Static analysis in Flambda 2

Pierre Chambart, Nathanaëlle Courant
and **Vincent Laviron** (OCamlPro)

Mark Shinwell and Leo White (Jane Street)

Cambium Seminar, Inria Paris, 2023-08-28

Introduction

- We will describe the static analysis used during the middle of the three passes of Flambda 2, the simplifier
- This was originally conceived as a type system for Flambda 2 terms
- However it looks like an abstract domain, in the sense of abstract interpretation
- The static analysis is written in OCaml (about 15kloc)
  - All data structures are immutable
- It does not depend on the actual term language itself, only a few basic types

Properties

We want to track the following properties:

- Numeric constants for constant propagation
- Block structures for simplifying accesses and removing allocation
- Functions and closures for enabling direct calls and inlining
- Variants (mixing integers and blocks) for dead code removal

Some of these properties are also used for unboxing.

The Flambda 2 simplifier communicates with the static analysis using a well-defined interface.

Numeric properties

We have several kinds of (unboxed) numbers:

- Floating-point numbers
- 64-bit, 32-bit and machine width integers
- Immediates (numbers with one fewer bit than word-size integers)

For each of them we use a domain of small finite sets.

Block structures

This is the domain used for tracking tuples, record values and module blocks.

- Only fixed-size blocks
- Arrays are handled separately
- No support for the contents of mutable records

Block structures: shapes

- We use a domain of shapes
- A shape can be a full shape, with a list of shapes for each of the fields...
- or a partial shape, with a partial map of index to known shapes, the size being at least that of the largest index

Full shapes arise from propagating the annotations of the OCaml type checker, or directly from allocation primitives.

Partial shapes are created from field accesses when the block doesn't have a known shape.

Relational information: examples

We would like to track relations between different fields:

```
let block_field_to_vars x y =
  let block = x, y in
  fst block + snd block

let field_to_field x =
  let block = x, x in
  match block with
  | A, A -> case_A ()
  | B, B -> case_B ()
  | A, B | B, A -> should_be_eliminated ()
```

Relational information

- We handle these relations using a domain tracking aliasing
- Basic elements are (possibly empty) sequences of projections from variables
- Equivalence classes are tracked using an union-find structure
- We are only tracking *must-alias* relations, not *may-alias* or *must-not-alias*

The actual implementation looks slightly different, we will come to this later

Functions

- For functions (which are all closed), we track their code if they are eligible for inlining
- We also track a summary capturing the relation between the inputs and outputs
  - Useful for reasoning about functors without inlining
- The summary is represented by an *environment extension*, which will be described later

Closures

For closures, we reuse part of the logic used for blocks (including for projections), but with a few differences:

- Indices are not integers, but symbolic labels called *value slots*
- The function associated with a given closure is also stored
- For handling mutually-recursive functions, an extra set of indices (*function slots*) is used to track accesses from one closure to another

In OCaml terms at runtime:

- Function slots $\approx$ code pointer sections of a closure block
- Value slots $\approx$ elements of a closure block's environment

Variants and disjunctions: Blocks

Variants are handled by splitting on their constructor, introducing disjunctions.

- For blocks, we split depending on their tag
- This also applies to block projections
- We also support blocks with unknown tags, which adds a little complexity

Variants and disjunctions: Constants

- Constant constructors are treated as immediates, so handled by the finite sets domain
- A top-level disjunction splits between the constant and block cases
- The two cases have an empty intersection, so domain operators act element-wise

More complicated disjunctions

- Note that although it is possible to have values of incompatible types flowing into the same variable (typically using GADTs), we do not support any other kind of disjunction.
- Such cases are represented with a Top value.

Propagation

A typical example:

```
let f x =
  let y = fst x in
  match y with
  | A -> assert (fst x == A)
  | _ -> ()
```

- Starting info: $\texttt{x} : (\top, \top)$ (from the Lambda value kinds)
- After binding $\texttt{y}$: alias relation $\texttt{x}[0] = \texttt{y}$
- In the first branch: $\texttt{y} : \texttt{A}$
- We want to infer $\texttt{x} : (\texttt{A}, \top)$
- This can be done using a *reduced product*

Reduced product

A *reduced product* is a standard way to combine two domains in abstract interpretation.

- The reduction operator can be defined as

$$\mathit{reduce}(A_l, A_r) = (\alpha_l(\gamma_l(A_l) \sqcap \gamma_r(A_r)), \alpha_r(\gamma_l(A_l) \sqcap \gamma_r(A_r)))$$

- In less formal terms, reduction **merges information from both sides** and **propagates any new constraints** back into their respective domains when possible
- Reduction is expensive in general
- It can be performed on demand, if the consumer knows where it is most useful (not the case here)

Implementation

We do not implement a full reduced product. Instead, we have:

- A domain of **aliases for variables** (no projections)
- A domain for **approximations** that can be either a variable or a concrete description
- Concrete descriptions for structures have approximations as sub-elements, allowing to encode projections
- As an invariant, each set of aliases has a single canonical element, and only this canonical element is allowed to have a concrete description
- Numerical constants are considered as predefined variables. Those are always the canonical alias

Revisiting the example

```
let f x =
  let y = fst x in
  match y with
  | A -> assert (fst x == A)
  | _ -> ()
```

- Starting info: $\texttt{x} : (\top, \top)$ (from the types)
- After binding `y`: $\texttt{x} : (=\texttt{y}, \top);\ \texttt{y} : \top$
- In the first branch: $\texttt{x} : (=\texttt{y}, \top);\ \texttt{y} : \texttt{A}$

Environments

- An environment is a **map from program names to approximations**, plus a **set of alias constraints**
- Environments can be enriched by either adding new variables with their approximations...
- ...or introducing constraints on one or more existing variables
- In the second case, this triggers a **meet**

More on “meet” in just a moment.

Levels: dual representation of environments

- For performance reasons, we also keep the **history of constraints** that were combined to produce the current environment
- At some synchronization points (decided by the consumer, in our case the simplification code), we keep a backup of the compact state at that point
- With that, we can easily extract and replay parts of the history.
- This is useful for join points, where we only have to join the part of the history that is not in common; and for extracting summaries out of the environments at the end of functions. More on this later

Meet

- One of the two main operations (the other being join)
- Meet is used to add constraints
  - In conditional branches
  - In some primitives (typically projections)
- Meeting two types in a given environment will return:
  - **A type** that subsumes both inputs
  - **An updated environment** that reflects the additional constraints
- Meeting two environments isn't very useful, and is not implemented

Meet examples

The meet algorithm takes as input an environment, and two approximations. It returns an approximation that subsumes both of the inputs, and an updated environment.

Examples:

$\{\mathtt{x} : (\mathtt{A}, \top);\ \mathtt{y} : \top\}$ $\quad =\mathtt{x} \sqcap (=\mathtt{y}, \mathtt{B})$

$\rightarrow$

$\{\mathtt{x} : (=\mathtt{y}, \mathtt{B});\ \mathtt{y} : \mathtt{A}\}$ $\quad =\mathtt{x}$

$\{\mathtt{x} : (=\mathtt{y}, \top);\ \mathtt{y} : \top;\ \mathtt{z} : \mathtt{A}\}$ $\quad =\mathtt{x} \sqcap (=\mathtt{z}, \top)$

$\rightarrow$

$\{\mathtt{x} : (=\mathtt{y}, \top);\ \mathtt{y} : \mathtt{A};\ \mathtt{z} : =\mathtt{y} \mid \mathit{aliases} : \{\mathtt{y}, \mathtt{z}\}\}$ $\quad =\mathtt{x}$

Meet algorithm part 1

Our current meet algorithm proceeds as follows:

- If any of the inputs is a variable, replace it with its canonical alias
- Fetch the corresponding concrete descriptions
- Do the relevant meet operation
  - This may call the generic meet algorithm recursively
  - Protection against cycles is needed because of the recursion inherent in closures
- This returns a new approximation and an updated environment

Meet algorithm part 2

- If exactly one of the inputs was a variable, replace the approximation for its canonical element by the new result in the updated environment. The result approximation is an alias to this variable
- If both inputs were variables, choose one to become the new canonical element and merge their equivalence classes
- The approximation for the non-canonical one is replaced in the result environment by an alias to the canonical one
- The result approximation is an alias to the canonical element

Environment extensions

- Environment extensions represent **sets of constraints** on a set of variables, expressed in the shape of a map from variables to approximations.
- The set of constrained variables is all of the keys of the map plus all free variables in the approximations.

Example:

```
let f x y = (x, y)
```

Summary: $\{\texttt{result} : (=x, =y)\}$

The summary can then be associated to the function in the environment.

Existentially-quantified variables

The environment can handle existentially-quantified variables.

- This allows the encoding of more constraints:
  $\{\mathtt{x} : (=\mathtt{y}, =\mathtt{y});\ \exists \mathtt{y} : \top\}$
- In practice, it is mostly used for variables no longer in scope
- Adapting environments to more restricted scopes (e.g. going under a lambda) becomes cheap
- This is done using a *lock* in the environment
- Environment extensions can also introduce existential variables

Extensions under disjunctions

- All disjunctions can have **extensions associated to individual cases**
- During meet, these extensions can be temporarily added while under the corresponding case
- If only one case remains possible, then the extension gets lifted to the result environment

These extensions are not inferred during join (too expensive).

Instead, during meet when several disjunction cases remain, extensions are extracted from the relevant result environments and stored with the cases.

This mostly matters during variant unboxing.

Extensions under disjunctions: example

Unboxing of variants relies on this feature for precise tracking of unboxed parameters to their contents.

Example:

```
let f cond y z =
  let r = if cond then A y else B z in ...

let f_unboxed cond y z =
  let tag, arg = if cond then A, y else B, z in ...
```

$\{\texttt{arg} : \top;\ \texttt{tag} : (\texttt{A}\{\texttt{arg} : =\texttt{y}\} \mid \texttt{B}\{\texttt{arg} : =\texttt{z}\})\}$

Join

- The other of the two main operations, alongside meet
- Join is used mostly when **merging branches** (i.e. at continuation handlers)
- The usual join takes two environments, and returns an environment that contains only properties that hold on both branches
- In practice our use of levels provides us with a common ancestor, which helps in keeping the complexity manageable
- Joining two types in a given environment is also implemented

Join algorithm

Join operates on two environments.

- **Find a common ancestor**. In practice, the consumer specifies it, but it would be possible to infer it
- Use the levels to efficiently factor each input into an **extension over the common ancestor**
- **Existentially quantify** all variables defined in the extensions. Variables occurring only on one side get introduced on the other side with a special approximation that behaves like a Bottom element
- **Join the individual approximations** associated to each variable
- **Add the resulting extension** to the ancestor environment

Join on approximations

The join on approximations works like this:

- Joining two concrete descriptions returns a concrete description, recursively calling join if needed
- **If only one side is a variable alias:** expand this alias in the corresponding input environment and join the concrete descriptions
- **If both sides are variable aliases:** if they share a common alias pick one of them as the return approximation. Otherwise expand the aliases in their respective environments and join the concrete descriptions

Derived numbers

- Programs can extract the *tag* of a given block and manipulate it as an integer
  - This often happens in code produced by the pattern matching compiler
- This introduces a **relation between numbers and blocks**
- This is currently handled by replacing the domain for immediates by either a concrete set or a relational constraint linking the block
- This works well for propagating information from the number to the block, not well for the reverse direction
- There is a similar problem and solution for the boolean identifying whether a value is a block or an integer

Derived numbers: alternatives

- A reduced product with a domain tracking these relations would work
- Another possibility is to consider finite sets as generic disjunctions and allow environment extensions on them
- The latter approach also opens possibilities for tracking other relations (equality, comparison)

Fixpoints and widening

- The domain was not designed for use in fixpoint computations
- There are possible implementations for a widening operator
- Comparison (inclusion check) would be particularly tricky to implement

Interface to the simplifier

The static analysis has a relatively small interface to the Flambda 2 simplifier. It provides the following operations on environments:

- **Creation** and **(de)serialization**
- **Updating**:
  - Adding variables
  - Meet to add any kind of constraint
  - Join to merge branches
  - Variable removal can be done independently (usually done during join)
- **Querying**:
  - Meet can be used for queries (looking at the resulting approximation, e.g. “is this a block?”)
  - Faster, specialised queries are also supported (less precise)

Example: Sum of squares using streams

We will see how the analysis helps us optimize this program:

```
let square x = x * x

let ints lo hi =
  unfold (fun i -> if i > hi then Empty else
      Cons (i, i + 1)) lo

let sum s = fold_left (+) 0 s

let foo () = sum (map square (ints 0 11))
```

This uses a minimal version of the `Sequence` module from the `Base` library. (We will talk about `Seq` later.)

Example: Minimal Base.Sequence

```
type ('a, 's) node = Empty | Cons of 'a * 's
type _ t = State :
  's * ('s -> ('a, 's) node)) -> 'a t

let rec fold_left f acc (State (s, next)) =
  match next s with
  | Empty -> acc
  | Cons (x, s') ->
    fold_left f (f acc x) (State (s', next))

let map f (State (s, next)) =
  State (s, fun s ->
    match next s with
    | Empty -> Empty
    | Cons (x, s') -> Cons (f x, s'))

let unfold f acc = State (acc, f)
```

Running example

```
let s1 = ints 0 11 in
let s2 = map square s1 in
sum s2
```

Environment:
`map`, `sum`, etc. omitted.

Running example

```
let f1 =
 ints.anon {hi=11}
in
let s1 = unfold f1 0 in
let s2 = map square s1 in
sum s2
```

Environment:

$$\texttt{f1}: \quad [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}]$$

Running example

```
let f1 =
 ints.anon {hi=11}
in
let s1 = State (0, f1) in
let s2 = map square s1 in
sum s2
```

Environment:

$$\texttt{f1}: \quad [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}]$$
$$\texttt{s1}: \quad (0, =\texttt{f1})$$

Running example

```
let f1 =
 ints.anon {hi=11}
in
let s1 = State (0, f1) in
let s2 =
 let State(s, next) = s1 in
 let f2 = map.anon
  {next=next,f=square}
 in
 State (s, f2)
in
sum s2
```

Environment:

f1:	$[\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}]$
s1:	$(0, =\texttt{f1})$
s:	0
next:	$=\texttt{f1}$

Running example

```
let f1 = ints.anon {hi=11} in
let s1 = State (0, f1) in
let f2 = map.anon
 {next=f1,f=square}
in
let s2 = State (0, f2) in
sum s2
```

Environment:

```
f1:  [ints.anon{hi ↦ 11}]
s1:  (0, =f1)
f2:  [map.anon
       {next ↦ =f1,
        f ↦ =square}]
s2:  (0, =f2)
```

Running example

```
let f1 = ints.anon {hi=11} in
let s1 = State (0, f1) in
let f2 = map.anon
 {next=f1,f=square}
in
let s2 = State (0, f2) in
fold_left (+) 0 s2
```

Environment:

f1:	[ints.anon{hi ↦ 11}]
s1:	(0, =f1)
f2:	[map.anon {next ↦ =f1, f ↦ =square}]
s2:	(0, =f2)
s:	0
next:	=f2

Running example

```
let f1 = ints.anon {hi=11} in
let s1 = State (0, f1) in
let f2 = map.anon
 {next=f1,f=square}
in
let s2 = State (0, f2) in
let State (s, next) = s2 in
loop (s=s,acc=0) {
 match next s with
 | Empty -> acc
 | Cons (x, s') ->
   let acc' = (+) acc x in
   loop (s=s', acc=acc')
}
```

Environment:

$$
\begin{array}{ll}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, =\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2}: & (0, =\texttt{f2})
\end{array}
$$

Running example

```
let f1 = ints.anon {hi=11} in
let s1 = State (0, f1) in
let f2 = map.anon {next=f1,
    f=square} in
let s2 = State (0, f2) in
loop (s=0,acc=0) {
 match f2 s with
 | Empty -> acc
 | Cons (x, s') ->
   let acc' = acc + x in
   loop (s=s', acc=acc')
}
```

We will now only consider the contents of the loop

Environment:

```
f1:  [ints.anon{hi ↦ 11}]
s1:  (0, =f1)
f2:  [map.anon
       {next ↦ =f1,
        f ↦ =square}]
s2:  (0, =f2)
```

Running example

```
match f2 s with
| Empty -> acc
| Cons (x, s') ->
  let acc' = acc + x in
  loop (s=s', acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1:} & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1:} & (0, =\texttt{f1}) \\
\texttt{f2:} & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2:} & (0, =\texttt{f2}) \\
\texttt{s:} & \top \\
\texttt{acc:} & \top
\end{array}
$$

Running example

```
let r =
 match f1 s with
 | Empty -> Empty
 | Cons (x, s') ->
   let y = square x in
   Cons (y, s')
in
match r with
| Empty -> acc
| Cons (x, s') ->
  let acc' = acc + x in
  loop (s=s', acc=acc')
```

Environment:

$$
\begin{aligned}
\texttt{f1}&: [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}&: (0, {=}\texttt{f1}) \\
\texttt{f2}&: [\texttt{map.anon} \\
&\quad \{\texttt{next} \mapsto {=}\texttt{f1}, \\
&\quad \texttt{f} \mapsto {=}\texttt{square}\}] \\
\texttt{s2}&: (0, {=}\texttt{f2}) \\
\texttt{s}&: \top \\
\texttt{acc}&: \top
\end{aligned}
$$

Running example

```
let p =
 if s < 11 then
  let t = s+1 in Cons (s,t)
 else Empty
in
let r =
 match p with
 | Empty -> Empty
 | Cons (x, s') ->
   let y = square x in
   Cons (y, s')
in
match r with
| Empty -> acc
| Cons (x, s') ->
  let acc' = acc + x in
  loop (s=s', acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, =\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2}: & (0, =\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{s}, =\texttt{t})
\end{array}
$$

Running example

```
let p =
 if s < 11 then
  let t = s+1 in Cons (s,t)
 else Empty
in
let r =
 match p with
 | Empty -> Empty
 | Cons _ ->
   let y = s * s in
   Cons (y, t)
in
match r with
| Empty -> acc
| Cons (x, s') ->
  let acc' = acc + x in
  loop (s=s', acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, {=}\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto {=}\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto {=}\texttt{square}\}] \\
\texttt{s2}: & (0, {=}\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}({=}\texttt{s}, {=}\texttt{t})
\end{array}
$$

Running example

```
let p =
 if s < 11 then
  let t = s+1 in Cons (s,t)
 else Empty
in
let r =
 match p with
 | Empty -> Empty
 | Cons _ ->
   let y = s * s in
   Cons (y, t)
in
match r with
| Empty -> acc
| Cons (x, s') ->
  let acc' = acc + x in
  loop (s=s', acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, =\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2}: & (0, =\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{s}, =\texttt{t}) \\
\texttt{y}: & \top \\
\texttt{r}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{y}, =\texttt{t})
\end{array}
$$

Running example

```
let p =
 if s < 11 then
  let t = s+1 in Cons (s,t)
 else Empty
in
let r =
 match p with
 | Empty -> Empty
 | Cons _ ->
   let y = s * s in
   Cons (y, t)
in
match r with
| Empty -> acc
| Cons _ ->
  let acc' = acc + y in
  loop (s=t, acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, {=}\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto {=}\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto {=}\texttt{square}\}] \\
\texttt{s2}: & (0, {=}\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}({=}\texttt{s}, {=}\texttt{t}) \\
\texttt{y}: & \top \\
\texttt{r}: & \texttt{Empty} \mid \texttt{Cons}({=}\texttt{y}, {=}\texttt{t})
\end{array}
$$

Running example

```
let r =
 if s < 11 then
  let t = s + 1 in
  let y = s * s in
  Cons (y, t)
 else
  Empty
in
match r with
| Empty -> acc
| Cons _ ->
  let acc' = acc + y in
  loop (s=t, acc=acc')
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, =\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2}: & (0, =\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{s}, =\texttt{t}) \\
\texttt{y}: & \top \\
\texttt{r}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{y}, =\texttt{t})
\end{array}
$$

Running example

```
if s < 11 then
 let t = s + 1 in
 let y = s * s in
 let acc' = acc + y in
 loop (s=t, acc=acc')
else
 acc
```

Environment:

$$
\begin{array}{rl}
\texttt{f1}: & [\texttt{ints.anon}\{\texttt{hi} \mapsto 11\}] \\
\texttt{s1}: & (0, =\texttt{f1}) \\
\texttt{f2}: & [\texttt{map.anon} \\
 & \ \{\texttt{next} \mapsto =\texttt{f1}, \\
 & \ \ \texttt{f} \mapsto =\texttt{square}\}] \\
\texttt{s2}: & (0, =\texttt{f2}) \\
\texttt{s}: & \top \\
\texttt{acc}: & \top \\
\texttt{t}: & \top \\
\texttt{p}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{s}, =\texttt{t}) \\
\texttt{y}: & \top \\
\texttt{r}: & \texttt{Empty} \mid \texttt{Cons}(=\texttt{y}, =\texttt{t})
\end{array}
$$

Running example

```
loop (s=0,acc=0) {
 if s < 11 then
  let t = s + 1 in
  let y = s * s in
  let acc' = acc + y in
  loop (s=t, acc=acc')
 else
  acc
}
```

Environment:

```
   f1: [ints.anon{hi ↦ 11}]
   s1: (0, =f1)
   f2: [map.anon
        {next ↦ =f1,
         f ↦ =square}]
   s2: (0, =f2)
    s: ⊤
  acc: ⊤
    t: ⊤
    p: Empty | Cons(=s, =t)
    y: ⊤
    r: Empty | Cons(=y, =t)
```

Stdlib.Seq.fold_left

```
let rec fold_left f acc seq =
  match seq () with
    | Nil -> acc
    | Cons (x, next) ->
        let acc = f acc x in
        fold_left f acc next
```

Simplifying this requires proving that `next` equals `seq` which is tricky, in addition to having function specialization in the simplifier, which is not yet there.

We can however prove this for the type in the example, and specialization is not required.

Conclusion

- This is not a finished work
  - Some features are written but not yet merged
  - There are features in the pipeline we haven't started on yet
  - Maybe some existing features will be altered in the future
- However the code is stable and fully reviewed
  - The API enables changes to be made locally without touching the rest of the system
- Many production systems are built with this (10,000+ instances)
  - Systems have been running now for several months
  - There have been no known miscompilations amongst these
  - Flambda 2 is expected to replace Closure and Flambda 1 completely at Jane Street during September