Combining uniqueness and linearity in one type system

Tanja Muller

January 20, 2025

Papers

- Uniqueness Logic
  Dana Harrington
  TCP 2006

Papers

- Uniqueness Logic
  Dana Harrington
  TCP 2006
  - Formalizes uniqueness in a logic, type system and category

Papers

- Uniqueness Logic
  Dana Harrington
  TCP 2006
  - Formalizes uniqueness in a logic, type system and category
- Linearity and Uniqueness: An Entente Cordiale
  Danielle Marshall, Michael Vollmer, Dominic Orchard
  European Symposium on Programming 2022

Papers

- Uniqueness Logic
  Dana Harrington
  TCP 2006
  - Formalizes uniqueness in a logic, type system and category
- Linearity and Uniqueness: An Entente Cordiale
  Danielle Marshall, Michael Vollmer, Dominic Orchard
  European Symposium on Programming 2022
  - Combines uniqueness and linearity in a single type system/programming language

Content

- Uniqueness and Linearity
  - What are these concepts + their use?
  - How do they relate?

Content

- Uniqueness and Linearity
  - What are these concepts + their use?
  - How do they relate?
- Logic
  - Compare Harrington's uniqueness logic with linear logic

Content

- Uniqueness and Linearity
  - What are these concepts + their use?
  - How do they relate?
- Logic
  - Compare Harrington's uniqueness logic with linear logic
- Typing rules
  - Uniqueness and linearity combined in typing rules from Marshall et al.

Content

- Uniqueness and Linearity
  - What are these concepts + their use?
  - How do they relate?
- Logic
  - Compare Harrington's uniqueness logic with linear logic
- Typing rules
  - Uniqueness and linearity combined in typing rules from Marshall et al.
- As programming language
  - Implement these typing rules in programming language
  - Performance

Linearity

- A linear variable must be **used exactly once**

Linearity

- A linear variable must be **used exactly once**
- (Affine: used at most once)

Linearity

- A linear variable must be **used exactly once**
- (Affine: used at most once)
- Example: a linear variable cake

Linearity

- A linear variable must be **used exactly once**
- (Affine: used at most once)
- Example: a linear variable cake
- "You can't have your cake and eat it too"

Linearity

- A linear variable must be **used exactly once**
- (Affine: used at most once)
- Example: a linear variable cake
- "You can't have your cake and eat it too"
- In Granule, where types are by default linear:

```
impossible : Cake -> (Happy, Cake)
impossible cake = (eat cake, have cake)
```

Use of linearity

- For example useful for file handling

```
twoChars : (Char, Char) <IO>
twoChars = let
h <- openHandle ReadMode "someFile";
(h, c1) <- readChar h;
(h, c2) <- readChar h;
() <- closeHandle h
in pure (c1, c2)
```

Use of linearity

- For example useful for file handling

```
twoChars : (Char, Char) <IO>
twoChars = let
h <- openHandle ReadMode "someFile";
(h, c1) <- readChar h;
(h, c2) <- readChar h;
() <- closeHandle h
in pure (c1, c2)
```

- Must always have only 1 file handle for a certain file

Use of linearity

- For example useful for file handling

```
twoChars : (Char, Char) <IO>
twoChars = let
h <- openHandle ReadMode "someFile";
(h, c1) <- readChar h;
(h, c2) <- readChar h;
() <- closeHandle h
in pure (c1, c2)
```

- Must always have only 1 file handle for a certain file
- Must always close the file when done with it

Use of linearity

- For example useful for file handling

```
twoChars : (Char, Char) <IO>
twoChars = let
h <- openHandle ReadMode "someFile";
(h, c1) <- readChar h;
(h, c2) <- readChar h;
() <- closeHandle h
in pure (c1, c2)
```

- Must always have only 1 file handle for a certain file
- Must always close the file when done with it
- Each time, new file handle created after use, and each file handle is **used exactly once**

Uniqueness

- A unique variable must have **at most one reference** to it

Uniqueness

- A unique variable must have **at most one reference** to it
- Different from affine types, which must be *used* at most once

Uniqueness

- A unique variable must have **at most one reference** to it
- Different from affine types, which must be *used* at most once
- Subtle difference:
  - Affine variables may not be copied
  - There may not exist copies of unique variables

Example uniqueness

- Example: unique variable for a cinema ticket

Example uniqueness

- Example: unique variable for a cinema ticket
- A unique ticket can be used at the cinema

Example uniqueness

- Example: unique variable for a cinema ticket
- A unique ticket can be used at the cinema
- It's fine if you don't use the ticket though

Example uniqueness

- Example: unique variable for a cinema ticket
- A unique ticket can be used at the cinema

- It's fine if you don't use the ticket though
- You can't sell a copy of your ticket to someone and still expect to be able to use it yourself at the cinema (it's only valid once)

Example uniqueness

- Example: unique variable for a cinema ticket
- A unique ticket can be used at the cinema

- It's fine if you don't use the ticket though
- You can't sell a copy of your ticket to someone and still expect to be able to use it yourself at the cinema (it's only valid once)
- If all types are by default unique:

```
impossible : Ticket -> (Cash, Ticket)
impossible ticket = (sell ticket, ticket)
```

Use of uniqueness

- Consider for example mutable arrays

Use of uniqueness

- Consider for example mutable arrays
- With lazy evaluation the order of actions is not always clear

Use of uniqueness

- Consider for example mutable arrays
- With lazy evaluation the order of actions is not always clear
- Example:

```
a = [1,1,3,4]
f(a, writeArray(a,1,2))
```

Use of uniqueness

- Consider for example mutable arrays
- With lazy evaluation the order of actions is not always clear
- Example:

```
a = [1,1,3,4]
f(a, writeArray(a,1,2))
```

- We want to prevent this

Use of uniqueness

- To fix this, you get a new array reference from read/write operations:

```
a0 = [15,25,30]
a1 = writeArray(a0, 35, 2)
```

Use of uniqueness

- To fix this, you get a new array reference from read/write operations:

```
a0 = [15,25,30]
a1 = writeArray(a0, 35, 2)
```

- If you know your reference is unique, you can directly update the array instead of having to copy the whole array

Use of uniqueness

- To fix this, you get a new array reference from read/write operations:

```
a0 = [15,25,30]
a1 = writeArray(a0, 35, 2)
```

- If you know your reference is unique, you can directly update the array instead of having to copy the whole array
- Therefore, only allow **at most one reference**

Relation between linearity and uniqueness

- Uniqueness: constraint about past
  - Guarantees that the reference has not yet been duplicated

Relation between linearity and uniqueness

- Uniqueness: constraint about past
  - Guarantees that the reference has not yet been duplicated
- Linearity: constraint about future
  - Guarantees that it will not be duplicated nor discarded from here on

Relation between linearity and uniqueness

- Uniqueness: constraint about past
  - Guarantees that the reference has not yet been duplicated
- Linearity: constraint about future
  - Guarantees that it will not be duplicated nor discarded from here on
- Consider the earlier example with mutable arrays

Relation between linearity and uniqueness

- Uniqueness: constraint about past
  - Guarantees that the reference has not yet been duplicated
- Linearity: constraint about future
  - Guarantees that it will not be duplicated nor discarded from here on
- Consider the earlier example with mutable arrays
  - Linearity is too restrictive: discarding arrays is no problem

Relation between linearity and uniqueness

- Uniqueness: constraint about past
  - Guarantees that the reference has not yet been duplicated
- Linearity: constraint about future
  - Guarantees that it will not be duplicated nor discarded from here on
- Consider the earlier example with mutable arrays
  - Linearity is too restrictive: discarding arrays is no problem
  - Linear (or affine) types are not strong enough: could have previously been a non-linear variable that was duplicated multiple times before being specialized to linear

Shared / non-linear variables

- Difference in how they relate to unrestricted variables

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear
  - Linear **cannot** be turned into non-linear, because then we can't guarantee they'll be used exactly once

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear
  - Linear **cannot** be turned into non-linear, because then we can't guarantee they'll be used exactly once
  - Non-linear **can** be turned into linear, since it doesn't matter for linearity what happened to it before

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear
  - Linear **cannot** be turned into non-linear, because then we can't guarantee they'll be used exactly once
  - Non-linear **can** be turned into linear, since it doesn't matter for linearity what happened to it before
- Relation unique vs shared (=non-unique)

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear
  - Linear **cannot** be turned into non-linear, because then we can't guarantee they'll be used exactly once
  - Non-linear **can** be turned into linear, since it doesn't matter for linearity what happened to it before
- Relation unique vs shared (=non-unique)
  - Unique **can** be turned into shared by "forgetting" the constraint about their past

Shared / non-linear variables

- Difference in how they relate to unrestricted variables
- Relation linear vs non-linear
  - Linear **cannot** be turned into non-linear, because then we can't guarantee they'll be used exactly once
  - Non-linear **can** be turned into linear, since it doesn't matter for linearity what happened to it before
- Relation unique vs shared (=non-unique)
  - Unique **can** be turned into shared by "forgetting" the constraint about their past
  - Shared **cannot** be turned into unique, since they might have been duplicated already

Paper: Uniqueness Logic

- *Uniqueness Logic* by Dana Harrington

Paper: Uniqueness Logic

- *Uniqueness Logic* by Dana Harrington
- Goal: formalize/interpret uniqueness in several ways
  - Logic
  - Typing rules
  - Category

Paper: Uniqueness Logic

- *Uniqueness Logic* by Dana Harrington
- Goal: formalize/interpret uniqueness in several ways
  - Logic
  - Typing rules
  - Category
- We will discuss some of the rules of this logic and compare them with linear logic

Comparison logic - introduction modalities

- Non-linear modality ! and shared modality $\circ$

Comparison logic - introduction modalities

- Non-linear modality ! and shared modality $\circ$
- Left/right introduction rules for both

Comparison logic - introduction modalities

- Non-linear modality ! and shared modality $\circ$
- Left/right introduction rules for both

$$\frac{\Gamma, P \vdash Q}{\Gamma, !P \vdash Q}\ !_L$$

$$\frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R$$

Linear Logic

Comparison logic - introduction modalities

- Non-linear modality ! and shared modality $\circ$
- Left/right introduction rules for both

$$\frac{\Gamma, P \vdash Q}{\Gamma, !P \vdash Q}\ !_L \qquad \frac{\Gamma, P \vdash Q^\circ}{\Gamma, P^\circ \vdash Q^\circ}\ \circ_L$$

$$\frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^\circ}\ \circ_R$$

Linear Logic Uniqueness Logic

Comparison logic - introduction modalities

- Non-linear modality ! and shared modality $\circ$
- Left/right introduction rules for both

$$\frac{\Gamma, P \vdash Q}{\Gamma, !P \vdash Q}\ !_L \qquad \frac{\Gamma, P \vdash Q^\circ}{\Gamma, P^\circ \vdash Q^\circ}\ \circ_L$$

$$\frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^\circ}\ \circ_R$$

Linear Logic | Uniqueness Logic

- introduction of ! on the left is unrestricted, while introduction of $\circ$ on the right is unrestricted

Comparison logic - contraction & weakening

- Thus, uniqueness and linearity behave dually w.r.t. their relation with unrestricted values

Comparison logic - contraction & weakening

- Thus, uniqueness and linearity behave dually w.r.t. their relation with unrestricted values
- However, for contraction and weakening, both modalities act exactly the same way

Comparison logic - contraction & weakening

- Thus, uniqueness and linearity behave dually w.r.t. their relation with unrestricted values
- However, for contraction and weakening, both modalities act exactly the same way

$$\frac{\Gamma, !P, !P \vdash R}{\Gamma, !P \vdash R}\ !_{\text{contraction}}$$

$$\frac{\Gamma \vdash R}{\Gamma, !P \vdash R}\ !_{\text{weakening}}$$

Linear Logic

Comparison logic - contraction & weakening

- Thus, uniqueness and linearity behave dually w.r.t. their relation with unrestricted values
- However, for contraction and weakening, both modalities act exactly the same way

$$\frac{\Gamma, !P, !P \vdash R}{\Gamma, !P \vdash R}\ !_{\text{contraction}} \qquad \frac{\Gamma, P^\circ, P^\circ \vdash R}{\Gamma, P^\circ \vdash R}\ {}^{\circ}{}_{\text{contraction}}$$

$$\frac{\Gamma \vdash R}{\Gamma, !P \vdash R}\ !_{\text{weakening}} \qquad \frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R}\ {}^{\circ}{}_{\text{weakening}}$$

Linear Logic   Uniqueness Logic

Comparison logic - contraction & weakening

- Thus, uniqueness and linearity behave dually w.r.t. their relation with unrestricted values
- However, for contraction and weakening, both modalities act exactly the same way

$$\frac{\Gamma, !P, !P \vdash R}{\Gamma, !P \vdash R}\ !_{\text{contraction}} \qquad \frac{\Gamma, P^{\circ}, P^{\circ} \vdash R}{\Gamma, P^{\circ} \vdash R}\ {}^{\circ}_{\text{contraction}}$$

$$\frac{\Gamma \vdash R}{\Gamma, !P \vdash R}\ !_{\text{weakening}} \qquad \frac{\Gamma \vdash R}{\Gamma, P^{\circ} \vdash R}\ {}^{\circ}_{\text{weakening}}$$

Linear Logic — Uniqueness Logic

- So linearity and uniqueness behave identically w.r.t. structural rules

Typing rules

- Harrington also made typing rules based on this

Typing rules

- Harrington also made typing rules based on this
- Closely related to the logic, through the Curry-Howard isomorphism
  - Types correspond to formulas in the logic

Typing rules

- Harrington also made typing rules based on this
- Closely related to the logic, through the Curry-Howard isomorphism
  - Types correspond to formulas in the logic
- Example (weakening rule):

$$\frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R} \quad \rightarrow \quad \frac{\overline{u} : \Gamma \vdash t : R}{\overline{u} : \Gamma, x : P^\circ \vdash t : R}$$

Typing rules

- Harrington also made typing rules based on this
- Closely related to the logic, through the Curry-Howard isomorphism
  - Types correspond to formulas in the logic
- Example (weakening rule):

$$\frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R} \quad \rightarrow \quad \frac{\overline{u} : \Gamma \vdash t : R}{\overline{u} : \Gamma, x : P^\circ \vdash t : R}$$

- Later, these typing rules are used by other researchers

Typing rules

- Harrington also made typing rules based on this
- Closely related to the logic, through the Curry-Howard isomorphism
  - Types correspond to formulas in the logic
- Example (weakening rule):

$$\frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R} \qquad \rightarrow \qquad \frac{\overline{u} : \Gamma \vdash t : R}{\overline{u} : \Gamma, x : P^\circ \vdash t : R}$$

- Later, these typing rules are used by other researchers
  - *Uniqueness Typing for Resource Management in Message-Passing Concurrency* (Hennessey et al.)

Typing rules

- Harrington also made typing rules based on this
- Closely related to the logic, through the Curry-Howard isomorphism
  - Types correspond to formulas in the logic
- Example (weakening rule):

$$\frac{\Gamma \vdash R}{\Gamma, P^\circ \vdash R} \quad \rightarrow \quad \frac{\overline{u} : \Gamma \vdash t : R}{\overline{u} : \Gamma, x : P^\circ \vdash t : R}$$

- Later, these typing rules are used by other researchers
  - *Uniqueness Typing for Resource Management in Message-Passing Concurrency* (Hennessey et al.)
  - *Linearity and Uniqueness: An Entente Cordiale* (Marshall et al.)

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard
- Goal: combine uniqueness and linearity in one system / programming language

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard
- Goal: combine uniqueness and linearity in one system / programming language
- Because it's faster, better performance

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard
- Goal: combine uniqueness and linearity in one system / programming language
- Because it's faster, better performance
  - With the guarantees from linearity/uniqueness, you don't have to account for when a value is non-linear/shared

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard
- Goal: combine uniqueness and linearity in one system / programming language
- Because it's faster, better performance
  - With the guarantees from linearity/uniqueness, you don't have to account for when a value is non-linear/shared
  - You don't have to copy an array to edit it if you know it's unique, then you can edit in-place

Paper: Linearity and Uniqueness

- *Linearity and Uniqueness: An Entente Cordiale*
  by Danielle Marshall, Michael Vollmer and Dominic Orchard
- Goal: combine uniqueness and linearity in one system / programming language
- Because it's faster, better performance
  - With the guarantees from linearity/uniqueness, you don't have to account for when a value is non-linear/shared
  - You don't have to copy an array to edit it if you know it's unique, then you can edit in-place
- They create a type system containing both unique and linear types (partly based on typing rules from Harrington)

Base system

- They use lazy evaluation

Base system

- They use lazy evaluation
- The base system is linearly typed

Base system

- They use lazy evaluation
- The base system is linearly typed
  - Already the default in the used programming language, Granule

Base system

- They use lazy evaluation
- The base system is linearly typed
  - Already the default in the used programming language, Granule
  - Avoids problems; for example if a product (t_1, t_2) of two linear values would be unique by default, then it can be converted to an unrestricted value and can then be duplicated

Modalities

- They have a uniqueness modality $*$

Modalities

- They have a uniqueness modality $*$
- They treat non-linearity and non-uniqueness as the same state: both are unrestricted

Modalities

- They have a uniqueness modality $*$
- They treat non-linearity and non-uniqueness as the same state: both are unrestricted
- Therefore, they have a single unrestricted modality !

Modalities

- They have a uniqueness modality $*$
- They treat non-linearity and non-uniqueness as the same state: both are unrestricted
- Therefore, they have a single unrestricted modality $!$
- Progression: unique $\rightarrow$ unrestricted $\rightarrow$ linear

Syntax

- Terms, usually denoted by t. For example:

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\, t_2$

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\, t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\,t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\ t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)
  - `unit`, which can be seen as the empty product

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\ t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)
  - `unit`, which can be seen as the empty product
  - some constructs like: copy t_1 as x in t_2, let $!x = t_1$ in t_2

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\ t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)
  - `unit`, which can be seen as the empty product
  - some constructs like: copy t_1 as x in t_2, let $!x = t_1$ in t_2
- Types

$$A, B ::= A \multimap B \mid A \otimes B \mid 1 \mid !A \mid *A$$

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\ t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)
  - `unit`, which can be seen as the empty product
  - some constructs like: copy t_1 as x in t_2, let $!x = t_1$ in t_2
- Types

$$A, B ::= A \multimap B \mid A \otimes B \mid 1 \mid !A \mid *A$$

- Typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]$$

Syntax

- Terms, usually denoted by t. For example:
  - lambda terms: x, $\lambda x.t$, $t_1\ t_2$
  - unrestricted terms $!t$, unique terms $*t$, borrowed terms $\&t$
  - products (t_1, t_2)
  - `unit`, which can be seen as the empty product
  - some constructs like: copy t_1 as x in t_2, let $!x = t_1$ in t_2
- Types

$$A, B ::= A \multimap B \mid A \otimes B \mid 1 \mid !A \mid *A$$

- Typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]$$

- Typing judgements $\Gamma \vdash t : A$

Typing contexts

- Non-linear assignments in a typing context are denoted by $x : [A]$

Typing contexts

- Non-linear assignments in a typing context are denoted by $x : [A]$
- These variables can be used multiple times

Typing contexts

- Non-linear assignments in a typing context are denoted by $x : [A]$
- These variables can be used multiple times
- Note: different from $x : !A$, which we'll see more clearly in the typing rules

Typing contexts

- Non-linear assignments in a typing context are denoted by $x : [A]$
- These variables can be used multiple times
- Note: different from $x : !A$, which we'll see more clearly in the typing rules
- We write $[\Gamma]$ to mark a context as containing only non-linear assignments (which includes the empty context)

Typing contexts

- Non-linear assignments in a typing context are denoted by $x : [A]$
- These variables can be used multiple times
- Note: different from $x : !A$, which we'll see more clearly in the typing rules
- We write $[\Gamma]$ to mark a context as containing only non-linear assignments (which includes the empty context)

Definition 1

Context addition $\Gamma_1 + \Gamma_2$ is the union of two contexts as long as every variable x that occurs in both contexts is assigned the same non-linear type in both contexts ($x : [A]$).

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \mathrm{VAR}$$

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \text{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \text{ABS}$$

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \mathrm{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \mathrm{ABS}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1 t_2 : B}\ \mathrm{APP}$$

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \mathrm{VAR} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \mathrm{ABS}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1 t_2 : B}\ \mathrm{APP}$$

- For variables, abstraction and application

Typing rules: λ-calculus

- The usual λ-calculus is typed by these three typing rules

$$\frac{}{[\Gamma], x : A \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \textsc{abs}$$

$$\frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1 t_2 : B}\ \textsc{app}$$

- For variables, abstraction and application
- Note that for VAR, the rest of the typing context has to be non-linear while the variable itself is linear

Typing rules: non-linear modality

- Introduction rule / promotion

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A} \, !_I$$

Typing rules: non-linear modality

- Introduction rule / promotion

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I \qquad\qquad \frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R$$

Typing rules: non-linear modality

- Introduction rule / promotion

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I \qquad\qquad \frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R$$

- Elimination rule

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \text{let } !x = t_1 \text{ in } t_2 : B}\ !_E$$

Typing rules: non-linear modality

- Introduction rule / promotion

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I \qquad\qquad \frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R$$

- Elimination rule

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ !x = t_1\ \mathsf{in}\ t_2 : B}\ !_E$$

- Dereliction rule: non-linear variables can be used linearly

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \mathrm{DER}$$

Typing rules: non-linear modality

- Introduction rule / promotion

$$\frac{[\Gamma] \vdash t : A}{[\Gamma] \vdash !t : !A}\ !_I \qquad\qquad \frac{!\Gamma \vdash P}{!\Gamma \vdash !P}\ !_R$$

- Elimination rule

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : [A] \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \text{let } !x = t_1 \text{ in } t_2 : B}\ !_E$$

- Dereliction rule: non-linear variables can be used linearly

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A] \vdash t : B}\ \text{DER} \qquad\qquad \frac{\Gamma, P \vdash Q}{\Gamma, !P \vdash Q}\ !_L$$

Typing rules: uniqueness modality

- Borrowing: "forget" uniqueness guarantee

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \& t : !A}\ \textsc{Borrow}$$

Typing rules: uniqueness modality

- Borrowing: "forget" uniqueness guarantee

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \text{BORROW} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^{\circ}}\ \circ_R$$

Typing rules: uniqueness modality

- Borrowing: "forget" uniqueness guarantee

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \textsc{Borrow} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^\circ}\ \circ_R$$

- Copying: make new variable with same value as non-linear term

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{copy}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{Copy}$$

Typing rules: uniqueness modality

- Borrowing: "forget" uniqueness guarantee

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \textsc{Borrow} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^\circ}\ \circ_R$$

- Copying: make new variable with same value as non-linear term

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{copy}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \textsc{Copy} \qquad\qquad \frac{\Gamma, P \vdash Q^\circ}{\Gamma, P^\circ \vdash Q^\circ}\ \circ_L$$

Typing rules: uniqueness modality

- Borrowing: "forget" uniqueness guarantee

$$\frac{\Gamma \vdash t : *A}{\Gamma \vdash \&t : !A}\ \text{BORROW} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma \vdash P^\circ}\ \circ_R$$

- Copying: make new variable with same value as non-linear term

$$\frac{\Gamma_1 \vdash t_1 : !A \qquad \Gamma_2, x : *A \vdash t_2 : !B}{\Gamma_1 + \Gamma_2 \vdash \mathsf{copy}\ t_1\ \mathsf{as}\ x\ \mathsf{in}\ t_2 : !B}\ \text{COPY} \qquad\qquad \frac{\Gamma, P \vdash Q^\circ}{\Gamma, P^\circ \vdash Q^\circ}\ \circ_L$$

- Necessitation: values without dependencies can be assumed unique

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \text{NEC}$$

From typing rules to programming language

- As application of these typing rules, we want to implement them into a programming language

From typing rules to programming language

- As application of these typing rules, we want to implement them into a programming language
- Then we can compare performance of a language with both unique and linear types

From typing rules to programming language

- As application of these typing rules, we want to implement them into a programming language
- Then we can compare performance of a language with both unique and linear types
- However, then we first need to define operational semantics

Heap model

- The paper made an operational heap model

Heap model

- The paper made an operational heap model
- Steps like:

$$H, x \mapsto_1 t \vdash x \quad \rightsquigarrow \quad H \vdash t$$

Heap model

- The paper made an operational heap model
- Steps like:

$$H, x \mapsto_1 t \vdash x \quad \rightsquigarrow \quad H \vdash t$$

- Also include arrays and operations on them
  - As a use case for uniqueness

Metatheory

- In the paper, a couple theorems are proven about the model

Metatheory

- In the paper, a couple theorems are proven about the model
  - Conservation: after reduction step, term still has same type in a typing context compatible with the heap

Metatheory

- In the paper, a couple theorems are proven about the model
  - Conservation: after reduction step, term still has same type in a typing context compatible with the heap
  - Progress: if well-typed term is not a value, another reduction step is possible

Metatheory

- In the paper, a couple theorems are proven about the model
  - Conservation: after reduction step, term still has same type in a typing context compatible with the heap
  - Progress: if well-typed term is not a value, another reduction step is possible
  - Soundness: if two well-typed terms are equivalent ($t_1 \equiv t_2$), then there is a value to which they both reduce

Metatheory

- In the paper, a couple theorems are proven about the model
  - Conservation: after reduction step, term still has same type in a typing context compatible with the heap
  - Progress: if well-typed term is not a value, another reduction step is possible
  - Soundness: if two well-typed terms are equivalent ($t_1 \equiv t_2$), then there is a value to which they both reduce
  - Uniqueness: if a term reduces to a unique value, then any unique array references from the incoming heap or new array references that occur in that value are still unique

Metatheory

- In the paper, a couple theorems are proven about the model
  - Conservation: after reduction step, term still has same type in a typing context compatible with the heap
  - Progress: if well-typed term is not a value, another reduction step is possible
  - Soundness: if two well-typed terms are equivalent ($t_1 \equiv t_2$), then there is a value to which they both reduce
  - Uniqueness: if a term reduces to a unique value, then any unique array references from the incoming heap or new array references that occur in that value are still unique
- Uniqueness theorem is incorrect!

Implementation

- Implemented in Granule: linearly typed language

Implementation

- Implemented in Granule: linearly typed language
- Operations for arrays: new, read, write, delete

Implementation

- Implemented in Granule: linearly typed language
- Operations for arrays: new, read, write, delete
- Write operation updates unique array destructively in place

Implementation

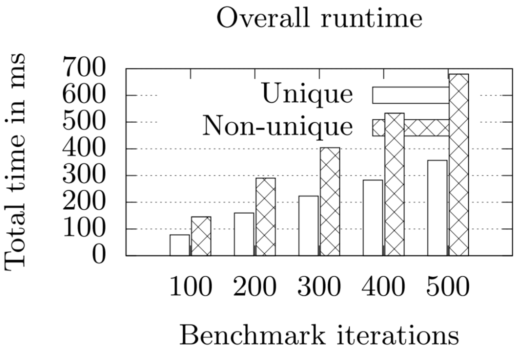
- Implemented in Granule: linearly typed language
- Operations for arrays: new, read, write, delete
- Write operation updates unique array destructively in place
  - Allowed because we have uniqueness guarantee

Implementation

- Implemented in Granule: linearly typed language
- Operations for arrays: new, read, write, delete
- Write operation updates unique array destructively in place
  - Allowed because we have uniqueness guarantee
- Compared performance with/without unique arrays

Performance

- Benchmark iteration: allocate list of 1000 arrays, populate arrays with values, traverse list to sum them up



Questions?

Counterexample uniqueness theorem

- Uniqueness: if a term reduces to a unique value, then any unique array references from the incoming heap or *new array references* that occur in that value are still unique
- Consider following heap + (well-typed) term

$$\emptyset \vdash *(\text{let } !x = \&(\text{newArray } 5) \text{ in } x) : *(\text{Array } A)$$

- Reduces to

$$a \mapsto_{\omega} \mathbf{arr}, x \mapsto_{\omega} a \vdash *a$$

- ω means that array reference a is not unique

Mistake in proof

- Proof of uniqueness theorem is by induction over typing rules
- Problem is with necessitation rule

$$\frac{\emptyset \vdash t : A}{[\Gamma] \vdash *t : *A}\ \textsc{nec}$$

- IH: "For a well-typed term $\Gamma \vdash t_1 : *B \ldots$"
- In the proof, they apply results of IH for $\emptyset \vdash t : A$ from $\textsc{nec}$
- A is not necessarily a unique type $*B$ for some B