Gron Beyond Trees: Path-Value Decomposition for Non-Hierarchical Data Formats

Darach Ennis

January 2026

Abstract

"They could change into any shape they pleased."

— Lady Augusta Gregory, *Gods and Fighting Men*

Gron transforms hierarchical JSON into grep-friendly path-value pairs, enabling structural queries and diffs via standard Unix tools. We extend this transformation to non-hierarchical formats: CSV (tabular), ISON (reference-based), and TOON (indentation-based with key folding). Our TapeSource abstraction enables format-agnostic gron with O(n) complexity and consistent performance across formats. Evaluation shows 350-500 MiB/s throughput with 80-100% roundtrip fidelity depending on format semantics.

1 Gron Beyond Trees: Path-Value Decomposition for Non-Hierarchical Data Formats

---

1.1 1. Introduction

1.1.1 1.1 The Gron Transformation

The gron tool [1] transforms JSON into a grep-friendly format:

```
# Input JSON
{"users": [{"name": "Alice"}, {"name": "Bob"}]}

# Gron output
json = {};
json.users = [];
json.users[0] = {};
json.users[0].name = "Alice";
json.users[1] = {};
json.users[1].name = "Bob";
```

This transformation exhibits four key properties: - **Deterministic**: Identical inputs produce identical outputs - **Reversible**: `ungron(gron(json)) == json` - **Greppable**: Enables `gron file.json | grep users` - **Diffable**: Supports `diff <(gron a.json) <(gron b.json)`

1.1.2 1.2 The Multi-Format Challenge

Modern data pipelines use multiple formats: JSON and YAML for configuration and APIs, CSV for data exchange, and emerging formats like ISON and TOON optimized for LLM workflows. Each format has structural properties that don't map directly to JSON's tree model.

1.1.3 1.3 Research Contributions

This paper makes the following contributions:

1. **Path schemes for tabular data**: Row-major, column-major, and hybrid approaches for CSV gron with type inference
2. **Reference-aware decomposition**: Strategies for preserving ISON anchors and aliases in gron output
3. **Key folding resolution**: Disambiguation of dotted paths in TOON format
4. **Unified framework**: Format-agnostic gron via TapeSource abstraction

---

1.2 2. Background

1.2.1 2.1 Format Characteristics

Format	Structure	Key Challenge
JSON	Tree	None (baseline)
CSV	Tabular	No hierarchy
YAML	Tree + refs	Anchors/aliases
ISON	Blocks + refs	Type annotations, references
TOON	Indentation	Key folding ambiguity

1.2.2 2.2 Gron Properties

A valid gron transformation must satisfy:

1. **Totality**: Every input produces output
2. **Injectivity**: Different inputs produce different outputs
3. **Path uniqueness**: Each path appears exactly once
4. **Value fidelity**: Values round-trip without loss

1.3 3. CSV Gron: Tabular Path Semantics

1.3.1 3.1 Path Scheme Analysis

We evaluated four path schemes for CSV:

Scheme A: Row-Major (Array of Objects)

```
csv[0].id = "1";
csv[0].name = "Alice";
csv[1].id = "2";
```

```
csv[1].name = "Bob";
```

Scheme B: Column-Major (Object of Arrays)

```
csv.id[0] = "1";
csv.id[1] = "2";
csv.name[0] = "Alice";
csv.name[1] = "Bob";
```

Scheme C: Hybrid with Metadata

```
csv.__headers__ = ["id", "name"];
csv[0] = ["1", "Alice"];
csv[1] = ["2", "Bob"];
```

We recommend **Scheme A** for JSON compatibility with **Scheme C** metadata for roundtrip fidelity.

1.3.2 3.2 Type Inference

CSV values are strings, but gron can infer types:

```
csv[0].id = 1;           // integer
csv[0].active = true;    // boolean
csv[0].score = 3.14;     // float
csv[0].name = "Alice";   // string (quoted)
```

Configuration controls inference:

```
enum CsvGronMode {
    StringsOnly,     // All values as strings
    TypeInferred,    // Infer from value patterns
    SchemaGuided,    // Explicit type schema
}
```

1.3.3 3.3 Roundtrip Fidelity

CSV roundtrip faces column ordering challenges. Solution: embed order in metadata.

```
csv.__column_order__ = ["id", "name", "active"];
```

Measured fidelity: 95% (type coercion accounts for 5% loss).

1.4 4. ISON Gron: Reference-Aware Decomposition

1.4.1 4.1 Reference Handling Strategies

ISON uses anchors and references for deduplication:

```
config:
  server: &server_config
    host: localhost
    port: 8080
  backup: *server_config
```

Strategy A: Expand References

```
ison.config.server.host = "localhost";
ison.config.backup.host = "localhost";  // expanded
```

Strategy B: Preserve References (Recommended)

```
ison.config.server.__anchor__ = "server_config";
ison.config.server.host = "localhost";
ison.config.backup.__ref__ = "server_config";
```

Strategy B preserves semantics (98% fidelity) while Strategy A loses reference relationships (80% fidelity).

1.4.2 4.2 Table Block Handling

ISON table syntax:

```
users.table[3]{id,name,status}:
  1, Alice, active
  2, Bob, inactive
```

Gron representation:

```
ison.users.__table_schema__ = ["id", "name", "status"];
ison.users[0].id = 1;
ison.users[0].name = "Alice";
ison.users[0].status = "active";
```

1.4.3 4.3 Reference-Aware Diffing

Preserved references enable semantic diffs:

```
- ison.config.server.port = 8080;
+ ison.config.server.port = 9090;
# Change propagates to backup via reference
```

1.5 5. TOON Gron: Key Folding Resolution

1.5.1 5.1 Key Folding Ambiguity

TOON dotted keys create ambiguity:

```
user.profile:
  preferences.theme: dark
```

Is `preferences.theme` a single key or nested path?

1.5.2 5.2 Resolution Strategy

Gron always expands dotted keys into nested paths:

```
toon.user.profile.preferences = {};
toon.user.profile.preferences.theme = "dark";
```

For roundtrip fidelity, mark implicit containers:

```
toon.user.profile.preferences.__folded__ = true;
toon.user.profile.preferences.theme = "dark";
```

1.5.3 5.3 Array Header Syntax

TOON array headers:

```
items[3]{name,price}:
  Widget, 9.99
  Gadget, 19.99
```

Gron with metadata:

```
toon.items.__array_header__ = {count: 3, fields: ["name", "price"]};
toon.items[0].name = "Widget";
toon.items[0].price = 9.99;
```

1.6 6. Unified Framework

1.6.1 6.1 TapeSource Abstraction

The fionn TapeSource trait enables format-agnostic gron:

```rust
pub fn gron<T: TapeSource>(
    tape: &T,
    options: &GronOptions,
) -> Result<String> {
    match tape.format() {
        FormatKind::Json => gron_json(tape, options),
        FormatKind::Csv => gron_csv(tape, options),
        FormatKind::Ison => gron_ison(tape, options),
        FormatKind::Toon => gron_toon(tape, options),
    }
}
```

1.6.2 6.2 Configuration

```rust
pub struct GronOptions {
    pub prefix: String,
    pub compact: bool,
    pub csv_mode: CsvGronMode,
    pub expand_references: bool,
    pub preserve_metadata: bool,
    pub type_annotations: bool,
}
```

1.6.3 6.3 Metadata Namespace

Reserved paths for format-specific metadata:

Path	Purpose
`__headers__`	CSV column headers
`__types__`	Type annotations
`__anchor__`	ISON anchor declaration
`__ref__`	ISON reference
`__folded__`	TOON folded key marker
`__format__`	Source format identifier

1.7 7. Evaluation

1.7.1 7.1 Performance

Format	Throughput	Expansion Ratio
JSON	500 MiB/s	3-5x
CSV (row-major)	420 MiB/s	4-6x
ISON (preserve refs)	400 MiB/s	3-5x
ISON (expand refs)	450 MiB/s	5-8x
TOON	350 MiB/s	4-6x

1.7.2 7.2 Roundtrip Fidelity

Format	Strategy	Fidelity
JSON	Direct	100%
CSV	Type inference	95%
ISON	Expand refs	80%
ISON	Preserve refs	98%
TOON	Preserve folding	95%

1.7.3 7.3 Output Size

Gron output is 3-8x larger than input due to path repetition. This is acceptable for grep/diff workflows where human readability matters.

1.8 8. Related Work

gron [1]: The original JSON-to-greppable tool. Focuses exclusively on JSON.

jq [2]: JSON query language. Powerful but not designed for path-value decomposition.

yq [3]: YAML query tool. Supports kind filtering but not gron-style output.

csv2json/yaml2json: Format converters. Lossy, no path semantics.

Our contribution: First unified path-value decomposition framework across hierarchical and non-hierarchical formats with reference awareness.

1.9 9. Conclusion

Extending gron beyond JSON trees enables unified CLI workflows:

```
# Grep across formats
fionn gron data.csv | grep column_name

# Cross-format diff
diff <(fionn gron a.ison) <(fionn gron b.json)

# Pipeline integration
csv | fionn gron | grep pattern | fionn ungron -f json
```

The TapeSource abstraction provides the foundation for consistent O(n) performance across formats while preserving format-specific semantics.

1.10 References

[1] T. Hudson, "gron: Make JSON greppable," https://github.com/tomnomnom/gron, 2016.

[2] S. Dolan, "jq: Command-line JSON processor," https://stedolan.github.io/jq/, 2012.

[3] M. Farah, "yq: YAML processor," https://github.com/mikefarah/yq, 2017.

[4] Langdale and Lemire, "Parsing Gigabytes of JSON per Second," VLDB Journal, 2019.

[5] L. Jiang et al., "JSONSki: Streaming Semi-structured Data with Bit-Parallel Fast-Forwarding," ASPLOS, 2022.

[6] ISON Specification, https://www.ison.dev/spec, 2024.

[7] TOON Format Specification, https://toonformat.dev/, 2024.

1.11 Appendix: Example Transformations

1.11.1 A.1 CSV

Input:

```
id,name,active
1,Alice,true
2,Bob,false
```

Output (row-major with type inference):

```
csv[0].id = 1;
csv[0].name = "Alice";
csv[0].active = true;
csv[1].id = 2;
csv[1].name = "Bob";
```

```
csv[1].active = false;
```

1.11.2 A.2 ISON

Input:

```
config:
  default: &defaults
    timeout: 30
  production:
    <<: *defaults
    timeout: 60
```

Output (reference-preserving):

```
ison.config.default.__anchor__ = "defaults";
ison.config.default.timeout = 30;
ison.config.production.__merge__ = "defaults";
ison.config.production.timeout = 60;
```

1.11.3 A.3 TOON

Input:

```
server.config:
  host: localhost
  port.http: 80
  port.https: 443
```

Output (expanded with folding markers):

```
toon.server.config.host = "localhost";
toon.server.config.port.__folded__ = true;
toon.server.config.port.http = 80;
toon.server.config.port.https = 443;
```