You Only Get One Shot

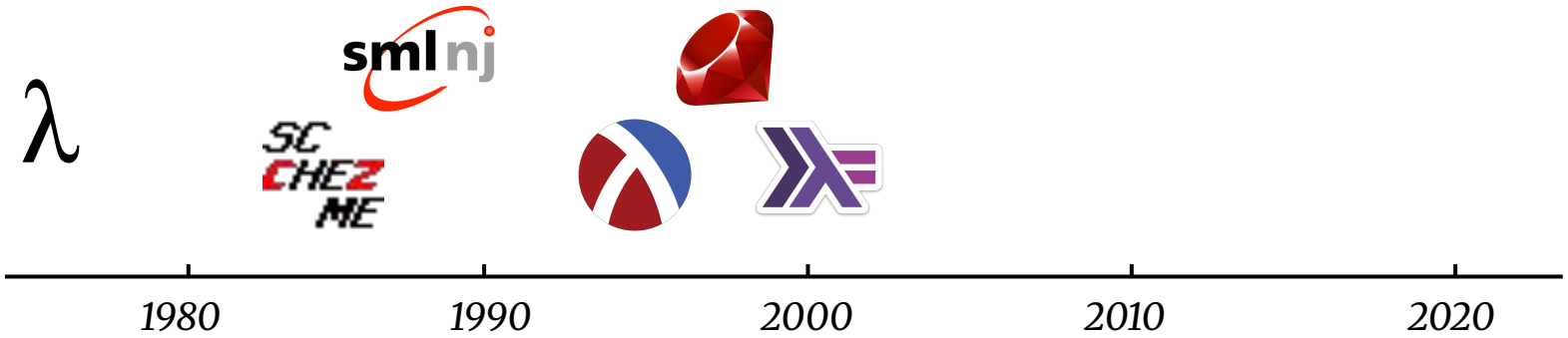
Reasoning About One-Shot Undelimited Continuations in Separation Logic

joint work with
presented by
on the

Xavier Leroy (*Collège de France & Inria*)
Paulo Emílio de Vilhena (*University of Surrey*)
25th of September, 2026

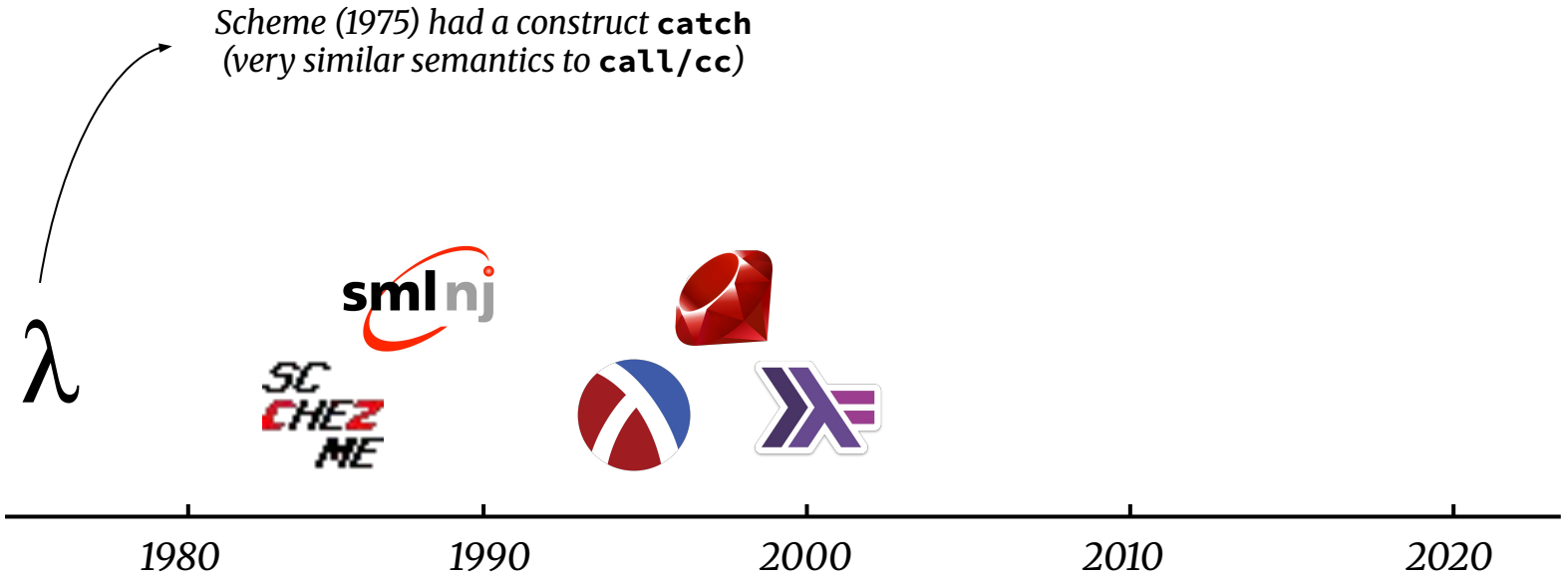
Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.



Overview

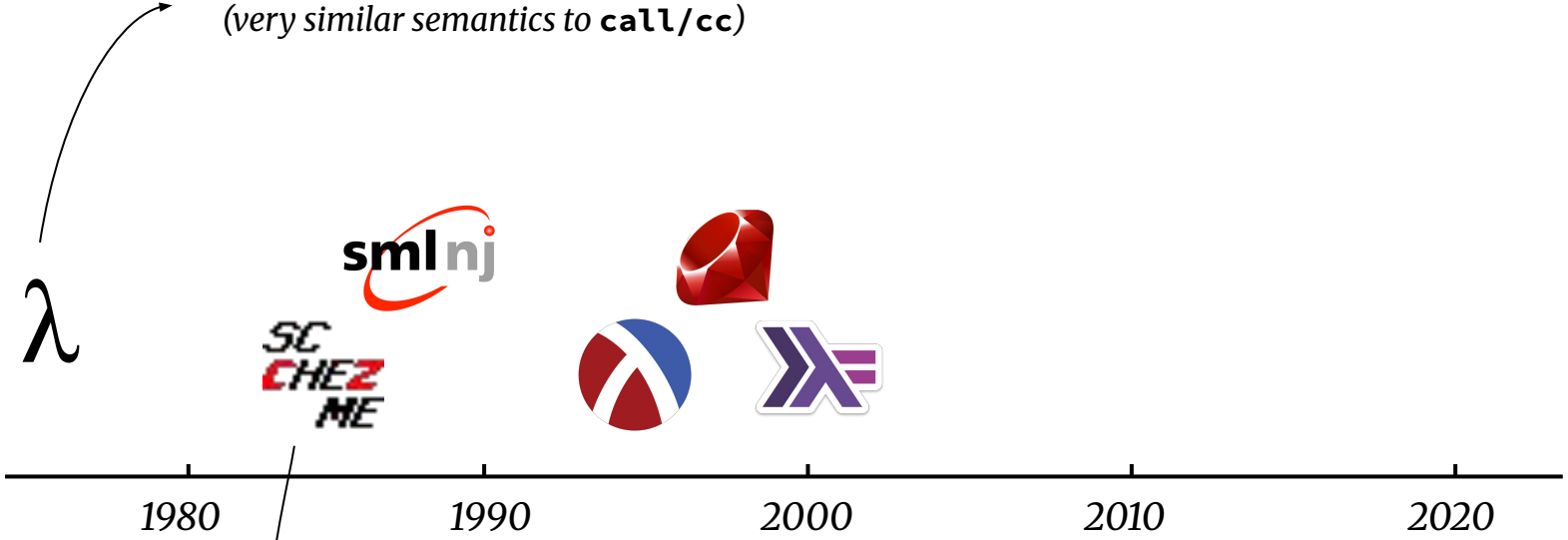
In this talk, we explore *control operators* in the **`call/cc`** *family*.



Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.

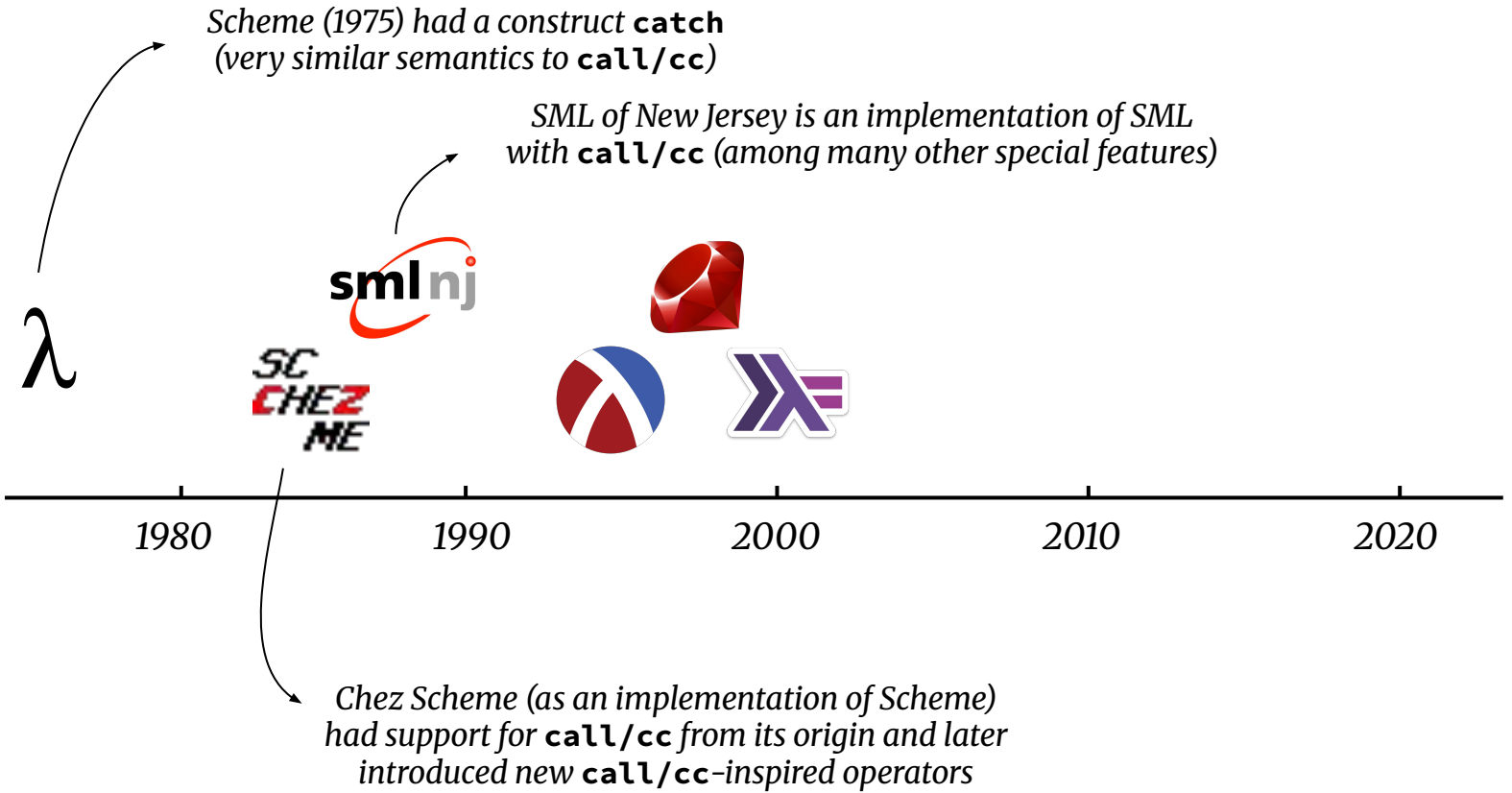
Scheme (1975) had a construct **`catch`**
(very similar semantics to **`call/cc`***)*



Chez Scheme (as an implementation of Scheme)
had support for **`call/cc`** *from its origin and later*
introduced new **`call/cc`***-inspired operators*

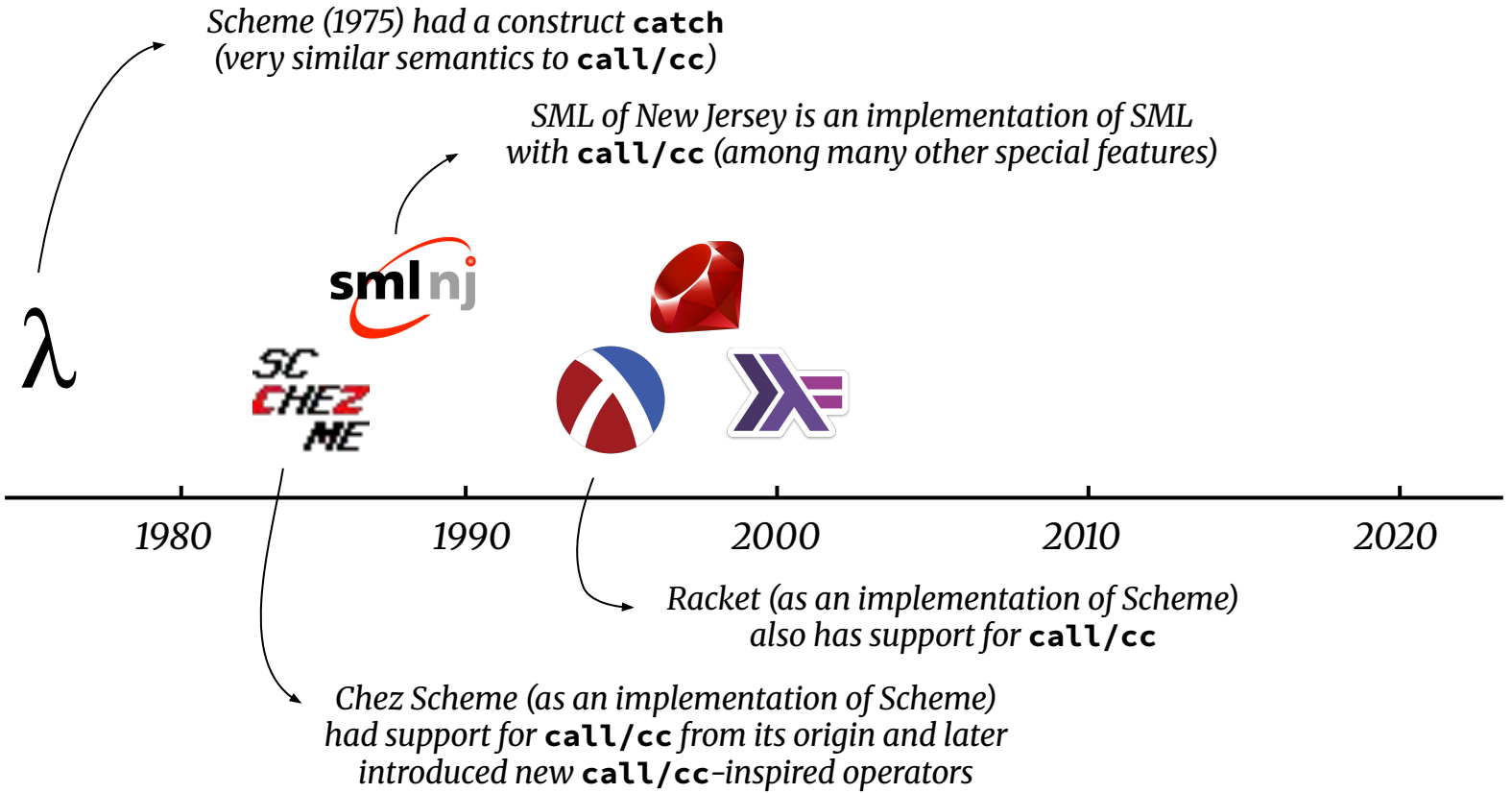
Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.



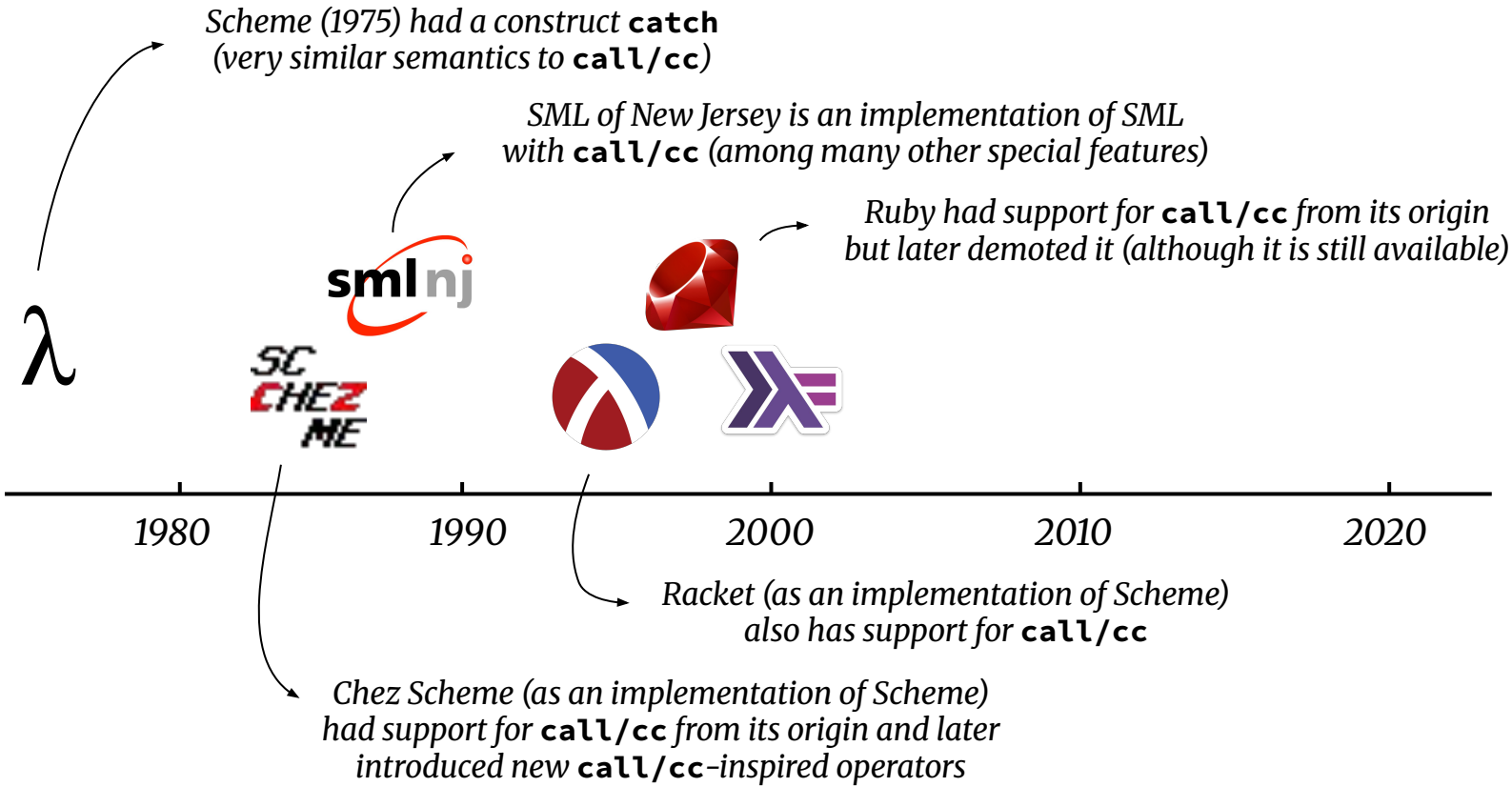
Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.



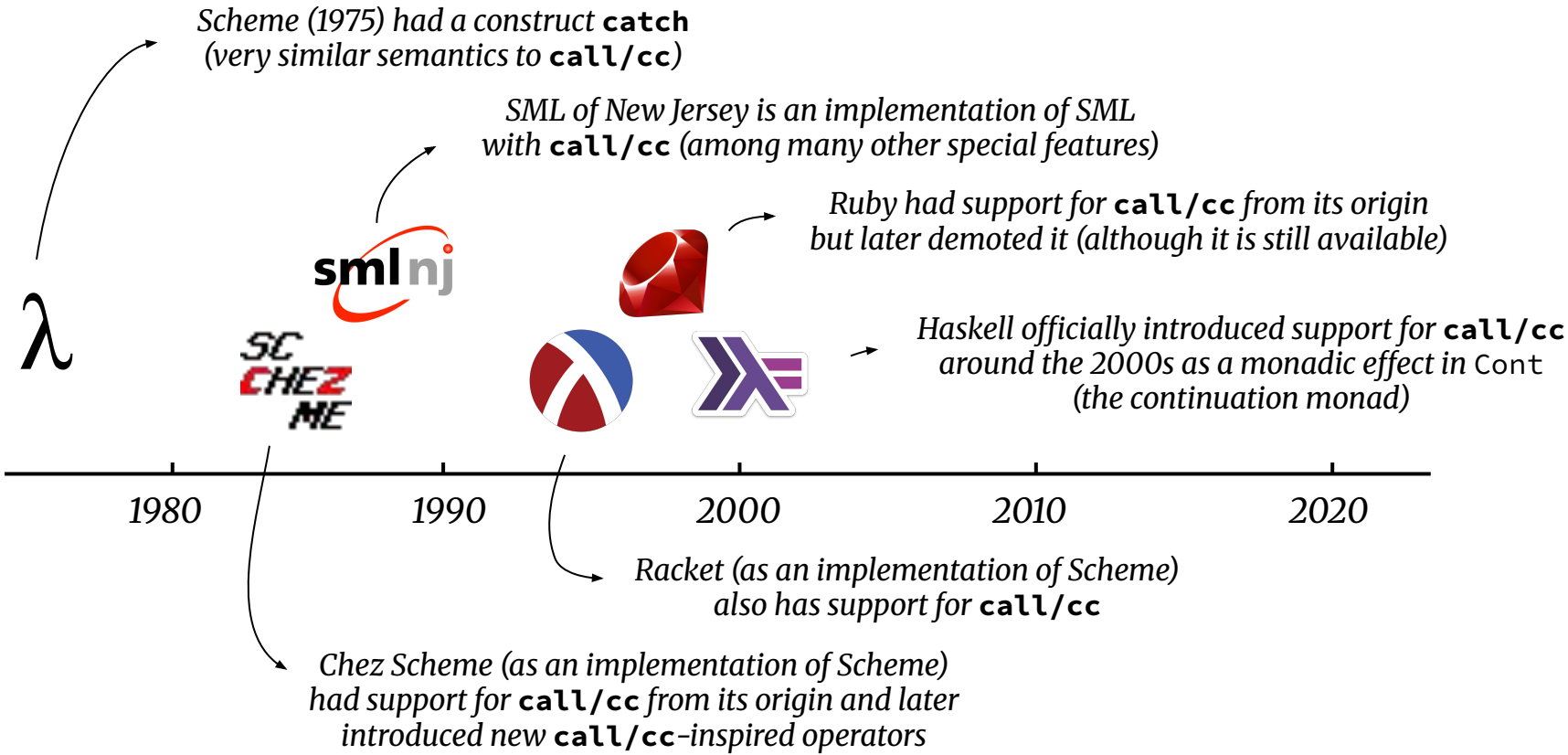
Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.



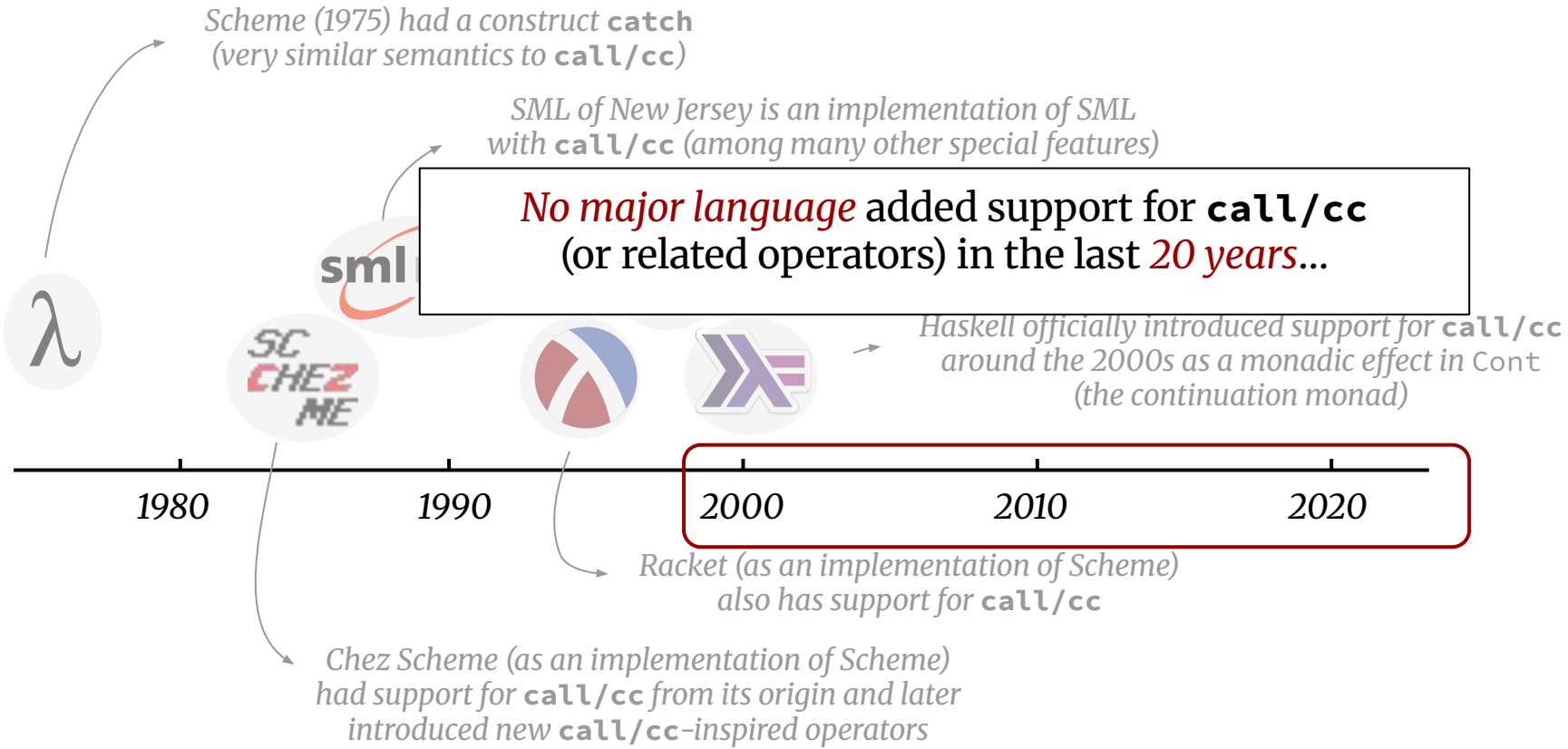
Overview

In this talk, we explore *control operators* in the **`call/cc`** *family*.

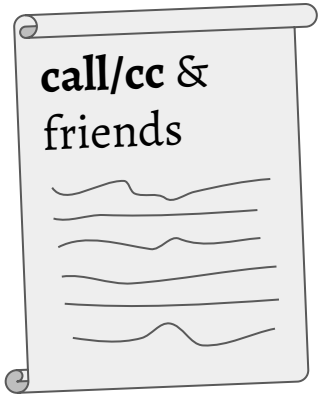


Overview

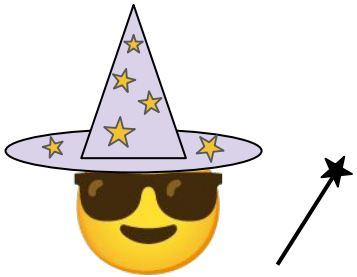
In this talk, we explore *control operators* in the **`call/cc`** *family*.




call/cc

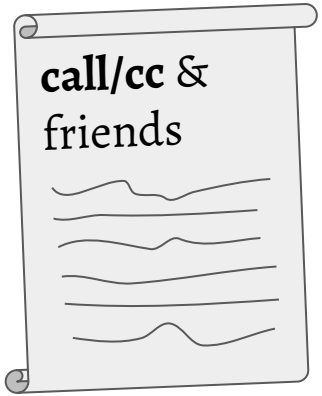


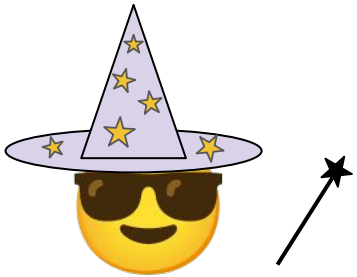
study **`call/cc`**



through the perspective of
separation logic

in search for
insight





study **`call/cc`** through the perspective of ***separation logic*** in search for ***insight***

1. Programming perspective on **`call/cc`** and continuations (in a ***fictional "OCaml-like language"***).
2. Logical perspective on **`call/cc`** and continuations.
3. Alternative semantics for **`call/cc`** and its logical principles.

1. Continuations: programming perspective

What is `call/cc`?

Continuations

The *continuation of an expression* `e` is what remains to be computed *after* `e` terminates.

Continuations

The *continuation of an expression* `e` is what remains to be computed *after* `e` terminates.

```
let x = e1 in e2
```

Examples.

- **Continuation of** e_1**.** What remains to be computed after e_1 terminates with y_1 is *"to bind* x *to* y_1 *and to evaluate* e_2*"*
- **Continuation of** e_2**.** What remains to be computed after e_2 terminates with y_2 is *"to return* y_2*"*

Continuations

The *continuation of an expression* e is what remains to be computed *after* e terminates.

```
let x = e1 in e2
```

Examples.

- **Continuation of** e_1**.** What remains to be computed after e_1 terminates with y_1 is *"to bind* x *to* y_1 *and to evaluate* e_2*"*
- **Continuation of** e_2**.** What remains to be computed after e_2 terminates with y_2 is *"to return* y_2*"*

Continuations

The *continuation of an expression* e is what remains to be computed *after* e terminates.

```
let x = e1 in e2
```

Examples.

- **Continuation of** e_1**.** What remains to be computed after e_1 terminates with y_1 is *"to bind x to y_1 and to evaluate e_2"*
- **Continuation of** e_2**.** What remains to be computed after e_2 terminates with y_2 is *"to return y_2"*

Continuations

The *continuation of an expression* e is what remains to be computed *after* e terminates.

```
let x = e1 in e2
```

Examples.

- **Continuation of** e_1. What remains to be computed after e_1 terminates with y_1 is *"to bind* x *to* y_1 *and to evaluate* e_2*"*
- **Continuation of** e_2. What remains to be computed after e_2 terminates with y_2 is *"to return* y_2*"*

Continuations

The *continuation of an expression* e is what remains to be computed *after* e terminates.

```
let x = e1 in e2
```

Examples.

- **Continuation of** e_1. What remains to be computed after e_1 terminates with y_1 is *"to bind* x *to* y_1 *and to evaluate* e_2*"*
- **Continuation of** e_2. What remains to be computed after e_2 terminates with y_2 is *"to return* y_2*"*

Continuations - Undelimited

The *continuation of an expression* e is what remains to be computed *after* e terminates.

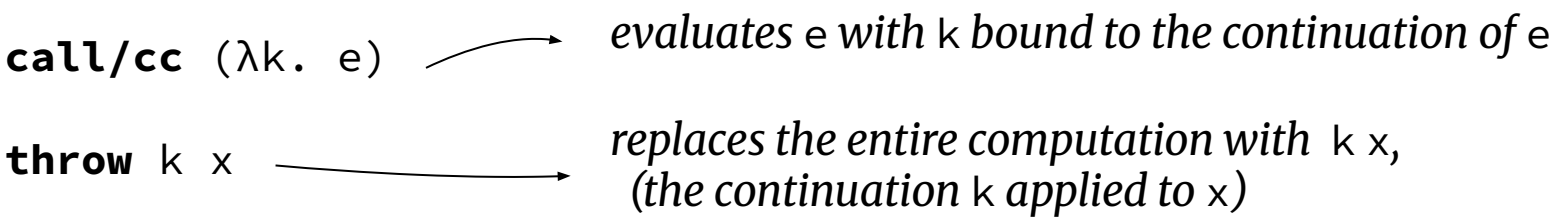
```
let x = e1 in e2    (* the complete program *)
```

Examples.

- **Continuation of** e_1. What remains to be computed after e_1 terminates with y_1 is *"to bind* x *to* y_1 *and to evaluate* e_2*"*
- **Continuation of** e_2. What remains to be computed after e_2 terminates with y_2 is *"to return* y_2*"*

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:



Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
```

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
```

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
```

⇝

```
let x = throw k 0 in x + 1
```

`k =` `λy. let x = y in x + 1`

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

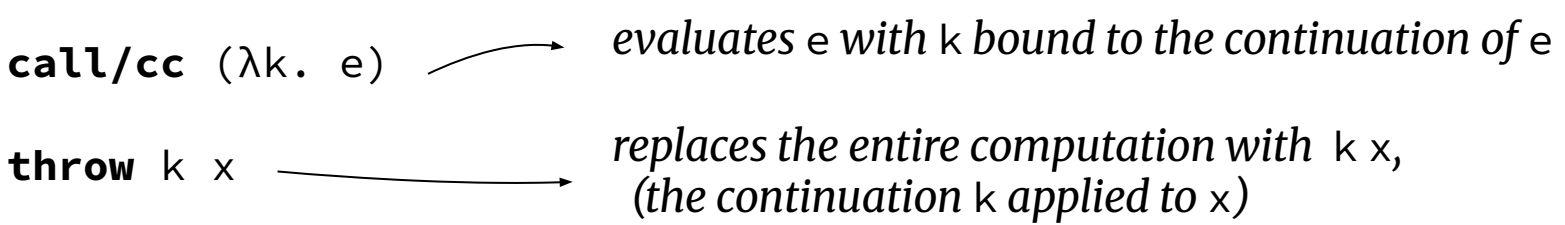
`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1          k = λy. let x = y in x + 1
```

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:



Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1          k = λy. let x = y in x + 1
  ⇝
k 0
```

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1
  ⇝
k 0
```

`k =` `λy. let x = y in x + 1`

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1
  ⇝
k 0
  ⇝
let x = 0 in x + 1
```

k = `λy. let x = y in x + 1`

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1          k = λy. let x = y in x + 1
  ⇝
k 0
  ⇝
let x = 0 in x + 1
```

Programming with continuations

The **`call/cc`** construct provides an interface for *programming with continuations*:

`call/cc` `(λk. e)` → *evaluates* `e` *with* `k` *bound to the continuation of* `e`

`throw` `k x` → *replaces the entire computation with* `k x`, *(the continuation* `k` *applied to* `x`*)*

Example.

```
let x = call/cc (λk. throw k 0) in x + 1
  ⇝
let x = throw k 0 in x + 1          k = λy. let x = y in x + 1
  ⇝
k 0
  ⇝
let x = 0 in x + 1
  ⇝
1
```

Intuition

continuation of `e`	=	*program point of* `e`
`call/cc` `(λk. e)`	=	*get* `e`*'s program point and bind it to* `k`
`throw` `k x`	=	*go to program point* `k`

> "**`call/cc`** *is the* **`goto`** *of lambda calculus"* — *Xavier Leroy*

Pros & Cons of **`call/cc`**

Powerful

- Cooperative concurrency
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- Backtracking
  - ↘ *save checkpoints via* **`call/cc`**
    retry via **`throw`**

Pros & Cons of `call/cc`

Powerful

- Cooperative concurrency
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- Backtracking
  - ↘ *save checkpoints via* **`call/cc`**
    retry via **`throw`**

Inefficient

↘ *Under the hood,*
`call/cc` *copies the control stack*

Pros & Cons of `call/cc`

Powerful

- Cooperative concurrency
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- Backtracking
  - ↘ *save checkpoints via* **`call/cc`**
    retry via **`throw`**

Inefficient

↘ *Under the hood,*
`call/cc` *copies the control stack*

Error-prone

↘ *Code that frees up resources might be part of a continuation that is never called.*
** Same problem occurs in any other interface for continuations...*

Pros & Cons of `call/cc`

Powerful

- Cooperative concurrency
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- Backtracking
  - ↘ *save checkpoints via* **`call/cc`**
    retry via **`throw`**

Inefficient

↘ *Under the hood,*
`call/cc` *copies the control stack*

Error-prone

↘ *Code that frees up resources might be part of a continuation that is never called.*
** Same problem occurs in any other interface for continuations...*

Difficult to ***reason about***

2. Continuations: logical perspective

A logical perspective

The *difficulty* in *reasoning about* **`call/cc`** & **`throw`**
can be understood from the perspective of *separation logic*.

Separation logic = Program logic + Assertions that express ownership

$\{P\}\ \mathtt{e}\ \{y.\ Q\}$

$\mathtt{l} \mapsto \mathtt{v},\ A * B,\ A \mathrel{-\!\!*} B, \ldots$

A logical perspective

We are going to study the effects of **`call/cc`** to program reasoning via two key *rules*: the *bind rule* and the *frame rule*.

Bind rule

The *bind rule* is an essential rule of program logics that allows *verification by parts*.

Bind

$$\frac{\{P\}\ e_1\ \{x.\ Q\} \qquad \forall x.\ \{Q\}\ e_2\ \{R\}}{\{P\}\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \{R\}}$$

Bind rule

The *bind rule* is an essential rule of program logics that allows *verification by parts*.

Bind

$$\frac{\{P\}\ e_1\ \{x.\ Q\} \qquad \forall x.\ \{Q\}\ e_2\ \{R\}}{\{P\}\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \{R\}}$$

Sequence (Derived*)

$$\frac{\{P\}\ e_1\ \{\_.\ Q\} \qquad \{Q\}\ e_2\ \{R\}}{\{P\}\ e_1; e_2\ \{R\}}$$

* *By exploiting the property that* $e_1; e_2$ *has the same semantics as* $\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$.

Frame rule

The *frame rule* is a crucial rule of separation logic that allows *local reasoning about state*.

Frame

$$\frac{\{P\}\ \mathtt{e}\ \{Q\}}{\{P * R\}\ \mathtt{e}\ \{y.\ Q\ y * R\}}$$

"Any resource R (disjoint from P) that is available now is also available after `e` *returns."*

Bind + frame + `call/cc` = unsound

What is the *final value* of `r`?

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

Bind + frame + `call/cc` = unsound

What is the *final value* of `r`?

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

With frame and bind rules, we can prove the final value of `r` is `1`:

- The *bind rule* lets us prove a specification for `f` *"out of context"*:

  ```
  {True} f () {_. True}
  ```

- The *frame rule* then lets us prove `f` *"preserves"* `r`:

  ```
  {r ↦ 1} f () {_. r ↦ 1}
  ```

Bind + frame + `call/cc` = unsound

What is the *final value* of `r`?

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

With frame and bind rules, we can prove the final value of `r` is `1`:

- The *bind rule* lets us prove a specification for `f` *"out of context"*:

  ```
  {True} f () {_. True}
  ```

- The *frame rule* then lets us prove `f` *"preserves"* `r`:

  ```
  {r ↦ 1} f () {_. r ↦ 1}
  ```

If we evaluate this program, however, we can see the final value of `r` is `2`!

What went wrong?

The continuation `λy. let f = y in incr r; g (); !r` is run *twice*!

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

What went wrong?

The continuation `λy. let f = y in incr r; g (); !r` is run *twice*!

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

The *first time* after `call/cc` returns and `f` is bound to `λ_. throw k (λ_. ())`:

$$\left(\begin{array}{l}\texttt{λy.}\\ \quad\texttt{let f = y in}\\ \quad\texttt{incr r; f (); !r}\end{array}\right)\texttt{(λ\_. throw k (λ\_. ()))}$$

What went wrong?

The continuation `λy. let f = y in incr r; g (); !r` is run *twice*!

```
let r = ref 0 in
let f = call/cc (λk. λ_. throw k (λ_. ())) in
incr r; f (); !r
```

The *first time* after **`call/cc`** returns and `f` is bound to `λ_. throw k (λ_. ())`:

```
(λy.
   let f = y in
   incr r; f (); !r) (λ_. throw k (λ_. ()))
```

The *second time* after **`throw`**, when `f` is bound to `λ_. ()`:

```
(λy.
   let f = y in
   incr r; f (); !r) (λ_. ())
```

3. Insights

One-shot continuations

The previous example suggests that continuations should be *one-shot*:
(i) a *continuation* should be *invoked at most once* ***and***
(ii) *normal termination* of `call/cc` should *count as one invocation*.

One-shot continuations

The previous example suggests that continuations should be *one-shot*:
(i) a *continuation* should be *invoked at most once* ***and***
(ii) *normal termination* of **`call/cc`** should *count as one invocation*.

In fact, Chez Scheme has a **`call/1cc`** construct with precisely this *one-shot restriction*!*

* *Although, in Chez Scheme, a one-shot continuation can be automatically promoted to a multi-shot continuation due to concerns related to the interaction between* **`call/1cc`** *and* **`call/cc`**.

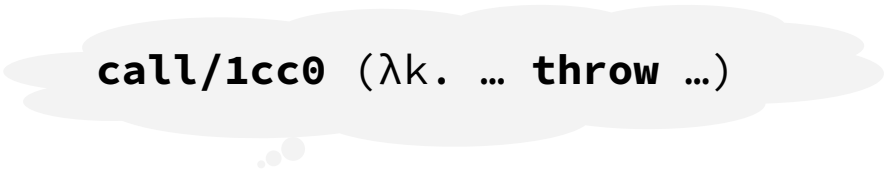
One-shot continuations

We go one step further and introduce **`call/1cc0`**,
where *normal termination* of the body of **`call/1cc0`** is *disallowed*.

One-shot continuations

We go one step further and introduce **`call/1cc0`**,
where *normal termination* of the body of **`call/1cc0`** is *disallowed*.

Why? Because it provides a *simpler rule* to follow:
"In the body of **`call/1cc0`***, whichever path we take must end with a* **`throw`***!"*

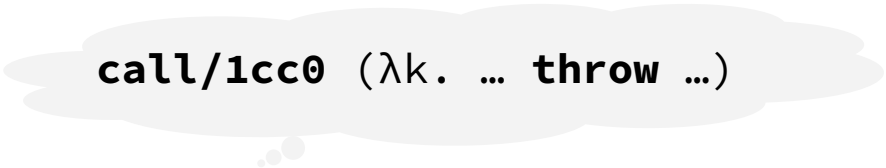


(Intuition)

One-shot continuations

We go one step further and introduce **`call/1cc0`**,
where *normal termination* of the body of **`call/1cc0`** is *disallowed*.

Why? Because it provides a *simpler rule* to follow:
"In the body of **`call/1cc0`***, whichever path we take must end with a* **`throw`***!"*



(Intuition)

Moreover, we can implement **`call/1cc`** on top of **`call/1cc0`**:

```
call/1cc (λk. e)  ≜  call/1cc0 (λk. let y = e in throw k y)
```

Pros & Cons of `call/1cc0`

Powerful

- Cooperative concurrency ✓
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion ✓
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators ✓
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- ~~Backtracking~~
  - ↘ ~~*save checkpoints via* **`call/cc`**~~
    ~~*retry via* **`throw`**~~

Pros & Cons of `call/1cc0`

Powerful

- Cooperative concurrency ✓
  - ↘ *threads as continuations stored in a queue*
    `yield` = **`call/cc`** + **`throw`**
- Control inversion ✓
  - ↘ *internal to external iterators*
    `iter` (`'a t -> ('a -> unit) -> unit`) *to streams*
- Python-style generators ✓
  - ↘ **`yield`** *(instead of* **`return`***)*
    `next` *(instead of calling a function)*
- ~~Backtracking~~
  - ↘ ~~*save checkpoints via* **`call/cc`**~~
    ~~*retry via* **`throw`**~~

Efficient

↘ **`call/1cc0`** *does not copy the stack*

Improved ***reasoning***

↘ *admits a logic with both frame and bind!*

+

simple reasoning rules!

4. Separation logic for one-shot continuations

Reasoning rules for `call/1cc0` & `throw`

permission to resume k

Call/1cc0

$$\frac{\forall k.\ \{P * \mathit{isCont}_1\ k\ Q\}\ e\ \{\_.\ \mathit{False}\}}{\{P\}\ \mathbf{call/1cc0}\ (\lambda k.\ e)\ \{Q\}}$$

`e` *cannot return normally*

Throw

$$\frac{}{\{Q\ y * \mathit{isCont}_1\ k\ Q\}\ \mathbf{throw}\ k\ y\ \{\_.\ \mathit{False}\}}$$

Q *was the postcondition of* `e`*,*
Now, it is the precondition of k

`isCont₁`

`isCont₁ k Q` asserts that k expects a value y such that `Q y`:
`Q` is the *precondition to resume* the continuation.

Observation 1. From the perspective of the logic,
the continuation of `e` is described *abstractly* by `e`'s postcondition `Q`,
rather than *concretely* as a programming value.

Key to the bind rule

Observation 2. If k needs `R`, and `R` is available now,
then `R` can be transferred to the continuation k:

$$R * \texttt{isCont₁}\ k\ (y.\ R * Q\ y) \mathrel{-\!\!*} \texttt{isCont₁}\ k\ Q$$

Key to the frame rule

Construction of the logic (in Iris)

Standard program-logic construction in Iris:

$$\{P\}\ e\ \{Q\} = \Box(P \mathrel{-\!\!*} WP\ e\ \{Q\})$$

a program specification is persistent

Construction of the logic (in Iris)

Standard program-logic construction in Iris:

$$\{P\}\ e\ \{Q\} = \square(P \mathrel{-\!\!*} WP\ e\ \{Q\})$$

a program specification is persistent

1. `wp e {Q}` (Iris standard weakest precondition)

   ⟶ Timany & Birkedal (2019): *frame* but *no bind*

Construction of the logic (in Iris)

Standard program-logic construction in Iris:

$$\{P\}\ e\ \{Q\} = \square(P \mathrel{-\!\!*} WP\ e\ \{Q\})$$

a program specification is persistent

1. `wp e {Q}` (Iris standard weakest precondition)

   ⟶ Timany & Birkedal (2019): *frame* but *no bind*

2. $\texttt{wp/cc}\ e\ \{Q\} = \forall k.\ (\square \forall y.\ Q\ y \mathrel{-\!\!*} \texttt{wp}\ k\ y\ \{\_.\mathit{True}\}) \mathrel{-\!\!*} \texttt{wp}\ k\ e\ \{\_.\mathit{True}\}$

   ⟶ de Vilhena (2022): *bind* but *no frame*

Construction of the logic (in Iris)

Standard program-logic construction in Iris:

$$\{P\}\ e\ \{Q\} = \square(P \mathrel{-\!\!*} WP\ e\ \{Q\})$$

a program specification is persistent

1. `wp e {Q}` (Iris standard weakest precondition)

   ⟶ Timany & Birkedal (2019): *frame* but *no bind*

2. $\texttt{wp/cc}\ e\ \{Q\} = \forall k.\ (\square \forall y.\ Q\ y \mathrel{-\!\!*} \texttt{wp}\ k\ y\ \{\_.\mathit{True}\}) \mathrel{-\!\!*} \texttt{wp}\ k\ e\ \{\_.\mathit{True}\}$

   ⟶ de Vilhena (2022): *bind* but *no frame*

3. $\texttt{wp/1cc0}\ e\ \{Q\} = \forall R.\ R \mathrel{-\!\!*} \texttt{wp/cc}\ e\ \{y.\ Q\ y * R\}$

   ⟶ de Vilhena & Leroy (2026): both *frame* and *bind*!

$$\mathit{isCont}_1\ k\ Q = \forall y.\ Q\ y \mathrel{-\!\!*} \texttt{wp/1cc0}\ k\ y\ \{\_.\ \mathit{False}\}$$

Construction of the logic (in Iris)

Standard program-logic construction in Iris:

1. wp

2. wp

3. wp

→ de Vilhena & Leroy (2026): both *frame* and *bind*!

$\mathit{isCont_1}\ k\ Q = \forall y.\ Q\ y \mathrel{-\!\!*} \mathtt{wp/1cc0}\ k\ y\ \{\_.\ \mathit{False}\}$

Connecting the logics

$$\frac{\mathtt{wp/1cc0\ e\ \{}Q\mathtt{\}}}{\mathtt{wp/cc\ e\ \{}Q\mathtt{\}}}$$

$$\frac{\forall R.\ R \mathrel{-\!\!*} \mathtt{wp/cc\ e\ \{}y.\ Q\ y * R\mathtt{\}}}{\mathtt{wp/1cc0\ e\ \{}Q\mathtt{\}}}$$

5. Application of the logic

Cooperative concurrency

```
let with_fork main = call/1cc0 (λreturn.
  (* queue of suspended threads *)
  let q = Queue.create () in

  (* resume a thread, if there is one, otherwise terminate *)
  let next () =
    if Queue.empty q then
      throw (Queue.take q) ()
    else
      throw return () (* exits if queue is empty *)
  in

  (* suspend one's own execution and resume a thread *)
  let yield () = call/1cc0 (λk. Queue.add q k; next ()) in
  let fork f = call/1cc0 (λk. Queue.add q k; f (); next ()) in

  main fork yield
)
```

Cooperative concurrency

Specification of `with_fork`:

$$\frac{\begin{array}{l}\forall I\ \mathit{fork}\ \mathit{yield}.\\ (\forall f.\ \{I\}\ f\ ()\ \{\_.\ I\} \mathrel{-\!\!*} \{I\}\ \mathit{fork}\ f\ \{\_.\ I\}) \mathrel{-\!\!*}\\ \{I\}\ \mathit{yield}\ ()\ \{\_.\ I\} \mathrel{-\!\!*}\\ \{I\}\ \texttt{main}\ \mathit{fork}\ \mathit{yield}\ \{\_.\ I\}\end{array}}{\{\mathit{True}\}\ \texttt{with\_fork main}\ \{\_.\ \mathit{True}\}}$$

Cooperative concurrency

Specification of `with_fork`:

$$\frac{\begin{array}{l}\forall I\ \mathit{fork}\ \mathit{yield}.\\ (\forall f.\ \{I\}\ f\ ()\ \{\_.\ I\} \mathrel{-\!\!*} \{I\}\ \mathit{fork}\ f\ \{\_.\ I\}) \mathrel{-\!\!*}\\ \{I\}\ \mathit{yield}\ ()\ \{\_.\ I\} \mathrel{-\!\!*}\\ \{I\}\ \texttt{main}\ \mathit{fork}\ \mathit{yield}\ \{\_.\ I\}\end{array}}{\{\mathit{True}\}\ \texttt{with\_fork main}\ \{\_.\ \mathit{True}\}}$$

The key idea in the verification is to instantiate I with a suitable definition:

$$\begin{array}{rcl} I & = & \mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True})\ * \\ & & \exists \mathit{ks}.\ \mathit{isQueue}\ q\ \mathit{ks}\ *\ \ast k \in \mathit{ks}.\ \triangleright\ \mathit{isCont}_1\ k\ (\lambda\_.\ I) \end{array}$$

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I}
  call/1cc0 (λk. Queue.add q k; f (); next ())
{_. I}
```

$$
\begin{aligned}
I \;\triangleq\;& \mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True}) \ast \\
& \exists \mathit{ks}.\ \mathit{isQueue}\ q\ \mathit{ks} \ast \mathbin{\text{✱}} k \in \mathit{ks}.\ \triangleright\ \mathit{isCont}_1\ k\ (\lambda\_.\ I)
\end{aligned}
$$

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I ∗ isCont₁ k (λ_. I)}
  Queue.add q k; f (); next ()
{_. False}
```

```
I  =  isCont₁ return (λ_. True) ∗
      ∃ks. isQueue q ks ∗ ✱k∈ks. ▷ isCont₁ k (λ_. I)
```

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I ∗ isCont₁ k (λ_. I)}
  Queue.add q k
{I}
  f ()
{I}
  next ()
{_. False}
```

```
I  =  isCont₁ return (λ_. True) ∗
      ∃ks. isQueue q ks ∗ ✱k∈ks. ▷ isCont₁ k (λ_. I)
```

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I ∗ isCont₁ k (λ_. I)}
  Queue.add q k
{I}
  f ()                                  by assumption
{I}
  next ()
{_. False}
```

$$I \;=\; \mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True}) \ast \exists \mathit{ks}.\ \mathit{isQueue}\ q\ \mathit{ks} \ast \mathbin{\ast\!\!\forall} k \in \mathit{ks}.\ \triangleright\ \mathit{isCont}_1\ k\ (\lambda\_.\ I)$$

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I ∗ isCont₁ k (λ_. I)}
  Queue.add q k
{I}
  f ()
{I}
  next ()
{_. False}
```

adding k to q preserves the queue invariant

$$I \;=\; \mathit{isCont}_1\ \mathit{return}\ (\lambda\_.\ \mathit{True}) \ast \exists \mathit{ks}.\ \mathit{isQueue}\ q\ \mathit{ks} \ast \mathbin{\text{✱}} k \in \mathit{ks}.\ \triangleright\ \mathit{isCont}_1\ k\ (\lambda\_.\ I)$$

Cooperative concurrency

Verification of `fork`:

```
{I} f () {_. I}
  -∗
{I ∗ isCont₁ k (λ_. I)}
  Queue.add q k
{I}
  f ()
{I}
  next ()
{_. False}
```

if q *is empty, then use* `isCont₁` *return* `(λ_. True)`
otherwise take k *from* q *such that* `isCont₁ k (λ_. I)`

$$I \;=\; \texttt{isCont}_1\ \textit{return}\ (\lambda\_.\ \texttt{True}) \;*\; \exists ks.\ \texttt{isQueue}\ q\ ks \;*\; \textbf{✱} k \in ks.\ \triangleright\ \texttt{isCont}_1\ k\ (\lambda\_.\ I)$$

6. Conclusion

Conclusion

Programming with continuations

- *Continuations* are a *powerful* programming concept.
- `call/cc` offers a flexible interface for continuations, but at the cost of *efficiency* and *ease of reasoning*.

Perspective of logics for `call/cc`

- From the perspective of separation logic, the *difficulty in reasoning* about `call/cc` is justified by the *absence* of either the *frame rule* or the *bind rule*.
- The key *counterexample* to both frame and bind rules relies crucially on continuations being used *more than once*.

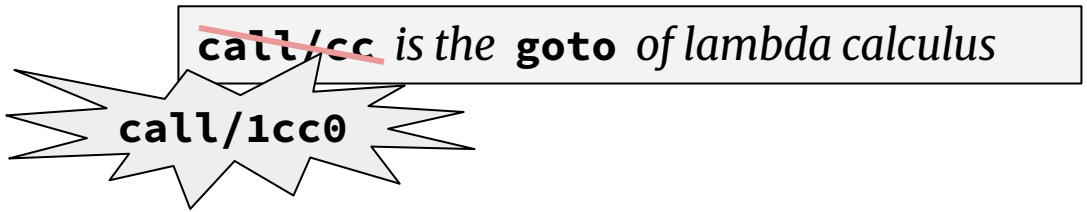
One-shot continuations

- Incorporating a *one-shot restriction* improves *performance*, *preserves* most of the interesting *applications* of `call/cc`, and admits a *separation logic* with both *bind* and *frame rules*!

Conclusion

`call/cc` *is the* **`goto`** *of lambda calculus*

Conclusion



Both **`call/1cc0`** and **`goto`**
(1) are *expressive*,
(2) can be *efficiently implemented*, and
(3) admit *simple reasoning rules*.

Questions