An Introduction to GraphQL
Tutorial at ISWC 2019, October 27, 2019

4. Fundamental Properties

Olaf Hartig[a], Ruben Taelman[b]

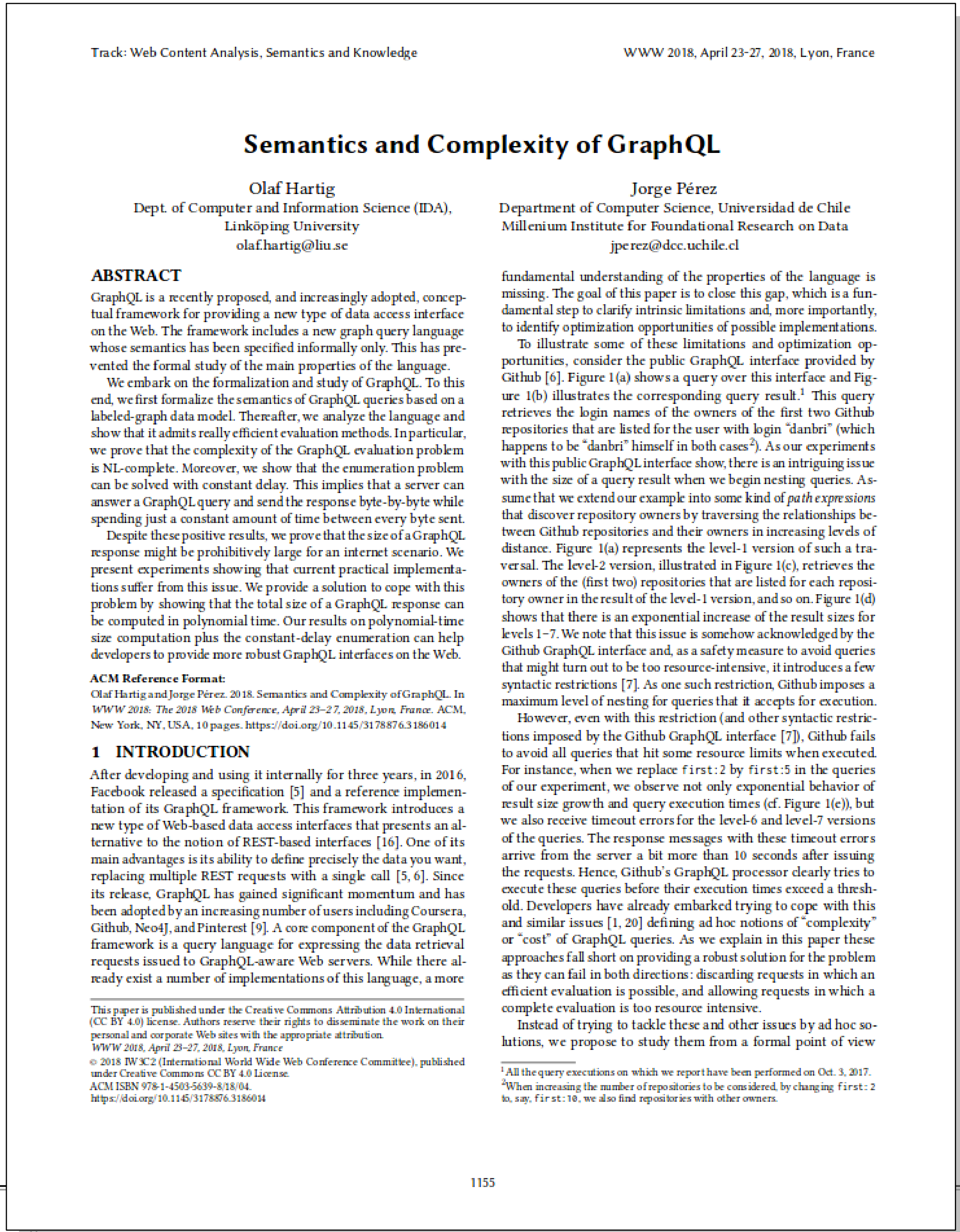
(a) Dept. of Computer and Information Science, Linköping University, Sweden

(b) Ghent University – imec – IDLab, Belgium

Joint work with **Jorge Pérez**
from the Universidad de Chile

Based on: O Hartig and J Pérez: *Semantics and Complexity of GraphQL.*
In Proceedings of The Web Conference 2018 (WWW 2018).

LINKÖPING UNIVERSITY



Semantics and Complexity of GraphQL

Olaf Hartig
Dept. of Computer and Information Science (IDA), Linköping University
olaf.hartig@liu.se

Jorge Pérez
Department of Computer Science, Universidad de Chile
Millenium Institute for Foundational Research on Data
jperez@dcc.uchile.cl

ABSTRACT

GraphQL is a recently proposed, and increasingly adopted, conceptual framework for providing a new type of data access interface on the Web. The framework includes a new graph query language whose semantics has been specified informally only. This has prevented the formal study of the main properties of the language.

We embark on the formalization and study of GraphQL. To this end, we first formalize the semantics of GraphQL queries based on a labeled-graph data model. Thereafter, we analyze the language and show that it admits really efficient evaluation methods. In particular, we prove that the complexity of the GraphQL evaluation problem is NL-complete. Moreover, we show that the enumeration problem can be solved with constant delay. This implies that a server can answer a GraphQL query and send the response byte-by-byte while spending just a constant amount of time between every byte sent.

Despite these positive results, we prove that the size of a GraphQL response might be prohibitively large for an internet scenario. We present experiments showing that current practical implementations suffer from this issue. We provide a solution to cope with this problem by showing that the total size of a GraphQL response can be computed in polynomial time. Our results on polynomial-time size computation plus the constant-delay enumeration can help developers to provide more robust GraphQL interfaces on the Web.

ACM Reference Format:
Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186014

1 INTRODUCTION

After developing and using it internally for three years, in 2016, Facebook released a specification [5] and a reference implementation of its GraphQL framework. This framework introduces a new type of Web-based data access interfaces that presents an alternative to the notion of REST-based interfaces [16]. One of its main advantages is its ability to define precisely the data you want, replacing multiple REST requests with a single call [5, 6]. Since its release, GraphQL has gained significant momentum and has been adopted by an increasing number of users including Coursera, Github, Neo4J, and Pinterest [9]. A core component of the GraphQL framework is a query language for expressing the data retrieval requests issued to GraphQL-aware Web servers. While there already exist a number of implementations of this language, a more fundamental understanding of the properties of the language is missing. The goal of this paper is to close this gap, which is a fundamental step to clarify intrinsic limitations and, more importantly, to identify optimization opportunities of possible implementations.

To illustrate some of these limitations and optimization opportunities, consider the public GraphQL interface provided by Github [6]. Figure 1(a) shows a query over this interface and Figure 1(b) illustrates the corresponding query result.[1] This query retrieves the login names of the owners of the first two Github repositories that are listed for the user with login "danbri" (which happens to be "danbri" himself in both cases[2]). As our experiments with this public GraphQL interface show, there is an intriguing issue with the size of a query result when we begin nesting queries. Assume that we extend our example into some kind of *path expressions* that discover repository owners by traversing the relationships between Github repositories and their owners in increasing levels of distance. Figure 1(a) represents the level-1 version of such a traversal. The level-2 version, illustrated in Figure 1(c), retrieves the owners of the (first two) repositories that are listed for each repository owner in the result of the level-1 version, and so on. Figure 1(d) shows that there is an exponential increase of the result sizes for levels 1–7. We note that this issue is somehow acknowledged by the Github GraphQL interface and, as a safety measure to avoid queries that might turn out to be too resource-intensive, it introduces a few syntactic restrictions [7]. As one such restriction, Github imposes a maximum level of nesting for queries that it accepts for execution.

However, even with this restriction (and other syntactic restrictions imposed by the Github GraphQL interface [7]), Github fails to avoid all queries that hit some resource limits when executed. For instance, when we replace `first:2` by `first:5` in the queries of our experiment, we observe not only exponential behavior of result size growth and query execution times (cf. Figure 1(e)), but we also receive timeout errors for the level-6 and level-7 versions of the queries. The response messages with these timeout errors arrive from the server a bit more than 10 seconds after issuing the requests. Hence, Github's GraphQL processor clearly tries to execute these queries before their execution times exceed a threshold. Developers have already embarked trying to cope with this and similar issues [1, 20] defining ad hoc notions of "complexity" or "cost" of GraphQL queries. As we explain in this paper these approaches fall short on providing a robust solution for the problem as they can fail in both directions: discarding requests in which an efficient evaluation is possible, and allowing requests in which a complete evaluation is too resource intensive.

Instead of trying to tackle these and other issues by ad hoc solutions, we propose to study them from a formal point of view

WWW 2018, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/10.1145/3178876.3186014

[1] All the query executions on which we report have been performed on Oct. 3, 2017.
[2] When increasing the number of repositories to be considered, by changing `first:2` to, say, `first:10`, we also find repositories with other owners.

Our Contributions in a Nutshell

Formal definition of the language

Study of **computational complexity**
(the language admits really efficient evaluation methods)

Solution to the problem of **large results**

How is the language defined in the spec?

2.4 Selection Sets

SelectionSet **:**
  { *Selection*$_{\text{list}}$ **}**

Selection **:**
  Field
  FragmentSpread
  InlineFragment

An operation selects the set of information it needs, and will receive exactly that information and nothing more, avoiding over-fetching and under-fetching data.

```
{
  id
  firstName
  lastName
}
```

In this query, the `id`, `firstName`, and `lastName` fields form a selection set. Selection sets may also contain fragment references.

How is the language defined in the spec?

2.4 Selection Sets

SelectionSet :
    { *Selection*$_{list}$ }

Selection :
    Field
    FragmentSpread
    InlineFragment

An operation selects the
nothing more, avoiding

```
{
  id
  firstName
  lastName
}
```

In this query, the `id`, `fi`
contain fragment refere

6.3 Executing Selection Sets

To execute a selection set, the object value being evaluated and the object type need to be known, as well as whether it must be executed serially, or may be executed in parallel.

First, the selection set is turned into a grouped field set; then, each represented field in the grouped field set produces an entry into a response map.

ExecuteSelectionSet(*selectionSet*, *objectType*, *objectValue*, *variableValues*) :

1. Let *groupedFieldSet* be the result of CollectFields(*objectType*, *selectionSet*, *variableValues*).
2. Initialize *resultMap* to an empty ordered map.
3. For each *groupedFieldSet* as *responseKey* and *fields*:
   a. Let *fieldName* be the name of the first entry in *fields*. Note: This value is unaffected if an alias is used.
   b. Let *fieldType* be the return type defined for the field *fieldName* of *objectType*.
   c. If *fieldType* is **null**:
      i. Continue to the next iteration of *groupedFieldSet*.
   d. Let *responseValue* be ExecuteField(*objectType*, *objectValue*, *fields*, *fieldType*, *variableValues*).
   e. Set *responseValue* as the value for *responseKey* in *resultMap*.
4. Return *resultMap*.

How is the

2.4 Selection Sets

SelectionSet :
{ $Selection_{list}$ }

Selection :
Field
FragmentSpread
InlineFragment

An operation selects the
nothing more, avoiding

```
{
  id
  firstName
  lastName
}
```

In this query, the `id`, `fi`
contain fragment refere

6.3 Ex

To e
wel

Firs
fiel

6.4 Executing Fields

Each field requested in the grouped field set that is defined on the selected objectType will resu
entry in the response map. Field execution first coerces any provided argument values, then re
value for the field, and finally completes that value either by recursively executing another sele
or coercing a scalar value.

ExecuteField(*objectType*, *objectValue*, *fieldType*, *fields*, *variableValues*) :

1. Let *field* be the first entry in *fields*.
2. Let *argumentValues* be the result of CoerceArgumentValues(*objectType*, *field*, *variableV*
3. Let *resolvedValue* be ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentVa*
4. Return the result of CompleteValue(*fieldType*, *fields*, *resolvedValue*, *variableValues*).

ExecuteSelectionSet(*selectionSet*, *objectType*, *objectValue*, *variableValues*) :

1. Let *groupedFieldSet* be the result of CollectFields(*objectType*, *selectionSet*, *variableValues*).
2. Initialize *resultMap* to an empty ordered map.
3. For each *groupedFieldSet* as *responseKey* and *fields*:
   a. Let *fieldName* be the name of the first entry in *fields*. Note: This value is unaffected if an alias is used.
   b. Let *fieldType* be the return type defined for the field *fieldName* of *objectType*.
   c. If *fieldType* is **null**:
      i. Continue to the next iteration of *groupedFieldSet*.
   d. Let *responseValue* be ExecuteField(*objectType*, *objectValue*, *fields*, *fieldType*, *variableValues*).
   e. Set *responseValue* as the value for *responseKey* in *resultMap*.
4. Return *resultMap*.

How is the

2.4 Selection Sets

SelectionSet :
  { *Selection*list }

Selection :
  Field
  FragmentSpread
  InlineFragment

An operation selects th
nothing more, avoidin

```
{
  id
  firstName
  lastName
}
```

In this query, the `id`, f
contain fragment refer

6.3 Ex

To e
wel

Firs
fiel

6.4 Executing Fields

Each field requested in the grouped field set that is defined on the selected objectType will resu
entry in the response map. Field execution first coerces any provided argument values, then re
value for the field, and finally completes that value either by recursively executing another sele
or coercing a scalar value.

ExecuteField(*objectType*, *objectValue*, *fieldType*, *fields*, *variableValues*) :

1. Let *field* be the first entry in *fields*.
2. Let *argumentValues* be the result of CoerceArgumentValues(*objectType*, *field*, *variableV*
3. Let *resolvedValue* be ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentVa*
4. Return the result of CompleteValue(*fieldType*, *fields*, *resolvedValue*, *variableValues*).

6.4.2 Value Resolution

While nearly all of GraphQL execution can be described generically, ultimately the internal system exposing the GraphQL interface must provide values. This is exposed via *ResolveFieldValue*, which produces a value for a given field on a type for a real value.

As an example, this might accept the *objectType* `Person`, the *field* `"soulMate"`, and the *objectValue* representing John Lennon. It would be expected to yield the value representing Yoko Ono.

ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*) :

1. Let *resolver* be the internal function provided by *objectType* for determining the resolved value of a field named *fieldName*.
2. Return the result of calling *resolver*, providing *objectValue* and *argumentValues*.

NOTE It is common for *resolver* to be asynchronous due to relying on reading an underlying database or networked service to produce a value. This necessitates the rest of a GraphQL executor to handle an asynchronous execution flow.

How is the

6.4 Executing Fields

Each field requested in the grouped field set that is defined on the selected objectType will resu
rces any provided argument values, then re
either by recursively executing another sele

ields, variableValues) :

1. Let *resolver* be the internal function providec
value of a field named *fieldName*.
2. Return the result of calling *resolver*, providir

2. Let *argumentValues* be the result of CoerceArgumentValues(*objectType*, *field*, *variableV*
3. Let *resolvedValue* be ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentVa*
4. Return the result of CompleteValue(*fieldType*, *fields*, *resolvedValue*, *variableValues*).

2.4 S

Field
FragmentSpread
InlineFragment

Firs
fielc

An operation selects th
nothing more, avoidin

```
{
  id
  firstName
  lastName
}
```

In this query, the `id`, f
contain fragment refer

6.4.2 Value Resolution

While nearly all of GraphQL execution can be described generically, ultimately the internal system exposing the GraphQL interface must provide values. This is exposed via *ResolveFieldValue*, which produces a value for a given field on a type for a real value.

As an example, this might accept the *objectType* `Person`, the *field* `"soulMate"`, and the *objectValue* representing John Lennon. It would be expected to yield the value representing Yoko Ono.

ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*) :

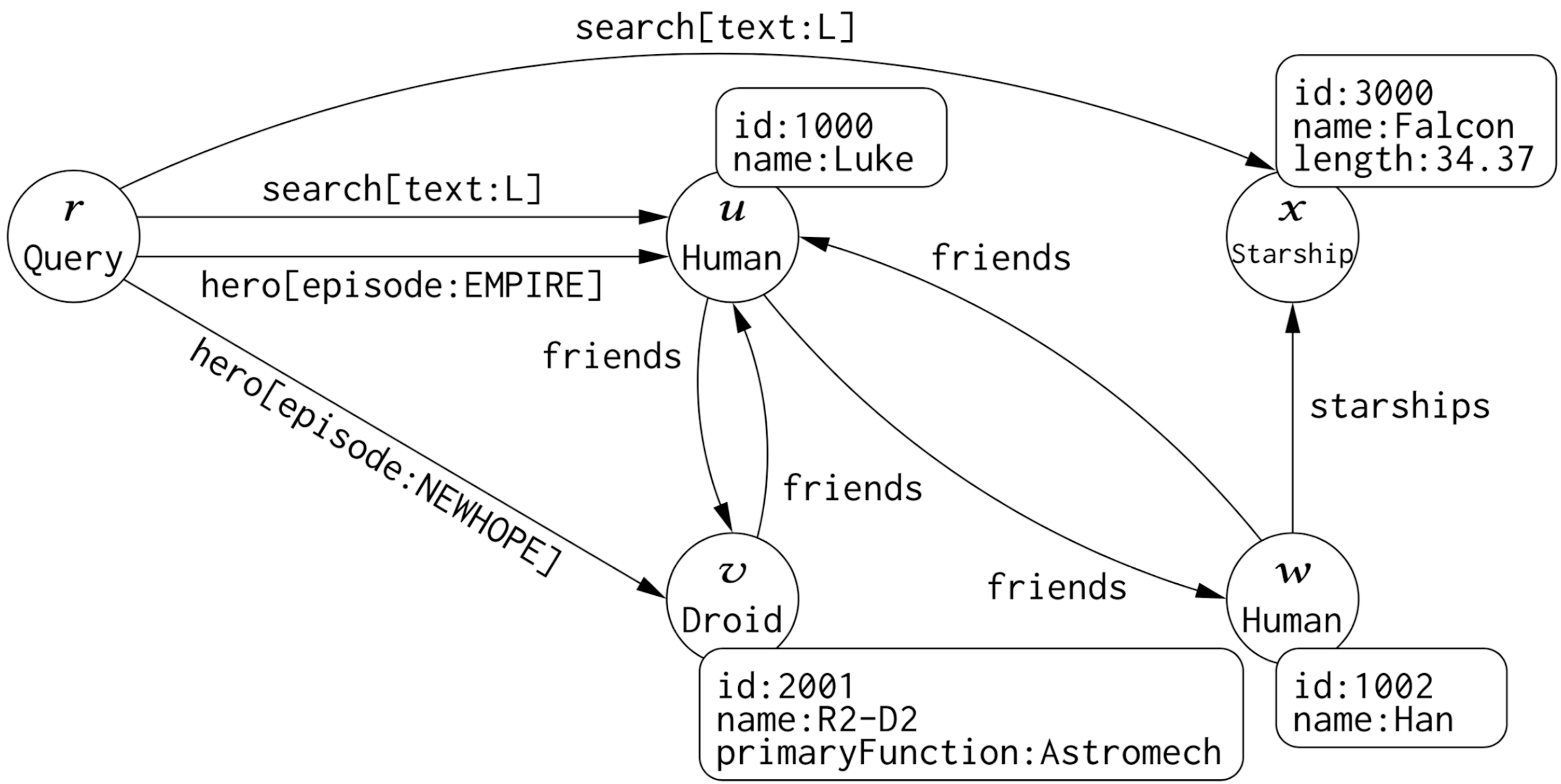
1. Let *resolver* be the internal function provided by *objectType* for determining the resolved value of a field named *fieldName*.
2. Return the result of calling *resolver*, providing *objectValue* and *argumentValues*.

NOTE It is common for *resolver* to be asynchronous due to relying on reading an underlying database or networked service to produce a value. This necessitates the rest of a GraphQL executor to handle an asynchronous execution flow.

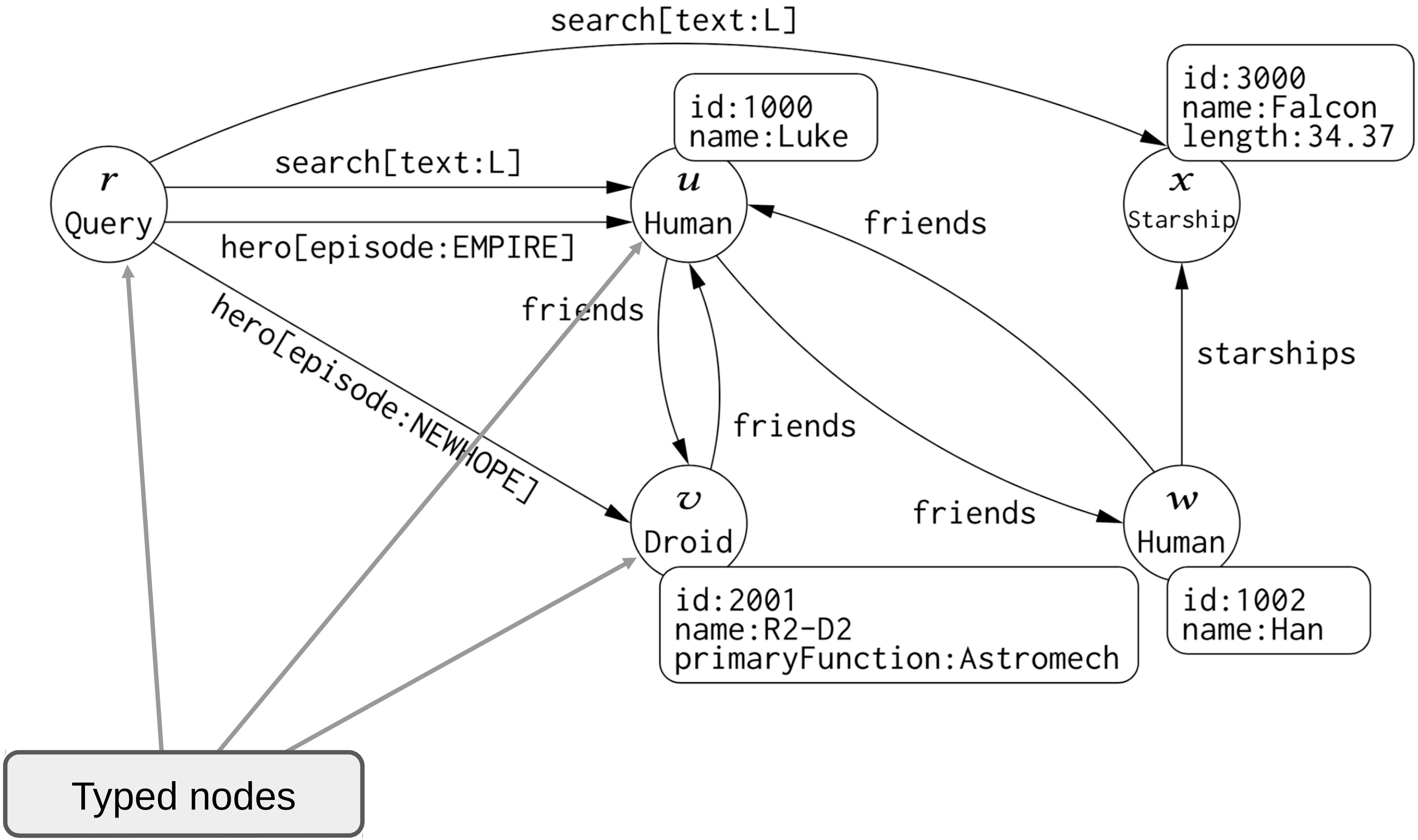
LINKÖP
UNIVER

Formalization of GraphQL

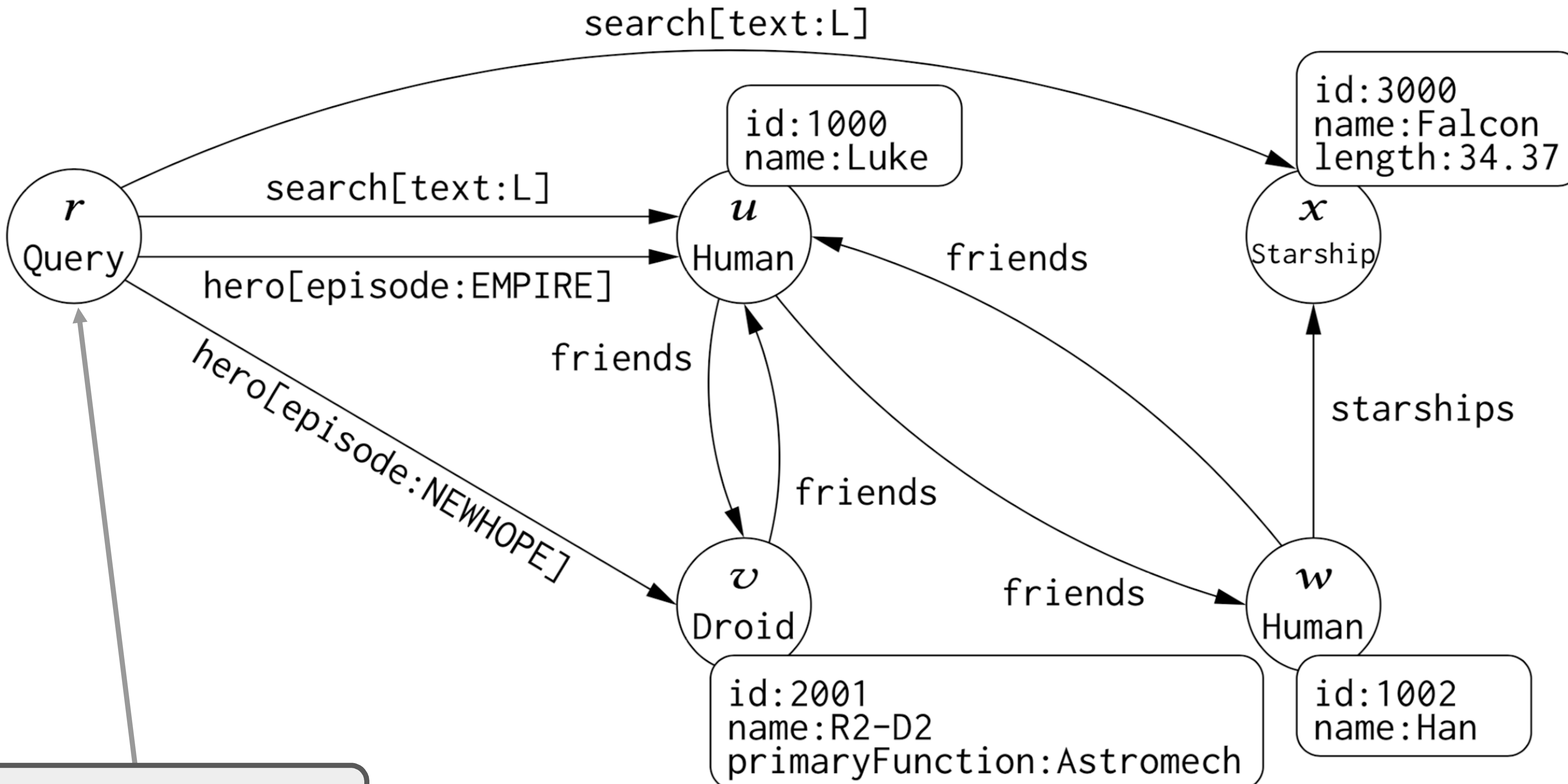
GraphQL Graphs (Our Formalization)



GraphQL Graphs (Our Formalization)



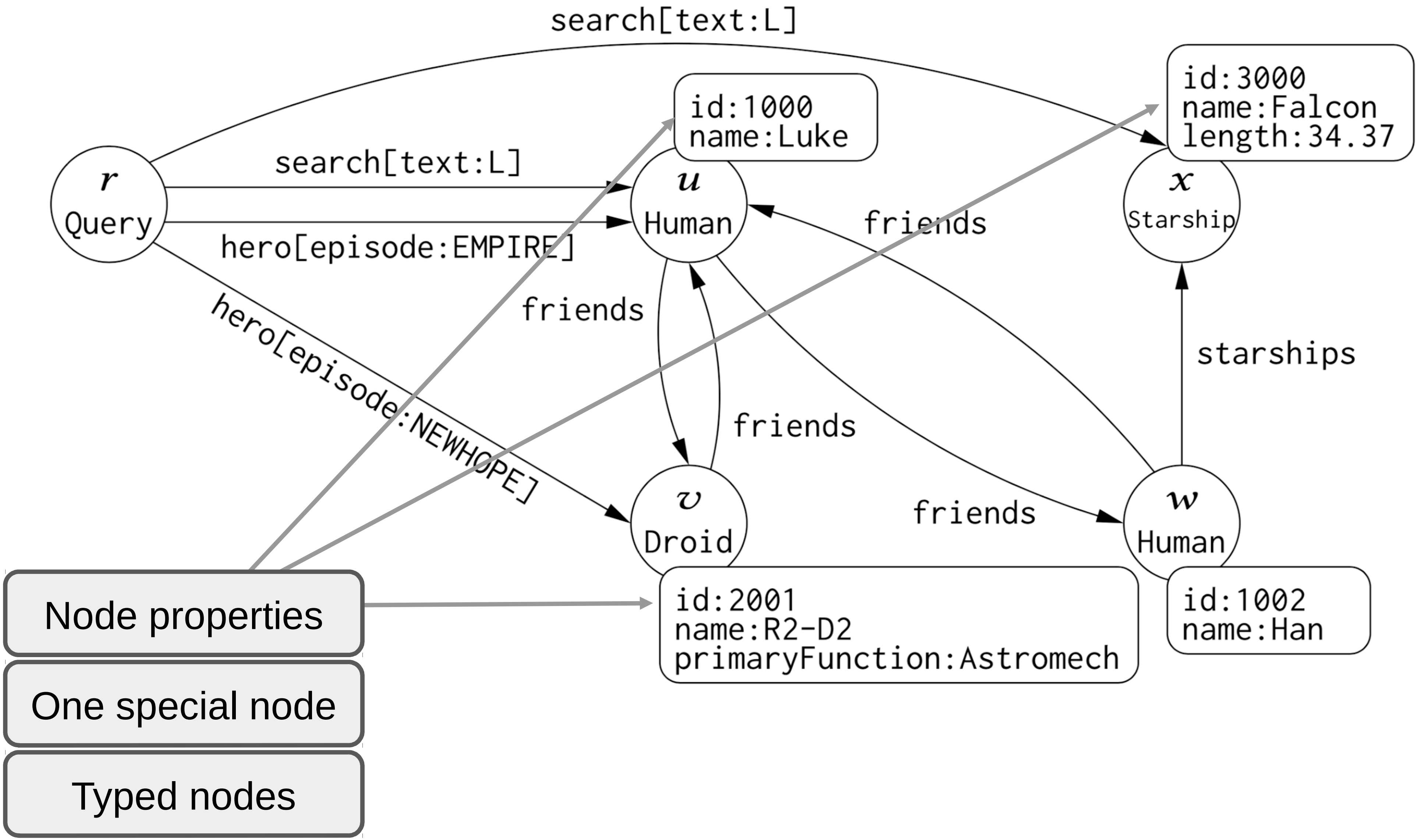
GraphQL Graphs (Our Formalization)



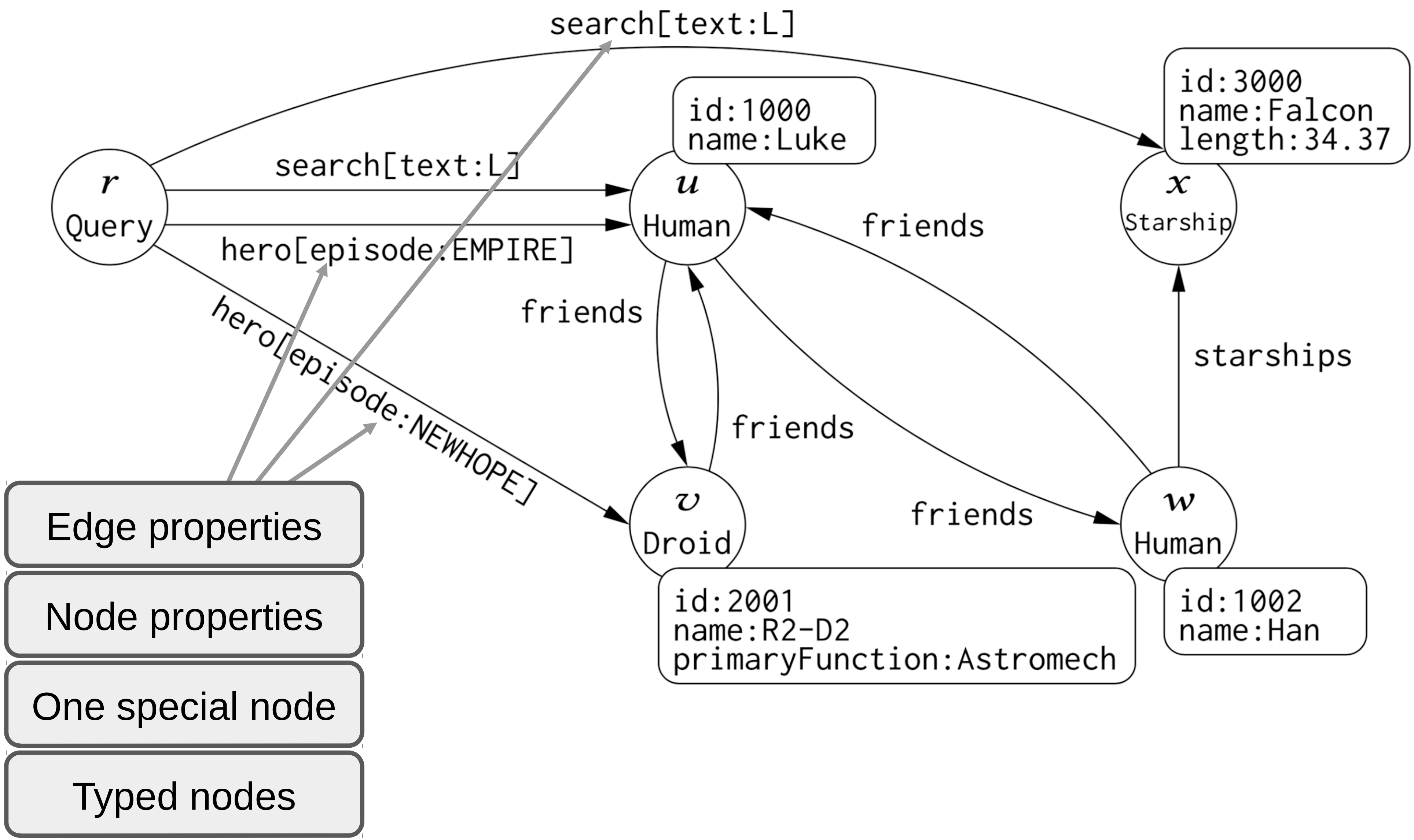
One special node

Typed nodes

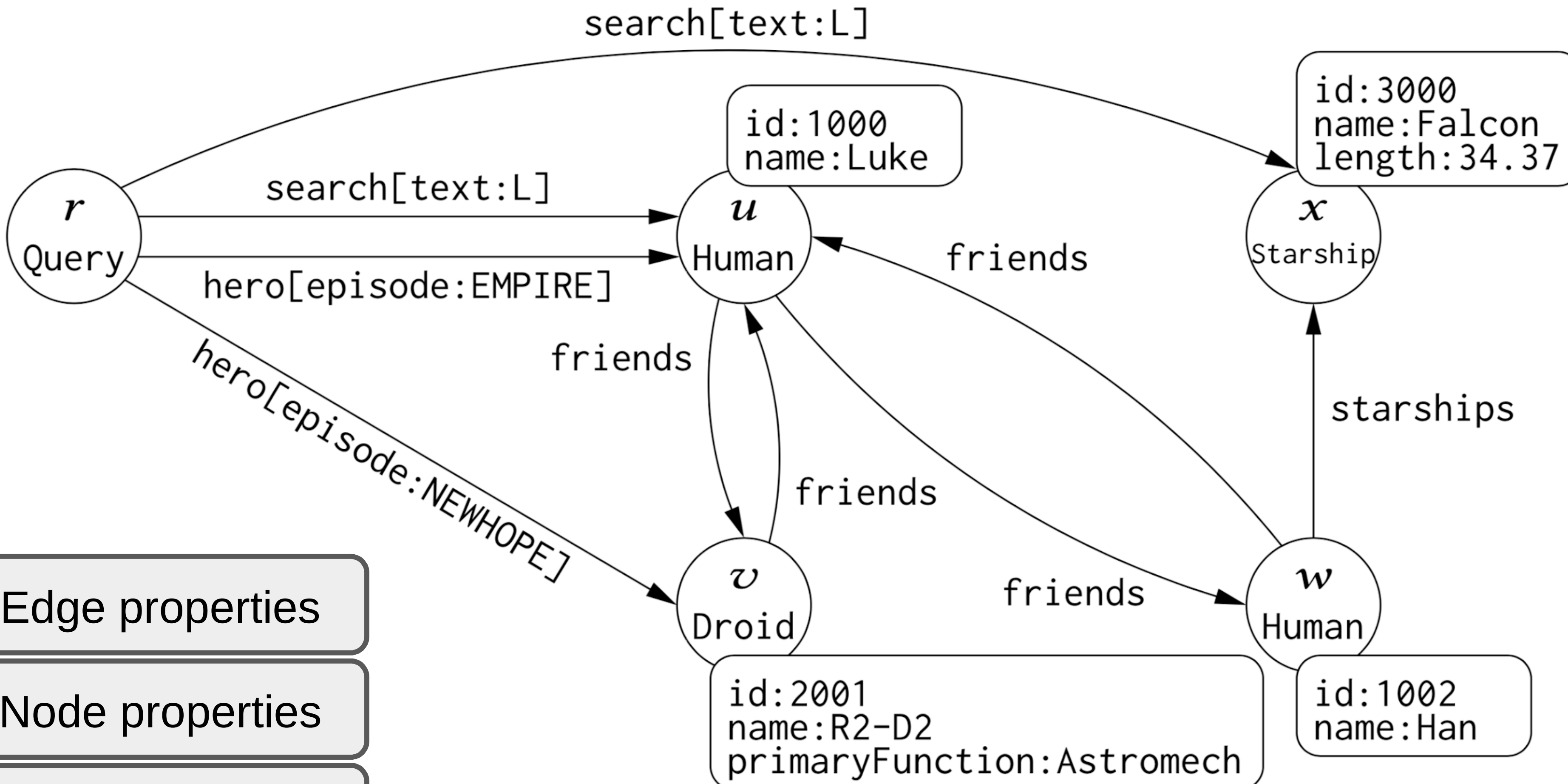
GraphQL Graphs (Our Formalization)



GraphQL Graphs (Our Formalization)



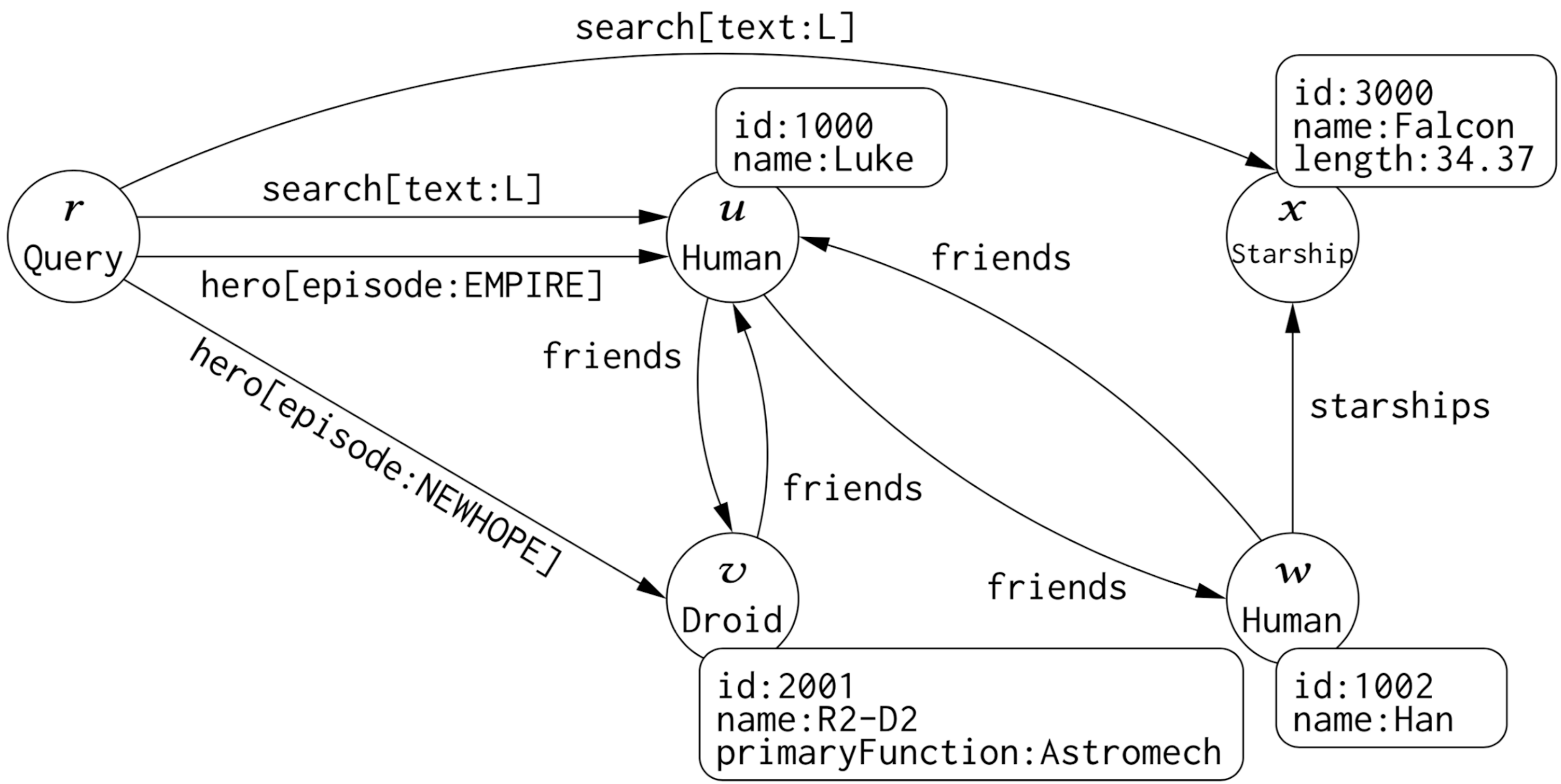
GraphQL Graphs (Our Formalization)



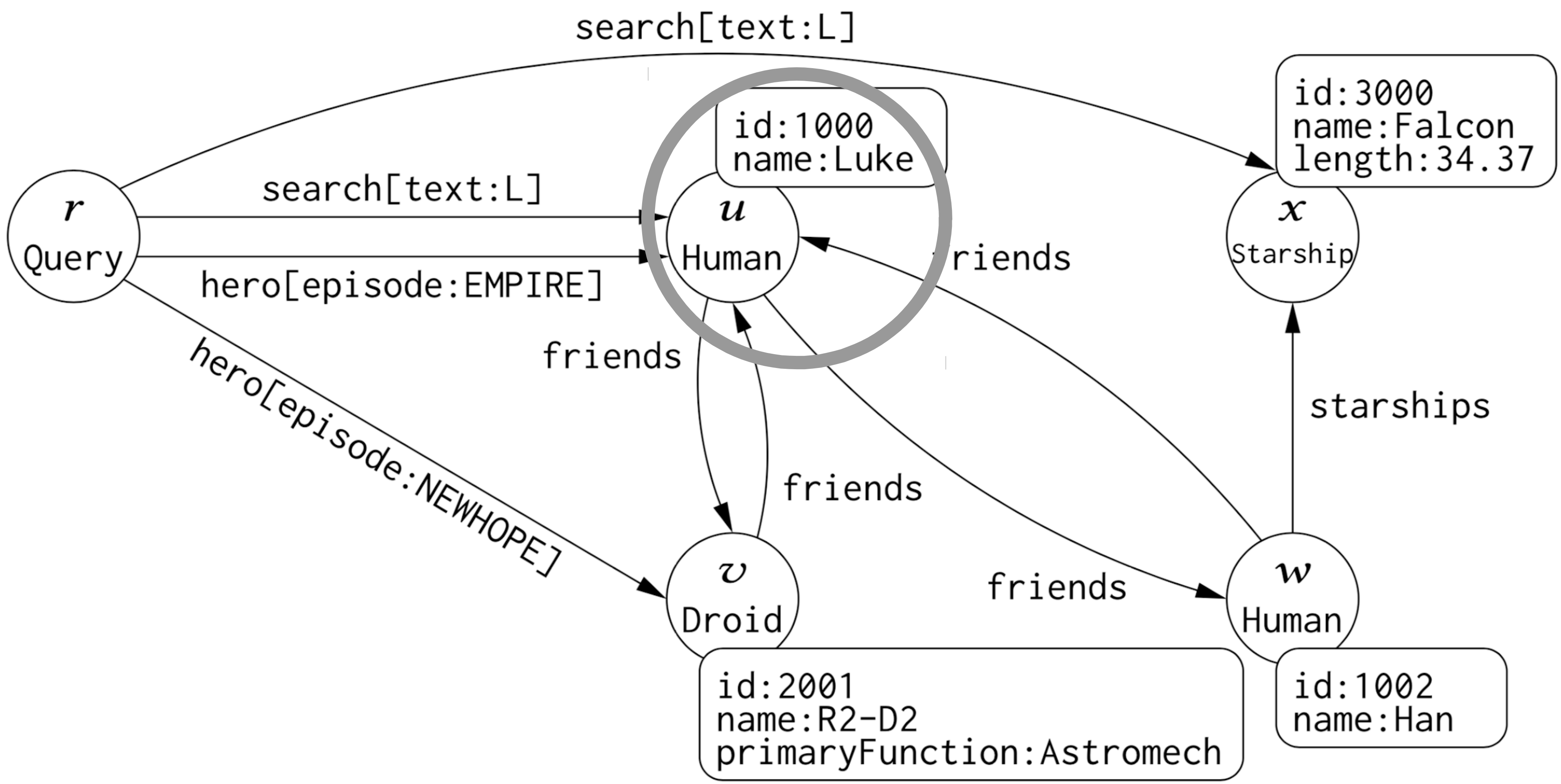
- Edge properties
- Node properties
- One special node
- Typed nodes

We also formalize the notions of *GraphQL schema* and *schema satisfaction* based on this data model

Formalization of Query Evaluation Function

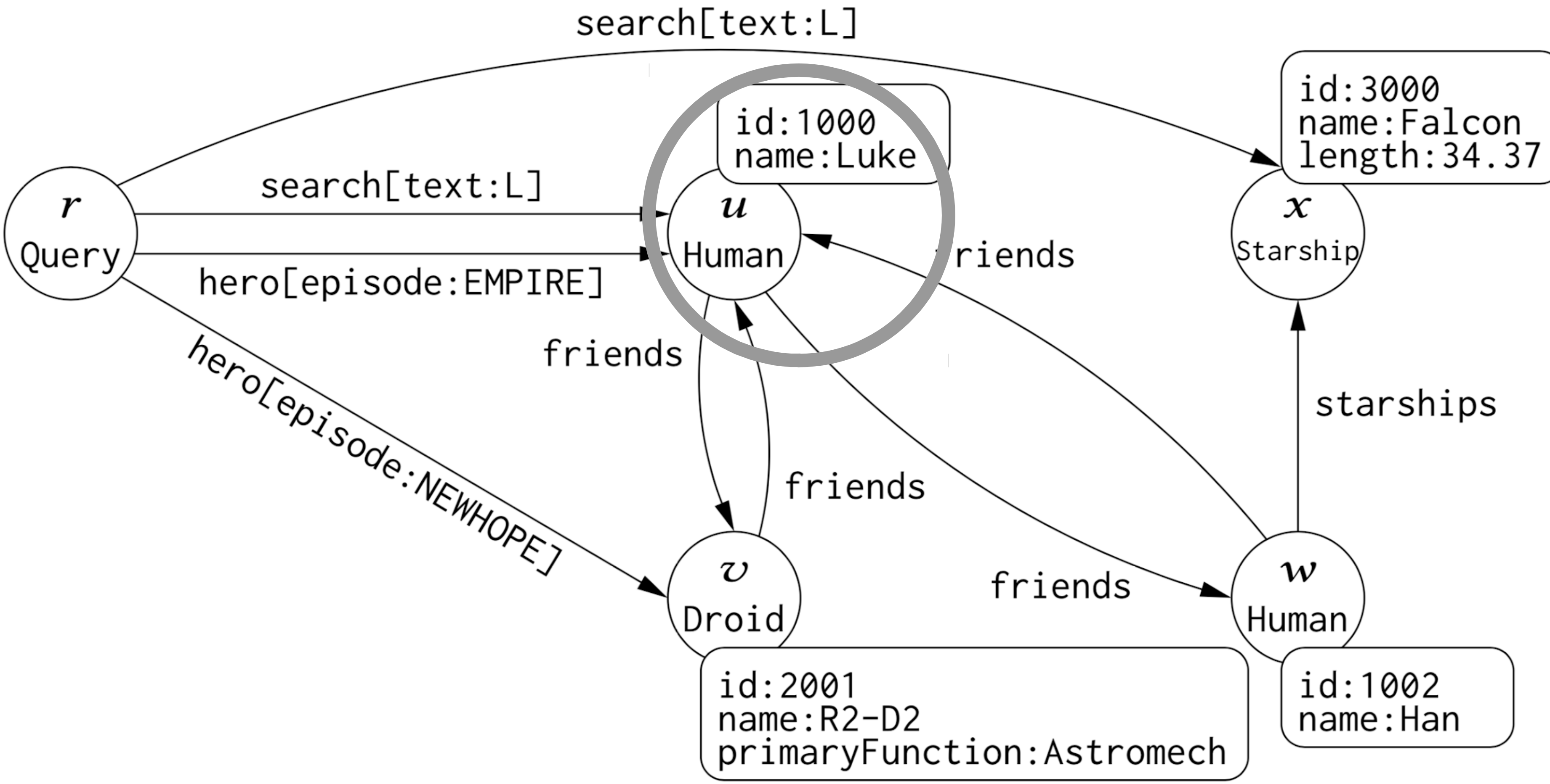


Formalization of Query Evaluation Function



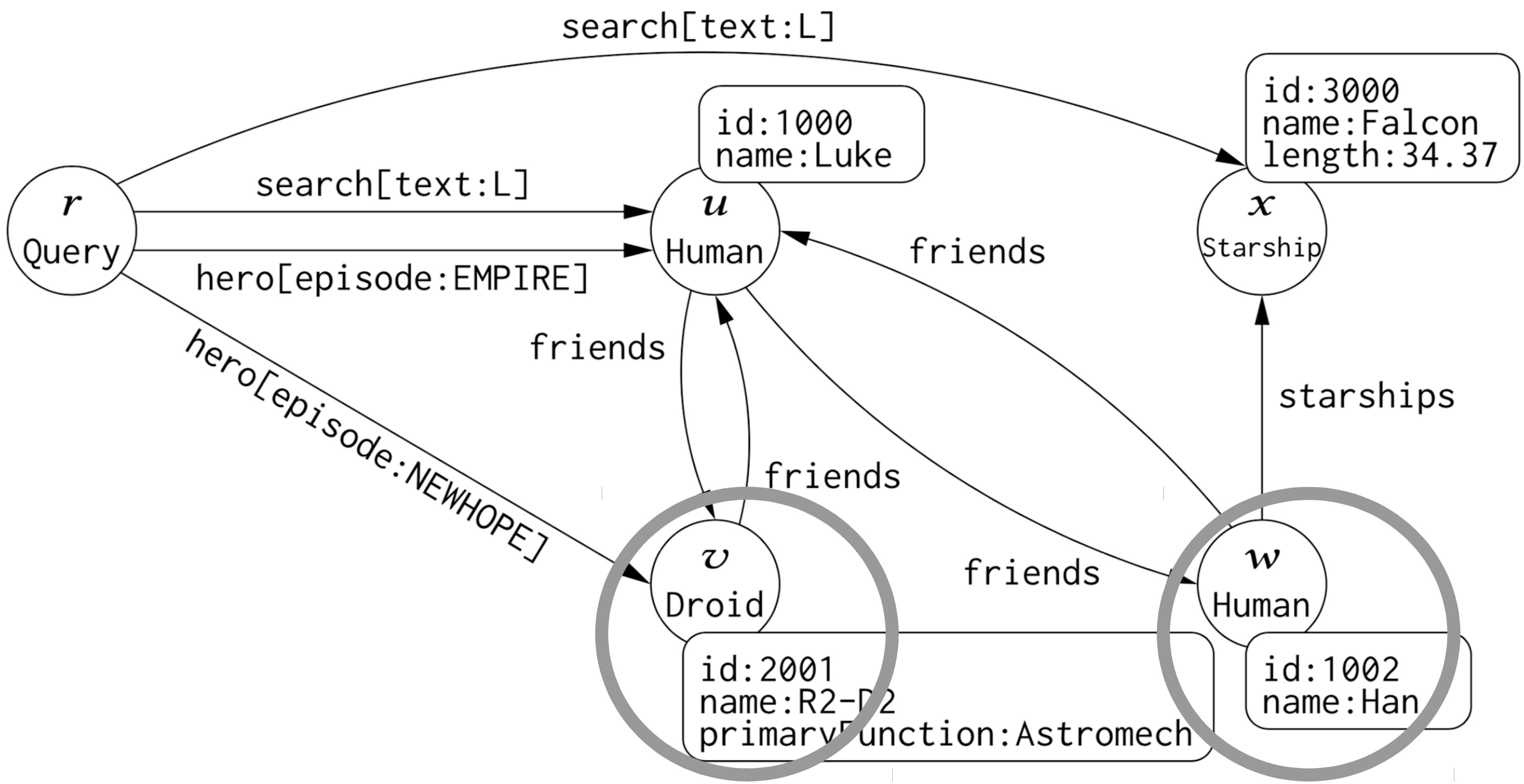
$[\![\text{friends \{ name \}}]\!]^u \quad =$

Formalization of Query Evaluation Function



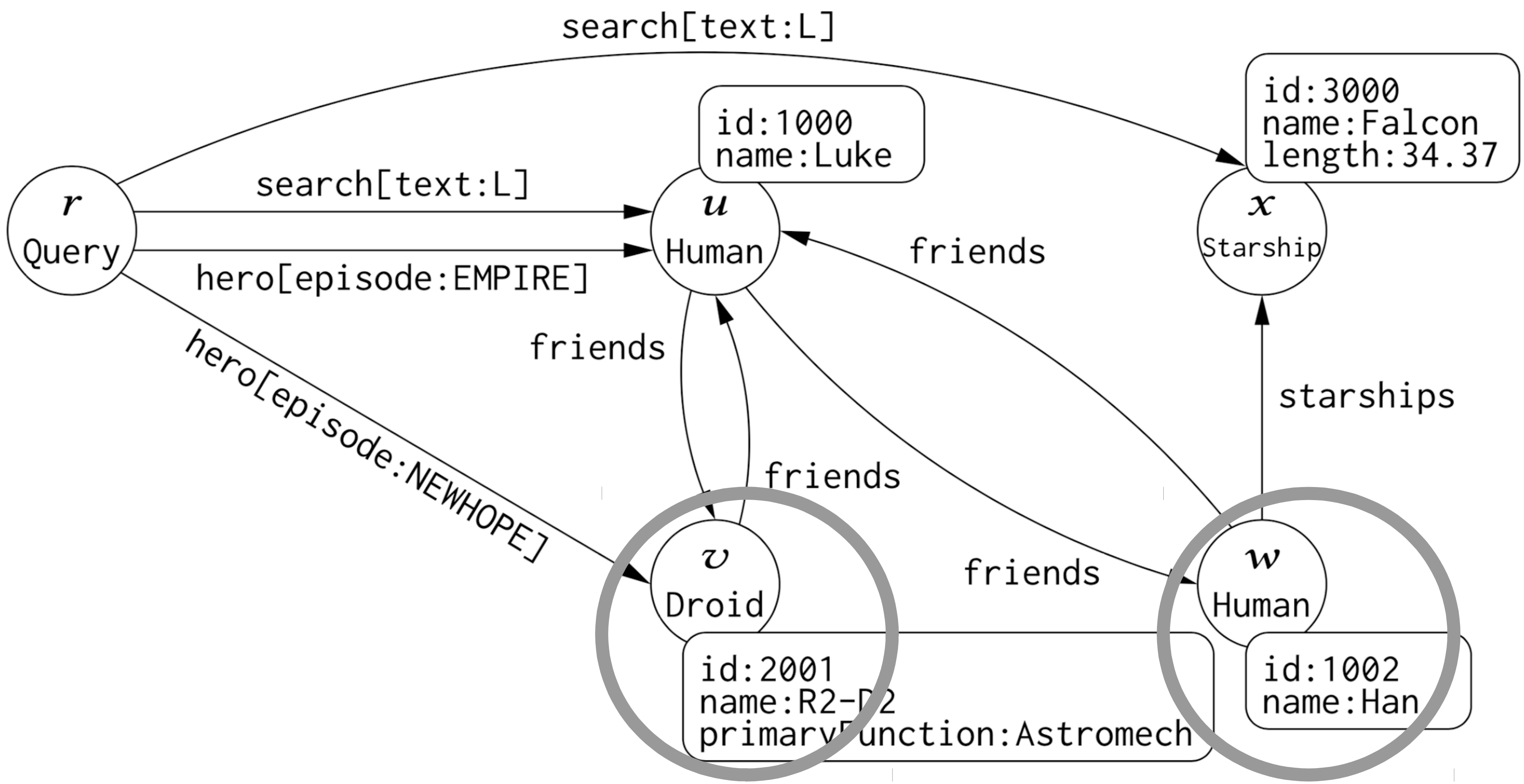
$[\![\text{friends \{ name \}}]\!]^u$ = friends: [ ]

Formalization of Query Evaluation Function



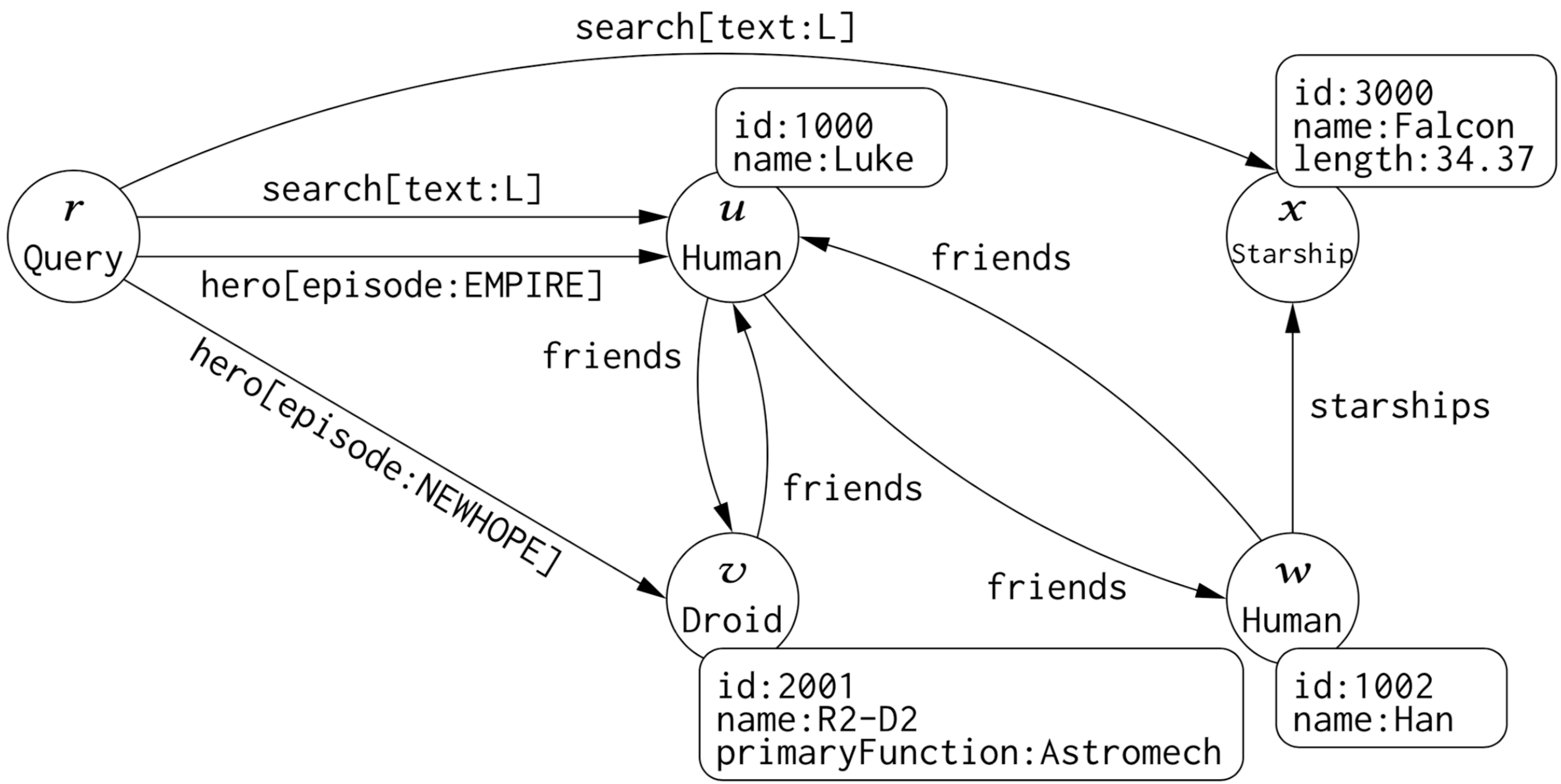
$$[\![\text{friends \{ name \}}]\!]^{u} \;=\; \text{friends: [\{ } [\![\text{name}]\!]^{v} \text{ \} \{ } [\![\text{name}]\!]^{w} \text{ \}]}$$

Formalization of Query Evaluation Function



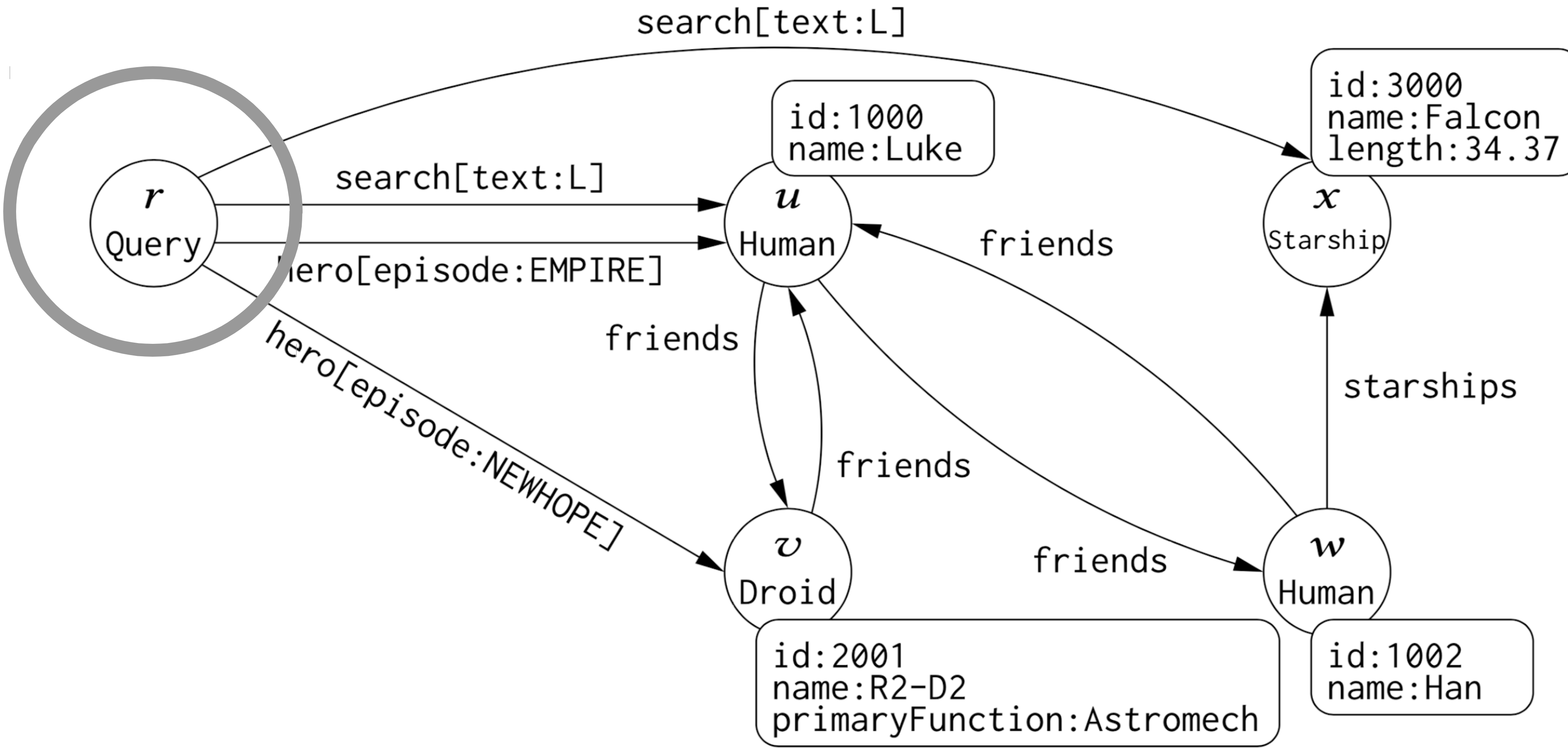
$[\![$friends { name }$]\!]^u$ = friends: [{name:R2-D2} {name:Han}]

Formalization of Query Evaluation Function



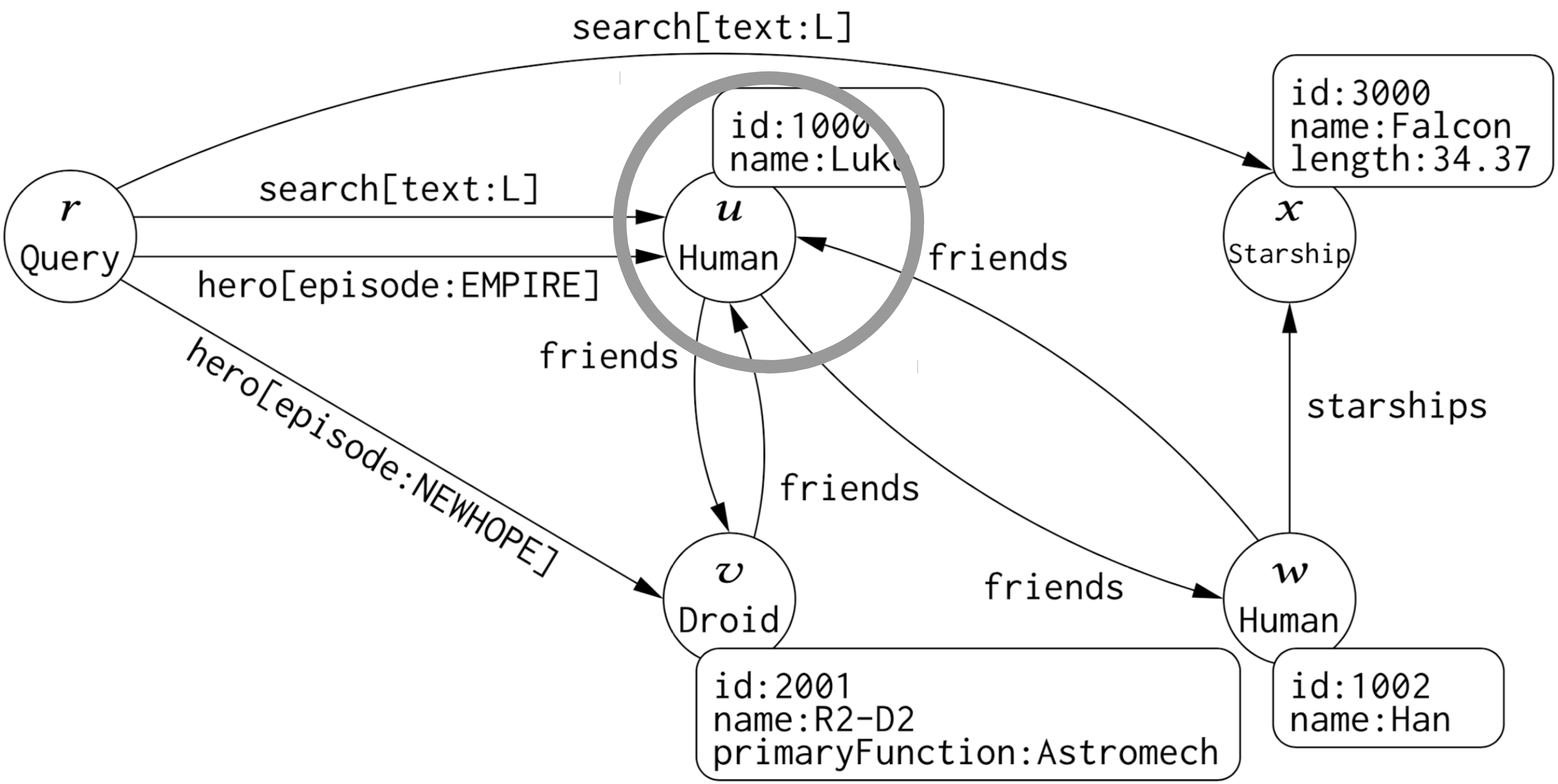
$[\![$ hero[episode:EMPIRE] { friends {name} } $]\!]$

Formalization of Query Evaluation Function



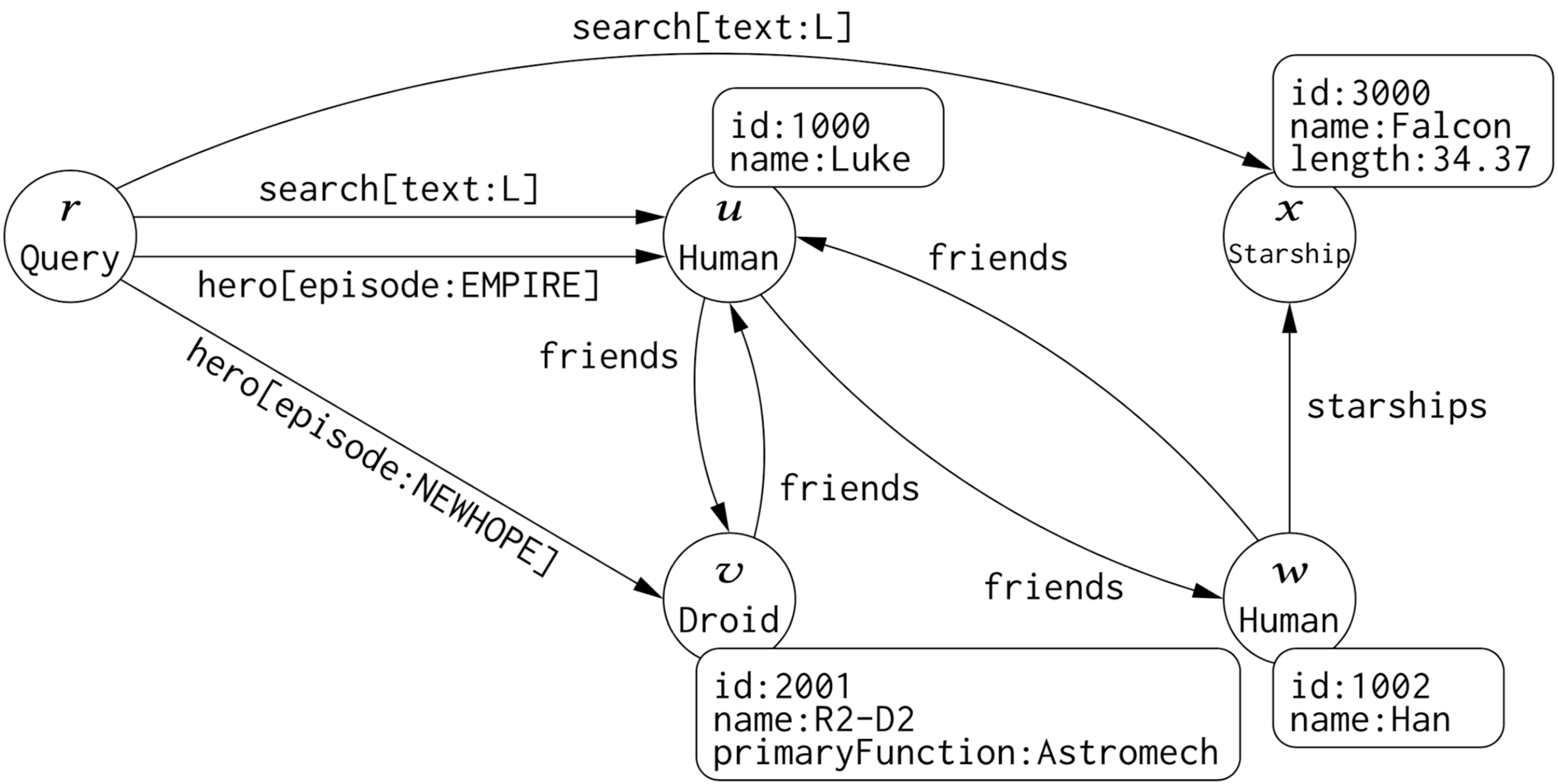
$[\![$ hero[episode:EMPIRE] { friends {name} } $]\!]^r$

Formalization of Query Evaluation Function



$[\![$ hero[episode:EMPIRE] { friends {name} } $]\!]^r$ = hero: { $[\![$ friends {name} $]\!]^u$ }

Formalization of Query Evaluation Function



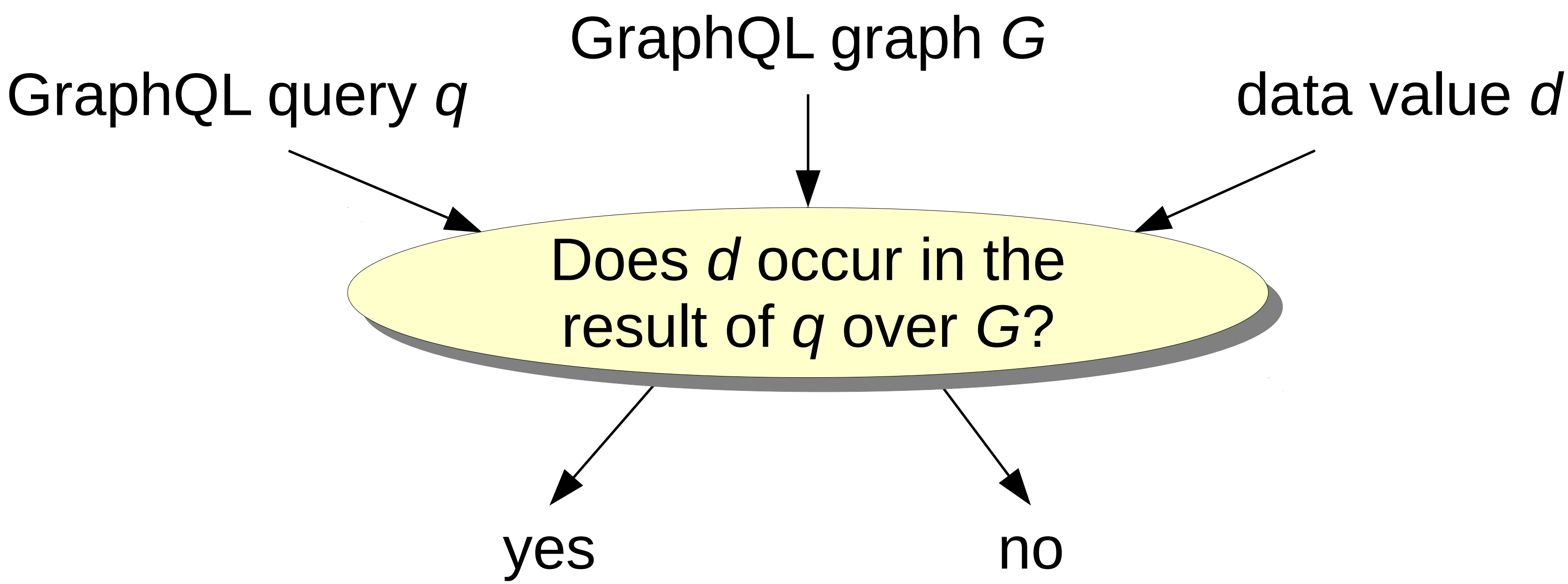
$\llbracket$ hero[episode:EMPIRE] { friends {name} } $\rrbracket^{r}$ = hero: { $\llbracket$ friends {name} $\rrbracket^{u}$ }

= hero: { friends: [{name:R2-D2} {name:Han}] }

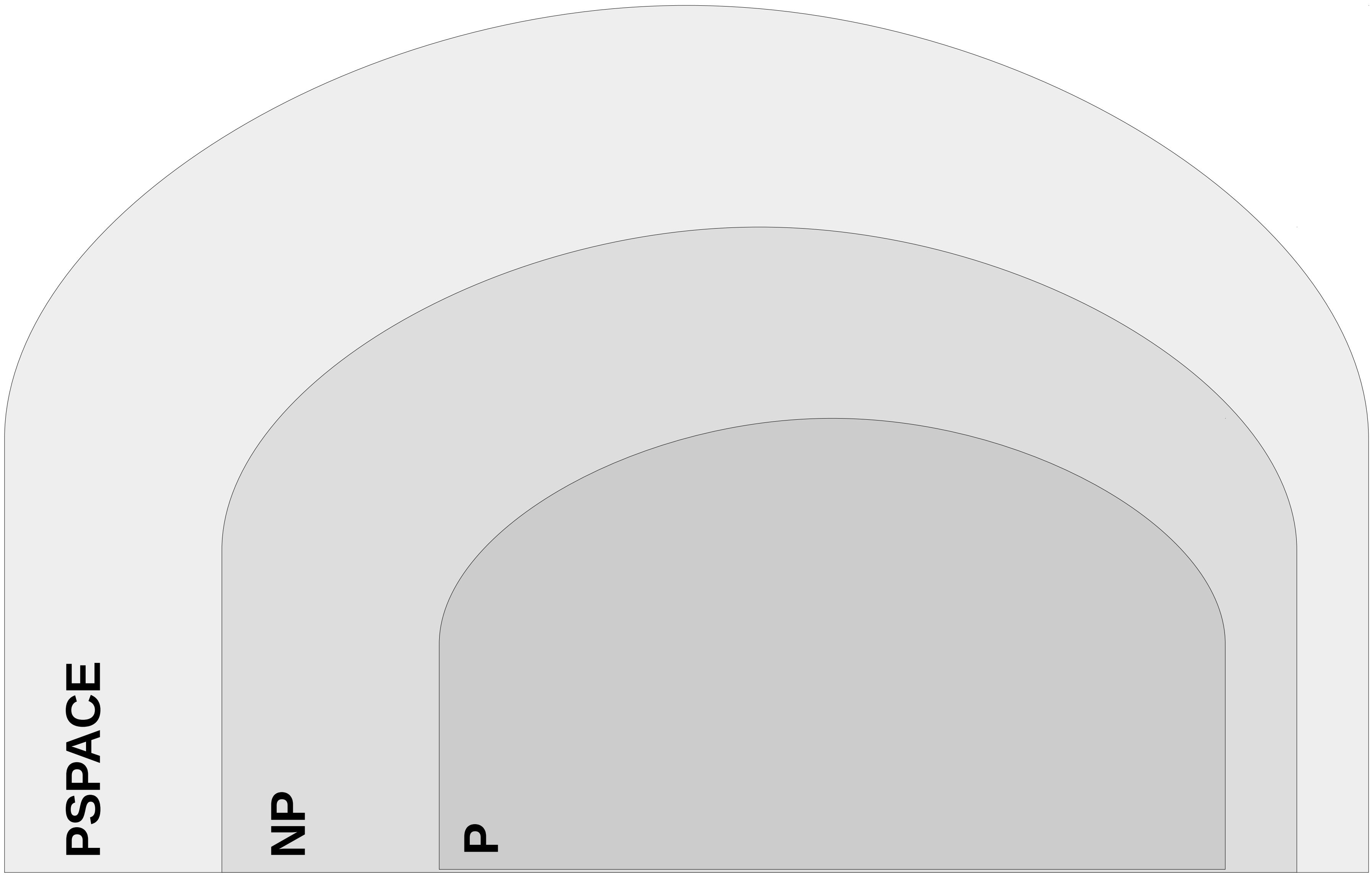
Complexity Analysis

Evaluation Problem

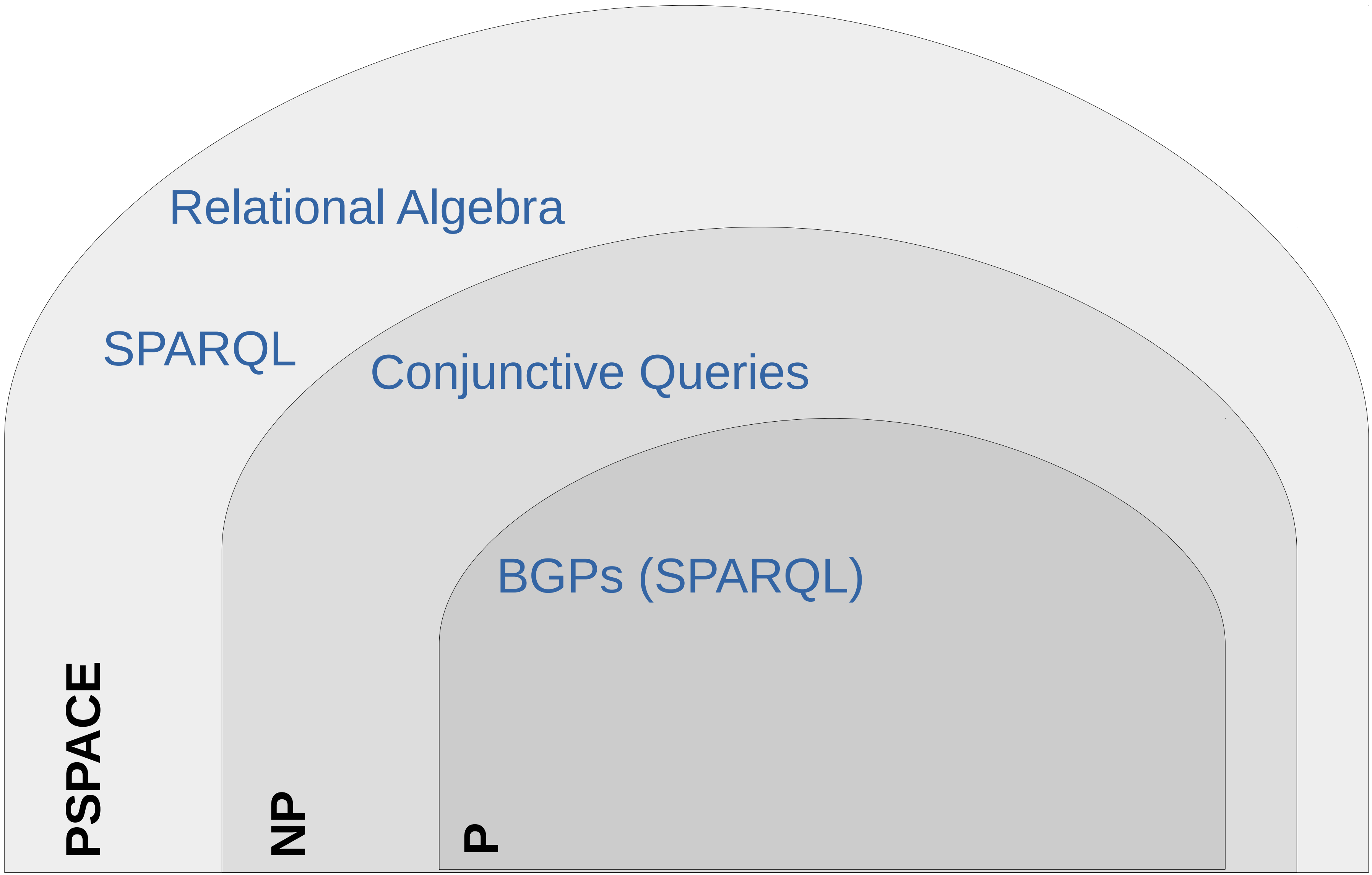
Evaluation Problem of GraphQL



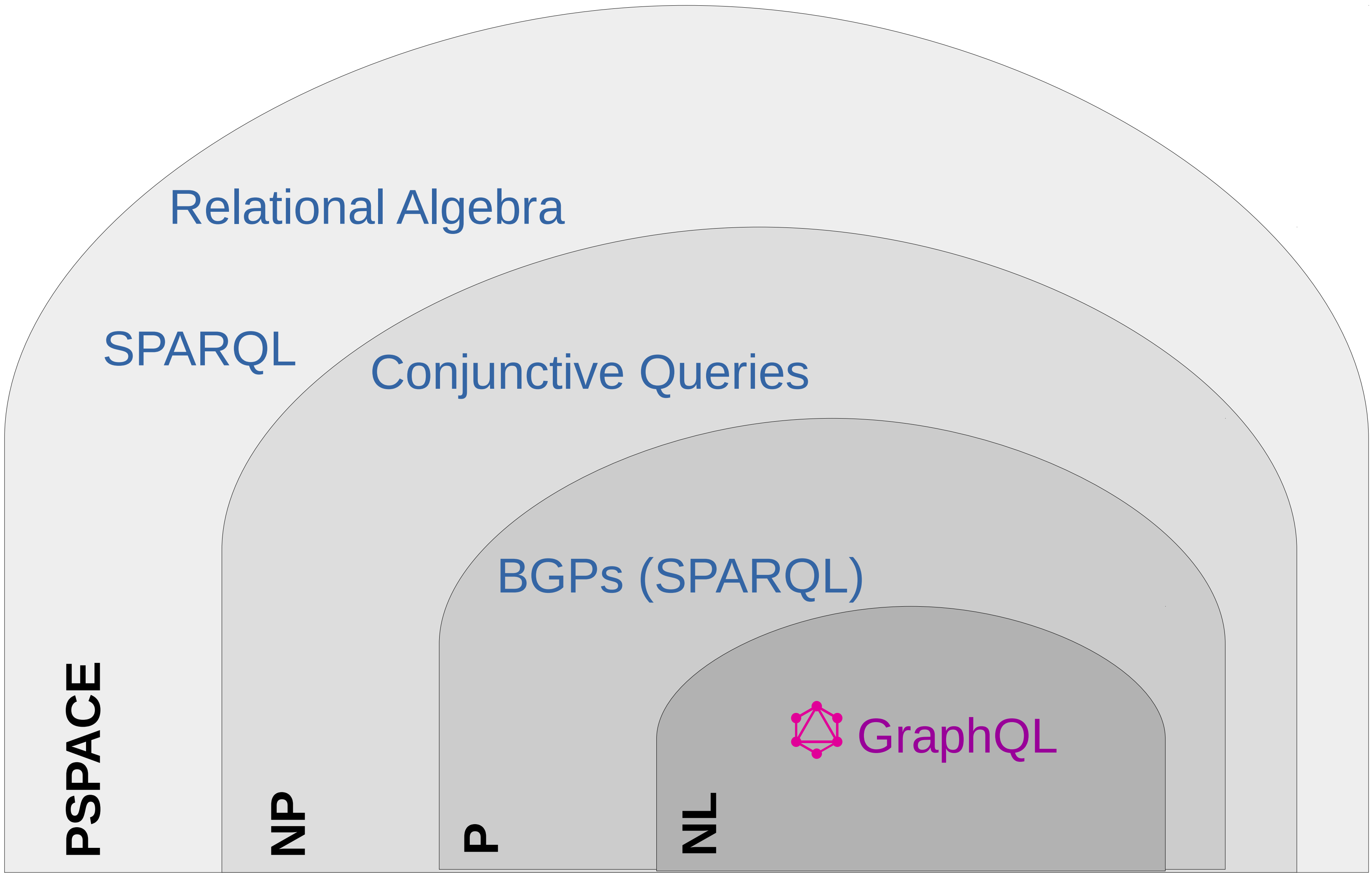
Complexity Classes



Complexity of Evaluation Problems



Complexity of Evaluation Problems



Complexity Analysis

Enumeration Problem

Non-Redundancy

Valid query

```
hero(episode: EMPIRE) {
    name
    friends {
        name
    }
    id
    name
    friends {
        id
    }
}
```

Invalid result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2}
        { name: Han}
    ]
    id: 1000
    name: Luke
    friends:  [
        { id: 2001}
        { id: 1002}
    ]
}
```

Non-Redundancy

Valid query

```
hero(episode: EMPIRE) {
    name
    friends {
        name
    }
    id
    name
    friends {
        id
    }
}
```

Correct result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2
          Id: 2001 }
        { name: Han
          Id: 1002 }
    ]
    id: 1000
}
```

Fields are *collected* before answering

Non-Redundancy

Non-redundant query

```
hero(episode: EMPIRE) {
    name
    friends {
        name
        id
    }
    id
}
```

Correct result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2
          Id: 2001 }
        { name: Han
          Id: 1002 }
    ]
    id: 1000
}
```

Fields are *collected* before answering

Another Complication: Type Restrictions

Valid query

```
hero(episode: EMPIRE) {
    name
    friends {
        on Droid { name }
        on Human { id }
        name
    }
}
```

Invalid result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2
          name: R2-D2 }
        { id: 1002
          name: Han }
    ]
}
```

Another Complication: Type Restrictions

Valid query

```
hero(episode: EMPIRE) {
    name
    friends {
        on Droid { name }
        on Human { id }
        name
    }
}
```

Correct result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2 }
        { id: 1002
          name: Han }
    ]
}
```

Another Complication: Type Restrictions

Valid query

```
hero(episode: EMPIRE) {
    name
    friends {
        on Droid { name }
        on Human { id name }
    }
}
```

Correct result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2 }
        { id: 1002
          name: Han }
    ]
}
```

Ground-Typed Normal Form

Ground-typed query

```
hero(episode: EMPIRE) {
    name
    friends {
        on Droid { name }
        on Human { id name }
    }
}
```

Correct result

```
hero {
    name: Luke
    friends:  [
        { name: R2-D2 }
        { id: 1002
          name: Han }
    ]
}
```

Eliminating Redundancies

Rewriting rules for queries

Every GraphQL **query** *q* **can be rewritten** into a query *q'* that is i) non-redundant and ii) in ground-typed normal form, such that

$$q \equiv q'$$

Advantage: **field collection is not needed** for non-redundant queries in ground-typed NF

Now to the Enumeration Problem

Let *q* be i) non-redundant and
ii) in ground-typed normal form

Result of *q* can be produced symbol by symbol with **only constant time between symbols**

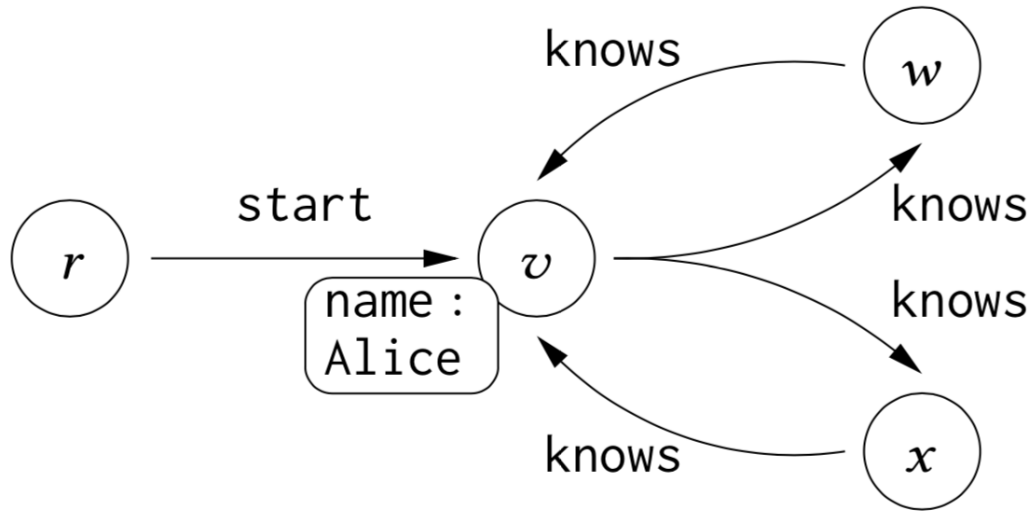
hero { friends: [{ name:R2-D2}] }

Time to produce the complete query result **depends linearly** on the size of this result

Complexity Analysis

Result Size

Results of GraphQL queries can be huge

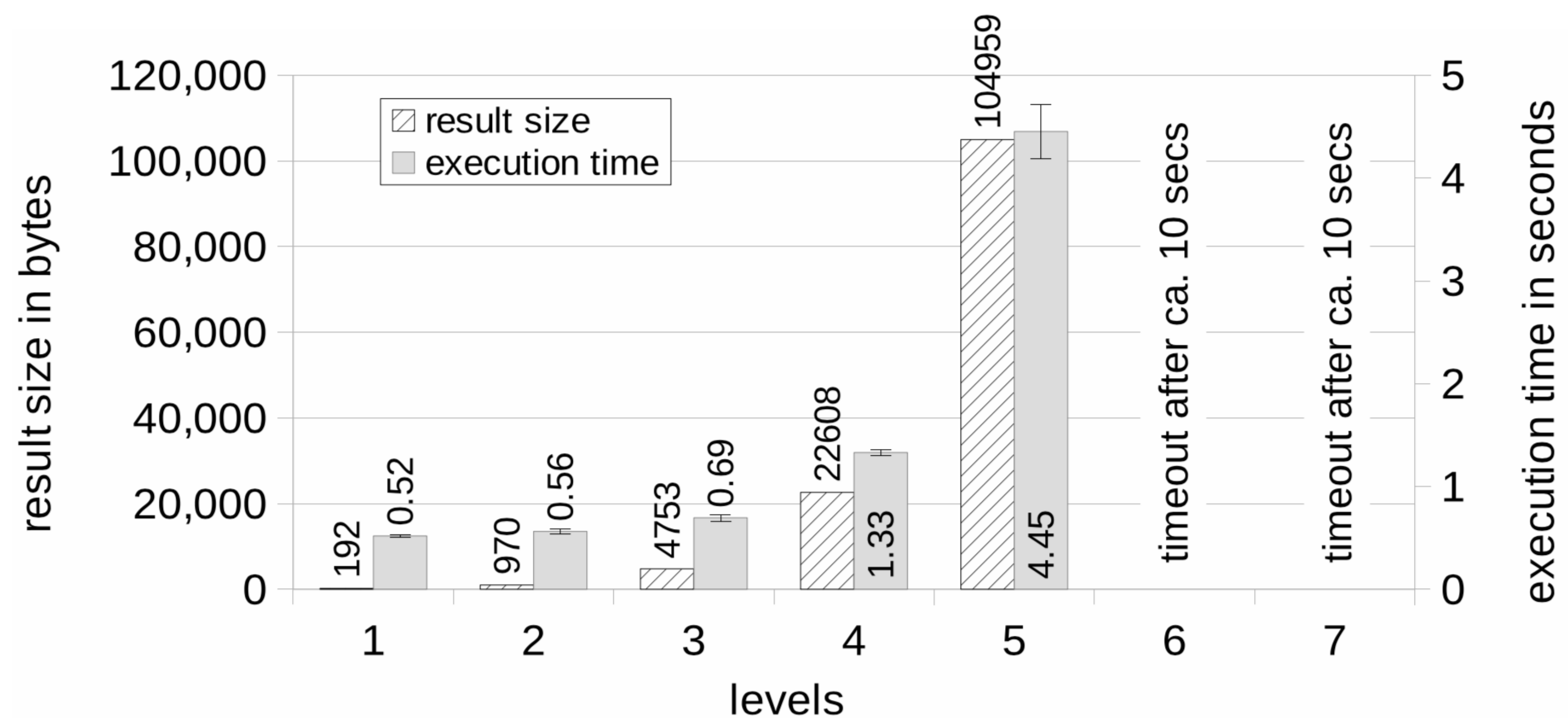


$$\text{start} \{ \underbrace{\text{knows} \{ \text{knows} \{ \ldots \{ \text{knows}}_{2N \text{ times}} \{ \text{name} \} \} \ldots \} \} \}$$

Alice appears 2^N times in the result

Huge results in practice: Github's GraphQL API

Owners of *first five* repos that user "danbri" contributes to, and the owners of *first five* repos that they contribute to, and so on...



Result sizes can be computed efficiently!!!

Let q be i) non-redundant and ii) in ground-typed normal form

Time to compute the size of the result of q over a graph G **depends linearly** on the product

$$(\text{size of } q) \times (\text{size of } G)$$

We provide an **algorithm** that achieves this complexity bound

Proposal for GraphQL Servers

First, compute the size of the result.

If too big, reject query.

Else, inform the size to the client, and

Send the result byte by byte.

(or use the size as basis of a billing model)

Summary

Our Results in a Nutshell

Formal definition of the language

- Property Graph-like data model
- Formal query semantics

Study of **computational complexity** (the language admits really efficient evaluation methods)

- Evaluation problem is NL-complete
- Enumeration of results is linear

Solution to the problem of **large results**

- Efficient algorithm to compute result size

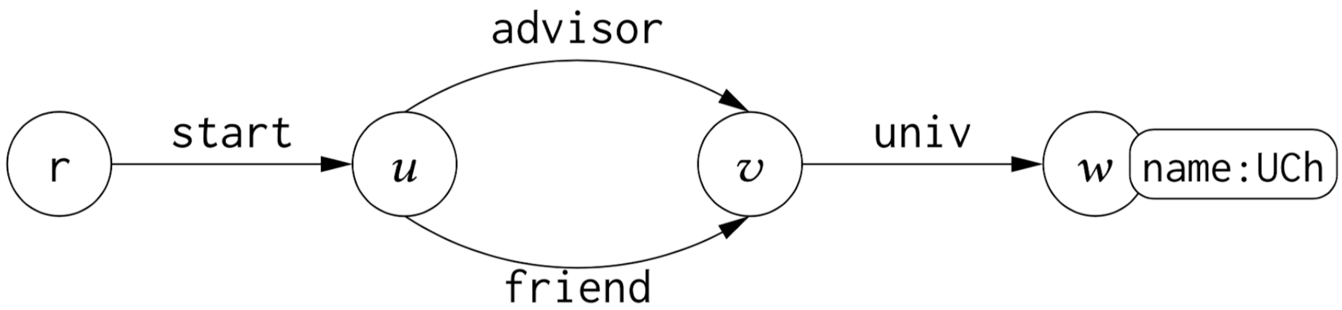
Olaf Hartig and Jorge Pérez: *Semantics and Complexity of GraphQL.* In The Web Conference 2018.

www.liu.se

LINKÖPING UNIVERSITY

Backup Slides

Result-Size Computation



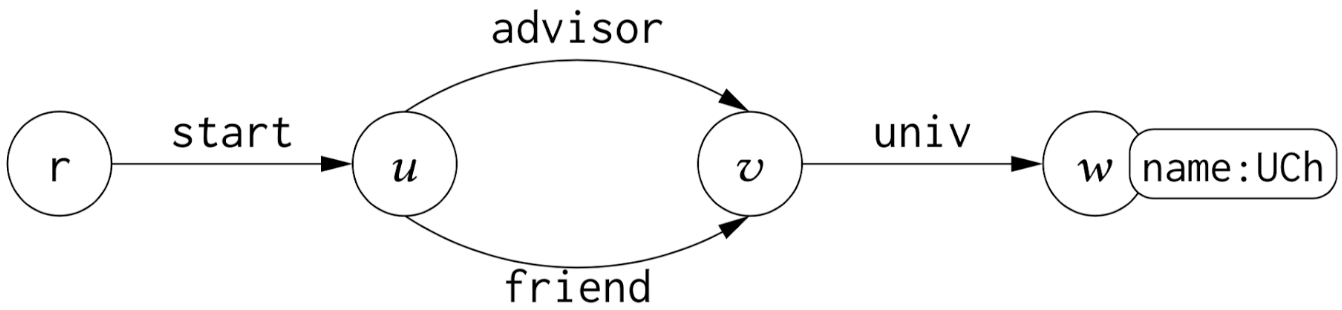
$$\text{start \{ } \underbrace{\text{advisor \{ univ \{ name \} \}}}_{q_2} \; \underbrace{\text{friend \{ univ \{ name \} \}}}_{q_3} \text{ \}}$$

Result:

start: { … … }

$$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$$

Result-Size Computation

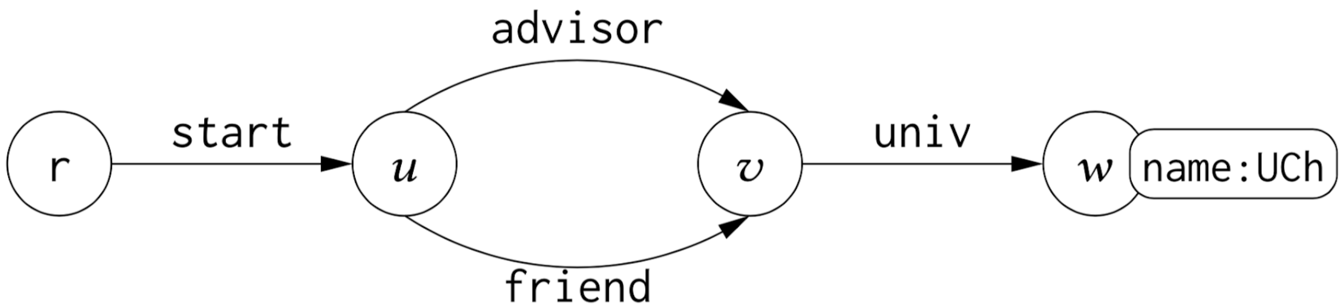


$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_2}\ \underbrace{\text{friend}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_3}\ \}$$

$\text{size}(q_2, u) =$ $\text{size}(q_3, u) =$

$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$

Result-Size Computation

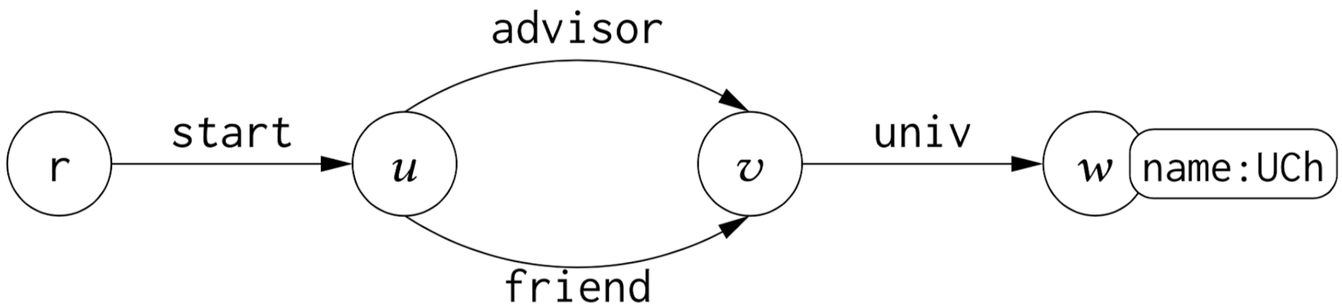


$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \text{name}\ \}}^{q_4}\ \}}_{q_2}\ \ \underbrace{\text{friend}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_3}\ \}$$

$\text{size}(q_2, u) = 4 + \text{size}(q_4, v)$ $\text{size}(q_3, u) =$

$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$

Result-Size Computation



$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \underbrace{\text{name}\ \}}_{q_5}}^{q_4}\ \}}_{q_2}\ \underbrace{\text{friend}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_3}\ \}$$

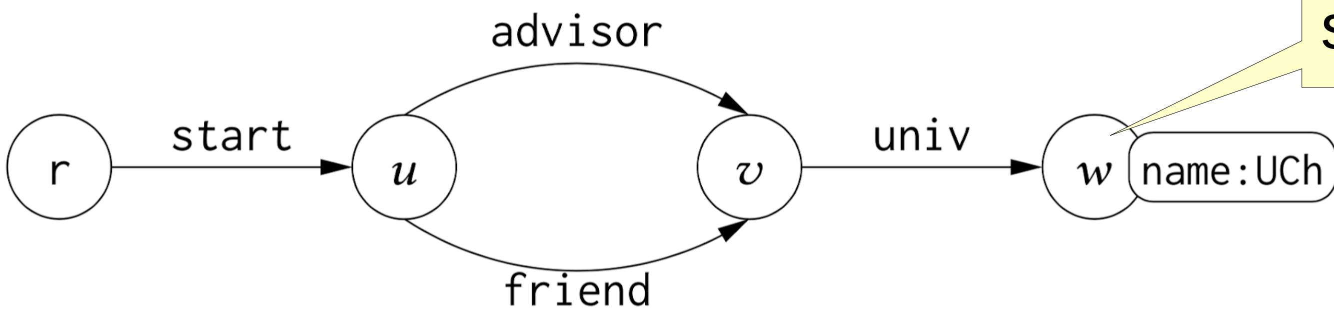
$$\text{size}(q_4, v) = 4 + \text{size}(q_5, w)$$

$$\text{size}(q_2, u) = 4 + \text{size}(q_4, v) \qquad \text{size}(q_3, u) =$$

$$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$$

Result-Size Computation

size(q_5) = 3



$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \underbrace{\text{name}}_{q_5}\ \}}^{q_4}\ \}}_{q_2}\ \ \underbrace{\text{friend}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_3}\ \}$$

$\text{size}(q_5, w) = 3$

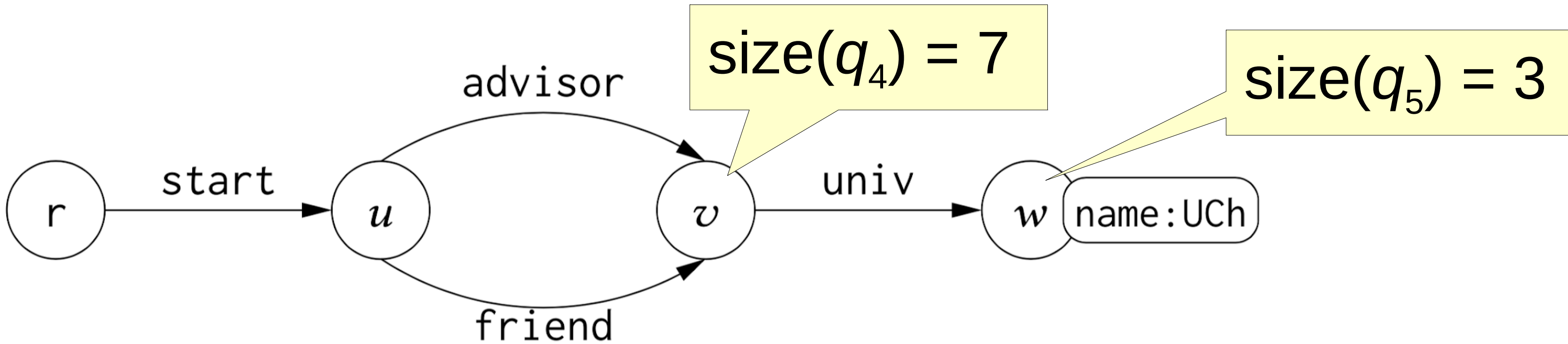
$\text{size}(q_4, v) = 4 + \text{size}(q_5, w)$

$\text{size}(q_2, u) = 4 + \text{size}(q_4, v)$

$\text{size}(q_3, u) =$

$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$

Result-Size Computation



$$\text{start} \; \{ \; \underbrace{\text{advisor} \; \{ \; \overbrace{\text{univ} \; \{ \; \underbrace{\text{name} \; \}}_{q_5}}^{q_4} \; \}}_{q_2} \;\; \underbrace{\text{friend} \; \{ \; \text{univ} \; \{ \; \text{name} \; \} \; \}}_{q_3} \;\; \}$$

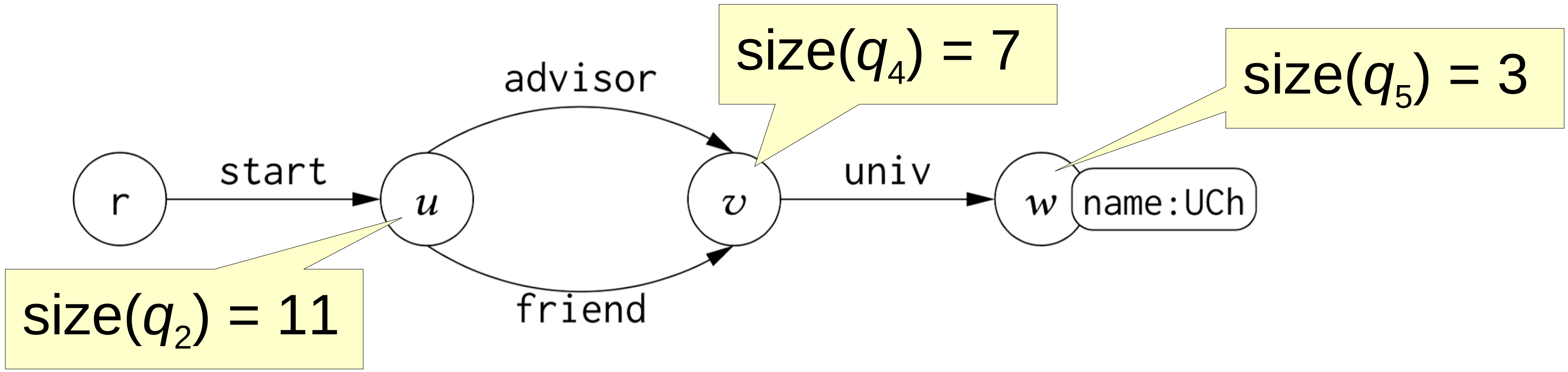
$$\text{size}(q_5, w) = 3$$

$$\text{size}(q_4, v) = 4 + \text{size}(q_5, w) = 4 + 3 = 7$$

$$\text{size}(q_2, u) = 4 + \text{size}(q_4, v) \qquad \text{size}(q_3, u) =$$

$$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$$

Result-Size Computation



$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \underbrace{\text{name}\ \}}_{q_5}}^{q_4}\ \}}_{q_2}\ \ \underbrace{\text{friend}\ \{\ \text{univ}\ \{\ \text{name}\ \}\ \}}_{q_3}\ \ \}$$

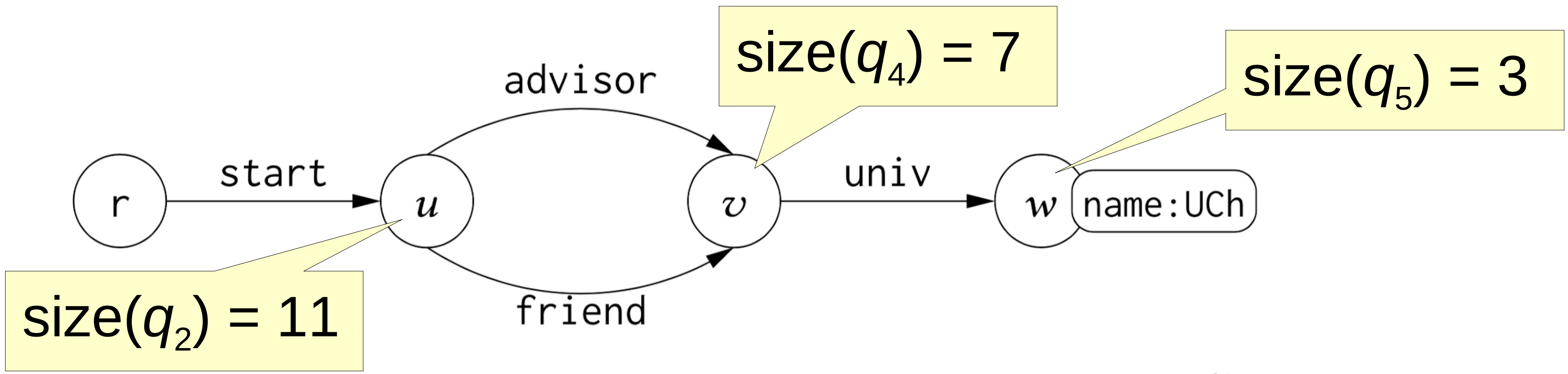
$\text{size}(q_5, w) = 3$

$\text{size}(q_4, v) = 4 + \text{size}(q_5, w) = 4 + 3 = 7$

$\text{size}(q_2, u) = 4 + \text{size}(q_4, v) = 11$ $\quad \text{size}(q_3, u) =$

$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$

Result-Size Computation



$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \underbrace{\text{name}\ \}}_{q_5}}^{q_4}\ \}\ \}}_{q_2}\ \ \underbrace{\text{friend}\ \{\ \overbrace{\text{univ}\ \{\ \text{name}\ \}}^{q_4}\ \}}_{q_3}\ \ \}$$

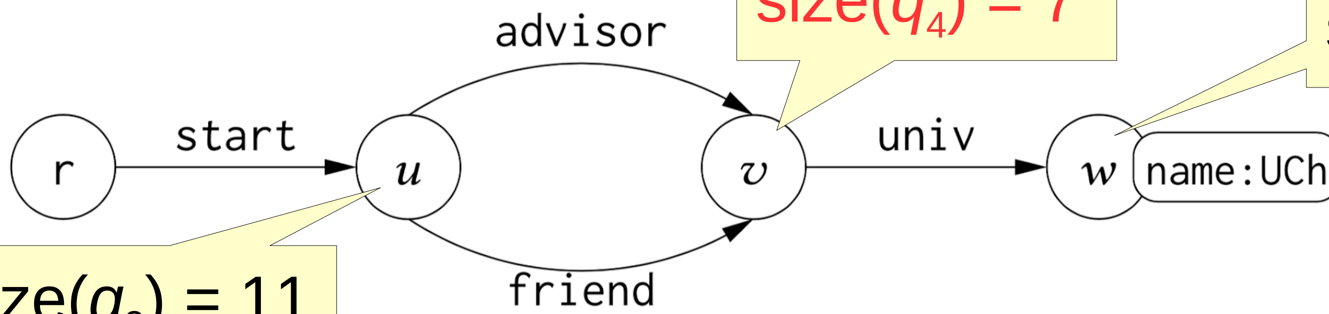
$\text{size}(q_5, w) = 3$

$\text{size}(q_4, v) = 4 + \text{size}(q_5, w) = 4 + 3 = 7$

$\text{size}(q_2, u) = 4 + \text{size}(q_4, v) = 11$ $\text{size}(q_3, u) = 4 + \text{size}(q_4, v)$

$\text{size}(q, r) = 4 + \text{size}(q_2, u) + \text{size}(q_3, u)$

Result-Size Computation



size(q_4) = 7

size(q_5) = 3

size(q_2) = 11
size(q_3) = 11

q_4 ⟷ q_4

start { $\underbrace{\text{advisor \{ } \overbrace{\text{univ \{ } \underbrace{\text{name}}_{q_5} \text{ \}}}^{q_4} \text{ \}}}_{q_2}$ $\underbrace{\text{friend \{ } \overbrace{\text{univ \{ name \}}}^{q_4} \text{ \}}}_{q_3}$ }

size(q_5, w) = 3

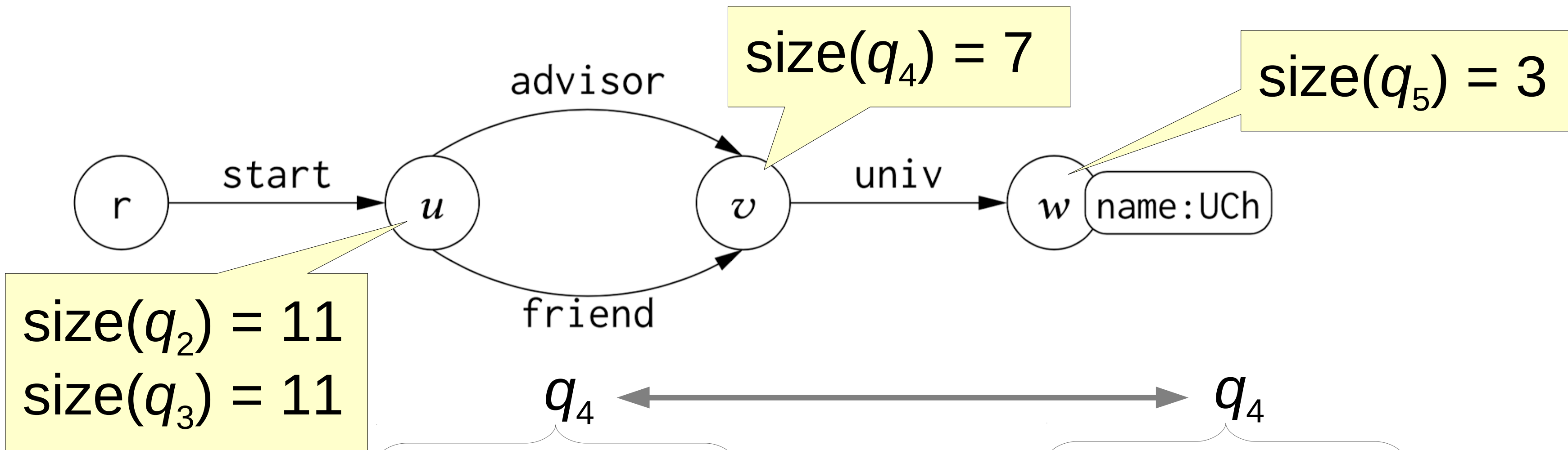
size(q_4, v) = 4 + size(q_5, w) = 4 + 3 = 7

size(q_2, u) = 4 + size(q_4, v) = 11

size(q_3, u) = 4 + size(q_4, v)
= 4 + 7 = 11

size(q, r) = 4 + size(q_2, u) + size(q_3, u)

Result-Size Computation



$$\text{start}\ \{\ \underbrace{\text{advisor}\ \{\ \overbrace{\text{univ}\ \{\ \underbrace{\text{name}\ \}}_{q_5}}^{q_4}\ \}\ \}}_{q_2}\ \ \underbrace{\text{friend}\ \{\ \overbrace{\text{univ}\ \{\ \text{name}\ \}}^{q_4}\ \}}_{q_3}\ \ \}$$

$$\text{size}(q_5, w) = 3$$

$$\text{size}(q_4, v) = 4 + \text{size}(q_5, w) = 4 + 3 = 7$$

$$\text{size}(q_2, u) = 4 + \text{size}(q_4, v) = 11 \qquad \text{size}(q_3, u) = 4 + \text{size}(q_4, v) = 4 + 7 = 11$$

$$\text{size}(q, r) = 4 + \underset{11}{\cancel{\text{size}(q_2, u)}} + \underset{11}{\cancel{\text{size}(q_3, u)}} = 26$$