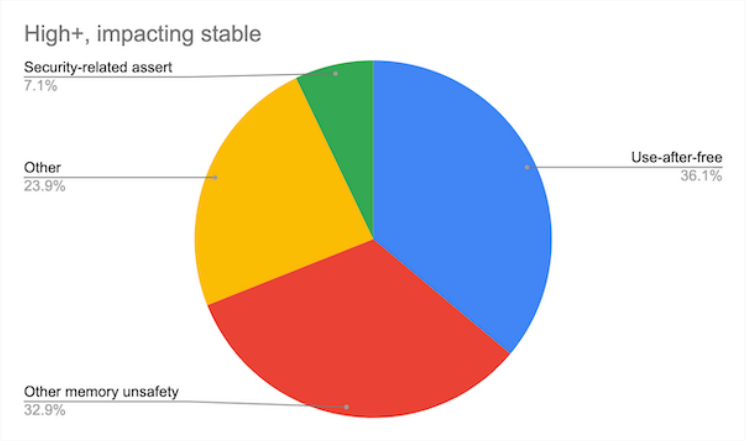
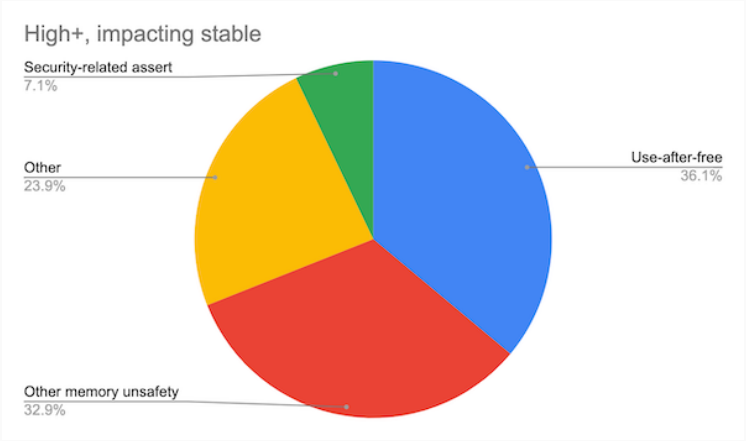
Kindly Bent To Free Us

Gabriel Radanne Hannes Saffrich Peter Thiemann

“high severity security bugs” in Chromium



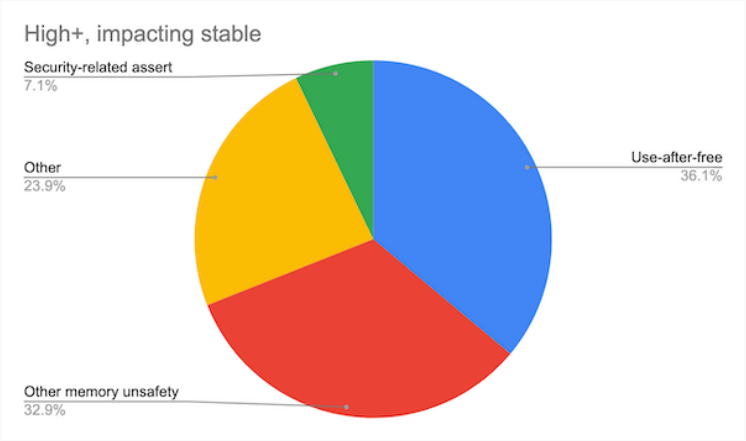
“high severity security bugs” in Chromium



Chromium is written in C/C++!

Surely these bugs don’t happen in high-level typed languages.

“high severity security bugs” in Chromium



Chromium is written in C/C++!

Surely these bugs don’t happen in high-level typed languages . . . right?

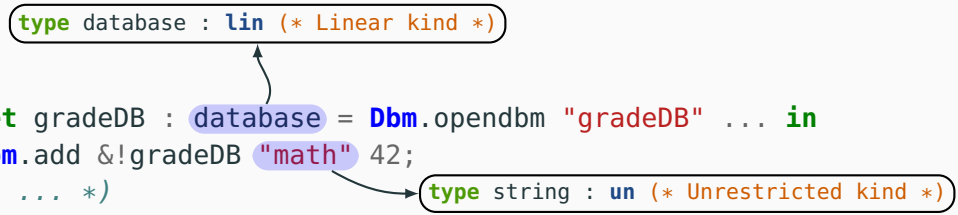
Let's write some OCaml code

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find gradeDB "literature") (* run-time error! *)
```

Affe!

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* ✘ compile-time error! *)
```

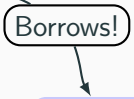
Affe!



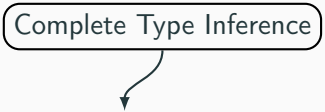
Kinds determine usage

Affe!

```
let gradeDB : database = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* ✘ compile-time error! *)
```



Affe!




```
let gradeDB = Dbm.opendbm "gradeDB" ... in
Dbm.add &!gradeDB "math" 42;
(* ... *)
Dbm.close gradeDB;
(* ... *)
print_int (Dbm.find &gradeDB "literature") (* ✘ compile-time error! *)
```

Table of contents

Linearity through kinds

Kinds 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:

```
type database : lin
type string : un
```

Linearity through kinds

Kinds 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:

```
type ('a : 'k) list : 'k
```

Linearity through kinds

Kinds 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:

```
type ('a : 'k) list : 'k
val create_list : ('a : un) => int -> 'a -> 'a list
```

Linearity through kinds

Kinds 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:

```
type ('a : 'k) list : 'k
val create_list : ('a : un) => int -> 'a -> 'a list
```

We also use **subkinding**: **un** $\leq$ **aff** $\leq$ **lin**

Table of contents

Functions and captures

```
let gradeDB = Dbm.open ...

let log_n_close msg =
  printf "Closing: %s" msg;
  Dbm.close gradeDB
```

Functions and captures

```
let gradeDB = Dbm.open ...

let log_n_close msg =
  printf "Closing: %s" msg;
  Dbm.close gradeDB
```

Functions and captures

```
let gradeDB = Dbm.open ...

let log_n_close msg =
  printf "Closing: %s" msg;
  Dbm.close gradeDB
```

Capture!

We infer the type:

```
val log_n_close : string -[lin]-> unit
```

Usage mode

Functions and captures

```
let gradeDB = Dbm.open ...

let log_n_close msg =
  printf "Closing: %s" msg;
  Dbm.close gradeDB
```

Capture!

We infer the type:

```
val log_n_close : string -lin-> unit
```

Warning: Does not say anything about the arguments!!

Usage mode

Table of contents

Borrows

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

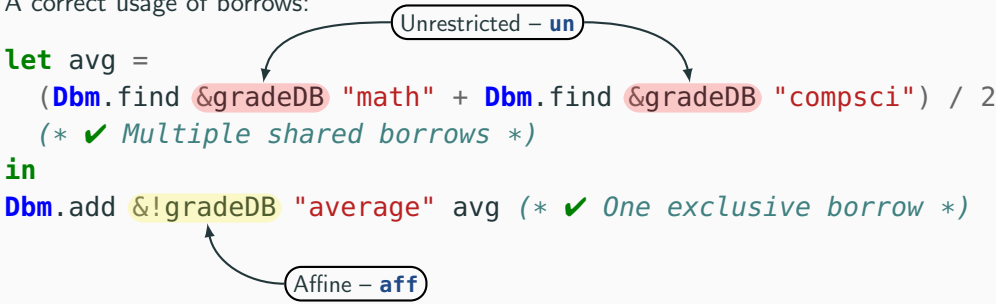
- **Shared** borrows `&a` are for observing the resource
- **Exclusive** borrows `&!a` are for modifying the resource

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:



Borrows – Example of uses

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

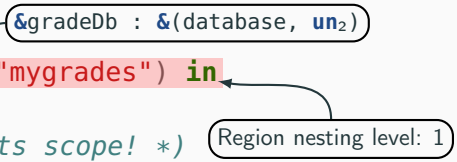
Rule 3: Borrows must not escape

```
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! *)
```



Indexed kinds ensure that borrows do not escape!

Borrows of index 2 cannot escape a region of index 1.

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  (Dbm.find &gradeDB "math" + Dbm.find &gradeDB "compsci") / 2

in
Dbm.add &!gradeDB "average" avg
```

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  let grade subject = Dbm.find &gradeDB subject in (* ✔ Capture *)
  (grade "math" + grade "compsci") / 2
in
Dbm.add &!gradeDB "average" avg
```

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

A simple use:

```
let gradeDB = ... in
let avg =
  let grade = Dbm.find &gradeDB in (* ✔ Partial application *)
  (grade "math" + grade "compsci") / 2
in
Dbm.add &!gradeDB "average" avg
```

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

A simple use:

```
let average db subjects =
  List.map (Dbm.find db) subjects / List.length subjects
let main () =
  let gradeDB = ... in
  let avg = average &gradeDB ["math";"compsci";...] in
  Dbm.add &!gradeDB "average" avg
```

Everything together

The `Database` API:

```
type database : lin
val find : &(database,'k) -> string -'k-> int
val add : &!(database,'k) -> string -'k-> int -'k-> unit
```

A simple use:


```
let average db subjects =
  List.map (Dbm.find db) subjects / List.length subjects
let main () =
  let gradeDB = ... in
  let avg = {| average &gradeDB ["math";"compsci";...] |} in
  {| Dbm.add &!gradeDB "average" avg |}
```

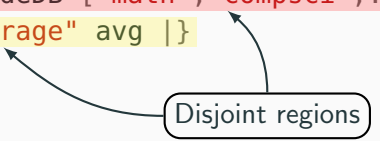
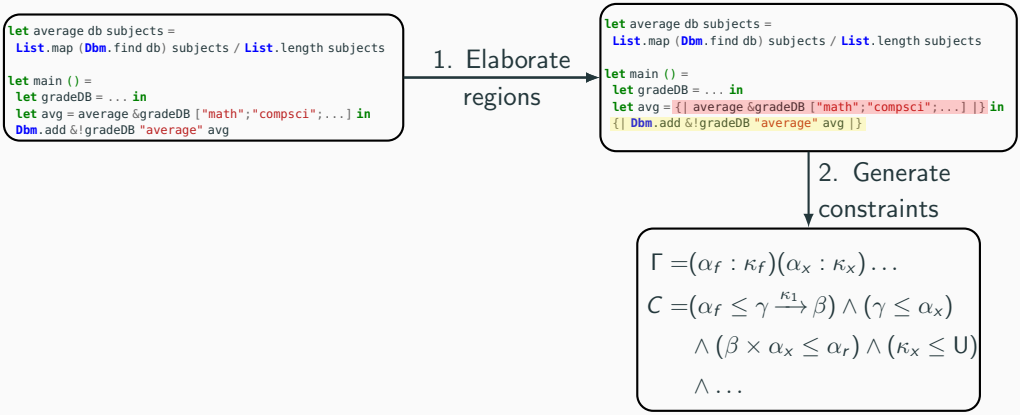


Table of contents

Inference in action

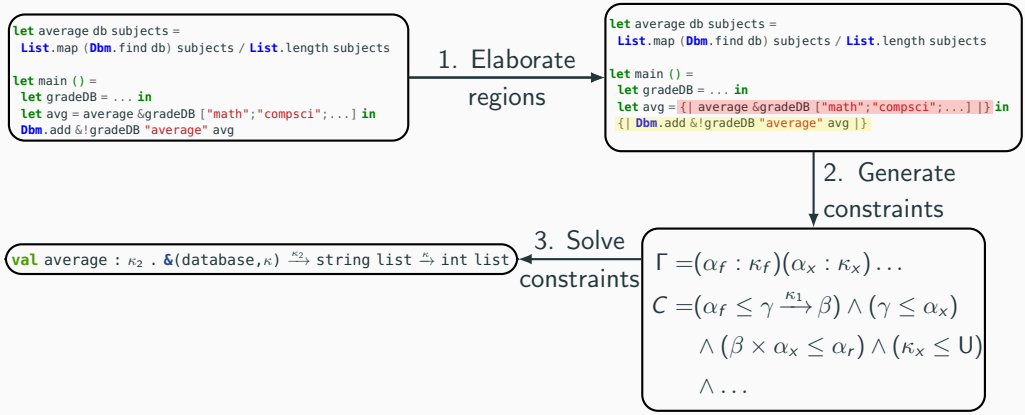
1. Infer the placement of region based on the position of borrows and the borrowing rules.

Inference in action



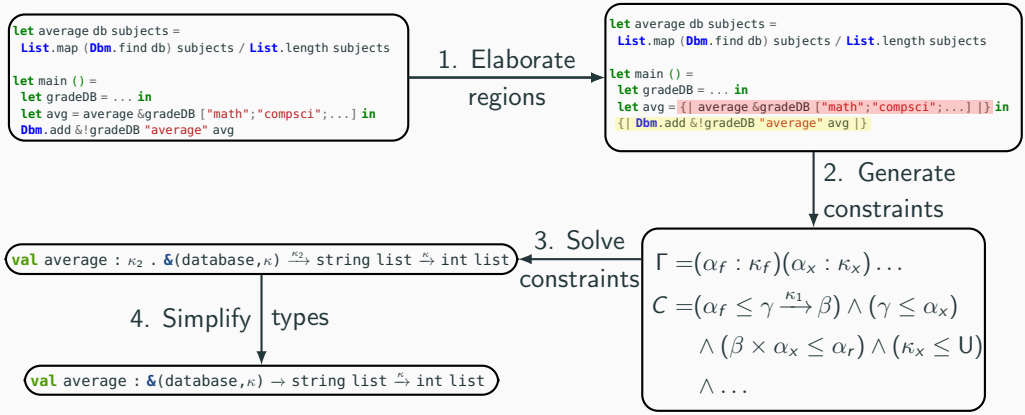
2. Generate custom constraints based on HM(X).

Inference in action



3. Solve the constraints using a custom algorithm and obtain principal type schemes.

Inference in action



4. Simplify the obtained type scheme by leveraging subkinding

The Affe language – Summary

Prototype: `https://affe.netlify.com/`

- ✔ Linearity, Closures, Borrows and Regions
  ⇒ Good support for both imperative and functional programming
- ✔ Support managed and unmanaged objects
- ✔ Principal type inference
- ✘ No flow sensitivity
- ✘ No concurrency story (yet)

Many examples in the paper: files, session types, semi-persistent arrays, iterators, connection pools, ...

The theory – Summary

- A Syntax-directed type system for Affe
  $\Rightarrow$ How to encode borrows into an ML-style type-system
- A formal semantics that takes allocations into account (+ proof of soundness)
- An inference algorithm for Affe:
  - An extension of HM(X) with kinds
  - A novel constraint systems to encode linearity and borrows
  - A constraint solving algorithm, and its proof of completeness

Close(Talk)