Flambda 2

Pierre Chambart and **Vincent Laviron** (OCamlPro)
Mark Shinwell (Jane Street)

Cambium Seminar, Inria Paris, 2023-06-26

History and context

What is Flambda 2?

- A new *middle end optimizer* that replaces Closure and Flambda 1
- Translates from Lambda to Cmm in two or three passes
- **Competes with Closure on compilation speed** in a "fast build" mode
- **Competes with Flambda 1 on runtime performance** in optimized mode
- Provides a general and extensible framework for optimization
- Real, critical systems compiled with it are running right now

Why Flambda 2?

The main motivation was limitations of Flambda 1

- The big one: in Flambda 1, most subexpressions were named, but *control points* (e.g. jump targets) were not named
- This made it difficult to write certain optimizations, for example match-in-match, or unboxing function return values
- Flambda 1 was also kind of ad-hoc and rather over-specialized. We wanted something more principled

We made various prototypes based on Flambda 1, mainly:

- CPS-based control flow optimisation passes
- adjusting the rather problematic closure representation

Eventually we decided a wholesale change of IR was best.

The new world

Structure of the middle end

There are three passes:

- `Lambda_to_flambda` converts from Lambda to Flambda 2
- `Simplify` performs lots of optimizations, in conjunction with a *type system* (abstract domain), which does not depend on the term language
- `To_cmm` produces Cmm code of a form which instruction selection can work on

In the fast-build "classic" mode, `Simplify` is not run.

Classic mode

- In Flambda 1, was trying to emulate the performance and optimisation features of the original compiler
- In Flambda 2, is meant as a fast but still decently good option, primarily for development builds
- It is compatible with the normal mode (not a configuration flag)
- In practice, it means bypassing Simplify completely, but adding a bit of extra work elsewhere

Style of the IR

- A hybrid of A-normal form (ANF), CPS and direct style
- Kennedy-style double-barrelled CPS (second-class normal and exception continuations): **every control point is named**
- No nested subexpressions: **every intermediate value is named**, as in ANF
- Still retains normal `let`-expressions, but the part after the = never has any control flow effects
- Primitive operations are close to Cmm primitives, but never have any control flow effects
- This IR is overall better for optimization than Flambda 1, though it makes some things harder (e.g. peephole optimizations).

Style of the IR: closures

- The IR has ways of creating and manipulating closures
- One reason for this is to be able to measure their cost when inlining
- Closures are represented separately from *code*: closed function bodies
- This is a major departure from Flambda 1. We should have used this approach in that system.
- Code gets to its own closure via an explicit `my_closure` parameter (like in Cmm)
- Code is named using *code IDs*. These correspond directly to function symbols in the resulting `.o` file

Closures are subtle and difficult to deal with.
However we have now tamed their complexity pretty well.

Rough shape of the IR

```
type expr =                    (* genuinely only six cases! *)
  | Let of ids * named * expr
  | Let_cont of k * vars * expr * expr
  | Apply of apply * k * exn_k * simples
  | Apply_cont of k * simples * trap_action
  | Switch of simple * (int * k * simple) list
  | Invalid of { message : string }
and named =
  | Simple of simple    (* var, symbol or const *)
  | Prim of ... (* arithmetic, load/store, ... *)
  | Set_of_closures of ...
  | Static_consts of ... static_const_or_code ...
and static_const_or_code =
  | Code of ... (* function body, sans closure *)
  | Static_const of ... (* like Cmm data items *)
```

Name binding for variables and continuations

- The bane of many compiler writers' lives, but not this time!
- Terms in the IR represent alpha-equivalence classes, enforced using abstract types
- Names are not unique, but are freshened when looking at them
- This means terms can just be assembled without either freshening or concern about name clashes
- Lazy freshening avoids expensive traversals
- Using permutations (as in nominal sets) instead of substitutions provides a second layer of assurance, as it is impossible to break alpha-equivalence using these

Example: match expression in IR

Switch is low level, only on naked integers.

```
match x with
 | None -> 1
 | Some s -> s
```

IR:

```
let i = is_int x
switch i
  | 0 -> apply_cont k_return 1
  | 1 -> apply_cont k_some     ; yes, we have goto!
where k_some =
  let r = field x 0 in
  apply_cont return r
```

Relational reasoning

```
let x = if b then None else Some 3 in
match x with
| None -> 3
| Some n -> n
```

`is_int x : =1` should imply `x : Block (Tag_0, =3)`

```
let i = is_int x
switch i
  | 0 -> apply_cont k_return 1
  | 1 -> apply_cont k_some
where k_some =
   let r = field x 0 in
   apply_cont k_return r
```

Translation from Lambda to Flambda 2

CPS conversion

- Continuations and proto-Flambda 2 terms are created:
  - `let_cont` of (non-trivial) `let`-bindings and code branches
  - Add return continuation to function parameters
- Mutable variables (local references) are turned into immutable variables and passed as extra continuation parameters when necessary. There are no mutable variables in Flambda 2.

Meanwhile, at the same time, a hand-deforested closure conversion algorithm runs.

Closure conversion

- Introduces closure and code bindings
- Produces actual Flambda 2 terms
- Removal of dead continuations arising from CPS conversion
- Localized unboxing
- Lifting of obviously-constant values

Classic mode features

Closure conversion also does most classic mode optimizations.

- Simple lifting of static data
  - Boxed values
  - Blocks (including modules)
  - Closures
- Tracking of value approximations:
  - Code for inlining or direct calls
  - Block fields to skip loading and yield lifted symbols
  - OCaml values for unboxing and static switch reduction
- Approximations translated to Flambda 2 types for `.cmx` export
  - `.cmx` compatibility **between classic and optimized modes**!
- Counting of continuation occurrences (needed for Cmm translation)

Flambda 2 to Cmm

Translation from Flambda 2 to Cmm

We're going to look at this before Simplify as it completes the classic mode pipeline.

- Eliminate continuations:
  - Direct translation to static exit/catch
  - Inline continuations used exactly once
- Produce expressions that `Selectgen` will like:
  - Substitute let-bound expressions used exactly once, and not crossing some boundaries (e.g. loops)
  - Allow some re-ordering if effects/coeffects allow it
  - This removes the need for the old `Un_anf` pass
- For exception handlers:
  - Use Cmm mutable variables for exception handler extra parameters introduced during closure conversion

Optimizations in To_cmm

- Remove unused value/function slots
- Statically allocate sets of closures with no value slots
- Switches
  - Generate an if-then-else directly, particularly when one discriminant is 0
  - Decide whether to use the tagged or untagged version of the scrutinee for binary switches (use the smallest expression)
- In classic mode, unbox some numbers:
  - Substitute boxed operations at their use-site
  - Rely on `Cmm_helpers` to unbox the substituted boxing (if applicable)
  - This can duplicate boxing operations

Simplify

High level view

The simplifier is in general a one-pass optimizer (although occasionally it may rerun itself on individual functions). It works in three stages:

- Downwards traversal in dominator order, collecting and using information in an abstract domain
- A couple of fast fixpoint calculations having reached the end
- Upwards traversal rebuilding the term

The simplifier is tail recursive, which is important for processing large source files

Inlining

One of the main optimizations the simplifier performs is inlining. It looks at:

- Inlining cost in code size (CPU fetch, binary size, compile time)
- Inlining benefits by specialising code to its context

To decide whether an inlining is worth doing:

- Heuristically decide when inlining is good (e.g. for very small functions)
- Inline then examine the result (Flambda 1 did this, but Flambda 2 can do the examination without actually rebuilding the proposed inlined term)

Forward pass: information collection

Traverse the control flow in dominator order

- Variables are freshened behind the scenes
- `let` introduces equations (meet) on variables in the domain
- `let_cont` is a join on the domain
- Dependencies between variables, symbols, etc are accumulated

Forward pass: transformation

- Inlining
- Partial- and over-application expansion
- Simplification of primitives based on information in the domain
- Decisions for static allocation (as in previous middle ends, statically-allocated values may be inconstant)
- Accumulation of symbol bindings corresponding to new static allocations
- Unboxing
- CSE

Fixpoints

We don’t do fixpoints on the abstract domain, but there are still some specific optimizations we need them for.

These are done once the “bottom” of the term has been reached.

- Dependency calculations, to work out which bindings are dead
- Local data flow analysis for mutable unboxing

Upward pass

- Rebuilding of the term
- Deletion of dead bindings
- Continuation simplifications (e.g. inlining continuations used exactly once)
- Rewriting of primitives for mutable unboxing
- Insertion of symbol bindings at the toplevel of the expression
- Computation of free variables
- Counting of continuation occurrences (needed for Cmm translation)

At one point we tried keeping free variable sets on the terms, but it was a performance disaster.

Examples of new optimizations

Match-in-match without allocation

```
type t = C | D | E
type s = A of int | B of int

let foo c a b =
  let m =
    match c with
    | C -> A a
    | D -> B b
    | E -> B (b + 1)
  in
  match m with
  | A x -> ...
  | B y -> ...
```

Post-inlining array operation specialization

```
let bar arr =
  for x = 0 to Array.length arr - 2 do
    arr.(x) <- arr.(x + 1)
  done

let foo (arr : int array) =
  ...;
  (bar [@inlined]) arr;
  ...
```

Loops like a C compiler (well, almost...)

```
let f cb i xs =
  (List.iter [@inlined])
    (fun [@inline] x ->
       cb (i + x))
    xs
```

```
        subq    $24, %rsp
        movq    %rax, (%rsp)
        movq    %rbx, 8(%rsp)
        movq    %rdi, 16(%rsp)
L100:
        testb   $1, %dil
        je      L101
        movl    $1, %eax
        addq    $24, %rsp
        ret
        .align  2
L101:
        movq    (%rsp), %rbx
        movq    (%rdi), %rax
        movq    8(%rsp), %rdi
        leaq    -1(%rdi,%rax), %rax
        movq    (%rbx), %rdi
        call    *%rdi
        movq    16(%rsp), %rdi
        movq    8(%rdi), %rdi
        movq    %rdi, 16(%rsp)
        jmp     L100
```

Improved mutable unboxing

```
type t = { a : int; b : float }

let[@inline] g r =
  r := { !r with b = !r.b +. 1. }

let f x =
  let r = ref x in
  for i = 0 to 10 do
    g r
  done;
  !r.b +. Float.of_int !r.a
```

Identity matches

```
type t = A | B | C
let f = function
  | A -> 0
  | B -> 1
  | C -> 2

type t1 = C of int | D of int
type t2 = Foo of int | Bar of int
let g = function
  | C x -> Foo x
  | D y -> Bar y
```

Local exceptions converted into jumps

```
exception Exit

let f x y =
  let r = ref x in
  try
    for i = 1 to y do
      if !r > 100 then raise_notrace Exit;
      r := !r + y
    done;
    !r
  with Exit -> !r
```

A couple of things are missing

Compared to Flambda 1, the following aren't yet implemented in Flambda 2:

- Unboxing of free variables of functions inside closures
- Specialization of functions given invariant parameters
- Lambda lifting (implemented in Flambda 1 but off by default, as it was never fully satisfactory)

The first of these is likely to come soon to Flambda 2.

Flambda 2 type system (abstract domains)

Abstract domain API

- Meet and join
- Queries (checking properties)
- Levels: backtracking/replaying functionality
- No fixpoints needed, so no widening or inclusion test

Full details to come in a follow-up talk, but we will whet the appetite here

Typical basic use

Reading the first field of a block `b`

```
let v = Variable.create "field" in
let env = TE.add_definition env v in
let t = T.immutable_block ~fields:[v] in
let env = TE.add_equation env b t in
```

More constructors

Reading from a block `b` at field `idx`
For performance reasons, we don't want to build large useless values

```
let v = Variable.create "field" in
let env = TE.add_definition env v in
let t = T.immutable_block_with_size_at_least
  ~n:(idx+1) ~field_n_minus_one:v in
let env = TE.add_equation env b t in
```

Inspection functions

Collection of light 'prove' and 'meet' functions that might avoid an expensive meet

```
let block_type = TE.find env b in
let meet_shortcut =
  meet_block_field_simple env block_type idx
in
match meet_shortcut with
| Known_result res -> ...
| Need_meet -> (* previous case *) ...
| Invalid -> Bottom
```

Reify

Ability to reconstruct terms from approximations

- Can replace primitives by their known result
- Used mostly for lifting allocations

Everything else

How do we make this go fast?

- Careful handling of names, as we have described
- Most identifiers (variables etc.) are represented by integers, with a hash table on the side providing more details when required
- Upon import of `.cmx` files any hash collisions are resolved via lazy renaming
- This means we can use Patricia trees for sets and maps
- Avoid default inlining parameters being too aggressive
- Aggressive pruning of `.cmx` file contents via reachability analysis
- Profiling with memtrace and perf (more to do here)

Terms of the IR are immutable to reduce the potential for error, but there is some local use of mutability for performance hidden under interfaces.

Flambda 2 and the rest of the OCaml system

Flambda 2 is well-isolated in its own directory. The interface is:

```
val lambda_to_cmm :
  ppf_dump:Format.formatter ->
  prefixname:string -> filename:string ->
  keep_symbol_tables:bool ->
  Lambda.program -> Cmm.phrase list
```

From the rest of the compiler it needs:

- The ability to generate direct push/pop trap primitives for exception handling in the backend
- A few minor changes in the front end of the compiler
- A relatively simple patch to improve handling of *asynchronous exceptions* (e.g. those arising from finalisers). We will present this upstream. The patch prevents GC safe points from having control flow effects.

Correctness

Three main test suites:

- normal compiler testsuite
- compiling all of OPAM
- compiling the Jane Street tree and running all the tests

The Jane Street tree is the most effective test suite: it's quite hard to cause a bug in the compiler and not have a failure. This is partially because there are very many tests that involve not only compilation but also execution.

There has not been a single failure amongst the Jane Street deployments so far. More than 8,000 critical systems compiled with Flambda 2 are running daily. This number will increase now that, for most systems built with Flambda 1, Flambda 2 is used instead.

Language extensions

Flambda 2 has (or will soon have) support for various interesting language extensions:

- Regions for non-escaping *local* allocations made on a separate stack (in production now)
- Unboxed tuples, including for function parameters and return values (nearly ready)
- SIMD vector types (in progress)

Implementing these new features in Flambda 2 has been generally straightforward (save for a few tricky details in the case of local allocations). Indeed for locals, the implementation in Flambda 1 turned out to be inhibiting existing optimizations, which was not the case in Flambda 2.

This project has taken longer than expected!

- Perhaps we should have tried harder to make incremental changes to Flambda 1, but the structure of the existing passes and the step change required in IR would have made this problematic
- We maybe should have tried to do only classic mode first
- We could next have done `Simplify` with a more straightforward type system, before improving the typing.
- Although we didn't come up with the idea of using the Lambda to Flambda 2 pass to implement classic mode optimizations until quite late on. (At first we thought we could still somehow use `Simplify` for classic mode, but that would have been too slow.)

It's unclear whether we would have reached a comparably good end result with a different programme of development.

Future directions

- We still need to do some code cleanup, but it's in a pretty good state
- This is to be expected when a new chunk of code arrives
- We plan to work on trying to simplify various parts of the system (for example the notion of 'symbol' can probably be replaced by variables using a special 'static' mode, which will avoid the special dominator scoping rule for symbols)
- There are many exciting optimizations on the list, for example partial dead code elimination, for better placement of allocations; and improved compilation of match-in-match.

Any questions?

Thanks to everyone who has contributed:

Guillaume Bury, Pierre Chambart, Nathanaëlle Courant,
Keryan Didier, Vincent Laviron (OCamlPro)

Xavier Clerc, Luke Maurer, Pierre Oechsel, Mark Shinwell,
Leo White (Jane Street)

plus anyone we have forgotten.

Extra slides

Levels

Hints to make join efficient.
Often joins are between two mostly identical states: A large shared context and a small diff on top.
Levels marks points where to look for shared context.

Some aliases have to be tracked

```
let f x y =
  let block = (x, y) in
  fst block + snd block
```

should turn into

```
let f x y =
  x + y
```

Types and environments

- Type: information known about one value
  - Singletons, finite sets
  - Block shapes
  - Aliases
  - And more
- Environment: mostly a map from variables to their types
- Also contains relational information

Main operations: Meet and 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

Main operations: Meet and Join

- Join is used mostly when merging branches (i.e. on 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 can provide us with a common ancestor, which helps keeping the complexity manageable
- Joining two types in a given environment is also implemented

Basic types: finite sets

- Used mostly for numbers, also for immutable strings
- Meet is the set intersection, Join is the set union
- Non-relational

Aliases: Environment

- Relational information, tracked globally in the environment
- Implemented using equivalence classes
- Meet merges equivalence classes, Join splits them

Aliases: Canonical elements

- Each equivalence class has a canonical element
- Only canonical elements have concrete types associated to them
- Non-canonical elements have *alias types* (singletons)

Aliases: Meet and Join

- Meet between two alias types merges the equivalence classes
- Join between two alias types is `Top` unless they're in the same equivalence class

Blocks

- Blocks have both positive and negative versions
- Positive block types represent full blocks as maps from indices (integers) to types
- Negative block types represent blocks where only part of the indices are known (the total size might not be known either)
- Both forms can represent projections using alias types in the map

Variants: representation

- Variants introduce disjunctive constraints:
  - Variant values can be either integers or blocks
  - Variant blocks can have different shapes depending on their tag
- Meet applies component-wise, with `Bottom` components removed, but the component-wise environments must be joined or dropped (see later for a better solution)
- Join applies component-wise

Variants: relational information

- Some variables can track the tag of a block, or whether a variant is an integer or a pointer
- The typing environment represents this relation in the types of the integer variable (not the blocks directly)
- Meet between a concrete type and a tag or isint type introduce constraints on the relevant variant type
- Meet between two tag or isint types could introduce various constraints, but it's not common so in practice it's not implemented
- Join is straightforward

Closures

- Very similar to regular blocks, with an extra field for the code
- Environment entries are indexed by value slot instead of integers
- Mutually recursive functions are indexed by function slot
- Support exists for disjunctions of closure types, similar to variant blocks
- Meet and Join use the same algorithms as for blocks

Environment extensions

- A special data structure that encodes partial environments
- As all relational information can be encoded as types, concretely it is a map from variables to types
- Initially returned by the meet functions instead of a full environment
- Now also stored in some disjunctive contexts to encode constraints that only apply in one case
- When a meet resolves a disjunction to a single case, the corresponding extension can be recovered and added to the result
- Useful for unboxing variants

Function return types

- Summary of the relations between inputs and outputs of a function
- Computed by extracting the environment for the return continuation
- Used for non-inlined direct function applications
- Disabled by default, can be activated for functors only or all functions
- Attached to code, not closures

Existential variables

- Environments can manipulate existentially-quantified variables
- This corresponds to variables no longer in scope
- Usually introduced during Join to keep relational information
- Can be explicitly projected out to make environments more compact
- Would cause trouble if we had to implement fixpoints

Optimisation: levels

- Environments have a dual representation
- They can be viewed as a single structure, with all constraints propagated, allowing fast queries
- They can be viewed as a sequence of variable introductions and refinments by constraints, allowing faster joins
- We use a data structure that can do both

Levels

- We use a notion of scopes, that are defined by the user (Simplify here), to split our environment
- An environment is then a map from scope to level (plus a single current level)
- A level stores, for the corresponding scope:
  - Which variables have been introduced
  - Which equations (or constraints) have been added
  - A compact representation of the environment at the end of the scope

Levels

- During Join, the user specifies a base scope that is guaranteed to be included on both sides
- For each side, we then merge all levels with later scopes into a single one, discarding the compact environments
- We then introduce all variables from both sides into the result, as existentials
- Existentials only present on one side are given a special `Bottom` type on the other side
- We then join all the constraints, keeping only constraints valid on both sides
- The result of the Join is the environment at the base scope, with the current level set to contain all the new variables and the common constraints
- The compact form is re-computed from that