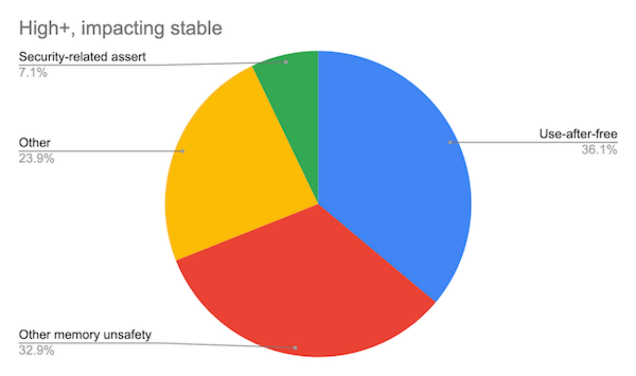
Linear types, Quésaco?!

Gabriel Radanne

Motivation ?

La sûreté mémoire est la source de nombreux bugs et failles de sécurité.

Classification récente (2015-2020) des “high severity security bugs” dans Chromium:



“Use after free”

Un “Use after free” en C

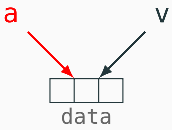
```c
char *s = malloc(len);
/* ... */
free(s);
/* ... */
s[i] // bug!
```

“Use after free”

Des vecteurs auto-redimensionnés en C

```c
struct vector {
  int* data; int limit; int size;
};

struct vector v = init();
/* ... */
int *a = v.data; // Pointe sur le contenu
/* ... */
push(v,2); // Le vecteur peut-être agrandi
/* ... */
a[i] // bug!
```

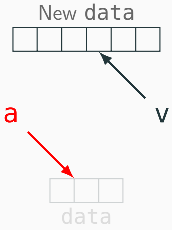


“Use after free”

Des vecteurs auto-redimensionnés en C

```c
struct vector {
  int* data; int limit; int size;
};

struct vector v = init();
/* ... */
int *a = v.data; // Pointe sur le contenu
/* ... */
push(v,2); // Le vecteur peut-être agrandi
/* ... */
a[i] // bug!
```



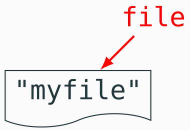
“Use after free”

Utilisons un langage plus sûr: OCaml!

“Use after free”

Utilisons un langage plus sûr: OCaml!

```
let file = open_out "myfile" in
write file "hello";
(* ... *)
close file;
(* ... *)
write file "world"; (* bug! *)
```



“Use after free”

Utilisons un langage plus sûr: OCaml!

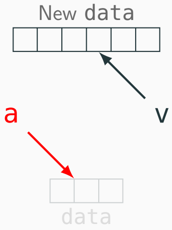
```
let file = open_out "myfile" in
write file "hello";
(* ... *)
close file;
(* ... *)
write file "world"; (* bug! *)
```

"Use after free"

Les types **affines** à la rescousse!

Les vecteurs en **Rust**

```
let mut v = vec![];
/* ... */
let a = &mut v[0]; // Pointe sur le contenu
/* ... */
v.push(12); // Le vecteur peut-être agrandi
/* ... */
a[1]; // ✘ Erreur de compilation !
```



The main idea

Main idea:
Limit usage of variables
We call such systems "**sub-structural**"

In the rest of this talk:

- We use the word "**resource**" for things we want to limit the usage of
- We use some imaginary ML-ish syntax

The main idea

Main idea:
Limit usage of variables
We call such systems "**sub-structural**"

In the rest of this talk:

- We use the word "**resource**" for things we want to limit the usage of
- We use some imaginary ML-ish syntax

Plan

Linear types: the beginning

Modality determine usage:

- Linear (`lin`): Used exactly once [1]
- Affine (`aff`): Used at most once [0 – 1]
- Unrestricted (`un`): Used arbitrarily many time [$0 - \infty$]

Examples:

- file descriptors are linear
- GC-managed strings are unrestricted

Linear types: A file API

Let's create an `Database` API together!

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
end
```

Linear types: A file API

Let's create an `Database` API together!

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
end

let main () =
  let a = Dbm.open "foo" in
  .... (* a is linear *)
  Dbm.close a
```

Linear types: A file API

Let's create an Database API together!

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
end

let main () =
  let a = Dbm.open "foo" in
  .... (* a is linear *)
  Dbm.close a ;
  f a (* ✘ No! *)
```

Linear types: A file API

How to read the array ?

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
  val find : t -> string -> int (* ? *)
end
```

Linear types: A file API

How to read the array ?

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
  val find : t -> string -> int (* ? *)
end
```

```
let main () =
  let gradeDB = Dbm.open "grades.db" in
  let x = Dbm.find gradeDB "math" in
  Dbm.close gradeDB (* ✘ No! *)
  print x
```

This doesn't work!

Linear types: A file API

How to read the array ?

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val open : filename -> t
  val close : t -> unit
  val find : t -> string -> t * int
end

let main () =
  let gradeDB = Dbm.open "grades.db" in
  let gradeDB, x = Dbm.find gradeDB "math" in
  Dbm.close gradeDB ;
  print x
```

Linear types: the conclusion

We know everything about linear types!

- Gives us safe manual allocations and IO
- Modality (linear, affine, unrestricted) to control uses

Let's dig a bit more

Linear types: the conclusion

We know everything about linear types!

- Gives us safe manual allocations and IO
- Modality (linear, affine, unrestricted) to control uses

Let's dig a bit more

Session types: Intro

Session types aims to describe *protocols* through types.

Example: Ordering coffee

1. Choose program
2. Add Cup
3. (Get Coffee) or (Not enough grains, Add grains, Get Coffee)

Our tools:

$!\tau.S$	Send some τ then continue with S.
$?\tau.S$	Receive some τ then continue with S.
$S \oplus S'$	Internal choice between S and S'.
$S \,\&\, S'$	Offer a choice between S and S'.

Session types: Intro

Session types aims to describe *protocols* through types.

Example: Ordering coffee

1. Choose program
2. Add Cup
3. (Get Coffee) or (Not enough grains, Add grains, Get Coffee)

Our tools:

$!\tau.S$	Send some τ then continue with S.
$?\tau.S$	Receive some τ then continue with S.
$S \oplus S'$	Internal choice between S and S'.
$S \,\&\, S'$	Offer a choice between S and S'.

Session types: the Types

$$\begin{aligned} !\tau.S \quad & \text{Send some } \tau \text{ then continue with } S. \\ ?\tau.S \quad & \text{Receive some } \tau \text{ then continue with } S. \\ S \oplus S' \quad & \text{Internal choice between } S \text{ and } S'. \\ S \,\&\, S' \quad & \text{Offer a choice between } S \text{ and } S'. \end{aligned}$$

User point of view:

$!\texttt{Program}.\ !\texttt{Cup}.$
$\quad (?\texttt{Coffee}.\ \texttt{End} \,\&\, !\texttt{Grains}.\ ?\texttt{Coffee}.\ \texttt{End})$

Coffee machine point of view:

$?\texttt{Program}.\ ?\texttt{Cup}.$
$\quad (!\texttt{Coffee}.\ \texttt{End} \oplus ?\texttt{Grains}.\ !\texttt{Coffee}.\ \texttt{End})$

Notion of **dual** of a type.

Session types: the Types

$!\tau.S$	Send some τ then continue with S.
$?\tau.S$	Receive some τ then continue with S.
$S \oplus S'$	Internal choice between S and S'.
$S \mathbin{\&} S'$	Offer a choice between S and S'.

User point of view:
$!$`Program`. $!$`Cup`.
($?$`Coffee`. End $\&$ $!$`Grains`. $?$`Coffee`. End)

Coffee machine point of view:
$?$`Program`. $?$`Cup`.
($!$`Coffee`. End $\oplus$ $?$`Grains`. $!$`Coffee`. End)

Notion of **dual** of a type.

Session types: the code

```
let request_coffee (ch : ... channel) program =
  let ch = send ch program in
  let ch = send ch my_favorite_cup in
  match test ch with
  | Coffee ch ->
    let coffee, ch = receive ch in
    close ch;
    coffee
  | NotEnoughGrain ch ->
    let grains =
      GrainProvider.coffee ()
    in
    let ch = send ch grains in
    let coffee, ch = receive ch in
    close ch;
    coffee
```

The operations:

```
type 'S ch
val send : (!'a. 'S) ch -> 'a -> 'S ch
val receive: (?'a. 'S) ch -> 'a * 'S ch
val test : ('S1⊕'S2) ch -> ('S1 ch | 'S2 ch)
val close : end ch -> unit
```

Session types: Linearity

For correction, channels **must** be linear!

- Must never skip/duplicate steps
- Must fully consume the channel

```
...
let ch1 = send ch program in
let ch2 = send ch my_favorite_cup in ✗
...
ignore ch2 ✗
```

The operations:

```
type 'S ch
val send : (!'a. 'S) ch -> 'a -> 'S ch
val receive: (?'a. 'S) ch -> 'a * 'S ch
val test : ('S1⊕'S2) ch -> ('S1 ch | 'S2 ch)
val close : end ch -> unit
```

Session types: the code

We assemble the various parts thanks to duals:

```
let main () =
  let ch, ch' = create () in
  fork (coffee_machine ch);
  request_coffee ch my_program
```

The operations:

```
type 'S ch
val send : (!'a. 'S) ch -> 'a -> 'S ch
val receive: (?'a. 'S) ch -> 'a * 'S ch
val test : ('S1⊕'S2) ch -> ('S1 ch | 'S2 ch)
val close : end ch -> unit

val create : unit -> 'S ch * (dual 'S) ch
```

Session types

- Linear types as a building block
- Static verification of conformance to “protocols”
- $\Rightarrow$ Encode state automatons in types

Billions of extensions (recursive, multi-party, multi-tiers, OOP, asynchronous, ...).
Some practical use in limited communities (OS $\cap$ Static typing = { Rust, Mirage, ...})

> « *But Gabriel, this code is too functional, it’s a PITA to write and it’s probably slow as $"£#!* »
>
> – The public, when I prepare my talk alone

Let's go imperative

The Database API is back:

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val create : int -> 'a -> t
  val close : 'a t -> unit
  val find : t -> string -> t -> int (* ??? *)
end
```

We need to pass the database around, this is very inconvenient.

⇒ We want to write imperative code to mutate the world!

Let's go imperative

The Database API is back:

```
module Dbm : sig
  type t : lin (* Databases are linear *)
  val create : int -> 'a -> t
  val close : 'a t -> unit
  val find : &t -> string -> int (* A borrow! *)
end

let main () =
  let gradeDB = Dbm.open "grades.db" in
  let x = Dbm.find &gradeDB "math" in
  Dbm.close gradeDB ;
  print x
```

Nice imperative-like code using borrows! ✔

Borrows

A borrow is a temporary loan of a resource `a`

- **Shared** borrows `&a` are for observing the resource
- **Exclusive** borrows `&!a` are for modifying the resource

Borrows

A borrow is a temporary loan of a resource a

- **Shared** borrows &a are for observing the resource
- **Exclusive** borrows &!a are for modifying the resource

A correct usage of borrows:

Unrestricted – un

```
let avg =
  (Dbm.find &gradeDB "math" + Dbm.find &gradeDB "compsci") / 2
  (* ✔ Multiple shared borrows *)
in
Dbm.add &!gradeDB "average" avg (* ✔ One exclusive borrow *)
```

Affine – aff

Rule 1: Cannot use a borrow and the resource itself simultaneously

```
let gradeDB = ... in
f (gradeDB, &gradeDB) (* ✘ Conflicting use and borrow! *)
```

Borrows – Example of uses

Rule 2: Cannot use an exclusive borrow and any other borrow simultaneously

```
let gradeDB = ... in
f (&!gradeDB, &gradeDB) (* ✘ Conflicting borrows! *)
```

Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =
  let gradeDB = ... in
  let x = (&gradeDb, "mygrades") in
  x
  (* ✘ Borrow escaping its scope! *)
```

Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =
  let gradeDB = ... in
  {| let x = (&gradeDb, "mygrades") in
  x |}
  (* ✘ Borrow escaping its scope! *)
```

Borrows – Example of uses

Rule 3: Borrows must not escape

```
let f () =
  let gradeDB = ... in
  {| let x = (&gradeDb, "mygrades") in
  x |}
  (* ✘ Borrow escaping its scope! *)
```

A Region!

Regions ensure that borrows do not escape!

Borrows

In Rust:

- Regions are not so lexical
- The compiler tries very hard to guess what the user meant
- Much more control over allocations, C++-like.

$\Rightarrow$ Lot's of tools to tangle yourself ... but safely!

That's all for safety, let's look at ***performances***!

Borrows

In Rust:

- Regions are not so lexical
- The compiler tries very hard to guess what the user meant
- Much more control over allocations, C++-like.

$\Rightarrow$ Lot's of tools to tangle yourself ... but safely!

That's all for safety, let's look at ***performances***!

Linearity for optimisations

Futhark is a *pure* functional language for GPGPUs.

Example: Radix sort in Futhark

```
let radix_sort_step [n] (xs: [n]u32) (b: i32): [n]u32 =
  let bits = map (\x -> (i32.u32 (x >> u32.i32 b)) & 1) xs
  let bits_neg = map (1-) bits
  let offs = reduce (+) 0 bits_neg
  let idxs0 = map2 (*) bits_neg (scan (+) 0 bits_neg)
  let idxs1 = map2 (*) bits (map (+offs) (scan (+) 0 bits))
  let idxs2 = map2 (+) idxs0 idxs1
  let idxs  = map (\x->x-1) idxs2
  scatter (copy xs) (map i64.i32 idxs) xs
```

Linearity used as an aliasing analysis

Can transform all these operation to in-place versions!

☺

Linearity for data-structures

We can use linearity to enforce hybrid data-structure performance contracts [Conchon and Filliâtre, 2007, Puente, 2017]

Example: Hash-Array-Mapped-Tries (HAMT)

- Persistent immutable operations
  `set : ('k, 'v) hamt -> 'k -> 'v -> ('k, 'v) hamt`
  For cold path, O(log(N)), some copies
  Use locks/copies, support concurrency and backtracking
- Transient mutable operations
  `set : ('k, 'v) hamt -> 'k -> 'v -> unit`
  For hot path, O(1), no copies
  Use (potentially dynamically-checked) linearity to allow in-place operations

$\Rightarrow$ Requires hybrid languages, with both linear and non-linear accesses and borrows.

Very promising lead, not fully realized yet.

Linear types: Why ?

Linear types have many uses:

- Direct uses for safety: channels, memory alloc, ...
  This has reached "mainstream" (Rust) ✔
- Advances safety uses: session types, type-states, ...
  This is still very active research. Some basic encoding exists.
- Optimisation uses
  Very promising prototypes, still requires key compiler/langage improvements

$\Rightarrow$ We have yet to find all the programming uses of linear types

This was **Why**

Let's now see **How**

Linear types: Why ?

Linear types have many uses:

- Direct uses for safety: channels, memory alloc, ...
  This has reached "mainstream" (Rust) ✔
- Advances safety uses: session types, type-states, ...
  This is still very active research. Some basic encoding exists.
- Optimisation uses
  Very promising prototypes, still requires key compiler/langage improvements

$\Rightarrow$ We have yet to find all the programming uses of linear types

This was **Why**

Let's now see **How**

How can I use my variables?

```
let r = create_resource()
begin
  shadok r; (* Can I pass it as argument? *)
  r (* And still use it after? *)
end
let x = (r, r) (* Can I duplicate it? *)
let f x = write r x (* Can I capture it? *)
let {foo; bar} = r (* Can I decompose it? *)
...
r (* Can I return it? *)
```

Humble beginnings: Linear logic

Simple questions on variable:

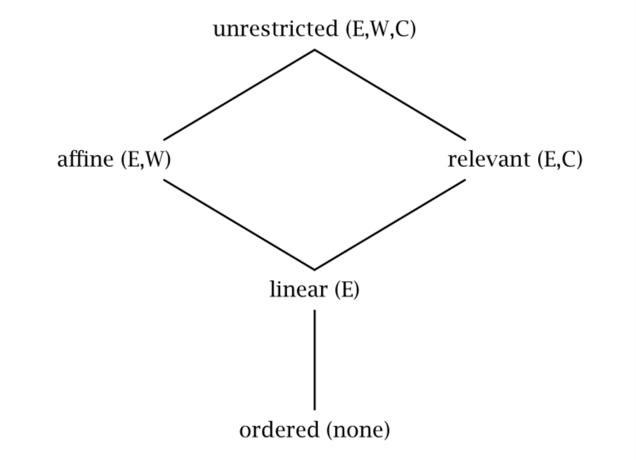
- Do I *have* to use it ? (Weakening)
- Can I use it several time ? (Contraction)
- Is the order of definition important ? (Exchange)

Weakening
$$\frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_1, (x : \tau), \Gamma_2 \vdash e : \tau}$$

Contraction
$$\frac{\Gamma_1, (x_1 : \tau), (x_2 : \tau), \Gamma_2 \vdash e : \tau}{\Gamma_1, (x : \tau), \Gamma_2 \vdash e[x_1, x_2 \to x] : \tau}$$

Exchange
$$\frac{\Gamma_1, (x_1 : \tau_1), (x_2 : \tau_2), \Gamma_2 \vdash e : \tau}{\Gamma_1, (x_2 : \tau_2), (x_1 : \tau_1), \Gamma_2 \vdash e : \tau}$$

The Sub-structural lattice



A simple linear calculus

1 Substructural Type Systems

David Walker

A simple linear calculus

$$q ::= un \mid lin \qquad \text{(Modality)}$$

$$e ::= c \mid x \mid e\ e' \qquad \text{(Expressions)}$$
$$\mid q\ \lambda(x : T).e$$
$$\mid q\ {<}e, e'{>}$$
$$\mid let\ x, y = e\ in\ e'$$

$$P ::= T * T \mid T \to T \qquad \text{(Pretypes)}$$

$$T ::= q\ P \qquad \text{(Types)}$$

$$\Gamma ::= (x : T)^* \qquad \text{(Environments)}$$

```
λ(x:un int).x+x ✔

λ(x:lin int).x+1 ✔
λ(x:lin int).x+x ✘
λ(x:lin int).3 ✘
```

```
(λz.λy.<free z,free y>) x x ✘

let r:lin int = 3 in
let f = λx.(r+x) in
<f 1, f 2> ✘
```

A simple linear calculus

$$
\begin{aligned}
q &::= un \mid lin && \text{(Modality)} \\
e &::= c \mid x \mid e\ e' && \text{(Expressions)} \\
&\mid q\ \lambda(x : T).e \\
&\mid q\ {<}e, e'{>} \\
&\mid let\ x, y = e\ in\ e'
\end{aligned}
$$

$$
\begin{aligned}
P &::= T * T \mid T \to T && \text{(Pretypes)} \\
T &::= q\ P && \text{(Types)} \\
\Gamma &::= (x : T)^* && \text{(Environments)}
\end{aligned}
$$

```
λ(x:un int).x+x ✔
```

```
(λz.λy.<free z,free y>) x x ✘
```

```
λ(x:lin int).x+1 ✔
λ(x:lin int).x+x ✘
λ(x:lin int).3 ✘
```

```
let r:lin int = 3 in
let f = λx.(r+x) in
<f 1, f 2> ✘
```

A simple linear calculus

$$\begin{aligned} q &::= un \mid lin && \text{(Modality)} \\ e &::= c \mid x \mid e\ e' && \text{(Expressions)} \\ &\mid q\ \lambda(x : T).e \\ &\mid q <e, e'> \\ &\mid let\ x, y = e\ in\ e' \end{aligned}$$

$$\begin{aligned} P &::= T * T \mid T \to T && \text{(Pretypes)} \\ T &::= q\ P && \text{(Types)} \\ \Gamma &::= (x : T)^* && \text{(Environments)} \end{aligned}$$

```
λ(x:un int).x+x ✔

λ(x:lin int).x+1 ✔
λ(x:lin int).x+x ✘
λ(x:lin int).3 ✘
```

```
(λz.λy.<free z,free y>) x x ✘

let r:lin int = 3 in
let f = λx.(r+x) in
<f 1, f 2> ✘
```

Playing with environments

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{q}\ \mathsf{T}_{11} \to \mathsf{T}_{12} \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathsf{T}_{11}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{T}_{12}} \quad \text{(T-APP)}$$

The most important ingredient: How to manipulate environments!

Playing with environments

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{q}\, \mathsf{T}_{11} \to \mathsf{T}_{12} \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathsf{T}_{11}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{T}_{12}} \quad \text{(T-APP)}$$

The most important ingredient: How to manipulate environments!

Context Split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\varnothing = \varnothing \circ \varnothing \quad \text{(M-EMPTY)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x}{:}\mathtt{un}\ \mathsf{P} = (\Gamma_1, \mathtt{x}{:}\mathtt{un}\ \mathsf{P}) \circ (\Gamma_2, \mathtt{x}{:}\mathtt{un}\ \mathsf{P})} \quad \text{(M-UN)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x}{:}\mathtt{lin}\ \mathsf{P} = (\Gamma_1, \mathtt{x}{:}\mathtt{lin}\ \mathsf{P}) \circ \Gamma_2} \quad \text{(M-LIN1)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, \mathtt{x}{:}\mathtt{lin}\ \mathsf{P} = \Gamma_1 \circ (\Gamma_2, \mathtt{x}{:}\mathtt{lin}\ \mathsf{P})} \quad \text{(M-LIN2)}$$

Secret Sauce 1: We restrict Contraction to specific variables during split

Sub-sumption

$$\frac{\Gamma_1 \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma_2 \vdash \mathtt{t}_2 : \mathtt{T}_2 \qquad \mathtt{q}(\mathtt{T}_1) \qquad \mathtt{q}(\mathtt{T}_2)}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{q}\ \mathtt{<t_1,t_2>} : \mathtt{q}\ (\mathtt{T}_1 \mathtt{*} \mathtt{T}_2)} \quad \text{(T-PAIR)}$$

where $q(T)$, if and only if $T = q'P$ and $q \sqsubseteq q'$

We have $\mathtt{lin} \sqsubseteq \mathsf{un}$

Secret Sauce 2: We can "upgrade" modality along the lattice

Playing with environments

$$\frac{\mathsf{un}\ (\Gamma_1, \Gamma_2)}{\Gamma_1, \mathsf{x}{:}\mathsf{T}, \Gamma_2 \vdash \mathsf{x} : \mathsf{T}} \quad \text{(T-VAR)} \qquad \frac{\mathsf{q}(\Gamma) \qquad \Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{q}\ \lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2 : \mathsf{q}\ \mathsf{T}_1 \to \mathsf{T}_2} \quad \text{(T-ABS)}$$

where $q(\Gamma)$, if for every binding $(x : T) \in \Gamma$ we have $q(T)$

Secret Sauce 3: We restrict Weakening to specific variables during usage/capture

Let's typecheck together

$$\frac{}{(x : \texttt{lin int}), (y : \texttt{un int}) \vdash \texttt{lin} <(\lambda(z : \texttt{lin int}).z + 1)\ (x + y), y> : ?}$$

Properties

We can prove:

- Decidability and completeness of typing
- Soundness with an oblivious λ-calculus semantics
- Soundness with a heap-aware λ-calculus semantics (which de-allocate linear resources aggressively)

Extensions

We can easily extend to:

- Algebraic data-types
- Polymorphism
- Arrays/references/...

Some more unusual ideas:

- "Managed" (GC) or "Ref counted" can also be modalities!
- Control space and time(!) complexity of programs
  Example: If we only use affine variables, programs are polynomial
- Modeling of stack allocations
  If we restrict Exchange, resources can only be removed in-order, like a stack.

$\Rightarrow$ Of great theoretical interest (c.f. Plume), but so far little used for programming.

Extensions

We can easily extend to:

- Algebraic data-types
- Polymorphism
- Arrays/references/...

Some more unusual ideas:

- "Managed" (GC) or "Ref counted" can also be modalities!
- Control space and time(!) complexity of programs
  Example: If we only use affine variables, programs are polynomial
- Modeling of stack allocations
  If we restrict Exchange, resources can only be removed in-order, like a stack.

$\Rightarrow$ Of great theoretical interest (c.f. Plume), but so far little used for programming.

Simple linear lambda calculus: Wrap up

The main principle of substructural type systems:
By restricting Contraction/Weakening/Exchange for *some* variables, we can control usage.

To design a new linear type system
You need to answer three questions:

- How to decide on which variables to apply Contraction?
- How to decide on which variables to apply Weakening?
- Can (and How) a variable change modality?
- How much polymorphism do you allow

This is where the design space explodes a little bit . . .

Simple linear lambda calculus: Wrap up

The main principle of substructural type systems:
By restricting Contraction/Weakening/Exchange for *some* variables, we can control usage.

To design a new linear type system
You need to answer ~~three~~ four questions:

- How to decide on which variables to apply Contraction?
- How to decide on which variables to apply Weakening?
- Can (and How) a variable change modality?
- How much polymorphism do you allow

This is where the design space explodes a little bit . . .

Simple linear lambda calculus: Wrap up

The main principle of substructural type systems:
By restricting Contraction/Weakening/Exchange for *some* variables, we can control usage.

To design a new linear type system
You need to answer ~~three~~ four questions:

- How to decide on which variables to apply Contraction?
- How to decide on which variables to apply Weakening?
- Can (and How) a variable change modality?
- How much polymorphism do you allow

This is where the design space explodes a little bit ...

The design space

The design space: Subsumption

Can (and How) a variable change modality?

Example with the linear λ-calculus:

```
let f
  : lin int -> lin int
  = λx.(2*x+1)

let x : un int = 3
let y = f x
```

The design space: Subsumption

Can (and How) a variable change modality?

- Unrestricted can become Linear/Affine
  $\Rightarrow$ **Linear/Affine** types
  Good for safety, can limit optimisations
  "Linearity means it will never be aliased in the future"
  Examples: Rust, Affe, Mezzo, Almost everything you know
- ~~Linear~~Unique can become Unrestricted
  $\Rightarrow$ **Unique** types
  Enable aggressive optimisations
  "Uniqueness means it was never aliased in the past"
  Examples: Futhark, Clean, Idris.
- Combination of both/neither/requires additional proofs

The design space: Subsumption

Can (and How) a variable change modality?

- Unrestricted can become Linear/Affine
  $\Rightarrow$ **Linear/Affine** types
  Good for safety, can limit optimisations
  "Linearity means it will never be aliased in the future"
  Examples: Rust, Affe, Mezzo, Almost everything you know
- ~~Linear~~Unique can become Unrestricted
  $\Rightarrow$ **Unique** types
  Enable aggressive optimisations
  "Uniqueness means it was never aliased in the past"
  Examples: Futhark, Clean, Idris.
- Combination of both/neither/requires additional proofs

The design space: Variables

How to decide on which variables to apply Contraction/Weakening?

	Examples	Pros	Cons
Types	Linear λ-calculus, Clean, Futhark, Object systems, C++'s `unique_ptr`	Simple	Inflexible
Arrows	Linear Haskell	Looks like linear logic	I stopped counting
Type classifiers (Kinds, Classes, Dependent, ...)	Rust, Affe, Alms, Idris, ...	Language integration Polymorphism	Complicated internals
Permissions/Logic assertions	Mezzo, (F*), ...	Expressive	Complicated to use
Discharge as Proof obligation	Separation logic, ATS, ...	You are in a proof assistant	

The design space: Variables

How to decide on which variables to apply Contraction/Weakening?

	Examples	Pros	Cons
Types	Linear λ-calculus, Clean, Futhark, Object systems, C++'s `unique_ptr`	Simple	Inflexible
Arrows	Linear Haskell	Looks like linear logic	I stopped counting
Type classifiers (Kinds, Classes, Dependent, ...)	Rust, Affe, Alms, Idris, ...	Language integration Polymorphism	Complicated internals
Permissions/Logic assertions	Mezzo, (F*), ...	Expressive	Complicated to use
Discharge as Proof obligation	Separation logic, ATS, ...	You are in a proof assistant	

The design space: Variables

How to decide on which variables to apply Contraction/Weakening?

	Examples	Pros	Cons
Types	Linear λ-calculus, Clean, Futhark, Object systems, C++'s `unique_ptr`	Simple	Inflexible
Arrows	Linear Haskell	Looks like linear logic	I stopped counting
Type classifiers (Kinds, Classes, Dependent, ...)	Rust, Affe, Alms, Idris, ...	Language integration Polymorphism	Complicated internals
Permissions/Logic assertions	Mezzo, (F*), ...	Expressive	Complicated to use
Discharge as Proof obligation	Separation logic, ATS, ...	You are in a proof assistant	

The lay of the land

Rust

Weakening:	Always (Affine!)
Contraction:	Controlled via traits `Copy` and `Clone`
Subsumption:	Only for borrows, via traits
Polymorphism:	Partially, via traits
Bonuses ✔:	Borrows, rich elision rules to avoid annotations, non-lexical regions, concurrency …
Maluses ✘:	No GC, Limited support for closures

The lay of the land

Affe (Kindly Bent to Free Us, ICFP2020)

Weakening: Contraction:	Kinds (`lin`,`aff`,`un`)
Subsumption:	Subkinding
Polymorphism:	Yes, polymorphic kinds
Bonuses ✔:	With GC, Borrows, Functional+Imperative prog, Full type inference
Maluses ✘:	With GC, Limited regions, No concurrency

Conclusion

We have explored linear types:

Why:

- Soundness. Partially achieved, but we can go futher!
- Performance (Compiler and Data-structures). Still WIP in many respect

How:

- Control how variable behaves
- Use language construct to decide which variable to control

Leads and WIP:

- Hybridization with other programming construct/style

A glimpse of the bleeding edge

Linear types + Static analysis:

In Rust, there is a construct: `unsafe`.

```
unsafe {
    let my_slice: &[u32] = slice::from_raw_parts(pointer, length);
    assert_eq!(some_vector.as_slice(), my_slice);
}
```

Allow to say "I know this piece of code doesn't respect the borrow checker, please let me"

⇒ Opportunity for static analysis: The RustBelt project!

We might want the same thing for functional linear languages.

A glimpse of the bleeding edge

Linear types + Static analysis:

In Rust, there is a construct: `unsafe`.

```
unsafe {
    let my_slice: &[u32] = slice::from_raw_parts(pointer, length);
    assert_eq!(some_vector.as_slice(), my_slice);
}
```

Allow to say "I know this piece of code doesn't respect the borrow checker, please let me"

⇒ Opportunity for static analysis: The RustBelt project!

We might want the same thing for functional linear languages.

Close(Talk)

References i

Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In Claudio V. Russo and Derek Dreyer, editors, *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, pages 37–46. ACM, 2007. doi: 10.1145/1292535.1292541. URL `https://doi.org/10.1145/1292535.1292541`.

Juan Pedro Bolívar Puente. Persistence for the masses: RRB-vectors in a systems language. *PACMPL*, 1(ICFP):16:1–16:28, 2017. doi: 10.1145/3110260. URL `https://doi.org/10.1145/3110260`.