Dataflow Facts as Stable Patterns

FABIAN LUKAS GRUBMÜLLER, ETH Zürich, Switzerland
JULES JACOBS, ETH Zürich, Switzerland

Constant propagation, value numbering, SSA construction, unboxing and contification are usually presented as separate analyses. We show that they can be seen as special cases of one analysis. The analysis finds a *pattern* (in the sense of ML pattern matching) that the arguments of every call to a block (a function, loop or branch target) match. A pattern can say that an argument is a constant, that two arguments are equal, that an argument is a pair, or that it is a known closure; rewriting the block to assume its pattern can then drop a constant parameter, merge two equal parameters, unbox a pair, and make a closure call a direct call. Patterns are *stable* if they hold for all calls/jumps that remain in the rewritten block. Stability is what we need for correctness: any set of stable patterns is sound for optimization. We give an algorithm that computes the most specific stable patterns, which generalizes many classical analyses. We prove correctness, optimality, and termination in Lean, for a core language. If we allow several patterns per block, the same algorithm becomes an inliner, a specializer of higher-order functions, or a partial evaluator, depending on the policy that chooses patterns. We illustrate these ideas: (1) in an optimizer for a dependently typed language, where it does monomorphization and compiles programs written with folds, streams and dictionaries of functions to tight loops, (2) by porting some of the ideas to Flambda 2 and Chez Scheme, to speed up some OCaml and Scheme benchmarks, (3) by comparing it experimentally to prior classical analyses.

1 Introduction

A compiler optimizes by rewriting: `2 - 1` to `1`, `a - a` to `0`, and `(a, b).0` to `a`. Such rewrite rules are local, so to make them fire across a program, we need to bring in global knowledge: that this parameter is always 1, that these two are always equal, that one is always a pair.

Finding these facts is hard, because a fact can depend on itself. For example, in the program below `x` is `1` on every iteration (we write `?x`, a *binder*, where a variable is bound, and plain `x` where it is used):[1]

```
main(?i) = loop(i, 1)
loop(?i, ?x) = if i == 0 then x else loop(i - 1, 2 - x)
```

A pessimistic analysis misses this fact: the recursive call passes `2 - x` for `x`, which is unknown as long as `x` is. An *optimistic* analysis sees `1` as the argument for `x` at the call from `main` and assumes that every later call will also pass `1`. Under this assumption the recursive call passes `2 - 1`, which is `1`, so the assumption holds.

Compilers do this with dataflow analysis [1, 42, 45, 85], which computes such optimistic facts at all points of the program. Each kind of fact has its own analysis via a lattice fixed point, such as SSA construction [8, 24], constant propagation [85], value numbering [1, 75, 79], unboxing [38], contification [31, 44] and the control-flow analysis of closures [78], and compilers run these analyses one after another [62, 84]. Facts of different kinds can depend on each other, and where the dependence runs in both directions, a combined analysis finds facts that the separate analyses, run one after another, miss [18, 54].

This paper is about one idea: dataflow facts can be represented as a *pattern* that the arguments of every call of a block match, where a block is a function, a loop or a branch target. These patterns are those of *pattern matching* in functional programming: a function is defined by clauses, each of

[1] Although we write programs as equations like high-level functional programs with tail calls, this is best seen as a low-level IR: a definition is a basic block, and a call of a block is a jump, not a function call. A block's parameters are the ϕ-functions of SSA, written as block arguments as in MLIR [47] or Cranelift [23] (§2.1).

Authors' Contact Information: Fabian Lukas Grubmüller, ETH Zürich, Switzerland; Jules Jacobs, ETH Zürich, Switzerland.

which has patterns in place of parameters and applies only to the calls whose arguments match them. To illustrate what this has to do with dataflow analysis, we return to `loop` and state that `x` is always `1` by writing the pattern `1` in place of `?x`, and substitute $x = 1$ in the body:

```
loop(?i, 1) = if i == 0 then 1 else loop(i - 1, 2 - 1)
```

Then *rewrite* with the compiler's rules: `2 - 1` folds to `1`:

```
loop(?i, 1) = if i == 0 then 1 else loop(i - 1, 1)
```

Read as a definition by pattern matching, it defines `loop` only for calls whose second argument is `1`, and for any other call the match fails. (We print clauses in blue to distinguish them from the source program.)

Defining `loop` this way is safe if no call ever fails to match. The program has two calls of `loop`: `main`'s call `loop(i, 1)` and `loop`'s own call in the simplified body, `loop(i - 1, 1)`, and both match. A match that never fails need not be performed, so the `1` does not need to be passed nor tested for. Deleting the match leaves

```
main(?i) = loop(i)
loop(?i) = if i == 0 then 1 else loop(i - 1)
```

What we just did is the following:

> *Write a pattern in place of a parameter of every block, as specific as it can be while every call in the program matches the pattern. Then remove matching pieces.*

This may seem like a very roundabout way to do constant propagation, but the beauty is that it goes much further, and the same idea can do minimal SSA construction, value numbering, unboxing, contification, control flow analysis of closures, and more.

For value numbering, take a loop with two parameters that are always equal:

```
main(?n) = loop(n, 0, 0)
loop(?i, ?a, ?b) = if i == 0 then a + (a - b) else loop(i - 1, a + 1, b + 1)
```

We state that `b` is always equal to `a` by writing the pattern `loop(?i, ?a, a)`. A plain name in a pattern matches only the value of that variable, here the parameter to its left. We thus optimistically substitute $b = a$ in the body, and `a + (a - a)` folds to `a`:

```
loop(?i, ?a, a) = if i == 0 then a else loop(i - 1, a + 1, a + 1)
```

Both calls match: `main` passes `0` twice, and the recursive call passes `a + 1` twice. Deleting the matching piece removes the third parameter:

```
main(?n) = loop(n, 0)
loop(?i, ?a) = if i == 0 then a else loop(i - 1, a + 1)
```

We just did *value numbering*: we removed a quantity that was equal to another, even around a loop.

For unboxing, take a loop that carries a pair:

```
main(?u, ?n) = loop(n, (0, u))
loop(?i, ?q) = if i == 0 then q.0 + q.1 else loop(i - 1, (q.0 + q.1, q.1))
```

Here `q` is always a pair, and its second field is always `main`'s `u`. One pattern can do both: a pair with a binder `?s` for the first field and the name `u` for the second. We substitute $q = (s, u)$ in the body, and `q.0` and `q.1` fold to `s` and `u`:

```
loop(?i, (?s, u)) = if i == 0 then s + u else loop(i - 1, (s + u, u))
```

Both calls pass a pair whose second field is `u`, so both match. With the match deleted, no pair is built: `loop` is passed only the first field, and it reads `u` from `main`, inside which we place it:

Table 1. What a pattern says about the calls of a block `f`, and the optimization that it gives. The rows above the line copy no code and need no inlining; those below copy a block for each of its patterns. Figure 1 has examples.

Pattern	What it says	Optimization
`f(?x, 1)`	an argument is always the same constant	constant propagation, dead-branch elimination
`f(?x, u)`	it is always a variable `u` that is in scope	copy propagation; minimal SSA, with no ϕ
`f(?x, x)`	two arguments are always equal	value numbering
`f((?a, ?b))`	it is always a pair	unboxing, argument promotion
`f(Some(?a))`	it is always built with the constructor `Some`	the case of a known constructor
`f(g⟨u⟩)`	it is always the same closure	a direct call: devirtualization, lambda dropping
`f(g⟨?a⟩)`	a closure of `g` whose capture varies	lambda lifting: `a` is passed, not a closure
`f(?x; k)`	`f` always returns to `k`	contification
`k(g⟨a⟩)`	`k` is always returned a closure of `g`, which it calls	uncurrying: the closure is not built
`f((g, h))`	it is always the same record of functions	devirtualization of calls through a dictionary
several patterns, chosen by a policy	a pattern per closure that is passed	specialization of higher-order functions
	a pattern per constructor that is passed	constructor specialization [66]
	a pattern per call site	inlining, jump threading
	a pattern per value of a marked parameter	partial evaluation, loop unswitching

```
main(?u, ?n) = loop(n, 0)
  where loop(?i, ?s) = if i == 0 then s + u else loop(i - 1, s + u)
```

We just unboxed the pair, and removed its second component entirely: a combination of unboxing and copy propagation.

The classical analyses find different kinds of pattern. A constant in a pattern is constant propagation. A plain name of a variable in scope is copy propagation, and in SSA it is a parameter that needs no ϕ (i.e., minimal SSA construction). A name bound by a binder to its left says that two arguments are always equal, which is value numbering. A constructor pattern is unboxing. A pattern can also describe a closure. `app(inc⟨u⟩)`, where `inc⟨u⟩` denotes partial application of `inc`, says that `app` is always passed a closure of the function `inc` that has captured the caller's `u`. Everything in such a closure is in scope where it is called, and just like unboxing can avoid building a pair, this can avoid building a closure, and calls to the closure can turn into direct calls. Similarly, in continuation-passing style, a return is a call of a closure too, so a pattern can say that a function always returns to the same place, which is contification. Table 1 lists what patterns can say and the analysis or transformation that each gives, and Figure 1 has examples. None of these copies code or relies on inlining: the output has each block of the program at most once.

We call the patterns that are safe to optimize with: *stable* patterns. Stable patterns are our analogue of dataflow lattice values at a fixed point: if we assume that the arguments of a block match its pattern, then the calls/jumps that the block makes match their target's pattern. Intuitively, if a set of patterns is stable, then at run time the arguments of calls match their block's pattern. In the unboxing example `loop(?i, (?s, u))` is a stable pattern, as is `loop(?i, ?q)`, but `loop(?i, (0, u))` is *not* stable: even if the first component of the pair is initially `0`, that no longer holds in the second iteration of the loop. This is similar to the closedness condition of partial deduction [58], used here as the correctness condition of a dataflow analysis. Stable patterns are always safe to use, no matter how you found them.

The difficult part: finding the patterns. So far we wrote the patterns down and checked that they are stable, which is the easy part: it takes one traversal of the program with local rewrites. Stability

Constant propagation

source: `scale(2, a) + scale(2, b) where scale(?k, ?x) = k * x`
analyzed: `scale(2, a) + scale(2, b) where scale(2, ?x) = 2 * x`
output: `scale(a) + scale(b) where scale(?x) = 2 * x`

A loop invariant; SSA without a ϕ

source: `loop(n, u) where loop(?i, ?a) = if i == 0 then a else loop(i - 1, a)`
analyzed: `loop(n, u) where loop(?i, u) = if i == 0 then u else loop(i - 1, u)`
output: `loop(n) where loop(?i) = if i == 0 then u else loop(i - 1)`

A result that is never built; contification

source: `let p = dup(y) in p.0 - p.1 where dup(?s) = (s, s)`
in CPS: `dup(y; k) where dup(?s; ?r) = r((s, s))`
`k(?p) = p.0 - p.1`
analyzed: `dup(y; k) where dup(y; k) = k((y, y))`
`k((y, y)) = 0`
output: `0`

A closure that disappears

source: `app(λ?x. x + u) where app(?f) = f(1)`
closure: `app(inc⟨u⟩) where app(?f) = f(1)`
`inc⟨?u⟩(?x) = x + u`
analyzed: `app(inc⟨u⟩) where app(inc⟨u⟩) = inc⟨u⟩(1)`
`inc⟨u⟩(1) = 1 + u`
output: `1 + u`

Fig. 1. Common optimization techniques expressed by patterns. We show the source, the patterns that our algorithm (§2.4) finds, and the output. The third program is first put in CPS, and the fourth has the λ as an explicit closure (§3.1): k is what the caller does with the result, and `inc⟨u⟩` is the block of the λ with the u it captured. More examples are in Table 3 (Appendix A).

suffices for correctness: given *any* stable patterns, deleting the matches, and the branches that the patterns decide, preserves the program's meaning (Theorem 5.2).

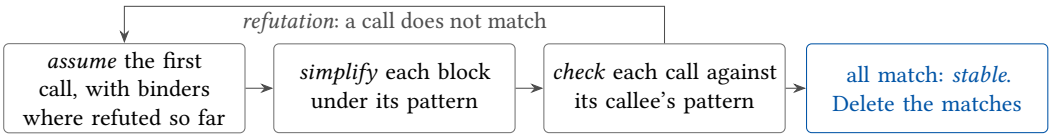


Fig. 2. The rounds of the algorithm. A refutation starts a new round; a round without a refutation ends the search.

The difficulty lies in finding the patterns in the first place, because a program has many choices of stable patterns and we want the most specific. Consider

```
main(?u, ?n) = loop(n, u, 1, Some((0, u)))
loop(?i, ?a, ?x, ?q) = case q of None → 0 | Some((?s, ?t)) →
    if i == 0 then s + t
    else if x == 1 then loop(i - 1, a, 2 - x, Some((s + a, t + (t - a))))
    else loop(i - 1, i, 99, q)
```

`x` is always 1, so the last line is dead; `a` is therefore always `main`'s `u`; and `q` is always `Some` of a pair whose second field `t` is `u`. These facts depend on themselves or on one another: `2 - x` is 1 only if `x` is, and `t + (t - a)` is `u` only if `t` and `a` are. So they have to be assumed before they can be confirmed. The following six patterns for `loop` are stable, in order of specificity:

`loop(?i, ?a, ?x, ?q)`	says nothing
`loop(?i, ?a, 1, ?q)`	what conditional constant propagation [85] finds
`loop(?i, u, 1, ?q)`	what Lemerre's analysis [54] finds
`loop(?i, u, 1, Some(?p))`	and the option is unboxed
`loop(?i, u, 1, Some((?s, ?t)))`	and the pair in it
`loop(?i, u, 1, Some((?s, u)))`	the most specific, which we find

With the matches of the last one deleted, the parameters of `loop` are its two binders, and the output is the loop of the unboxing example:

```
main(?u, ?n) = loop(n, 0)
  where loop(?i, ?s) = if i == 0 then s + u else loop(i - 1, s + u)
```

This combines constant propagation, dead-branch elimination, the detection of a loop-invariant parameter and the unboxing of a loop-carried option of a pair. None of the seven published analyses that we ran on this loop finds all of it, and LLVM at `-O3` finds all but the last step: it does not see that `t` is always `u`.

The usual way to compute a fixed point need not find the most specific patterns: a pattern can keep a value computed under a false assumption and so become too general, and in some programs the answer depends on the order of the visits (§2.5). In a monotone framework [42] every order gives the same result, but patterns do not fit this framework, because a pattern can refer to a variable of another block. Seen through patterns, the same effect shows in a published analysis: the results of its released implementation, written as patterns, depend on the iteration strategy and are not always the most specific (§7).

Instead, we give an optimistic fixed-point algorithm that finds the most specific patterns by *refutation* (Figure 2). *Assume* the strongest thing that could be true: a block's pattern is exactly the arguments of the calls that reach it before it is simplified, with binders where they differ. *Simplify* the blocks under those assumptions and look at the calls the program still makes. A call that fails to match the pattern of its callee *refutes* it: the pattern gets binders where the two differ, and this attempt, a *round*, is discarded with its computed values. Rounds are *repeated* until nothing

more is refuted; the patterns are then stable. Every refutation is forced by a call that the simplified program makes, so the result is the most specific stable patterns, whatever the order of the visits (Theorem 5.3).

Higher-order patterns. A pattern can contain a closure, and this makes the analysis interprocedural and higher-order. We write functions in continuation-passing style [3, 44], in which a function returns by calling its continuation parameter, and we treat a closure as a constructor whose tag is its target block and whose fields are what it captured. A capture that is a variable in scope, or is computed from such variables, is not passed. A capture that varies becomes a binder, and its value is passed in place of the closure: a stream `from⟨?a⟩` that goes around a loop becomes the loop variable `a` (§3.2). With such patterns the same fixed point is interprocedural: the pattern of a function describes its calls from all its callers, and a continuation in it says where the function returns. When a function always returns to the same place, its return becomes a direct jump there, which is contification [31, 44]. A closure that *escapes* needs a further rule (§3.1).

Specialization: several patterns for a block. Some facts hold only for some callers: in a `fold` that is called with two different λs, the closure cannot be specialized away. We therefore let a block have several patterns. A *specialization policy* says which part of a call is its *context*, such as the labels of the closures that it passes, and calls with different contexts go to different *versions* of the block, each with a separate pattern (Figure 6). This is the only way in which a block is copied. The recursive call of `fold` passes the same closure, so it goes to the same version, and the loop is specialized as a whole. Two call sites that pass the same closure share one version, where an inliner makes a copy for each (§4.1). The policy works as a dial on the same fixed point: with one version per block it is a dataflow analysis, and the lower rows of Table 1 list its other settings, up to a partial evaluator that specializes an interpreter away (§4.2). Specialization itself is an old idea [16, 20, 66]. Here it helps improve the analysis's fixed point.

Contributions. The main contribution is the idea that patterns give an elegant unifying account of many classical intra- and interprocedural analyses and transformations, and they extend naturally to higher-order programs. Concretely:

(1) **One idea in place of many analyses** (§2). Stable patterns for parameters do the work of constant propagation, value numbering, SSA construction, unboxing, contification and more, in one optimistic fixed point parameterized by local rewrite rules, which is 130 lines of OCaml without comments in the reference implementation (Appendix E).
(2) **Interprocedural and higher-order** (§3). We analyze across calls and returns and through closures uniformly, via continuation-passing style.
(3) **Specialization as more patterns** (§4). A specialization policy gives a block a pattern per context, in the same fixed point: per closure or constructor passed it is a specializer, per call site an inliner, and per value a partial evaluator.
(4) **A mechanized theory** (§5). We prove in Lean, for a core language with one pattern per block: (1) *Stable patterns are true at runtime:* every value that reaches a block when the program runs is an instance of the block's pattern. (2) *They are safe to use:* the transformation by stable patterns is semantically correct, so correctness needs only a stability check. (3) *There are most specific stable patterns:* they are an instance of any other stable patterns. (4) *And the algorithm finds them:* it terminates, and in whatever order it visits the blocks, it ends with them (Theorems 5.1 to 5.3).
(5) **Implementation** (§6). We provide a reference implementation and test it on a large suite of examples, and compare what it finds with analyses from the literature. We use the algorithm as the only optimizer in a compiler for a dependently typed language, where it also does

prog	::=	$\ell(\bar{x}) = b \ \ldots$	blocks; the first is the entry
b	::=	$\texttt{let}\ x = e\ \texttt{in}\ b \mid e(\bar{e}) \mid \texttt{case}\ e\ \{C_i \mapsto \ell_i(\bar{e}_i)\} \mid e$	bodies: lets, then a jump, a case or a result
e	::=	$n \mid x \mid C(\bar{e}) \mid \ell\langle\bar{e}\rangle \mid e \ominus e$	expressions
t	::=	$n \mid C(\bar{t}) \mid \ell\langle\bar{t}\rangle \mid ?h \mid h \mid ? \mid t \ominus t$	terms; $?h$ binder, h plain name, ? unknown

Fig. 3. Programs and terms. In a body, x is a parameter of the block or a variable bound by one of its lets. $\ell\langle\bar{e}\rangle$ is the block ℓ with its first parameters supplied, a closure (§3); we write ℓ when none are supplied. The operations $\ominus$ are primitives such as + and ==; they are pure and total. An unknown ? is a value from code that we do not analyze, and different unknowns are different values. From §3 on, a result is a jump to a continuation.

monomorphization, and many of our programs written with a fold, a stream, a monad or a dictionary of functions compile to the same assembly as a hand-written loop. We also port parts of it to Flambda 2 and Chez Scheme (which already optimize well), cutting Menhir's wall time by ca. 20% and making a Scheme benchmark 10.5 times faster.

2 Finding the Patterns by Refutation

We begin with first-order programs: blocks that bind parameters and end in a jump to a block or in a result. A loop is a block that jumps to itself. §3 extends the algorithm to functions that return and to functions that are passed as values, and §4 lets a block have more than one pattern. The clauses and outputs in the listings are printed by the reference implementation.

2.1 Programs

A program (Figure 3) is a list of blocks. The body of a block is a sequence of lets, followed by a jump, a case or a result. Blocks are closed: a body uses only the block's own parameters and the variables that its lets bind.[2] An arm of a case jumps to a block, which receives the arm's arguments followed by the fields of the constructor that the arm matches. In this section a jump goes to a label; §3 lets jump targets be values. The listings write the blocks of a case's arms in place: `if` e `then` b_1 `else` b_2 for a case on a test, and `case` e `of` $C(\bar{x}) \to b$ for a case on a constructor.

A program whose jumps go to labels is in SSA form with block arguments [4, 43, 69]. The loop of §1 can be written in three ways:

assignments and jumps

```
   x := 1
L: if i == 0 goto E
   i := i - 1
   x := 2 - x
   goto L
E: return x
```

SSA

```
L: i1 = φ(i0, i2)
   x1 = φ(1, x2)
   if i1 == 0 goto E
   i2 = i1 - 1
   x2 = 2 - x1; goto L
E: return x1
```

blocks with arguments

```
main(?i) = loop(i, 1)
loop(?i, ?x) =
  if i == 0 then x
  else loop(i - 1, 2 - x)
```

These are simply three equivalent ways to write the same low-level program. The pattern `loop(?i, 1)` says that $x_1 = \phi(1, x_2)$ is the constant 1.

We simplify expressions with the compiler's local rewrite rules, such as constant folding, $x - x = 0$ and $x + 0 = x$. We write $nf(e)$ for e simplified in this way, so $nf(2 - 1) = 1$ and $nf(u + (u - u)) = u$, and we write $e \equiv e'$ when $nf(e) = nf(e')$. The rules are a parameter of the algorithm, and the proofs assume conditions on them, which the reference implementation's rules meet.

[2] This is a simplifying assumption for exposition; the algorithm works fine if blocks can refer to other blocks' parameters directly.

2.2 Patterns

An *assignment* Q gives each block ℓ a pattern $Q[\ell]$, which has one term p for each parameter of ℓ:

$$p \ ::= \ ?h \ \mid \ h \ \mid \ n \in \mathbb{Z} \ \mid \ C(p_1, \ldots, p_n) \ \mid \ p_1 \ominus p_2$$

As in §1, $?h$ is a binder, and h is a plain name: it refers to a binder of this pattern or of another block's.

A call with arguments $\bar{a}$ *matches* the pattern $Q[\ell]$ if $\bar{a} \equiv Q[\ell]\theta$ for some substitution θ of terms for the binders of $Q[\ell]$. Otherwise the call *refutes* the pattern.

The *join* $a \sqcup b$ of two terms, for a block ℓ, is

$$a \sqcup b \ = \ \begin{cases} a & \text{if } a = b, \text{ with no binder of } \ell \text{ and no unknown in } a, \\ C(a_1 \sqcup b_1, \ldots, a_k \sqcup b_k) & \text{if } a = C(\bar{a}) \text{ and } b = C(\bar{b}), \\ ?h_{a,b} & \text{otherwise,} \end{cases}$$

where $h_{a,b}$ is a binder of ℓ, the same binder for the same pair a, b. Two tuples are joined component by component: $(0, 0) \sqcup (1, 1) = (?h, h)$, and $(0, 0) \sqcup (1, 2) = (?h_1, ?h_2)$. The join of two calls is the most specific pattern that both match, their least general generalization [67, 72].

2.3 Stable assignments

To *walk* a block under its pattern is to simplify its body: each parameter is replaced by its term in the pattern, and each expression is normalized with *nf*. If a case's scrutinee is then a constructor, the walk follows the matching arm, and otherwise each arm, with unknown fields. The jumps that the walk arrives at are the *calls* that the block makes under the pattern.

Definition 2.1 (Stable). The call of the entry, with unknown arguments, is *reached*. A block is reached if a reached call calls it, and a call is reached if the walk of a reached block arrives at it. An assignment is *stable* if every reached call matches the pattern of the block it calls.

Which calls are reached depends on the patterns, and the patterns must match the reached calls, so stability is a fixed-point condition, like that of an inductive invariant.

2.4 The algorithm

The algorithm searches for a stable assignment in *rounds* (Figures 2 and 5; §3.2 and §4.2 add the lines for closures and for policies).

A round. A round begins with the call of the entry (line 4). The algorithm then repeatedly picks a block that has been reached and not yet walked, and walks it (line 5). The block's pattern for the round is fixed when its walk begins (line 6). It is the join of $F[\ell]$, the calls that have reached the block so far in this round, with $\mathit{refuted}[\ell]$, which has binders where earlier rounds refuted the block's pattern. A call that the walk arrives at (line 8) is treated in one of two ways. If the walk of the block that it calls has not begun, the call is joined into that block's F (line 12). If the walk has begun, as it has for a loop's call of itself, the call is checked against the block's pattern (line 13), and a call that does not match refutes the pattern (line 14). The pattern does not change during the round. A round without a refutation has checked that the patterns are stable, and the search stops. Otherwise a new round starts from the call of the entry, and it keeps only *refuted* from the rounds before.

What is kept between rounds. When calls refute the pattern of ℓ, we join them with the pattern and keep the *skeleton* of the result as $\mathit{refuted}[\ell]$. It has the binders of ℓ, their uses and the constructors above them, with $\cdot$, read *nothing*, in place of the rest. The skeleton of $(1, (?h, h))$ is $(\cdot, (?h, h))$. A

Program

```
main(?u, ?n) = loop(n, u, 1, Some((0, u)))
loop(?i, ?a, ?x, ?q) = case q of None → 0 | Some((?s, ?t)) →
    if i == 0 then s + t
    else if x == 1 then loop(i - 1, a, 2 - x, Some((s + a, t + (t - a))))
    else loop(i - 1, i, 99, q)
```

Round 1 `main` is walked under `main(?u, ?n)`: the arguments of the entry are unknown. Its call `loop(n, u, 1, Some((0, u)))` is the first to reach `loop`, so it is `loop`'s pattern. Then `loop` is

```
loop(n, u, 1, Some((0, u))) = if n == 0 then u
                              else loop(n - 1, u, 1, Some((u, u)))
```

(`x == 1` holds, so the last line is dead; `2 - 1` folds to 1, and `u + (u - u)` to u)

	i	a	x	q = Some((s, t))	
loop's pattern	n	u	1	0	u
the call in its body	n - 1	u	1	u	u
the skeleton that is kept	?i	·	·	?s	·

checked: differs at two places, so it *refutes* the pattern

binders where the call differed; no values

Round 2 `main` is walked as before. The join of its call with the skeleton, `loop(?i, ·, ·, Some((?s, ·)))`, is `loop`'s pattern. Then `loop` is

```
loop(?i, u, 1, Some((?s, u))) = if i == 0 then s + u
                                else loop(i - 1, u, 1, Some((s + u, u)))
```

loop's pattern	?i	u	1	?s	u
the call in its body	i - 1	u	1	s + u	u

checked: *matches*. Nothing is refuted, so the patterns are stable

Output

```
main(?u, ?n) = loop(n, 0)
  where loop(?i, ?s) = if i == 0 then s + u else loop(i - 1, s + u)
```

Fig. 4. A run of the algorithm on the four-parameter `loop` of §1, from the program to the output. A grey cell is where the call differs from the pattern, and a shaded cell is a binder.

pattern is a claim about the calls of the block, which one call can refute. A skeleton is a claim about the block's stable patterns: they have a binder at or above each binder of the skeleton, and different binders where the skeleton has different ones. Where the skeleton has · it claims nothing. In particular it does not keep a value that the calls of this round agreed on, because that value was computed under the patterns of this round, and the round has refuted some of them. If the value still holds, the next round finds it again from its own calls; in the example of §2.5 it does not. Later calls can only add binders to a skeleton. The join of · with a term is the term with binders in place of its unknowns and of whole operations that contain unknowns. So in the next round the pattern has a binder where the skeleton has one, and elsewhere it is the join of that round's calls. A round that refutes ℓ makes $\mathit{refuted}[\ell]$ strictly more general, which can happen only finitely many times, so the algorithm terminates without widening (Theorem 5.3).

A run. Figure 4 follows the algorithm on the four-parameter loop of §1. The first round assumes that `x` is 1, that `a` is `main`'s `u` and that `q` is `Some` of a pair whose second field is `u`, and no call contradicts these. The recursive call differs at `i` and `s`, which become binders in the next round.

```
refuted[ℓ] := nothing, for every block ℓ     -- the only state kept between rounds
repeat                                        -- a round
  F[ℓ] := nothing, for every block ℓ; no block is walked
  call(entry, unknowns)
  while a block ℓ is reached and not walked:        -- any one
    Q[ℓ] := refuted[ℓ] ⊔ F[ℓ]; R[ℓ] := Q[ℓ]  -- its pattern for this round
    walk ℓ's body, its parameters bound to Q[ℓ], normalizing by nf;
      at each jump ℓ'(ā) it arrives at:  call(ℓ', ā)
until the round refuted nothing
return rewrite(Q)
call(ℓ, ā):
  ℓ's walk has not begun:  F[ℓ] := F[ℓ] ⊔ ā
  ā is Q[ℓ]θ:              nothing to do
  otherwise:               R[ℓ] := R[ℓ] ⊔ ā; refuted[ℓ] := skeleton(R[ℓ])  -- refuted
```

Fig. 5. The algorithm for first-order programs. $\sqcup$ is the join (§2.2; the text gives its value on *nothing*), and $R[\ell]$ is the join of ℓ's pattern with the calls that refuted it in this round. At a case, the walk calls the arms' blocks as §2.3 describes.

2.5 The most specific stable assignment

A program can have several stable assignments; §1 lists six patterns for a loop. The algorithm searches for the most specific, and a refutation shows that all stable patterns have binders where the skeleton has them. Write $p \sqsubseteq q$ when p is at least as specific as q, with *nothing* more specific than any term. Throughout the search, for each block,

$$\text{the skeleton} \sqsubseteq \text{the pattern of the round} \sqsubseteq \text{every stable pattern.}$$

A pattern stands for a set of calls, its instances, and a skeleton stands for a set of patterns, the candidates that the refutations so far have not ruled out. A round tries the most specific candidate. When a round refutes nothing, its patterns are stable, so they are the most specific stable patterns. We call this assignment the *optimum* O_{pat}. It does not depend on the order in which the algorithm walks the blocks. Theorem 5.3 proves this for a core language, with the rewrite rules of the reference implementation.

Why start over? The usual way to reach a fixed point keeps what it has found and repairs it: when a call fails to match, weaken the pattern of the block it calls, and revisit what depends on that block. For patterns it can miss the most specific ones, and in some programs its result depends on the order of the visits. Consider

```
main() = f(1)        f(?x) = g(x)        g(?b) = if b == 1 then f(2) else b
```

Both ways begin alike. `f` is assumed to be `f(1)`, whose body calls `g(1)`, so `g` is assumed to be `g(1)`; under that the test is true, and the call `f(2)` does not match `f(1)`. An iteration that repairs in place now weakens `f` to `f(?x)` and revisits it. `f` now calls `g(x)`, while `g`'s pattern is still `g(1)`. `1` and `x` have nothing in common, so their join is a new binder of `g`, `g(?b)`. That is stable, but it is not the most specific. The `1` in `g`'s old pattern was never a fact about `g`: it was `f`'s `x`, under an assumption that has since been refuted. In this program the stale `1` weakens `g`'s pattern whichever block is visited first. In others it happens under some orders and not others: on a program of a dozen lines, the two iteration strategies of Lemerre's artifact give different results for this reason.

The algorithm keeps from the first round only that `f`'s parameter must be a binder, and starts again from `f(?x)`. `f`'s body calls `g(x)`, the first call to reach `g`, so `g`'s pattern is the plain name `x`:

its argument is always `f`'s `x`. Since `g` is reached only from `f`'s body, it can be nested in `f`, where `x` is in scope, and `g` is passed nothing. Its clause is `g(x) = if x == 1 then f(2) else x`.

Because the pattern of the round is at least as specific as every stable pattern, a call that refutes it is also made under a stable assignment, perhaps with less specific arguments, and there it must match. So the stable pattern has a binder where the refutation puts one, and the skeleton stays below it, as does the pattern of the next round, the join of the skeleton with such calls (Theorem 5.3). This holds of all the refutations in a round, so a round does not have to stop at the first.

2.6 The rewrite

Given a stable assignment, the rewrite removes what the patterns have made redundant, as in §1. Take the last round of Figure 4. The clause is `loop(?i, u, 1, Some((?s, u))) = ...`, and the two calls of `loop` are `loop(n, u, 1, Some((0, u)))` and `loop(i - 1, u, 1, Some((s + u, u)))`. Both calls have `u`, `1` and `Some((_, u))` where the pattern has them, as stability requires. So the rewrite deletes these parts from the pattern and from both calls. What is left of the pattern is its binders, `loop(?i, ?s)`, and what is left of each call is what it passes for the binders, `loop(n, 0)` and `loop(i - 1, s + u)`: the figure's output.

In general, the body of a block in the output is its body as the last round's walk simplified it. The parameters of the block are its binders, and a call passes, for each binder, the term that matched it. A binder is kept only if it is *used*: its value is needed by a case that remains, by a result, by a constructor value that the output builds, or by the argument for a used parameter of a callee (the least such set). A plain name of another block, like `u` above, is not passed. A block that is called from a single jump in another block is inlined there, so no code is copied. The entry keeps all its parameters and is not inlined.

Why a name of another block is in scope. The blocks of the input are closed (§2.1), yet `loop`'s pattern uses `u`, a parameter of `main`, and so does `loop`'s body in the output. This is safe because the pattern is stable. Every call of `loop` passes the term `u` where the pattern has `u`. A block can pass `u` only if it binds `u` or has `u` in its own pattern, and in the second case the same holds of its callers. So every path of calls from the entry to `loop` goes through `main`, and when `loop` runs, `u` is the parameter of the last run of `main` on that path. As in SSA, a block may use the variables of the blocks that dominate it; here the algorithm finds the dominance itself. The rewrite nests `loop` inside `main`, where `u` is in scope. Leißa and Griebler [50] scope SSA for higher-order programs in this way, by the variables that a block uses and not by control flow. Terms carry names from block to block only during the analysis. When the output runs, a call passes values for the callee's binders, and each is computed in the caller, where the names it uses are in scope.

3 Functions as Values

3.1 Continuations and closures

We put programs in *continuation-passing style* [3, 44]. A function takes an extra parameter, written after a semicolon: its *continuation*, the block to jump to with the result. A result e becomes a jump $k(e)$ to that parameter, and the rest of the caller after a call becomes a separate block, `k`, which the caller passes (on the right):

```
main(?y) = let p = dup(y) in p.0 - p.1
dup(?s) = (s, s)
```

```
main(?y; ?ret) = dup(y; k⟨ret⟩)
dup(?s; ?r) = r((s, s))
k⟨?ret⟩(?p) = ret(p.0 - p.1)
```

Everything is now jumps: a call, a return, a loop and a branch are all jumps to blocks, so one analysis rule can cover all control flow, within a function and across functions.

The block `k` uses `ret`, so it takes `ret` as a parameter, which we write in angle brackets, and `main` supplies it when it passes `k`. We treat lambdas in the same way. Instead of having lambdas in the source, we extract all lambdas $\lambda(p_1, \ldots, p_n).\text{body}[x_1, \ldots, x_n]$ to definitions $f\langle x_1, \ldots, x_n\rangle(p_1, \ldots, p_n) = \text{body}[x_1, \ldots, x_n]$, where x_i are the free variables. The expression $f\langle x_1, \ldots, x_n\rangle$ then allocates the closure, and we call $x_1, \ldots, x_n$ its *captures*.

A closure is data, so we allow a pattern to match on it, similar to how patterns can already match on data constructors: `add⟨5⟩` matches only a closure of the block `add` that captured 5, and `add⟨?y⟩` matches any closure of `add` and binds what it captured.

A jump through a variable, such as `f(1; r)`, does not know what block will run. If the enclosing block's pattern has a closure for `f`, substitution puts that closure where `f` was, and the jump becomes a direct call of the closure's block, with the captures and then the jump's arguments. Like any call, it must match that block's pattern. Consider a function passed to another function and called there, `main(?u) = app(λ?x. x + u)` with `app(?f) = f(1)`. We call the λ's block `inc`:

```
main(?u; ?ret) = app(inc⟨u⟩; ret)
app(?f; ?r) = f(1; r)
inc⟨?u⟩(?x; ?k) = k(x + u)
```

The only call of `app` is `main`'s, so `app`'s pattern is `app(inc⟨u⟩; ret)`. Substituting it into `f(1; r)` gives `inc⟨u⟩(1; ret)`, which is a call of `inc`, and `inc`'s body `k(x + u)` becomes `ret(1 + u)`. Neither `app`'s nor `inc`'s pattern has a binder, and each has one call, so the output is `main(?u; ?ret) = ret(1 + u)` (§2.6). We've avoided building the closure entirely.

The same step applies to returns. In the first listing, `dup`'s continuation is always `k⟨ret⟩`, so its return becomes a call of `k`; this is contification [31]. That call gives `k` the clause `k⟨ret⟩((y, y)) = ret(0)`, in which the pair is part of the pattern and is therefore never built (Figure 1 shows this clause without `ret`).

A closure *escapes* when we lose track of it: the program returns it, passes it to code that we do not analyze, or passes it where a pattern has only a binder. Code that holds an escaped closure may call it with any arguments. We treat this as a further call of the closure's block, with what is known of the closure's captures and unknowns for the other parameters. In short:

(1) Closures $f\langle x_1, \ldots, x_n\rangle$ are treated just like data constructors $F(x_1, \ldots, x_n)$
(2) A call to a known closure becomes a direct call
(3) When a closure escapes, that counts as a fictitious call with unknown arguments

These rules turn the algorithm into a higher-order analysis that can eliminate closures by passing the varying parts of their captures instead, and turn indirect calls into direct calls.

3.2 Closures in the algorithm

With closures the target of a jump is a term c, and line 8 of Figure 5 becomes the first three lines below.

```
15 at each jump c(ā) that the walk arrives at:
16   c is a closure g⟨f̄⟩:  call(g, f̄ ā)
17   c is not a closure:  every closure in ā escapes
18 escape(g⟨f̄⟩):         call(g, f̄ with its names unknown, unknowns)
19 a closure that a join (lines 6, 12) or θ (line 13) puts into a binder escapes
```

A jump through a closure $g\langle\bar{f}\rangle$ is a call of g with the captures $\bar{f}$ and then the jump's arguments (line 16); a jump to a label is the case without captures. If the target is not a closure, the block that will run is unknown, and each closure in the arguments escapes (line 17). A closure also escapes when a join or the match of line 13 puts it, or a term that contains it, into a binder (line 19). An escaped closure counts as a call of its block (line 18), with unknowns for the other parameters, and

with an unknown for each binder or plain name in the captures (the same unknown for the same name). What is known of the captures, such as a constant or two equal captures, therefore stays in the block's pattern. The names become unknowns because an escaped closure may be called after the blocks that bind them have finished. The block's pattern then has binders at those captures, and the closure the output builds stores their values. A closure that does not escape needs no unknowns: each jump through it has the closure's term as its target, so the names in its captures are in scope at the jump, as in §2.6. Definition 2.1 counts the calls of escaped closures as reached too, and each must match the pattern of the block it calls. The join treats a closure like a constructor value: two closures of the same block are joined field by field.

In the rewrite (§2.6), a closure is built where the source builds it, and only if it escapes; it stores the values that its block's pattern binds at the captures. A binder is also used if a jump whose target is not known to be a closure needs it, or if a closure that the output builds stores it. The block of an escaped closure keeps all its parameters after the captures and is not inlined, because code that we do not analyze may call it.

We give two examples, written without continuations. In the first, `sum` passes a closure that has captured `c` to `fold`, a higher-order fold over a range of integers:

```
sum(?c, ?n) = fold(scale⟨c⟩, 0, 1, n)                scale⟨?c⟩(?s, ?i) = s + c * i
fold(?f, ?s, ?i, ?n) = if i > n then s else fold(f, f(s, i), i + 1, n)
```

Every call of `fold` passes `scale⟨c⟩` for `f`, so `fold`'s pattern is `fold(scale⟨c⟩, ?s, ?i, n)`, with binders only for `s` and `i`. Under it `f(s, i)` is a direct call of `scale`. The output is a loop over numbers that uses `sum`'s `c` and does not build the closure:

```
sum(?c, ?n) = fold(0, 1)
  where fold(?s, ?i) = if i > n then s else fold(s + c * i, i + 1)
```

In the second example the closure changes on every iteration of a loop. A stream is a closure that returns a pair of its first value and the stream of the rest: `from⟨a⟩` is the stream of `a`, `a + 1`, and so on. The loop `sum` adds up the first `i` values of a stream `g`:

```
main(?n) = sum(from⟨1⟩, n, 0)                          from⟨?a⟩() = (a, from⟨a + 1⟩)
sum(?g, ?i, ?acc) = if i == 0 then acc else let p = g() in sum(p.1, i - 1, acc + p.0)
```

In the first round `sum`'s pattern is the arguments of `main`'s call. Under it `g()` is a direct call of `from`, and its return is known, as `dup`'s was in §3.1: `p` is the pair `(1, from⟨2⟩)`. The recursive call is therefore `sum(from⟨2⟩, n - 1, 1)`, which refutes the pattern. Both calls pass a closure of `from`, so the join keeps `from` and puts a binder at its capture: the second round's pattern is `sum(from⟨?a⟩, ?i, ?acc)`. Now `p` is `(a, from⟨a + 1⟩)`, and the recursive call is `sum(from⟨a + 1⟩, i - 1, acc + a)`, which matches. The output passes the capture where the source passed the stream:

```
main(?n) = sum(1, n, 0)
  where sum(?a, ?i, ?acc) = if i == 0 then acc else sum(a + 1, i - 1, acc + a)
```

No closure and no pair is built.

4 Specialization Policies

4.1 More than one pattern for a block

The fold of §3.2 became a loop over numbers because it had one caller, so its pattern had the function that it was passed. Here is the same fold, used twice:

```
fold(?f, ?s, ?i, ?n) = if i > n then s else fold(f, f(s, i), i + 1, n)

stats(?n) = let p = fold(λ(?s, ?i). (s.0 + i, s.1 + i * i), (0, 0), 1, n) in
```

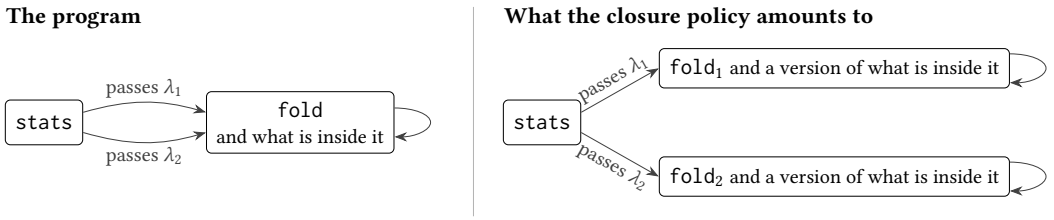


Fig. 6. What a specialization policy amounts to (arrows are calls). On the left both calls flow into `fold`, and its pattern is their join. With the closure policy, a call goes to a version of `fold` for the closure that it passes, made when the algorithm first reaches such a call. On the right, fewer calls flow into each block.

```
let h = fold(λ(?h, ?i). h * 31 + i, 0, 1, n) in
(p.1 - p.0) + h
```

With one pattern for each block, this fold does not become such a loop. Both of `stats`'s calls flow into `fold`, and its pattern is their join. `f` is one λ at the first call and another at the second, so the pattern has a binder there and the jump through `f` is not a direct call. Likewise `s` is a pair and then a number, and the two calls return to different places.

Had the program been written with two copies of `fold`, one per call, both copies would again have one caller, and their patterns would have the functions they are passed. A *specialization policy*, or policy for short, says which part of a call is its *context*. We give a block a copy per context, a *version* of the block, and send a call to the version for its context (Figure 6). A version is a block like any other. Its pattern is the join of the calls that go to it, and it is made when a round first reaches a call with its context.

With the *closure policy*, the context of a call consists of the labels of the closures that it passes, other than its continuation, and not their captures. `stats`'s two calls pass λ_1 and λ_2, so they go to two versions of `fold`, and both versions get a clause (`s1` and `s2` are `stats`'s two continuations):

```
fold(λ1, (?a, ?b), ?i, n; s1) = if i > n then s1((a, b))
    else fold(λ1, (a + i, b + i * i), i + 1, n; s1)
fold(λ2, ?h, ?i, n; s2) = if i > n then s2(h)
    else fold(λ2, h * 31 + i, i + 1, n; s2)
```

The first version returns by a jump to `s1`, which starts the second loop. Its argument has the pattern `(a, b)`, so the pair is never built. With the matches deleted, the two clauses are two loops over numbers, as written by hand:

```
stats(?n) = fold1(0, 0, 1)
  where fold1(?a, ?b, ?i) = if i > n then fold2(0, 1)
                            else fold1(a + i, b + i * i, i + 1)
        where fold2(?h, ?i) = if i > n then (b - a) + h
                              else fold2(h * 31 + i, i + 1)
```

4.2 How a call finds its version

The version that a call goes to is its *route*. The algorithm finds routes from the program and policy, not the patterns: it walks the body of a version knowing only its context, and takes the context of each call from that walk. In the version of `fold` for λ_1, `f` is λ_1. The recursive call passes `f`, so it has the same context and goes back to the same version. The blocks that `fold` is made of get versions for λ_1 in the same way (Figure 6). A closure that is known only from a pattern, for example one

Table 2. The six specialization policies, all with the same fixed point. The in-place and value policies are used with a bound on versions (§4.2).

A version per …(policy)	The algorithm is then	Example
block (constant)	a dataflow analysis	§2, §3
closure passed (closure)	a higher-order specializer	`fold` becomes two loops
constructor passed (constructor)	a call-pattern specializer [66]	a loop per state
closure and continuation (return)	the same, per caller	a result pattern per return point
closure and call site (in-place)	an inliner	each call has its own version
value of marked parameters (value)	a partial evaluator	an interpreter is specialized away

returned by another call, is not in the context. A closure gets a version of its block where it is built, the *closure's version*, as if building it were a call that passes only the captures.

Routes could use the patterns instead, and the output would still be correct. But then a pattern that becomes more general can move a call to another version, and the version the call leaves can get a more specific pattern. A program can then have two stable assignments, neither at least as specific as the other, and the order of the walk decides which is found (Appendix C). With routes that do not use the patterns, versions and their routes form a program (the right of Figure 6), and the algorithm runs on it as on any other, as long as every jump through a closure goes to the closure's version. The theorems of §5 are for one version per block; we state none for policies.

With versions in place of blocks, line 16 becomes

```
20   c is a closure g⟨f̄⟩:  call(v, f̄ ā), v the jump's route, or else the closure's version
```

A jump to a label, or through a closure built in the same block, always has a route. Any other jump through a variable has a separate route, a version of the closure's block for a context from the jump, only if the context of the jump's version gives the closure's label and the policy also uses another part of the jump: a closure or constructor that it passes, its continuation, its site, or a marked value (below). Otherwise it goes to the closure's version. In `fold`'s version for λ_1, the closure policy uses nothing else of `f(s, i)`, which goes to the closure's version in `stats`; with the return policy, its continuation gives it a separate route. The jump takes this route only if the pattern of the jump's version has the closure. With a binder there, as when closures of λ_1 from two versions of `stats` meet, the closure escapes, a call of the closure's version. So the patterns decide which version gets the closure's calls, and the patterns found are stable but need not be the most specific.

The reference implementation has six policies (Table 2). The *constant* policy uses nothing, so a block has one version. The *constructor* policy adds to the closure policy the constructor of each argument. The *return* policy adds the label of the continuation, so a function has a version per return point. With these four a context holds at most one label or constructor name per parameter, so a block has finitely many versions and the search terminates as in §2.4. This is why the closure policy leaves out the captures: a loop that wraps its closure in a new closure on every call would otherwise get a version per call. The *value* policy adds to the return policy the values of the parameters that the program marks (with `@[specialize (x)]` on their block). The *in-place* policy adds to the closure policy the call site, for blocks that are not recursive. These two are used with a bound on the number of versions of a block, past which a call goes to the version with the empty context.

5 Theory

We proved the theorems of this section in Lean 4 [27]. They are about a core of the IR of Figure 3, in continuation-passing style (§3.1), with constructors, closures and operations. We first say how a

program runs and how the algorithm runs in Lean, and then state three theorems. The appendix has the definitions and the statements in full (Appendices B.3 and B.5).

How a program runs. The semantics is a machine. Its state is the block that is running, with values for the block's captures and parameters. A step either evaluates an equality test of the block's body and continues in the branch taken, or does what the body ends in: it halts with a value, calls a block with new values, or jumps through a closure, which runs the closure's block with its captures. An operation on values is evaluated by the rewrite rules that the analysis uses. A program has *entries*, each a call of its entry block with given values. The machine is deterministic, so a run from an entry either halts with a value or never halts.

How the algorithm runs. The Lean formalization leaves the order of steps open. It defines three kinds of steps on the state of Figure 5, and a *run* of the algorithm is any sequence of them. A *forward* step joins the arguments of a reached call into the F of the block it calls. A *walk* step makes a block's F its pattern for the round. A *restart* takes any set of calls containing one that does not match, joins each into the pattern of the block it calls, keeps the skeletons of the results, and discards the rest of the round. A join names each new binder after a place where the binder occurs. The algorithm of Figure 5 is one such run: it restarts at the end of a round, with all the calls that did not match. When no step is possible, every reached call matches the pattern of the block it calls, so the patterns are stable (Definition 2.1).

Theorem 5.1 (Stable patterns hold when the program runs). *If the patterns of a program are stable, then at each step of a run from an entry, the values of the running block's captures and parameters are an instance of the block's pattern.*

Theorem 5.2 (Stable patterns are safe to use). *If the patterns of a program are stable, then from each entry the program rewritten by them, with the matches deleted, the decided tests removed and the blocks nested (§2.6), halts with the same value as the program, or neither halts.*

Theorem 5.2 holds however the patterns were found, so a compiler can find them in any way, check that they are stable, and use them.

Patterns P are *at least as specific* as patterns Q if the blocks reached under P are reached under Q, and a single substitution turns the patterns of Q into those of P.

Theorem 5.3 (The algorithm finds the most specific stable patterns). *Take a program with finitely many blocks and entries. Every run of the algorithm terminates with patterns that are stable, and they are at least as specific as any stable patterns of the program.*

Theorems 5.1 and 5.2 are proved by a simulation, and Theorem 5.3 by showing that the patterns of a round stay at least as specific as any stable patterns, as §2.5 explains.

The theorems are about the Lean formalization of the algorithm; the OCaml reference implementation and our compiler's optimizer we test. The theorems are also for one pattern per block. We have not proved them for specialization policies, and for some policies Theorem 5.3 fails: with a policy that gives a jump through a closure a separate route (§4.2), a program can have no most specific stable patterns, and what the algorithm finds can depend on the order of its steps. A theory of policies is future work.

6 Use Cases

We look at the algorithm in three settings: as the only optimizer of a compiler, as an addition to two mature optimizers, and compared with the published analyses that it is related to. Appendices D and E have the measurements and the examples of other papers.

6.1 Eliminating abstractions

Our compiler for a dependently typed language uses stable patterns as the only optimization (before LLVM). It has no separate monomorphization, inlining, constant propagation, unboxing or closure optimization. Types are values in its IR, so monomorphization is just a specialization policy on type arguments: it specializes a generic function to the types it is used at.

We wrote twelve small programs twice: *with* an abstraction (e.g., a fold, a stream, a monad, or a dictionary of functions) and as a loop written by hand. For 9 of the twelve, the two programs compile to the same IR up to names and to the same assembly. In the other three, a pull stream and the option monad keep their constructors in registers, and a loop in a state monad is still far slower than a handwritten one; making it fast is future work.

With a specialization policy on the values of marked parameters, the compiler can turn an interpreter applied to a constant program into an efficient loop: the interpreter, the program and the program counter are gone. Our benchmarks that pass closures, use generic code or interpret a constant program execute up to 295× fewer instructions than without the optimizer.

6.2 Ports to mature optimizers

Flambda 2 [64], the optimizer of OxCaml, and Chez Scheme already optimize well, so we asked what the algorithm adds to them, and ported two parts of it into each. The first is the closure policy of §4.1, for direct calls to functions that the host compiler does not inline. The second is refutation, for the shapes of loop parameters: a parameter of a loop is assumed to have the shape of the values that enter the loop, for example a boxed float, until a back edge passes a value of another shape.

In Flambda 2 at `-O3`, the closure policy alone makes Menhir take 18.8–24.0% less wall time (inlining a critical function does the same), and angstrom 18.1–25.8% less. Four of 27 Sandmark [65] programs take 21–73% less wall time, three numerical ones through loop shapes and one through versions, and the others change by at most 2.7%. In Chez Scheme, at its default optimize-level, `mbrot` of the r7rs-benchmarks runs 10.5× faster: its loop variables are flonums (boxed floats) because the back edge passes flonums. This is the circular dependence of §1. Most programs' running times, however, do not change much. This matches the literature: optimistic analysis typically has only modest benefits in practice [9, 19], although in certain cases it is critical.

The ports do cost compile time. The Flambda 2 port adds 11% to Menhir's compile time and 9.0% to its code size, and up to 64% extra compile time for some other programs, mostly because register allocation takes twice as long on a large function; the analysis itself is cheap. The Chez port adds 9% compile time (Table 8). These are for illustration purposes; we do not claim that these are industrial strength implementations.

6.3 How often published analyses find the most specific patterns

The most specific stable patterns are our own measure of an analysis. Published analyses were designed with other aims, so as an experiment we measured how often they nevertheless find them. We implemented seven of them on our IR: Simpson's value numbering in reverse postorder (RPO), Cooper and Simpson's over strongly connected components (SCC) [79], the SSA construction of Braun et al. [8], Lemerre's analysis [54], Flambda 2 [64], SCCP [85] and Click's combined analysis [18, 19]. We ran them on generated small scalar programs and small programs with constructors. For each program, a brute force search finds the most specific stable patterns (the optimum of §2.5), and Figure 7 shows how often the result of an analysis, written as patterns, is that optimum.

Click's combined analysis finds the optimum on all scalar programs, but misses it on some with constructors. Lemerre's analysis misses it on 0.7% of the scalar programs, where it keeps a

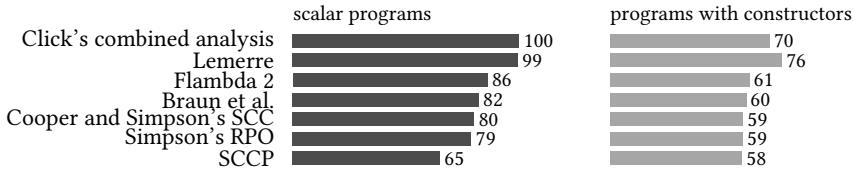


Fig. 7. Share of programs (%) on which each published analysis finds the most specific stable patterns. Ours is not shown: it finds them on every program, as Theorem 5.3 predicts.

redundant ϕ. SCCP, value numbering and Braun et al.'s algorithm miss it on 1.2% of them when run in turn [18] to a fixed point, in the best order for each program. A miss is not an error: the results of these analyses are correct, and less specific by our measure. As Theorem 5.3 predicts, our own implementation finds the optimum on all of them, whatever the order of the program's functions.

We also ran the reference implementation on 43 examples that the papers and manuals of these and other analyses use to show what they do. For 33 of them the output is the optimized program they expect: for 21 with one version per block, and for 12 more with a policy (all but one of those copy code too). Most of the rest need a fact from a test (§7) or from the heap, which patterns do not express yet; extending them to such facts is future work.

7 Related Work

Optimistic analyses within a function. The closest work is Lemerre's [54]. He presents SSA translation, optimistic global value numbering, SCCP and loop-invariant code motion as a single abstract interpretation [21] of a function. Where control flow merges, his analysis keeps a symbolic expression if the incoming paths agree on it, and otherwise introduces a variable for the value; these variables are the ϕs. Ours combines the first three, and differs in three ways. Its patterns contain constructors, so it unboxes. It analyzes across functions and closures, and makes versions. We prove that, under the conditions of §5, its result is the most specific in any order of the walks. Lemerre's paper states that its translation has the classic minimality property of SSA, and remarks that the choice of iteration strategy does not affect the result [54, §1; extended version, Appendix E]. We found that neither statement holds of his artifact [53] (Appendix D): on one program of a dozen lines the iterative strategy places a ϕ that the recursive strategy does not, and on another both place $x_2 = \phi(x_1, x_2)$, where minimal SSA [24] has no ϕ. The outputs of the artifact are correct. In our terms such a ϕ is a parameter that the analysis introduced in an earlier iteration and kept. Our analysis makes a binder only where a call contradicts a pattern, and it discards what was computed under the refuted pattern (§2.5), so no binder is left over from an earlier round. By Theorem 5.3 the result is the most specific stable assignment for any order of the walks, and it has no redundant set of parameters. Lemerre's analysis also moves loop-invariant computations, which ours does not. Lesbre and Lemerre [56] also compute an analysis and a transformation in one fixed point. Nexis [74] proves in Lean that two forms of code motion are optimal.

Click and Cooper [18, 19] showed that, when transfer functions are monotone, the fixed point of a combined optimistic analysis is at least as good as any sequence of the separate analyses. Ours is a combined analysis in this sense, over a domain that also has constructors, closures and plain names. Lerner, Grove and Chambers [55] compose analyses that are written separately; we have one analysis, because all our facts are patterns.

Facts from tests. Wegman and Zadeck observe [85, §2.2] that inside a branch an analysis can use the fact that the test succeeded, for example that $x = 0$ in the true branch of `x == 0`. A pattern describes the arguments that calls pass to a block, and the outcome of a test is not an argument, so

our patterns do not carry such facts yet. SCCP with them finds a constant that is not in O_{pat} on 8.1% of our scalar programs and 2.1% of those with constructors. Versions can express them. When the true branch of a test is a version that is passed what the test established, our analysis finds these constants on the scalar programs. Keeping a most specific result in that setting is future work.

Herbrand equalities. Kildall [45], Gulwani and Necula [36] and, across procedures, Müller-Olm, Seidl and Steffen [63] optimistically compute the Herbrand equalities of a program, those that hold with operations uninterpreted. They find equalities with operations: in a loop that starts with `(a, b)` equal to `(u, u + 1)` and passes `(a + 1, b + 1)`, they find $b = a + 1$, and our output passes two parameters. Our join does not look inside operations, so its result does not depend on the order of the walks. Conversely, we evaluate the tests whose outcome our patterns determine, and these analyses treat tests as nondeterministic.

Equality saturation. Equality saturation [81, 86] represents in an e-graph the forms of a program that its rewrite rules can produce. Our analysis keeps a single form, the program simplified under its patterns. Arbore, Cheung and Willsey [5] add optimistic reasoning to equality saturation within a function, and so prove that two loop variables are equal when they start equal and are updated in the same way. Our analysis gives two such variables one binder (§1), also across functions.

Cloning, and versions by context. SpecConstr [66] makes a specialized copy of a recursive function for each constructor pattern with which the function's own body calls it; with the constructor policy, our analysis produces that paper's results on two of its three examples. Procedure cloning [20], GCC's IPA-CP [33], LLVM's function specialization [59], selective specialization [28] and customization [14, 15] choose what to clone from interprocedural constants [12] and other facts at call sites, from profiles, or from receiver types. A policy, like the choice of contexts in k-CFA [78], is a fixed rule. Our analysis finds contexts as it runs, but not from the patterns. A context holds only the parts of a call the policy uses, such as labels of the closures the call passes, so within a version we still join calls and refute patterns. Basic block versioning [16, 17] makes a version of a block for each type context that reaches it; its context is what is known on entry to the block, so no join takes place within a version.

Deciding what to specialize. Supercompilers [41, 80, 82] and online partial evaluators [40] generalize a growing configuration against an earlier one, using the least general generalization [58, 67], which that literature calls the most specific generalization. Our join is also the most specific generalization, applied when a call does not match the pattern of its block. These systems are polyvariant: a function gets a specialized copy for each configuration, and which copies are made depends on when generalization is applied, so their results are not ordered by one notion of best. A monovariant abstract interpretation of a logic program [10], which has also been combined with partial deduction [57], keeps one description of the calls of a predicate, and computes the most specific one as the least fixed point of a monotone analysis, in any order of iteration. Our analysis is monovariant in this sense, but its patterns can also name a variable of another block, so that the block is not passed that variable. With such names the analysis is not monotone, and the order of iteration affects the result of the usual fixed-point computation (§2.5). The most specific stable patterns still exist, and the restarts find them (Theorem 5.3). AnyDSL [49], which is built on Thorin [51], decides at each call site which parameters to specialize, using filters that the programmer writes or that it infers; in MimIR [52] a filter decides whether a call is inlined. In our analysis, the policy determines which calls of a block go to the same version, and the analysis then runs on each version. Specializing an interpreter to the program that it runs is the first Futamura projection [32], which Truffle does by partial evaluation in a production virtual machine [87]. Like the value policy, weval [29] clones the blocks of an interpreter for each value of a context that the interpreter sets.

Production optimizers. Among production compilers, Flambda 2 [48, 64] is the closest. When several jumps go to one continuation, it joins the information that it has about their arguments: if two parameters receive the same value at each jump it records that they are equal, and if a parameter always receives the same variable of an enclosing scope it records that the parameter is that variable. A pattern expresses these facts by one binder at two positions and by a plain name of another block. Flambda 2 does this for continuations that are not recursive, and treats the parameters of a loop as unknown, except for those that it finds invariant. Our analysis adds policies and optimism about loop parameters, and our port adds a closure policy and the refutation of loop shapes to Flambda 2 (§6.2).

Fusion. Stream fusion [22] removes the intermediate streams of a pipeline with rewrite rules that a library supplies, together with the compiler's general optimizations. Our analysis removes the stream of §3.2 with no rule about streams. On that paper's `append` of two streams it makes one loop, with the return policy, but the loop still builds its step values, and on `concatMap` the inner stream is still built. Removing these too is future work.

Assume and refute. Houdini [30] is given a program and a finite set of candidate annotations. It assumes all of them and repeatedly removes those that a verifier refutes, until the rest are inductive [46]. Our analysis also assumes and refutes, with two differences. It builds its candidates by joining the arguments of calls. And the program that it checks changes: a round simplifies the program under the current assumptions, so a refutation changes which calls are checked.

Fixed-point solvers. A generic local solver [76] keeps the values that it has computed, and when a value changes it evaluates again what depends on that value. §2.5 gives an example on which this does not compute the most specific patterns. The reason is that a pattern can contain the name of a variable of another block: when a block's old pattern is joined with the arguments of a new call, such a name can be lost, although it would be present if the pattern were computed again from the calls. Amato et al. [2] restart when narrowing lowers the value at a loop head, and lose the guarantee of termination. Ours restarts each round with only its refutations, and terminates, as a pattern can be refuted finitely often.

Closures. Defunctionalization [73] and MLton's flow-directed closure conversion [13] also represent a closure as a constructor whose tag identifies its code, and compile an indirect call to a dispatch on the tag. We keep closures as terms during the analysis, and make a call direct when its target normalizes to a closure. A closure in a pattern has one label. Flow analyses such as lambda-set specialization [7], webs [68] and the closure optimization of Reppy, Shivers and Zhong [70] keep a set of labels, and so still resolve a call when closures of two blocks reach the same parameter. Patterns that hold a set of closures are future work; today a policy separates such calls into versions (§4).

Returns. In our analysis a return is a jump through the continuation parameter; for control flow this is as precise as direct style in a monovariant analysis [25, 61]. For data flow, the result of a function has one pattern for all of its callers, unless a policy gives it a version for each return point; CFA2 [83] and pushdown CFA [34] match each return with its call [71, 77].

The optimizations that patterns give. The kinds of pattern correspond to optimizations that have their own literature. A returned constant is what the return jump functions of interprocedural constant propagation compute [12], and a returned constructor that is not built is the constructed product result [6]. A pair passed as its fields is interprocedural unboxing [35], and a known continuation is contification [31, 44]. A closure that disappears because its variables are in scope is lambda dropping [26], and one reduced to its captures is lambda lifting [39]. A case on a constructor

that every caller passes is the case of a known constructor [60], across a call and without inlining. And a translation to SSA may pass all variables to all blocks and leave the rest to the algorithm, which removes the redundant ϕs, including those that are redundant only as a group, for which Braun et al. [8] need a pass over strongly connected components.

8 Conclusion

Constant propagation, value numbering, SSA construction, unboxing, contification and the control-flow analysis of closures are usually separate analyses. We have shown that they can be unified using *stable patterns*. The most specific stable pattern exists, and we can find it by refutation: assume the best, simplify under it, and weaken only when a newly found call contradicts us. A pattern can contain a closure, and then the analysis is interprocedural and higher-order: it works across functions, calls known closures directly, and eliminates closure allocations. A block can have several patterns, and then the same fixed point is a specializer of higher-order functions, an inliner, or a partial evaluator. There are interesting avenues for future work, such as taking into account if-conditions, extending the analysis to heap values, more precise analysis of sets of closures (as in k-CFA), full Herbrand equalities, and finding refutations in a more incremental way, rather than starting afresh each round.

AI Use

AI coding agents (Anthropic's Claude and OpenAI's Codex) wrote, under the authors' direction and review, the reference implementation, the compiler's specialization pass, the Lean development, the comparison harness and its models, the ports, the demo, and the experiment scripts. Lean's kernel checks the proofs.

Data-Availability Statement

The supplementary material of this submission is anonymized and holds the Lean development (five files, Lean 4.35, no `sorry`; axioms `propext`, `Quot.sound` and `Classical.choice`), which `lake build` checks, and a one-page demo that runs the algorithm on the paper's examples in a browser. An artifact exists beyond this, and we will submit it for artifact evaluation: the reference implementation of §2 with its tests, and every example of §1 to §4 and of Appendix A as a file with what the implementation prints; the comparison harness, its models of the published algorithms, and the raw output from which Figure 7 is generated; the measurements behind every other table; and the scripts that regenerate every table in this paper from those files. The compiler with the free-abstraction programs, and the OCaml and Chez Scheme patches, are not part of that artifact; they are available on request for artifact evaluation.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting Equality of Variables in Programs. In *Symposium on Principles of Programming Languages (POPL)*. 1–11. doi:10.1145/73560.73561

[2] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. 2016. Efficiently Intertwining Widening and Narrowing. *Science of Computer Programming* 120 (2016), 1–24.

[3] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

[4] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20. doi:10.1145/278283.278285

[5] Russel Arbore, Alvin Cheung, and Max Willsey. 2026. Optimism in Equality Saturation. *Proceedings of the ACM on Programming Languages* 10, PLDI (2026). doi:10.1145/3808302

[6] Clem Baker-Finch, Kevin Glynn, and Simon Peyton Jones. 2004. Constructed Product Result Analysis for Haskell. *Journal of Functional Programming* 14, 2 (2004), 211–245. doi:10.1017/S0956796803004751

[7] William Brandon, Benjamin Driscoll, Frank Dai, Wilson Berkow, and Mae Milano. 2023. Better Defunctionalization through Lambda Set Specialization. *Proc. ACM Program. Lang.* 7, PLDI, Article 146 (2023), 24 pages. doi:10.1145/3591260

[8] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction (CC) (LNCS, Vol. 7791)*. 102–122. doi:10.1007/978-3-642-37051-9_6

[9] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724.

[10] Maurice Bruynooghe. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming* 10, 2 (1991), 91–124. doi:10.1016/0743-1066(91)80001-T

[11] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. doi:10.1145/321239.321249

[12] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. 1986. Interprocedural Constant Propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. ACM, 152–161. doi:10.1145/13310.13327

[13] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Programming Languages and Systems, 9th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 1782)*. Springer, 56–71. doi:10.1007/3-540-46425-5_4

[14] Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 146–160. doi:10.1145/73141.74831

[15] Craig Chambers and David Ungar. 1990. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 150–164. doi:10.1145/93542.93562

[16] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 37)*. 101–123. doi:10.4230/LIPIcs.ECOOP.2015.101

[17] Maxime Chevalier-Boisvert and Marc Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *European Conference on Object-Oriented Programming (ECOOP) (LIPIcs, Vol. 56)*. 7:1–7:24. doi:10.4230/LIPIcs.ECOOP.2016.7

[18] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (1995), 181–196. doi:10.1145/201059.201061

[19] Clifford Noel Click, Jr. 1995. *Combining Analyses, Combining Optimizations*. Ph. D. Dissertation. Rice University.

[20] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. 1992. Procedure Cloning. In *International Conference on Computer Languages (ICCL)*. 96–105. doi:10.1109/ICCL.1992.185472

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL)*. 238–252. doi:10.1145/512950.512973

[22] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 315–326. doi:10.1145/1291151.1291199

[23] Cranelift project, Bytecode Alliance. 2026. Cranelift IR Reference. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/ir.md. Accessed 2026-11-03 (file last changed in commit `fed404610`).

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. doi:10.1145/115372.115320

[25] Daniel Damian and Olivier Danvy. 2003. Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation. *Journal of Functional Programming* 13, 5 (2003), 867–904. doi:10.1017/S0956796802004379

[26] Olivier Danvy and Ulrik P. Schultz. 2000. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *Theoretical Computer Science* 248, 1–2 (2000), 243–287. doi:10.1016/S0304-3975(00)00054-2

[27] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Conference on Automated Deduction (CADE) (LNCS, Vol. 12699)*. 625–635. doi:10.1007/978-3-030-79876-5_37

[28] Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 93–102. doi:10.1145/207110.207119

[29] Chris Fallin and Maxwell Bernstein. 2025. Partial Evaluation, Whole-Program Compilation. *Proc. ACM Program. Lang.* 9, PLDI, Article 160 (2025), 24 pages. doi:10.1145/3729259

[30] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity (FME) (LNCS, Vol. 2021)*. 500–517. doi:10.1007/3-540-45251-6_29

[31] Matthew Fluet and Stephen Weeks. 2001. Contification Using Dominators. In *International Conference on Functional Programming (ICFP)*. 2–13. doi:10.1145/507635.507639

[32] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. doi:10.1023/A:1010095604496 Reprint of the 1971 paper.

[33] GCC developers. 2025. Interprocedural constant propagation (`ipa-cp.cc`). GCC 15.1.0, `gcc/ipa-cp.cc`, https://gcc.gnu.org.

[34] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-Flow Analysis for Free. In *Symposium on Principles of Programming Languages (POPL)*. 691–704. doi:10.1145/2837614.2837631

[35] Neal Glew and Leaf Petersen. 2012. Type-Preserving Flow Analysis and Interprocedural Unboxing (Extended Version). arXiv:1203.1986

[36] Sumit Gulwani and George C. Necula. 2004. A Polynomial-Time Algorithm for Global Value Numbering. In *Static Analysis Symposium (SAS) (LNCS, Vol. 3148)*. 212–227. doi:10.1007/978-3-540-27864-1_17

[37] Sumit Gulwani and George C. Necula. 2007. A Polynomial-Time Algorithm for Global Value Numbering. *Science of Computer Programming* 64, 1 (2007), 97–114. doi:10.1016/j.scico.2006.03.005

[38] Fritz Henglein and Jesper Jørgensen. 1994. Formally Optimal Boxing. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 213–226. doi:10.1145/174675.177874

[39] Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, Vol. 201)*. Springer, 190–203. doi:10.1007/3-540-15975-4_37

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

[41] Peter A. Jonsson and Johan Nordlander. 2009. Positive Supercompilation for a Higher Order Call-By-Value Language. In *Symposium on Principles of Programming Languages (POPL)*. 277–288. doi:10.1145/1480881.1480916

[42] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7, 3 (1977), 305–317. doi:10.1007/BF00290339

[43] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Workshop on Intermediate Representations (IR)*. 13–22. doi:10.1145/202529.202532

[44] Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *International Conference on Functional Programming (ICFP)*. 177–190. doi:10.1145/1291151.1291179

[45] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages (POPL)*. 194–206. doi:10.1145/512927.512945

[46] Shuvendu K. Lahiri and Shaz Qadeer. 2009. Complexity and Algorithms for Monomial and Clausal Predicate Abstraction. In *Conference on Automated Deduction (CADE) (LNCS, Vol. 5663)*. 214–229. doi:10.1007/978-3-642-02959-2_18

[47] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[48] Vincent Laviron, Pierre Chambart, and Mark Shinwell. 2023. Flambda 2 Types: An Abstract Domain for Static Analysis of Functional Programs. Talk at the ML Family Workshop (ICFP 2023).

[49] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 119 (2018). doi:10.1145/3276489

[50] Roland Leißa and Johannes Griebler. 2026. SSA without Dominance for Higher-Order Programs. *Proc. ACM Program. Lang.* 10, PLDI (2026), 24 pages. doi:10.1145/3808286

[51] Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A Graph-Based Higher-Order Intermediate Representation. In *International Symposium on Code Generation and Optimization (CGO)*. 202–212. doi:10.1109/CGO.2015.7054200

[52] Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2025. MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age. *Proceedings of the ACM on Programming Languages* 9, POPL (2025). doi:10.1145/3704840

[53] Matthieu Lemerre. 2022. SSA Translation is an Abstract Interpretation (artefact). Zenodo. doi:10.5281/zenodo.7152678

[54] Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (2023), 30 pages. doi:10.1145/3571258

[55] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing Dataflow Analyses and Transformations. In *Symposium on Principles of Programming Languages (POPL)*. 270–282.

[56] Dorian Lesbre and Matthieu Lemerre. 2024. Compiling with Abstract Interpretation. *Proc. ACM Program. Lang.* 8, PLDI, Article 162 (2024), 26 pages. doi:10.1145/3656392

[57] Michael Leuschel. 2004. A Framework for the Integration of Partial Evaluation and Abstract Interpretation of Logic Programs. *ACM Trans. Program. Lang. Syst.* 26, 3 (2004), 413–463.

[58] Michael Leuschel and Maurice Bruynooghe. 2002. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming* 2, 4–5 (2002), 461–515. doi:10.1017/S147106840200145X

[59] LLVM developers. 2025. Function specialization pass (`FunctionSpecialization.cpp`). LLVM 20.1.0, `llvm/lib/Transforms/IPO/FunctionSpecialization.cpp`, https://llvm.org.

[60] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3140587.3062380

[61] Jan Midtgaard and Thomas P. Jensen. 2009. Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 287–298.

[62] MLton developers. 2025. MLton: a whole-program optimizing compiler for Standard ML. http://mlton.org; SSA passes `Redundant`, `Flatten`, `LoopInvariant`, `Contify`.

[63] Markus Müller-Olm, Helmut Seidl, and Bernhard Steffen. 2005. Interprocedural Herbrand Equalities. In *European Symposium on Programming (ESOP) (LNCS, Vol. 3444)*. 31–45. doi:10.1007/978-3-540-31987-0_4

[64] OCaml Flambda 2 developers. 2026. Flambda 2: the optimizer of the OxCaml compiler. https://github.com/oxcaml/oxcaml, `middle_end/flambda2`, commit `9cdd33d`. The n-way join of continuation uses is `types/env/join_env.ml`; loops are handled in `simplify/simplify_let_cont_expr.ml`.

[65] OCaml Labs and Tarides. 2024. Sandmark: a benchmark suite for the OCaml compiler. https://github.com/ocaml-bench/sandmark, commit `5605805`.

[66] Simon Peyton Jones. 2007. Call-Pattern Specialisation for Haskell Programs. In *International Conference on Functional Programming (ICFP)*. 327–337. doi:10.1145/1291151.1291200

[67] Gordon D. Plotkin. 1970. A Note on Inductive Generalization. In *Machine Intelligence 5*, Bernard Meltzer and Donald Michie (Eds.). Edinburgh University Press, 153–163.

[68] Benjamin Quiring, David Van Horn, John Reppy, and Olin Shivers. 2025. Webs and Flow-Directed Well-Typedness Preserving Program Transformations. *Proc. ACM Program. Lang.* 9, PLDI, Article 177 (2025), 25 pages. doi:10.1145/3729280

[69] Fabrice Rastello and Florent Bouchez Tichadou (Eds.). 2022. *SSA-based Compiler Design*. Springer.

[70] John Reppy, Olin Shivers, and Byron Zhong. 2026. Flow-Analysis-Based Closure Optimization. *Proc. ACM Program. Lang.* 10, PLDI, Article 184 (2026), 26 pages. doi:10.1145/3808262

[71] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Symposium on Principles of Programming Languages (POPL)*. 49–61. doi:10.1145/199448.199462

[72] John C. Reynolds. 1970. Transformational Systems and the Algebraic Structure of Atomic Formulas. In *Machine Intelligence 5*, Bernard Meltzer and Donald Michie (Eds.). Edinburgh University Press, 135–151.

[73] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference, Volume 2*. ACM, 717–740. doi:10.1145/800194.805852

[74] Martin Rinard. 2026. Program Analysis with Prophecy and History Variables in the Nexis Compiler. arXiv:2607.23033.

[75] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Symposium on Principles of Programming Languages (POPL)*. 12–27. doi:10.1145/73560.73562

[76] Helmut Seidl and Ralf Vogler. 2021. Three Improvements to the Top-Down Solver. *Mathematical Structures in Computer Science* 31 (2021), 1090–1134. doi:10.1017/S0960129521000499

[77] Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, Steven S. Muchnick and Neil D. Jones (Eds.). Prentice-Hall, 189–234.

[78] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. Ph. D. Dissertation. Carnegie Mellon University. Technical report CMU-CS-91-145.

[79] Loren Taylor Simpson. 1996. *Value-Driven Redundancy Elimination*. Ph. D. Dissertation. Rice University.

[80] Morten Heine Sørensen and Robert Glück. 1995. An Algorithm of Generalization in Positive Supercompilation. In *International Logic Programming Symposium (ILPS)*, John W. Lloyd (Ed.). MIT Press, 465–479. doi:10.7551/mitpress/4301.003.0048

[81] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM.

[82] Valentin F. Turchin. 1986. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.* 8, 3 (1986), 292–325. doi:10.1145/5956.5957

[83] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming (ESOP) (LNCS, Vol. 6012)*. 570–589. doi:10.1007/978-3-642-11957-6_30

[84] Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *ACM SIGPLAN Workshop on ML (invited talk)*. 1. doi:10.1145/1159876.1159877

[85] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. doi:10.1145/103135.103136

[86] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 23 (2021). doi:10.1145/3434304

[87] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3062341.3062381

A More optimizations as clauses

Table 3 lists the examples of this paper: what the patterns express in each, the policy it needs, and where it is shown. All outputs are from the reference implementation (`examples/` in the artifact).

Table 3. The examples of this paper. "Constant" is the policy with one version for each block.

What the patterns express	Usual name	Policy	Where
A parameter that is always the same constant	constant propagation	constant	Figure 1
A loop-invariant parameter, with no ϕ	minimal SSA construction	constant	Figure 1
Two parameters that are always equal	value numbering	constant	§1
A pair passed as its fields	unboxing, scalar replacement, argument promotion	constant	§1
A result that is never built	contification	constant	Figure 1, §3.1
A call through a closure that becomes a direct call	lambda dropping	constant	Figure 1, §3.1
Four facts that depend on each other, the last of which LLVM misses	a combined analysis	constant	§1, Figure 4
An invariant around two functions that call each other	copy propagation across calls	constant	Figure 8
A closure reduced to its environment	lambda lifting	constant	Figure 8
A returned constant	constant propagation through returns	constant	Figure 8
SSA from passing every variable to every block	minimal SSA construction	constant	Figure 8
A fold with a closure becomes a loop over numbers	specialization to a function argument	constant	§3.2
A stream (each closure returns the next) becomes a loop variable	closure elimination	constant	§3.2
An escaped closure keeps what is known of its captures		constant	§3.2
A fold used with two functions becomes two loops, with no pair between them	specialization of higher-order functions	closure	§4.1
An interpreter on a constant program becomes that program's loop	the first Futamura projection	value	Appendix D.2
A matcher by derivatives becomes its automaton	partial evaluation	value	below
The examples of other papers on specialization and analysis	call-pattern specialization, customization, cloning	several	Appendix E

Figure 1 showed four programs as clauses. Figure 8 has four more, in the same form.

A matcher that becomes its automaton. A regular expression is matched against a string by taking its derivative by each character in turn [11]:

```
match(?r, ?s) = case s of Nil → nullable(r) | Cons(?c, ?t) → match(deriv(r, c), t)
main(?s) = match(Star(Seq(Chr(a), Chr(b))), s)
```

With the value policy and `r` marked, `match` gets a version for each regular expression that it is called with. There are three, $(ab)^*$, $b(ab)^*$ and the empty set, and in each of them `nullable` and `deriv` are computed away. What is left is the automaton, one loop for each state:

```
main(?s) = m₁(s)
m₁(?s) = case s of Nil → 1 | Cons(a, ?t) → m₂(t) | Cons(b, ?t) → m₀(t)
m₂(?s) = case s of Nil → 0 | Cons(a, ?t) → m₀(t) | Cons(b, ?t) → m₁(t)
m₀(?s) = case s of Nil → 0 | Cons(a, ?t) → m₀(t) | Cons(b, ?t) → m₀(t)
```

	a call	*the block it calls*
A loop invariant, around two functions that call each other		
source	`ev(n, u)`	`ev(?n, ?a) = if n == 0 then a else od(n - 1, a)`
	`od(n, u)`	`od(?m, ?b) = if m == 0 then 0 else ev(m - 1, b)`
clauses		`ev(?n, u) = if n == 0 then u else od(n - 1, u)`
		`od(?m, u) = if m == 0 then 0 else ev(m - 1, u)`
output	`ev(n)`	`ev(?n) = if n == 0 then u else od(n - 1)`
	`od(n)`	`od(?m) = if m == 0 then 0 else ev(m - 1)`
A closure reduced to its environment		
source	`use(add⟨a⟩)`	`use(?f) = f(10)`
	`use(add⟨b + 1⟩)`	`add⟨?y⟩(?z) = z + y`
clauses		`use(add⟨?y⟩) = 10 + y`
output	`use(a)`	`use(?y) = 10 + y`
	`use(b + 1)`	
A returned constant		
source	`let z = count(n)`	`count(?i) = if i == 0 then 0 else count(i - 1)`
	`in z + a`	
clauses		`count(?i; k) = if i == 0 then k(0) else count(i - 1; k)`
		`k(0) = a`
output	`count(n)`	`count(?i) = if i == 0 then a else count(i - 1)`
SSA from passing every variable to every block		
source	`join(x, y, 1)`	`join(?a, ?b, ?t) = a + b + t`
	`join(x, y, 2)`	
clauses		`join(x, y, ?t) = x + y + t`
output	`join(1)`	`join(?t) = x + y + t`
	`join(2)`	

Fig. 8. Four more optimizations as clauses, in the form of Figure 1. In the third row k is what the caller does with the result, `k(?z) = z + a`.

B Further results

B.1 No redundant parameters

This subsection is about first-order programs in direct style without operations, for which the Lean development proves analogues of Theorems 5.1 to 5.3 (Appendix B.4). Braun et al. call a set of ϕs redundant if they refer only to each other and one outside value [8]. Likewise for binders: if every entry and live site gives each binder of a set S either a member of S or one term t that mentions neither S nor a version whose pattern mentions S, then collapsing S to t preserves stability (`collapse_valid`). So the optimum has no such set whose t is a constant, a constructor term or a binder that occurs in it, a plain name of another block included (`no_redundant`): it is minimal SSA, and more, since folding merges constant and equal ϕs too.

B.2 Escaped values

The escapes of §3.2 are a predicate on terms. Under Q, a term *has escaped* if it is an integer, a binder or a plain name, $\cdot$, a constructor that is not a closure whose fields have escaped, an operation whose operands have escaped, or a closure of a reached version g that satisfies two conditions. First, each binder in the pattern of g is a binder of g, and at each parameter the pattern has the binder named by that parameter, which does not occur at the capture positions. Second, the captures of the closure are an instance of g's patterns at the capture positions, by a substitution whose terms have escaped. A version with an escaped closure can therefore be called with arbitrary arguments. Escape is checked on the terms that the patterns give, normalized, so an operation escapes only when its operands do, even where its value contains no closure.

B.3 The language, and the proof of Theorems 5.1 and 5.2

The language in full. A program gives each block g a body b, a tree of equality tests whose leaves are halts, *calls* $g(\bar{t})$ of a block by its label, and *jumps* $t(\bar{t})$ through a term. Its variables x are the *positions* of g: two captures, then its parameters (§3.1). Terms are built from integers, binary constructors, variables, operations and the symbol $\cdot$ of skeletons. A closure is a constructor term $c(l, r)$ whose tag c is the label of a block g; a jump through it runs g with the captures l and r at its first two positions. Each block that closures are made of (a label of a tag) takes two parameters after the captures, a value and a continuation: a tuple carries several values, and a continuation receives a constant as its second. Other blocks take any number. A call passes a term without $\cdot$ for each position of the block it calls, a jump two such terms, and a target or a result is $\cdot$ or has no $\cdot$. A closure or constructor with fewer than two fields is padded with constants. The rewrite rules *nf* are those of the reference implementation, over any arithmetic on the naturals in which 0 and 1 are units, $m - m = 0$ and 0 absorbs, such as 64-bit wrap-around below 2^{64}: constant folding, $x - x = 0$, the laws of 0 and 1, and equality of equal terms or of two ground terms; the reference implementation itself folds with OCaml's 63-bit integers. An operation on values means what the rules make of it. Two more operations select the left and the right field of a data constructor (`Node.Ref.select_con`). Data constructors have separate tags, which the theorems require not to be labels of blocks. A `case` that binds fields can then be written as a test of the scrutinee against the constructor of its two selected fields, followed by uses of those fields (`Node.Ref.case_decided`). The programs of §2 whose cases bind no fields are programs of this language, with a result as a halt, and §3.1 puts a call that returns into this form. For first-order programs in direct style, whose calls may return, the development proves analogues of the theorems (Appendix B.4).

The semantics is a deterministic small-step machine. A state is the running block with the values of its positions, and a test compares closures as constructors. A run halts with a value or diverges. It can also *exit*, when it jumps through a value that is not a closure of the program, which models a call of unknown code. A program has a set of entries, tuples of values without $\cdot$ for the positions of its entry block. The unknowns of Figure 5 are modeled by two entries whose join has a different binder at each position, such as $(0, 1)$ and $(2, 3)$, and by the values that unknown code passes.

The translation from Figure 3, which removes lets and writes constructors as pairs, and the reference implementation are related to the theorems only by the tests of §6.

Stability in full. Patterns and instances are defined as in §2.2, and each block has one version. An assignment Q is a set of versions, the reached ones, with a pattern for each. The walk of §2.3 is replaced by two definitions. A *site* of a version v is a call, jump or halt in v's body, together with the tests on the path from the start of the body to it and the branch taken at each. A site is *dead* under a pattern if, with the pattern in place of the positions, one of these tests is decided and the site is on the other branch. A test is decided when its two sides are the same term, or when they differ at some place in a constructor or an integer. The escapes of §3.2 become a predicate on terms, those safe to hand to unknown code: a term *has escaped* if it is an integer, a binder or a plain name, $\cdot$, a constructor that is not a closure or an operation whose fields or operands have escaped, or a closure of a reached version whose pattern has at each parameter the binder named by that parameter and whose captures have escaped (Appendix B.2). Q is *stable* if the following hold. Each entry, and at each site of a reached version v that is not dead under v's pattern, each call and each jump through a term that v's pattern shows to be a closure of g, goes to a reached version and is an instance of its pattern by a substitution that puts only escaped terms into binders. At any other jump the target and the arguments, and at a halt the result, have escaped (`CpsNode.Valid`).

The theorem uses a *plain rewrite* by Q, simpler than the one in §2.6: it does not nest versions, omit unused binders or pass whole values. The parameters of a version are the binders of its pattern,

the plain names of other versions included; in its body each position is replaced by its term in the pattern, and a call passes, for each binder of the callee, the term substituted for it. A jump through a term that the pattern shows to be a closure of g becomes a direct call of g, a jump to its label. Tests are kept.

What the language omits. Compared with Figure 3, the language has no `let` of an expression, which substitution removes; no `case` that binds fields, in place of which it has the two selection operations; and only binary constructors and closures, with which others are written as nested pairs.

The proof. The proof is a simulation. An activation of the program is related to an activation of the output when the two are at the same point of the same version, and the values at the positions of the first are the version's pattern with the parameters of the second substituted for its binders. A step of the program corresponds to one step of the output, and at a call the relation holds for the new activation because, by stability, the arguments are an instance of the callee's pattern. The relation also requires the output's parameters to hold values that have escaped, and a call from unknown code is covered because the version of an escaped closure has distinct binders at its parameter positions, so any arguments are an instance of its pattern. The same definition of stability thus applies to a whole program and to a library whose functions are called by unknown code. The reference implementation's rules are sound and keep escape (`Node.Ref.conditions`). (Lean: `CpsNode.matches`, `CpsNode.preserves`, `CpsNode.escapes`, `CpsNode.reenter`.)

B.4 The nested output

For first-order programs in direct style, in which a call may return, the development proves analogues of Theorems 5.1 to 5.3, with and without operations, in which the rewrite also replaces a test that the pattern decides by the branch taken (`Sem.preserves`, `Sem.every_schedule_preserves`, `Node.preserves`, `Node.Batch.every_schedule_terminatesN`), and the following.

Theorem 5.2 is proved for two rewrites, for programs with closures and operations. The plain one keeps the tests and passes the plain names of other versions as parameters; for it the program and the output halt and exit at the same step (`Chip.main`, part (1) below). The other removes each test that the pattern decides, and nests a version inside the versions that bind its plain names, so that it takes their values from the nearest enclosing activation of such a version and is passed nothing for them. For this output a nesting exists for each stable assignment, and under any such nesting the program and the output halt and exit with the same values, the output in at most as many steps, since a removed test is a step that is not taken (`Chip.nested_output`; its parts are `Chip.output` and `Chip.nested`). In both rewrites a closure that is built stores all its captures. Storing only the values of the binders at the captures is proved without operations (`CpsStore.preserves`); with operations it is not, because the language can test a closure against another value, which a closure that stores less answers differently (`Chip.store_operations_false`). The rewrite of §2.6 also drops binders that are not used and inlines a block that is called from one place; neither proved rewrite does.

For first-order programs in direct style the nested output is proved too, with finitely many versions and entries. The version that binds a plain name of $Q[v]$ lies on each path from an entry to v (`Sem.owner_on_path`), so nesting the versions along their dominator tree puts each such name in scope (`Sem.dom_matches`, `DomTree.sem_dominator_preserves`).

B.5 The proof of Theorem 5.3

The theorems in full. In Lean, Theorems 5.1, 5.2 and 5.3 are one statement with three parts: (1) is Theorem 5.2, (2) is Theorem 5.3 and (3) is Theorem 5.1, each also for a program that exits to

unknown code. Take a program of this language in which no tag of a data constructor is the label of a block. (1) Let Q be stable, and take an entry. The program and the program rewritten by Q halt or exit at the same step of the run from it with the same values. (2) Let the program have finitely many blocks and entries, and let a join name each binder it makes by a place at which the binder occurs (a position and a path through constructors), and a binder at the top of a parameter by the top of a parameter. Then each run of the algorithm, under any schedule, has a number of steps bounded by a function of the program, and a run in which no step is possible is in a finished round. The patterns of a finished round are the least stable assignment O_{pat}, and those of any two finished rounds are each at least as specific as the other. (3) In (1), at each step the values at the positions of the running version are its pattern in Q under a substitution of escaped values, a result has escaped, and so have the target and the arguments of an exit. If a closure that has escaped is called from outside the program with escaped arguments, at most as many as it takes, (1) and (3) hold of that run. (Lean: `Chip.main`.)

Beyond finiteness, the statement's only condition is the calling convention above, which any program meets once its closures take a value and a continuation. The model differs from the reference implementation in two places: with closures, the reference implementation names a binder by its first occurrence, which can be in a capture and need not be at the top of a parameter; and where no rule applies to an operation on values, the model gives $\cdot$, where the reference interpreter is stuck.

The proof. The invariant of the proof says that the patterns in a run are at no point more general than a stable assignment requires. Fix a stable assignment Q'. The invariant is that there is one substitution τ, the same for all versions, such that $Q'[v]\tau$ is the round's pattern for each walked version v, and such that the arguments joined so far into $F[v]$ are an instance of $Q'[v]$ that agrees with τ on the plain names of other versions. It holds throughout a round, including a round that a restart later ends. Consider a site of a walked version that is not dead under the round's pattern. Its arguments under that pattern are, by τ, an instance of its arguments under Q'. The site is not dead under Q' either, because a test decided under a pattern is decided the same way under each instance (condition (N) below, and, for a test whose sides differ, that constructors are free). Since Q' is stable, the arguments are an instance of the pattern of the callee in Q'. The join of two instances of a pattern that agree on the plain names of other versions is again an instance of the pattern. This preserves the invariant at a walk step, and at a restart it shows that each new binder is at a place where Q' has a binder. The invariant has one more clause: each term that τ substitutes for a binder of Q' has escaped under Q'. The clause is needed for a jump that the round's pattern shows to be a call of g when the pattern in Q' does not: under Q' the closure has escaped, so $Q'[g]$ already accepts the call. Instances and the order are taken up to nf, and the proof uses the restriction that no binder of a version occurs below an operation in its pattern.

For termination, say that a place in the pattern of v is *open* if it is at or inside a binder of $\mathit{refuted}[v]$. A restart either makes a place open that was not, or gives different binders to two open places that had the same binder. It therefore increases the number of open places plus the number of pairs of open places with different binders. This number is bounded, because the depth of the terms that a run constructs is bounded by a function of the program. So the number of restarts is bounded, and between two restarts a round has at most one step for each version, site and entry. Names as part (2) asks for them can always be chosen (`CpsNode.Batch.go_pgood`), and the invariant holds without them.
(Lean: `Chip.main`, `CpsNode.forcedN`, `CpsNode.leastN`, `CpsNode.uniqueN`, `CpsNode.Batch.go_pgood`, `CpsNode.Batch.every_schedule_terminates`.)

The proof assumes conditions on the rules, which those of the reference implementation meet over any arithmetic in which 0 and 1 are units, $m - m = 0$ and 0 absorbs (`Node.Ref.conditions`). Condition **(N)** is that normalizing commutes with instantiation: $nf((nf\ t)\tau) = nf(t\tau)$ for a substitution τ of terms for binders, which excludes a rule that applies to a term with a binder and not to its instances. The others are that terms related by $\equiv$ have the same value, nf is a congruence, introduces no binders, does not deepen a term or produce $\cdot$, leaves constructors free, keeps the operands of a normal operation normal, and brings no closure to where it escapes.

C Routes that do not follow the caller's version

When a jump through a closure has a separate route, the patterns that the algorithm finds are stable, but a program can have two stable assignments and none that is at least as specific as both (`PolicyClo.Static.NestedCapture.no_least`), and the order of the walk can decide which one is found (`PolicyClo.Known.overshoot`).

If a route depends on more than the caller's version and the site, there may be no optimum:

```
f(?c, ?t, ?x, ?y) = if c then f(c, x.0, x, 99) else f(c, y.0, 99, y)
-- specialized by t's tag; entry f(U, A, (A, 0), (A, 0))
```

If the first back site stays in `f[A]`, then `y` is a binder, `y.0` is unknown, and the second site goes to a generic version; or the reverse. The two answers are incomparable, and, when a round is abandoned at its first refutation, the order of the arms decides. Each supports itself: more versions make patterns more specific, which keeps contexts known, which keeps the versions apart.

D Evaluation details

The four-parameter loop of §1. Table 4 runs our analysis and our models of seven published analyses on

```
main(?u, ?n) = loop(n, u, 1, Some((0, u)))
loop(?i, ?a, ?x, ?q) = case q of None → 0 | Some((?s, ?t)) →
    if i == 0 then s + t
    else if x == 1 then loop(i - 1, a, 2 - x, Some((s + a, t + (t - a))))
    else loop(i - 1, i, 99, q)
```

SCCP finds the constant, and Lemerre's and Click's analyses also that `a` is `u`; none of the seven models looks inside `q` on this loop. We also wrote the loop in C, with the option and the pair as structs: clang 17 at `-O2` and at `-O3` removes `a`, `x`, the test and the structs, and keeps a loop that replaces `t` by `2 * t - u` on each iteration. With `t` in place of `t + (t - a)` it reduces the whole to `(n + 1) * u`.

D.1 Models of published analyses: details

Lemerre's losses on scalar programs. They are a stale self-ϕ, $x_2 = \phi(x_1, x_2)$, which the repair $\phi(x, self) = x$ removes under the recursive strategy (under the iterative one, the analysis with the repair need not terminate), and two-ϕ cycles that need Braun et al.'s redundant-set removal. The self-ϕ does not come from our model: the author's artifact produces it too [53]. We built the artifact and ran its four configurations (the iterative and the recursive strategy on Bourdoncle's weak topological order, each with and without constant propagation) on three C functions of about a dozen lines (`data/compare/lemerre-artifact` in our artifact has the inputs, the outputs and how to reproduce them). Two of them are

```
uint8_t func_1(void) {
  uint8_t b = 1;
```

```
uint8_t func_1(void) {
  uint8_t x = 1;
```

Table 4. What each algorithm knows of `loop`'s parameters in the four-parameter loop of §1. h_k is a parameter of `loop` (equal k, one parameter); u is `main`'s. From the models of §6.3; every row is a stable assignment, and the reference implementation prints the first.

Algorithm	`i`	`a`	`x`	`q`	parameters left
Ours (the optimum)	h_1	u	1	$Some((h_2, u))$	2
Lemerre	h_1	u	1	h_2	2
Click	h_1	u	1	h_2	2
Flambda 2	h_1	h_2	h_3	h_4	4
Braun et al.	h_1	h_2	h_3	h_4	4
Simpson RPO	h_1	h_2	h_3	h_4	4
Cooper and Simpson SCC	h_1	h_2	h_3	h_4	4
SCCP	h_1	h_2	1	h_3	3

```
  uint8_t y = 0;
  while (b != 5) b = b + 1;
F: ;
G: y = 0 - y;
  if (b == 0) goto G;
H: if (b == 0) goto H;
  if (b == 1) goto F;
  return y;
}
```

```
L1: ;
L2: if (x == 0) goto L2;
  if (x == 2) goto END;
  x = 0;
  goto L1;
END: return x;
}
```

The artifact has no input operation, so the loop on `b` on the left keeps `b` unknown to constant propagation; on the right the runs are without constant propagation, which takes both edges of every branch. On the first, a loop that assigns nothing follows a loop that negates a variable: the iterative strategy places a ϕ at the head of the second loop, with itself as one operand, and the recursive strategy places none there, with constant propagation on or off. On the second, an inner loop that assigns nothing sits inside an outer loop, and both strategies place $x_2 = \phi(x_1, x_2)$ at the inner loop's head, where the standard placement [24] has no ϕ. Every output computes the right values; they differ in their ϕs. The cause is the one of §2.5. The artifact keeps the abstract store of each edge across iterations, and its join makes a new variable where two incoming expressions differ. A back edge that still carries the store of an earlier iteration is joined with the new value, and the variable that this makes is never removed. The extended version of the paper remarks that the choice of iteration strategy does not affect the result of the analysis [54, Appendix E], and its introduction says that the translation has the classic minimality property of SSA. The first run differs between the strategies, and the second has a ϕ where the dominance-frontier placement has none.

The oracle and the models. Besides the scalar programs and those with constructors there is a third set, of about twenty-six thousand *random* programs. The constructor programs are enumerated in seven tiers, the two largest sampled at every 20th and every 4th program. The oracle stops at 3,000 assignments, which happened on 3 programs, all of them random; there it takes the one assignment of fewest parameters. A result is optimal if it is stable, matches a concrete run on up to six inputs, and equals the oracle's where it reaches. Each model sees the program as one control-flow graph: a function is a block, its parameters ϕs, every call site an edge. Where a source is ambiguous we took the reading it most directly supports, or the generous one, and the artifact documents each reading. Each model was run on its source's own examples, and Lemerre's was compared with his

artifact. The model of Flambda 2 was also compared with the compiler: we wrote the programs of a sample as OCaml, each as one function whose blocks are local functions that the compiler turns into continuations, and read from its simplified output what it knows of each parameter. That is possible for programs that are typable in OCaml and make only tail calls. On 123 programs with constructors the model and the compiler disagree on whether a parameter is unboxed on one, and on 97 scalar programs on whether a constant is found on two; the compiler also unboxes a parameter that is entered with an opaque value, by loading its fields at the entry, which the programs of our sets cannot express, and uses facts from tests, in a few cases to find more than the model and in two to find less. Of the programs where the optimum unboxes a parameter, about one in a hundred is typable; on those the model unboxes about half.

Some of the losses are facts about parameters that nothing reads. With dead parameters removed from both sides, Lemerre's analysis is still worse than the optimum on 13% of the programs with constructors, down from 24%, and Flambda 2 and Braun et al. on 29%, down from 39% and 40%. On scalar programs the removal changes almost nothing.

Pipelines. Click's argument [18] is that one fixed point beats the analyses in sequence. We ran SCCP, Simpson's value numbering and Braun et al. as a pipeline, each stage on the program the stages before it rewrote, in all six orders, each iterated to a fixed point. Even taking the best order per program, the pipelines stay below the optimum on 1.2% of scalar programs, 29% with constructors and 0.35% of random ones, and never know more than ours. The smallest scalar program on which every pipeline of §6.3 loses is `f(?a0, ?a1; ?k) = let r = f(1, 1) in f(a0 - a1, 0; k)` from `f(u, u; K)`: Our analysis gets (h, h), but every pipeline keeps two parameters, since the congruence and the fold need each other.

Click's combined analysis. The strongest scalar baseline is Click's combined analysis [18, 19]: constants, reachability and congruence in one optimistic fixed point. Modeled on the whole program as one graph, it reaches the optimum on every scalar and random program, and on 70% with constructors. It misses nearly a hundred and fifty thousand, among them all those where the optimum unboxes. Click's thesis has no constructors, so that row is our extension of his analysis, and it depends on the reading: one that also lets the analysis know which constructor a value has reaches 76%, and with dead parameters removed the model we report is still worse on 19%. So what separates our analysis from our model of an optimistic combined analysis is constructors and unboxing, and, against an intraprocedural implementation, plain names across functions.

D.2 A compiler's only optimizer: details

Free abstraction. Twelve pairs of programs compute the same thing, one through an abstraction and one as the loop a programmer would write: folds with closures, map, filter and sum as push and as pull streams, a state monad as a loop body and as the loop itself, an option monad, and dictionaries as records of functions. In 9 of the 12 pairs the specialized IR is identical, and so is the assembly (Table 5). Each is one of the mechanisms of §3 and §4: a closure's label is used by the default policy, so the fold gets a version per λ; a captured variable is a field of the closure's term, which is in the pattern; a dictionary passed as the same constant at every site is a constant of the version; a loop's state pair is never built.

Three differ. *Pull streams and the option monad* keep their `Next` and `Some` values, in registers, with nothing allocated. In the option monad a continuation is passed one constructor on one path and another on the other, and the two meet in one parameter. A policy on that parameter's tag, which is case-of-case, removes the values from the IR in an experiment, and the assembly stays the same, since LLVM already decides those cases. In the pull stream the `Next` values that stay are those of a loop's parameter, which receives the results of a function with several callers, and that

Table 5. Abstract programs against the loops written by hand, compiled by the compiler's variant and clang -O2 (x86-64). "Identical" compares the specialized IR up to names, and the entry function in the assembly with labels and comments aside. Cycles per iteration of the abstract version, the least of five runs under `perf stat` on one core; ratio to the hand-written one.

Abstract / direct	Specialized IR	Assembly	Cycles/it	Ratio
`fold_hash` / `loop_hash`	identical	identical	3.00	1.00
`fold_hash_by` / `loop_hash_by`	identical	identical	4.00	1.00
`fold_squares` / `loop_squares`	identical	identical	1.15	1.00
`push_odd_squares` / `loop_odd_squares`	identical	identical	1.43	1.00
`pull_odd_squares` / `loop_odd_squares`	Next values stay	differs	3.00	2.10 (instr 1.92)
`monadic_body` / `by_hand`	identical	identical	1.72	1.00
`monadic_loop` / `by_hand`	closures, apply	differs	525.22	305.36 (instr 32.53)
`by_monad` / `by_matches`	Opt values stay	differs	1.34	0.58 (instr 0.78)
`dict_hash` / `loop_hash`	identical	identical	3.00	1.00
`dict_pair` / `loop_pair`	identical	identical	3.00	1.00
`dict_two` / `loop_two`	identical	identical	3.50	1.00
`fold_two` / `loop_two`	identical	identical	3.50	1.00

Table 6. Three interpreters applied to a constant program computing `h := h * 31 + i`, against the direct loop: cycles per iteration of the loop (four interpreted instructions), at $n = 10^7$, before the value policy (an earlier build of the compiler) and after.

	Before		With the value policy		
Program	cycles/it	× direct	cycles/it	× direct	allocations
`direct`			3.02	1	0
`interp_list`	148	49	3.02	1.00	1
`interp_words`	24.6	8.2	3.02	1.00	1
`interp_tree`	233	77	43.83	15	3

policy does not remove them. *The monadic loop* returns a closure, which the join of two returns makes a binder, so its λs escape (§3.2) and the loop runs through closures.

Interpreters on constant programs. With the value policy, an interpreter applied to a constant program should become the program compiled [32]. We wrote three interpreters: a register machine whose program is a list of instruction constructors (`interp_list`), the same machine with four words per instruction in a buffer (`interp_words`), and a `while` language over a syntax tree with variables by name in an association list (`interp_tree`). `run` is specialized on the program and the program counter; a site's next `pc` is `pc + 1` or a jump target read from the constant program, so routes are fixed by the caller's version and the site. Structural recursion on a constant is walked in place, so `fetch prog 3` unfolds to the instruction, and a rule answers a read through a chain of buffer writes with the last write at that index, so a buffer of constant words folds.

Table 6 gives the results. `interp_list` and `interp_words` specialize to the same IR: the program is gone, and one version of the loop remains with `pc` at the loop's test. clang promotes the registers from their buffer, and the assembly's loop is the direct loop's ten instructions, at the same 3 cycles per iteration. Before the value policy they were 49 and 8.2 times slower (in an earlier build of the compiler). `interp_tree` goes from 77 to 15 times slower: the tree and the evaluator are gone, but the environment remains, an association list. The list is *partially static* [40], and a context here is wholly known.

D.3 The ports: details

The port. The Flambda 2 port is over a thousand lines inside its simplifier. When Simplify does not inline a direct call, the port sends it to a version of the callee specialized on the closures it passes: on a closure's symbol, or on its code, with the captured values as new parameters. A recursive call with the same context goes back to the version. Loop parameters get shapes: if every entry passes a block of one tag, or a boxed float, the parameter is assumed to be one, so Flambda 2 unboxes it; a back edge that passes something else refutes it, and the loop is simplified again. A version is kept only if it removes an indirect call, allocation or branch. The compiler bootstraps with it.

Table 7. The Flambda 2 port against stock OxCaml (`9cdd33d`, `-O3`): the median change in user-space instructions and wall time over pairs of runs interleaved on one CPU of an AMD Ryzen 9 7950X3D. Outputs are identical, except zdd's weak-table statistics. Steps: geometric means of the change, between each configuration's fastest of nine runs, from OCaml 5.4 without Flambda to Flambda 1, to Flambda 2, to the port.

Program	Workload	Instructions	Wall time	Runs
Menhir	`sysver.mly`	−23.5%	−24.0%	15
	`sql-parser.mly`	−17.7%	−18.8%	15
angstrom	JSON	−10.2%	−18.1%	15
	HTTP	−15.0%	−25.8%	15
markup.ml	HTML	−0.4%	−0.3%	11
	XML	−0.7%	+1.1%	11
jsonm	decode	−0.5%	+3.4%	11
	round trip	−0.4%	+1.7%	11
earley	`pa_ocaml`	0.0%	−3.4%	11
pprint	render	−1.6%	−0.1%	11
gen	bench	−0.3%	−0.3%	11
ocamllex	3 lexers	−0.2%	−0.4%	11
Sandmark	`qr_decomposition`	−28.4%	−72.9%	15
	`crout_decomposition`	−17.9%	−38.0%	15
	`zdd`	−34.4%	−25.3%	15
	`naive_multilayer`	−7.8%	−20.8%	15
	23 others, at most	±1.0%	±2.7%	5–7
`ocamlopt`	8 of its modules	−4.7 to −0.1%	–	3
Steps	Menhir, angstrom	−9.2 / −7.9 / −16.7%	−16.8 / −6.1 / −21.1%	9
	the port's 4 Sandmark	−22.0 / −24.3 / −22.8%	−16.4 / +0.3 / −47.7%	9
	all but `sequence_cps`	−10.4 / −17.1 / −4.7%	−9.0 / −8.5 / −9.2%	9

Cost. The reference implementation runs fourteen small abstract programs with five of its six policies, with their hand-written loops, in half a second together, and the largest random programs of its tests, about two hundred lines that become several hundred versions, in about a second each. It finishes each round and then starts a new one, as Figure 5 does; resuming a round from the walk of the version that was refuted, as the compiler's variant does, gives the same output on every program we compared and halves the number of walks. *The compiler's variant* takes at most 12% of the instructions of a compile after type checking on the compiler's seven benchmarks, and up to 15% on a ladder of copies of one of them. Its output is smaller than with the variant off on about half of our benchmarks and larger on a sixth, at two thirds of the size in geometric mean: removing closures removes code. *The Flambda 2 port* costs compile time, most of it on one large file (the paragraph *Flambda 2: compile time*, below), and adds 9% to Menhir's code. *The Chez port* adds 9% to compile time. The implementation of our analysis that §6.3 uses, which abandons a round at its

first refutation, takes fewer than two rounds on average on scalar programs, because refutations are few: without constructors at most $\sum_v \max(0, 2\,\mathit{arity}(v) - 1)$, and with them $2^n - 1$ on a program of size $O(n)$ whose optimum has 2^n binders.

D.4 Costs

Table 8. What each implementation costs. The compiler's variant: instructions of the variant on the benchmarks, and LLVM's `opt` on the same programs' IR. The ports: against the stock compiler, at `-O3` for Flambda 2.

The compiler's variant	instructions per benchmark	4.5–70 million
	share of a compile after type checking	at most 12%
	against LLVM's `sccp`, `gvn`; against its `-O2`	1.0–6.6×; 7–21%
	growth on k copies of a parser, up to $k = 64$	$k^{1.01}$
Our analysis (§6.3)	rounds on scalar programs: mean; most	1.74; 5
	rounds with constructors: mean; most	1.98; 5
The Flambda 2 port	compile CPU time: Menhir; the other programs	+11%; +64%
	code size: the compiler; Menhir; angstrom	+8.7%; +9.0%; +10.0%
The Chez port	compile time; code size	+9%; +0.1%

Flambda 2: compile time. Over three builds at `-O3`, the port adds 11% to Menhir's compile CPU time (7% wall, the same 0.37 GB peak) and 64% to that of the other programs of Table 7 but Sandmark and `ocamlopt`, nearly all in earley's `pa_ocaml.ml` (63 s and 2.6 GB instead of 33 s and 1.7 GB); the other files take 23% more. In `pa_ocaml.ml` all of the extra time is in one function, the functor that builds the grammar. The port makes a version of `List.map` for a known closure. Flambda 2 deems `List.map` itself never inlinable, and the version is small, so it inlines the version, about 1,250 times in this function, which grows from about 310 to about 360 KB of code. Register allocation for it takes 56 s where it took 25 s, and Simplify 13 s where it took 5 s. The port's own work is small there: it tried 317 versions and kept 19, and with versions switched off the compile takes as long as with the port off. A change that keeps Flambda 2 from inlining such versions removes most of this cost, and with it angstrom's gain, so we have not adopted it (`data/ocaml/ctime2` in the artifact).

E Other Papers' Examples

The examples of Figure 1 and Appendix A are ours. This appendix takes the examples of others. For each analysis and specializer that this paper relates our analysis to, we took the program that its own paper or documentation uses to present it, wrote it as input for the reference implementation, and ran it. A loop of assignments becomes a block whose parameters are the loop's variables, a test that the source leaves open becomes a test of an unknown input, and for the examples of SSA construction every block takes every variable. Each file in the artifact (`examples/lit/`) says where in its source the example is and what was added to make it a program; its output is checked to be what the tool prints, and the artifact lists which sources were read directly and which figures were taken from the transcriptions made for the models of §6.3. We call an example *found* when the output is the result that the source states for its own optimization.

Of 43 examples from 21 sources (Table 9), 33 are found: 21 with one version per block, and 12 with a policy. 4 are found in part and 6 are not.

One version per block. The examples of optimistic value numbering are found, and so are those of SSA construction: Braun et al.'s redundant cycle of two ϕs, which their algorithm removes by a separate pass over strongly connected components, is two parameters that every call passes the same variable. Fluet and Weeks's Figure 7, the example that separates their dominator analysis

Table 9. Other papers' examples in the reference implementation. Policy: the first policy with which the output is the source's result (constant: one version per block); a dash where no policy gives it.

Source	Its example	Policy	Output
Wegman, Zadeck [85]	Figs. 9, 10, 11: a constant behind a dead branch, around a loop, through a ϕ	constant	the source's
	Fig. 4: a constant known from the test that guards a branch	–	needs a fact from the test
	Fig. 13: a dead exit, and a constant after the loop known from its test	constant	in part: the dead exit goes
Alpern et al. [1]	Fig. 5: variables congruent only optimistically	constant	the source's
Simpson [79]	Fig. 4.8: equal induction variables	constant	the source's
Click [19]	Figs. 3.9, 3.11: constants, dead code and congruence, each needing the others	constant	the source's
Braun et al. [8]	Figs. 2, 4, 5 and the ϕ cycle of §3.2	constant	the source's, with no pass over components
Lemerre [54]	Fig. 1, the running example	constant	the source's
Flambda 2 [64]	its reference's join of two `Some`	constant	no `Some` built
	unboxing with an opaque arm; match forwarding	constructor	the source's
	match simplification: a parameter that is 0 or 1	–	needs a set of constants
Gulwani, Necula [36, 37]	Fig. 2 (2004): $x = y$ and $z = F(x, x)$ where two paths meet	constant	$x = y$ only
	Fig. 11 (2007): an equality around a loop	constant	the source's
Fluet, Weeks [31]	four examples, among them Fig. 7, which needs dominators	constant	the source's
Kennedy [44]	§5 and §5.1: contification	constant	the source's
	§5.2: `unifV`, called between calls that return	–	keeps its continuation
Appel [4]	its program, with a ϕ for every variable in every block	constant	the source's ϕs
Peyton Jones [66]	`drop`, `sum_append`	constructor	the source's
	`last`	–	needs a fact from the test
Jones et al. [40]	`power`, Ackermann, `lookup`, an interpreter for Turing machines	value	the source's
Fallin, Bernstein [29]	Fig. 6: an interpreter loop	value	the program's loop
Brandon et al. [7]	`twice` with several λs	closure	a version per λ, all calls known
Chambers, Ungar [14]	`sumTo:`	constructor	a version per receiver class
Cooper et al. [20]	`matrix300`	value	the source's, in more versions
Chevalier-Boisvert, Feeley [16]	`sum` with a type test before each operation	constructor	four of five tests gone; boxes stay
Dean et al. [28]	`overlaps` on sets of unknown class	–	needs a fact from the test
Coutts et al. [22]	`append` of two streams	return	one loop; steps still built
	`concatMap`	–	the inner stream is built

from the two analyses before it, is found because a function's continuation is a parameter like any other: three functions reached only through tail calls from `f` get `f`'s continuation as a plain name.

The constructor policy. The policy that reads the tag of each constructor argument, as the closure policy reads the label of each closure, gives the results of call-pattern specialization [66] on two of that paper's three examples. `drop`, whose counter the source boxes, becomes a loop over a machine integer, and `sum_append`, a loop over a state that is `Left` of one list and then `Right` of the other, becomes two loops and builds neither. The same policy gives a version of a method per class of

its receiver, which is customization [14], and a version of a join point per constructor passed to it, which decides a match in each arm of the test before it.

What the other examples need. Nearly all of them need one of three things. The first is a fact from a test: that inside an arm the scrutinee is what the arm matched (§7), which is the whole of `last`, of Wegman and Zadeck's Figure 4 and of two of MLton's passes. The second is knowledge of a value that no call passes: the cells of a list in the heap, or a stream's step function inside a record. The third is a return point that knows that the continuation it stored is that of the function that built it, which `unifV` needs.