A low-level look at A-normal form

WILLIAM J. BOWMAN, University of British Columbia, Canada

A-normal form (ANF) is a widely studied intermediate form in which local control and data flow is made explicit in syntax, and a normal form in which many programs with equivalent control-flow graphs have a single normal syntactic representation. However, ANF is difficult to implement effectively and, as we formalize, unsafe for scoped effects such as scoped region-based allocation. The problem, as has often been observed, is the normalization of commuting conversions.

We argue that the traditional view of ANF, informed by high-level languages, is wrong. By studying the low-level intensional aspects of ANF, we can derive a normal form in which normalizing commuting conversion is easy, does not require join points, or code duplication, or renormalization after inlining, and is safe for scope. We formalize the connection between ANF and monadic form and their intensional properties, derive an imperative ANF, and design a compiler pipeline from an untyped λ-calculus with scoped regions, to monadic form, to a low-level imperative monadic form in which A-normalization is trivial and safe. We prove that any such compiler preserves, or optimizes, stack and memory behaviour compared to ANF.

The main take-away from this work is that, in general, monadic form should be preferred over ANF, and A-normalization should only be done in a low-level imperative intermediate form. This maximizes the advantages of each form, and avoids all the standard problems with ANF.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Formal software verification**; *Software performance*.

Additional Key Words and Phrases: Compilers, Optimization, A-normal form, CPS, Monadic Form, Intermediate Representation, Normal Form, Normalization

ACM Reference Format:
William J. Bowman. 2024. A low-level look at A-normal form. 1, 1 (April 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Intermediate representations, forms, and languages are used to simplify program analysis, optimization, and compilation, *e.g.*, by (1) explicating abstractions, such as evaluation order or data flow, into explicit representations in syntax; (2) normalizing programs, so that many programs equal under some equivalence class have a single normal representation; or (3) providing practical equational theories for optimization or transformation.

A-normal form (ANF) is an intermediate representation widely studied in compilation [3, 8, 11, 13, 14, 17]. ANF can be described syntactically as an untyped λ-calculus with **let** where all *computations* must take *values* as their operands, and intermediate computations are explicitly sequenced using **let**. This makes data and local control flow (everything except returning from a call) explicit in the syntax, simplifying analysis, optimization, and compilation.

For example, consider the following two equivalent terms, where Listing 2 is the A-normalization of Listing 1.

Author's address: William J. Bowman, University of British Columbia, Vancouver, Canada, wjb@williamjbowman.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(+ (let (x (f 5)) 0) 6)
```

Listing 1. λ-calculus

```
(let (x (f 5)) (+ 0 6))
```

Listing 2. ANF

Code generation from the Listing 2 is simpler than Listing 1, since the computation (+ 0 6) can be compiled to something like **mov** y 0*; add y 6*. Optimization is simpler; (+ 0 6) is trivially equal to 6, whereas optimizing the original term requires some additional control and data flow analysis. ANF explicates many stack frames into syntax, so the ANF abstract machine runs with a smaller stack, in general, than the λ-calculus machine. In this example, the λ-calculus term evaluates the first operand of + to a value, after pushing the frame $(+ \cdot 6)$, while the ANF term never pushes this frame.

ANF has several known disadvantages. We discuss these in detail, but in short: ANF is not closed under β-reduction (complicating inlining) and compiling branching expressions into ANF requires care to avoid duplicating expressions or introducing procedure calls.

ANF suffers at least one additional disadvantage that is not discussed in the literature: transformation into ANF is *unsafe for scope*. In Listing 2, the scope of x is extruded past the addition expression compared to Listing 1. Effects attached to lexical binding can be reordered.

For example, we can translate Listing 1 and Listing 2 into a language with the lexically scoped region system of Tofte and Talpin [20], where (**letregion** *r e*) allocates a new region *r* for use in *e* and frees that region when the expression returns, and (**@** *r e*) allocates in *r* the value resulting from evaluating *e*. All values must be explicitly allocated and passed by reference.

```
(letregion r1
 (@ r1 (+ (letregion r2
           (let (x (f (@ r2 5)))
            (@ r1 0)))
          (@ r1 6))))
```

Listing 3. λ-calculus with Regions

```
(letregion r1
 (letregion r2
  (let (x1 (@ r2 5))
   (let (x (f x1))
    (let (x2 (@ r1 0))
     (let (x3 (@ r1 6))
      (@ r1 (+ x2 x3))))))))
```

Listing 4. ANF with Regions

The term in Listing 3 allocates an inner region *r2*, allocating 5 in region *r2*, calling *f* before allocating a result in the outer region *r1*. *r2* is freed when the computation of *x* completes. However, in Listing 4, *r2* is not freed until the end of the program. We must make the allocation and free operations explicit before we can safely (as in safe for space [19]) target ANF.

Typically, ANF is contrasted with continuation-passing stlye (CPS), another syntactic discipline popular as an intermediate form, which also explicates control and data flow. We ignore CPS until Subsection 6.1, but instead consider a more related alternative: monadic form.

Monadic form is the syntactic discipline induced by Moggi's monadic meta-language [15] when treating all non-value expressions as effectful computations (consider, *e.g.*, partiality as the effect). We can pronounce the monadic bind as **let**, and implement monadic return as an untagged inclusion of values into computations. For example, Listing 1 (reproduced in Listing 5 for comparison) could be implemented in monadic form as either Listing 6 or Listing 7, since all ANF terms are also in monadic form. Neither term would be produce by the usual translation of Listing 1, but they're useful examples.

```
(+ (let (x (f 5))
     0)
   6)
```

Listing 5. λ-calculus

```
(let (y (let (x (f 5))
             0))
  (+ y 6))
```

Listing 6. A monadic equivalent

```
(let (x (f 5))
 (let (y 0)
  (+ y 6)))
```

Listing 7. An ANF equivalent

Monadic form suffers none of the disadvantages of ANF. However, it is *less normal*. Being less normal, transformations and analyses can be *less obvious* in monadic form compared to ANF. In Listing 6, the computation (+ y 6) only takes values as operands, but it's not obvious how to optimize the expression. In Listing 7, we can obviously inline y, since it is bound to a constant.

ANF is monadic form with all commuting conversions normalized. Listing 6 and Listing 7 are equal by associativity of bind, one of the commuting conversions. ANF also normalizes commuting conversions for conditionals, such as (**let** (x (**if** v e_1 e_2)) e_3) ≡ (**if** v (**let** (x e_1) e_3) (**let** (x e_2) e_3). Duplicating e_3 is undesired, so typically this is abstracted into a continuation, called a *join point*, as in (**let** (x (**if** v e_1 e_2)) e_3) ≡ (**let** (j ($\boldsymbol{\lambda}$ (y) e_3)) (**if** v (**let** (x e_1) (j x)) (**let** (x e_2) (j x)))).

This difference between ANF and monadic form, normalization of commuting conversions, is at the heart of ANF. It is both why ANF is attractive as an intermediate form, and causes many of the problems of ANF.

This is not a novel observation. Kennedy [11] observes that normalizing commuting conversions causes all the well known problems with ANF, before resorting to CPS. Maurer et al. [14] point out these problems as well, observing that while join points avoid duplication, they inhibit optimizations, and create a join point calculus to recover these optimizations in something ANF-like.

The problem with these commuting conversions is that, while the programs are extensionally equal even when reasoning about monadic effects, they are intentionally different when we consider details related to efficient execution and compilation. Duplicating code may be extensionally fine, but it's intensionally bad. Commuting conversions are also not equal for otherwise-effectful programs that use monadic form or ANF as intermediate forms but not to express monadic effects. That is why when we add scoped regions, A-normalization causes a new problem.

In this paper, we formalize a novel observation: join points are unnecessary and commuting conversions are not a problem. Instead, commuting conversions are a specification, not a literal implementation technique. By fully understanding the intentional aspects of commuting conversions, we can gain the benefits of ANF with none of the drawbacks. We make these intentional aspects formal using abstract machines. To contrast to Maurer et al. [14], our motto is: *work in monadic form, but think in abstract machines.* If we think of (**let** (x (**if** v e_1 e_2)) e_3) as its computation in an abstract machine, we might render it as (**begin** (**set**! x (**if** v e_1 e_2)) e_3), which by (a low-level interpretation of) associativity is the same as (**begin** (**if** v (**set**! x e_1) (**set**! x e_2)) e_3). No duplication, no join points needed. In fact, this idea is understood by some compiler writers; the high-performance Chez Scheme compiler uses a similar transformation (Subsection 6.4). Our paper performs a rational reconstruction and explanation of what some compiler writers already do.

Concretely, our contributions are:

(1) A novel formalization of monadic form as a subset of A-normal form, which is important for clarifying the distinction between ANF and monadic form, and identifying the source of problems with ANF (Section 2).
(2) A novel analysis of the abstract machine of high-level ANF and monadic form, with which we formalize how ANF optimizes stack usage (Section 3).

148
149
150
151
152
153
154
155
156
157
158
159
160
161
162

$$
\begin{array}{rcl}
v & ::= & \iota \mid x \mid (\boldsymbol{\lambda}\ (x)\ e) \\
e & ::= & v \mid (op\ \vec{e}) \mid (e\ e) \mid (\mathbf{let}\ (x\ e)\ e) \mid (\mathbf{if0}\ e\ e\ e) \\
E & ::= & \cdot \mid (\mathbf{let}\ (x\ E)\ e) \mid (\mathbf{if0}\ E\ e\ e) \mid (E\ e) \mid (v\ E) \mid (op\ \vec{v}\ E\ \vec{e}) \\
O & ::= & v \mid op
\end{array}
$$

$$
\begin{array}{rcll}
E[(\mathbf{let}\ (x\ e_1)\ e_2)] & \longrightarrow_A & (\mathbf{let}\ (x\ e_1)\ E[e_2]) & A_1 \\
 & & \text{where } E \neq \cdot & \\
E[(\mathbf{if0}\ v\ e_1\ e_2] & \longrightarrow_A & (\mathbf{if0}\ v\ E[e_1]\ E[e_2]) & A_2 \\
 & & \text{where } E \neq \cdot & \\
E[(O\ \vec{v})] & \longrightarrow_A & (\mathbf{let}\ (x'\ (O\ \vec{v}))\ E[x']) & A_3 \\
 & & \text{where } E \neq \cdot, E \neq E'[(\mathbf{let}\ (x\ \cdot)\ e)], \text{fresh } x' &
\end{array}
$$

Fig. 1. A-normalization for λ-calculus

163 (3) A novel formalization of a counterexample to the correctness of A-normal form with respect
164 to scoped effects (*unsafe for scope*), which is important for understanding limitations of
165 ANF in compilation (Subsection 3.2).
166 (4) A novel formalization of imperative variants of monadic form and ANF derived from the
167 machine semantics and admissible equations, and an algorithm for normalizing commuting
168 conversions without join points or duplication. These are important as normal forms for
169 compilation, optimization, and analysis as they enable the advantages of ANF but suffers
170 none of the known disadvantages (Section 4).
171 (5) A novel analysis of the abstract machine for imperative monadic form, with which we
172 prove that using monadic form followed by imperative A-normalization has the same or
173 better performance characteristics as ANF. In particular, we show that any such compiler (1)
174 avoids code duplication and join point introduced by commuting conversion in high-level
175 ANF; (2) preserves the stack behaviour of ANF; and (3) preserves the memory usage of
176 scoped regions, unlike the ANF compiler (Section 4).
177 (6) A model compiler designed to use monadic form as a high-level intermediate language,
178 and imperative ANF as a low-level IL. We argue that monadic form followed by imperative
179 A-normalization simplifies compiler implementation compared to ANF (Section 5).
180
181 All machines, reduction systems, languages, compilers, examples, and counterexample are imple-
182 mented in a PLT Redex [6, 12], available in the anonymous supplementary materials.
183
184 ## 2 A-NORMAL AND MONADIC FORM, FORMALLY
185 ### 2.1 A-normal Form
186 A-normal form (ANF) is often called "administrative normal form" or sometimes "administrative
187 form", but it is important and useful to think about ANF not as a vague form related to administrative
188 reductions, but formally and precisely as a normal form with respect to a set of reductions, as it
189 was originally formalized [8].
190 Formally, A-normal form was introduced as the form normal with respect to the set of reductions
191 $A = \{A_1, A_2, A_3\}$, defined in Figure 1. A term e represents an arbitrary λ-calculus expression. An
192 operator op is some n-ary primitive operator, such as addition, and must appear in operator position.
193 An ι is some ground value, such as a natural number, and a value v is any syntactic value. We
194 use the slight abuse of notation $(O\ \vec{e})$ to mean an application—of either a function or an operator,
195 and restrict the function position to a value as necessary. The A-reductions use the call-by-value
196

$$
\begin{array}{lrcl}
\textit{(Values)} & V & ::= & \iota \mid x \mid (\boldsymbol{\lambda}\ (x)\ M) \\
\textit{(Computations)} & N & ::= & V \mid (V\ V) \mid (op\ \overrightarrow{V}) \\
\textit{(Configuration)} & M & ::= & N \mid (\textbf{let}\ (x\ N)\ M) \mid (\textbf{if0}\ V\ M\ M)
\end{array}
$$

Fig. 2. A-normal form

evaluation contexts E to identify a non-value in evaluation position, and lift and bind it explicitly. This way, the data flow and local control flow is made explicit in the syntax.

The A-reductions require some side conditions about the evaluation context to ensure non-circular rewrites, and therefore to guarantee termination. We assume uniqueness of names and consider terms up to α-equivalence, as is standard.

If we *normal*ize a λ-calculus expression by reducing the transitive compatible closure of the set A, we reach a *normal* form with respect to A, *i.e.*, A-normal form, described syntactically by the grammar in Figure 2. The non-terminal N represents computations, while M represents program configurations that sequence computations.

As an example of A-normalization, consider the term in Listing 8, which A-normalizes to the term in Listing 9 by A_3 and then A_1. All computations are explicitly sequenced using **let**, so all operands are values.

```
(+ (+ 2 2)
   (let (x 1)
    (f x)))
```

Listing 8. λ-calculus example

```
(let (x1 (+ 2 2))
 (let (x 1)
  (let (x2 (f x))
   (+ x1 x2))))
```

Listing 9. A-normalization

Unfortunately, ANF has some drawbacks. ANF is not closed under β-reduction, complicating its calculus. The term $(\textbf{let}\ (x\ ((\boldsymbol{\lambda}\ (x')\ M)\ V))\ x)$ ought to be β-equivalent to $(\textbf{let}\ (x\ M[x' := V])\ x)$, where x' is substituted by v in M. But this expression is invalid since M cannot appear on the right-hand side of **let**. The ANF β-equivalence must renormalize all commuting conversions. This is a drawback as β-equivalence models inlining optimizations, and renormalization is inconvenient and expensive for a compiler. The A_2 rule causes exponential code duplication by duplicating the continuation E. Consider Listing 11. The term *LARGE* is duplicated 2^3 times while A-normalizing Listing 10—2^n where n is equal to the occurrences of **if** for which *LARGE* is in the evaluation context. Compilers using ANF avoid this using join points, but this canonical solution causes more problems; we discuss this in Section 5.

```
(let (x (if0 (if0 (if0 0 0 1)
                  0
                  1)
             0
             1))
  LARGE)
```

Listing 10. Nested Branching

```
(if0 0 (if0 0
         (if0 0 (let (x 0) LARGE) (let (x 1) LARGE))
         (if0 1 (let (x 0) LARGE) (let (x 1) LARGE)))
        (if0 1
         (if0 0 (let (x 0) LARGE) (let (x 1) LARGE))
         (if0 1 (let (x 0) LARGE) (let (x 1) LARGE))))
```

Listing 11. ANF Exponential Duplication

There are further benefits and problems with ANF, but these only become obvious when we consider machines for executing in ANF. We return to these in Section 3.

246
247
248
249
250

$$\begin{array}{rrcl} \textit{(Value)} & U & ::= & \iota \mid x \mid (\boldsymbol{\lambda}\ (x)\ C) \\ \textit{(Computations)} & C & ::= & U \mid (U\ U) \mid (op\ \vec{U}) \mid (\textbf{let}\ (x\ C)\ C) \mid (\textbf{if0}\ U\ C\ C) \end{array}$$

Fig. 3. Monadic Form Syntax

251
252

2.2 Monadic Form

253 254 255 256

Because ANF is not merely a syntactic description, but the normal form of a set of reductions, we can tweak those reductions to study alternatives normal forms. In fact, monadic form can be defined as a subset of the A-reductions. We can work backwards from its syntax to the reductions for which monadic form is normal.

257 258 259 260

We describe monadic form with the grammar in Figure 3, where we syntactically separate values U and effectful computations C. If we consider the effect as partiality, then we are explicitly forcing terms to value before calling each operation. In this definition, **let** corresponds to monadic bind, and return is implicit by the untagged inclusion of U in C.

261

This form is preserved under the monad laws and commuting conversions.

262 263 264 265 266 267

$$(\textbf{let}\ (x\ U)\ E[x]) \equiv E[U] \tag{Left Identity}$$

$$(\textbf{let}\ (x\ C)\ x) \equiv C \tag{Right Identity}$$

$$(\textbf{let}\ (y\ (\textbf{let}\ (x\ C)\ C_1))\ C_2) \equiv (\textbf{let}\ (x\ C)\ (\textbf{let}\ (y\ C_1)\ C_2)) \tag{Associativity}$$

$$(\textbf{let}\ (x\ (\textbf{if0}\ U\ C_1\ C_2))\ C) \equiv (\textbf{if0}\ U\ (\textbf{let}\ (x\ C_1)\ C)\ (\textbf{let}\ (x\ C_2)\ C)) \tag{Commute}$$

268

Both sides of the equations are in monadic form.

269 270 271

ANF is a normalization of monadic form: it normalizes the two commuting conversions, Equation Associativity and Equation Commute. The left-hand side of each of two rules is *not* in ANF, while the right-hand side is.

272 273

There is one further commuting conversion common in compilation literature that is inexpressible in our formalization of monadic form.

274

$$(\textbf{if0}\ (\textbf{if0}\ e\ e_1\ e_2)\ e_4\ e_5) \equiv (\textbf{if0}\ e\ (\textbf{if0}\ e_1\ e_4\ e_5)\ (\textbf{if0}\ e_2\ e_4\ e_5)) \tag{Case-of-Case}$$

275 276 277 278

Neither side is valid in our monadic form, since **if** must branch on a value, but **if** is a computation. Both monadic form and ANF normalize this equation; we return to it in Subsection 6.2, as it's important for optimization.

279 280 281 282 283 284

We can derive monadic form as a normal form as follows. First, we modify the definition of evaluation contexts. ANF was defined in a call-by-value[1] setting and, following CPS, concerned with internalizing the evaluation order. Monadic form is different: it (non-strictly) expresses composing effectful computations, ordering *only* the effects. This is reflected directly in the monad laws: to reach monadic form, we should not force terms to either side of the laws, and both computations and values are allowed on the right-hand side of **let**.

285 286 287 288 289 290 291

We define the non-strict monadic evaluation contexts E^b, and the B-reductions for them, in Figure 4. We omit **let**, but otherwise E^b is the same as E. The **let** evaluation context in ANF is responsible for normalizing commuting conversions, which each have a non-value in the right-hand side of a **let**. Recall Listing 10 has a conditional expression in the strict evaluation context $(\textbf{let}\ (x\ \bullet)\ e)$. We define the set $B = \{B_1, B_2, B_3\}$ of monadic reductions. These force a non-value in evaluation position to a value, just as like A-reductions, but using monadic evaluation contexts. B_1 and B_3 are essentially unchanged, but we drop one now-unnecessary termination side condition

292 293

[1]ANF doesn't require strict evaluation; *e.g.*, let could be non-strict in the operational semantics for terms in ANF. But ANF is normal with respect to A, which is defined using strict evaluation contexts.

294

295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343

$$
\begin{array}{rcll}
E^b & ::= & \cdot \mid (\textbf{if0}\ E^b\ e\ e) \mid (E^b\ e) \mid (v\ E^b) \mid (op\ \vec{v}\ E^b\ \vec{e}) & \\
E^b[(\textbf{let}\ (x\ e_1)\ e_2)] & \longrightarrow & (\textbf{let}\ (x\ e_1)\ E^b[e_2]) & B_1 \quad \text{where } E^b \neq \cdot \\
E^b[(\textbf{if0}\ v\ e_1\ e_2)] & \longrightarrow & (\textbf{let}\ (x'\ (\textbf{if0}\ v\ e_1\ e_2))\ E^b[x']) & B_2 \quad \text{where } E^b \neq \cdot\text{, fresh } x' \\
E^b[(O\ \vec{v})] & \longrightarrow & (\textbf{let}\ (x'\ (O\ \vec{v}))\ E^b[x']) & B_3 \quad \text{where } E \neq \cdot\text{, fresh } x'
\end{array}
$$

Fig. 4. B-normalizations for λ-calculus

restricting E^b. These two rules explicitly force operands to values via bind. B_2 is an optimization of A_2 to avoid code duplication. B_2 produces a monadic form expression, but not an expression in A-normal form. While we could use A_2 in B (they produce equal terms by Equation Commute), B_2 avoids the problems of A_2 and is more faithful to monadic form since it *avoids* unnecessarily normalizing the equation.

Some B-normal forms are A-normal, but not all, and all A-normal forms are B-normal. Consider our examples from earlier. Listing 8 has the same B-normal form and A-normal form, given in Listing 9. The second example, Listing 10, has a different B-normal form. Its A-normal form (Listing 11) included exponential code duplication, but its B-normal form (Listing 13) does not. Instead, the B-normal form normalizes Equation Case-of-Case by introducing an intermediate **let**, which is required by monadic form, but does not normalize Equation Associativity or Equation Commute.

```
(let (x (if0 (if0 (if0 0 0 1) 0 1) 0 1))
  LARGE)
```

Listing 12. Nested Branching λ-calculus

```
(let (x (let (x1 (if0 0 0 1))
          (let (x2 (if0 x1 0 1))
            (if0 x2 0 1))))
  LARGE)
```

Listing 13. B-normalized

```
(if0 0 (let (x1 0) (if0 x1 (let (x2 0) (if0 x2 (let (x 0) LARGE) (let (x 1) LARGE)))
                           (let (x2 1) (if0 x2 (let (x 0) LARGE) (let (x 1) LARGE)))))
       (let (x1 1) (if0 x1 (let (x2 0) (if0 x2 (let (x 0) LARGE) (let (x 1) LARGE)))
                           (let (x2 1) (if0 x2 (let (x 0) LARGE) (let (x 1) LARGE))))))
```

Listing 14. B- then A-normalized

B-normal form avoids code duplication by binding all computations, including **let** and **if**, rather than only binding values and primitive operations. This interpretation makes sense monadically, but violates the ANF discipline, which seeks to make the order of evaluation of the inner **let**s syntactically explicit by lifting them.

The specification of ANF and monadic form as a reduction system is useful for formalization. It enables separating A-normal form into its component pieces. As we will see, the reduction systems also serve as specifications so that we can prove certain properties of *any* compiler that targets the normal forms.

Now that we know, formally, what these normal forms are, let us turn to their intensional properties in the form of their machine semantics.

3 MACHINE SEMANTICS OF ANF

In this section, we introduce abstract machines for ANF and monadic form. We formalize two important facts: (1) ANF is an optimization, but (2) ANF is unsafe for scope. We show that A-normalization optimizes stack usage—there are fewer frames in use after A-normalization compared

$$
\begin{array}{llll}
\text{Frame} & F & ::= & (\cdot\ e) \mid (v\ \cdot) \mid (\mathbf{let}\ (x\ \cdot)\ e) \mid (\mathbf{if0}\ \cdot\ e\ e) \mid (op\ \vec{v}\ \cdot\ \vec{e}) \\
\text{Kontinuation (as stack of frames)} & K & ::= & \mathbf{mt} \mid\ F :: K
\end{array}
$$

$\boxed{e; K \rightarrow_{\lambda} e; K}$

$$
\begin{array}{rcll}
((\lambda\ (x)\ e)\ v); K & \rightarrow_{\lambda} & e[x := v]; K & \beta \\
(\mathbf{let}\ (x\ v)\ e); K & \rightarrow_{\lambda} & e[x := v]; K & \zeta \\
(\mathbf{if0}\ 0\ e_1\ e_2); K & \rightarrow_{\lambda} & e_1; K & \textsc{ifz} \\
(\mathbf{if0}\ v\ e_1\ e_2); K & \rightarrow_{\lambda} & e_2; K & \textsc{ifnz} \quad \text{where } v \neq 0 \\
(op\ \vec{v}); K & \rightarrow_{\lambda} & \delta[\![op\ \vec{v}]\!]; K & \textsc{primop} \\
v; (F :: K) & \rightarrow_{\lambda} & F[v]; K & \textsc{pop} \\
F[e]; K & \rightarrow_{\lambda} & e; (F :: K) & \textsc{push} \quad \text{where } e \neq v
\end{array}
$$

Fig. 5. λ-calculus CK Machine

to monadic form, in general. However, we then introduce a region calculus with scoped regions, and show that A-normalization of the region calculus results in more regions and longer region lifetimes than in monadic form, in general. We call this *unsafe for scope*, in the sense of unsafe for space [19]: A-normalization is unsafe for scope-based effects.

3.1 ANF vs λ Machines

We start with a CK machine [5–7] for arbitrary λ-calculus expressions, defined in Figure 5. An expression e represents the code pointer. Intuitively K is the rest of the computation, *i.e.*, the kontinuation, but we represent it as a stack of frames F to better study the effect of ANF on the control stack.

The machine transitions are completely standard. β-reduction performs capture-avoiding substitution as a metafunction over terms, replacing a variable by a value. Primitive operators evaluate by some denotation defined by $\delta[\![_]\!]$. Non-redexes push, and values pop, until the machine terminates with a value and an empty kontinuation **mt**. The rule PUSH is deceptively complex; it parses a term into a frame F and subterm e, an operation most real machine do not support directly.

Our example from Listing 8 evaluates in the CK machine are follows.

$$
\begin{array}{llll}
 & (+\ (+\ 2\ 2)\ (\mathbf{let}\ (x\ 1)\ (f\ x))); & \mathbf{mt} & \\
\rightarrow_{\lambda} & (+\ 2\ 2); & ((+\ \cdot\ (\mathbf{let}\ (x\ 1)\ (f\ x)))::\mathbf{mt}) & \textsc{push} \\
\rightarrow_{\lambda} & 4; & ((+\ \cdot\ (\mathbf{let}\ (x\ 1)\ (f\ x)))::\mathbf{mt}) & \textsc{primop} \\
\rightarrow_{\lambda} & (+\ 4\ (\mathbf{let}\ (x\ 1)\ (f\ x))); & \mathbf{mt} & \textsc{pop} \\
\rightarrow_{\lambda} & (\mathbf{let}\ (x\ 1)\ (f\ x)); & ((+\ 4\ \cdot)::\mathbf{mt}) & \textsc{push} \\
\rightarrow_{\lambda} & (f\ 1); & (+\ 4\ \cdot)::\mathbf{mt} & \zeta
\end{array}
$$

After, A-normalization (Listing 9), (1) all computations are either on the right-hand side of a **let**, so we can combine the rules PRIMOP and ζ, or in tail position (the injection from N into M)[2] so can be evaluated in place without a PUSH; (2) all operands are values, so evaluating an operator need never push a frame. Both (1) and (2) imply the only place a frame can be pushed is in a non-tail function call.

Therefore, we can define the *optimized* and *simpler* machine for A- and B-normal forms, *i.e.*monadic forms, in Figure 6. We slightly abuse notation to define two different machines $\rightarrow_{anf}$ for A-normal forms, and $\rightarrow_{bnf}$ for B-normal forms. Since $\rightarrow_{anf}$ is a subset of $\rightarrow_{bnf}$ (except for differences in metavariables), we present only one of the $\rightarrow_{bnf}$ rules formally. The machine is optimized in

[2]This makes tail calls a semantically distinct concept in this machine, and not an optimization.

393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441

$$\boxed{C;K \to_{_nf} C;K}$$

$(\mathbf{let}\ (x'\ ((\boldsymbol{\lambda}\ (x)\ M)\ V))\ M');K$	$\to_{_nf}$	$M[x := V];(\mathbf{let}\ (x'\ \bullet)\ M')::K$	CALL	
$V;(F :: K)$	$\to_{_nf}$	$F[V];K$	RETURN	
$((\boldsymbol{\lambda}\ (x)\ M)\ V);K$	$\to_{_nf}$	$M[x := V];K$	TAIL-CALL	
$(\mathbf{let}\ (x\ V)\ M);K$	$\to_{_nf}$	$M[x := V];K$	MOVE	
$(\mathbf{if0}\ 0\ M_1\ M_2);K$	$\to_{_nf}$	$M_1;K$	IFZ	
$(\mathbf{if0}\ V\ M_1\ M_2);K$	$\to_{_nf}$	$M_2;K$	IFNZ	where $V \neq 0$
$(\mathbf{let}\ (x\ (op\ \vec{V}))\ M);K$	$\to_{_nf}$	$M[x := \delta[\![op\ \vec{V}]\!]];K$	PRIMOP	
$(op\ \vec{V});K$	$\to_{_nf}$	$\delta[\![op\ \vec{V}]\!];K$	TAIL-PRIMOP	
$(\mathbf{let}\ (x\ C_1)\ C_2);K$	$\to_{bnf}$	$C_1;(\mathbf{let}\ (x\ \bullet)\ C_2)::K$	BIND	where $C_1 \neq N$

Fig. 6. *A*- and *B*-normal forms CK Machines

the sense that it requires strictly fewer transitions to evaluate ANF terms compared to the λ CK machine, and simpler in the sense that the machine never needs to decompose a term into frame and expression but looks only at the top-level computation, which maps well to a real machine. In fact, the machine is so straightforward, one might imagine that it could be interpreted as specifying a low-level register-transfer language; we do that in Section 4, after studying the high-level ANF machine in more detail. The BIND rule evaluates monadic terms that are not *A*-normal. For non-*A*-normal terms, the RETURN rule is both the return instruction for calls, but also the monadic return.

Now, Listing 9 evaluates as follows.

$$\begin{array}{llll} & (\mathbf{let}\ (x_1\ (+\ 2\ 2))\ (\mathbf{let}\ (x\ 1)\ (\mathbf{let}\ (x_2\ (f\ x))\ (+\ x_1\ x_2)))); & \mathbf{mt} & \\ \to_{anf} & (\mathbf{let}\ (x\ 1)\ (\mathbf{let}\ (x_2\ (f\ x))\ (+\ 4\ x_2))); & \mathbf{mt} & \text{PRIMOP} \\ \to_{anf} & (\mathbf{let}\ (x_2\ (f\ 1))\ (+\ 4\ x_2)); & \mathbf{mt} & \text{MOVE} \end{array}$$

The intermediate push and pop transitions are eliminated.

Monadic form does not remove all intermediate push and pop transitions, in general. It does reduce some stack usage; recall that the above example is also *B*-normal, so *B*-normalization also eliminates all of its stack usage. However, monadic form still supports binding non-trivial computations, such as $(\mathbf{let}\ (x\ (\mathbf{if0}\ 0\ 0\ 1))\ C)$ (as in Listing 13), and so the BIND rule is needed and pushes a frame. Only the optimization (2) applies in a *B*-normal machine, but not optimization (1).

We formalize this in Theorem 3.1. We define the metafunction MAX-STACK on machine traces $\mathcal{D}$ as follows, and prove that *A*-normalizing any *B*-normal form optimizes stack usage.

$$\begin{array}{ll} \text{MAX-STACK}[\![_]\!] : (\mathcal{D} : (C;K \to^*_{_nf} C;K)) \to \mathbb{N} & \\ \text{MAX-STACK}[\![C;K \to^0_{_nf} C;K]\!] & = 0 \\ \text{MAX-STACK}[\![C;K \to_{_nf} C';K' \to^*_{_nf} C'';K'']\!] & = \text{MAX}[\![\text{LEN}[\![K]\!], \text{MAX-STACK}[\![C';K' \to^*_{_nf} C'';K'']\!]]\!] \end{array}$$

THEOREM 3.1 (*A*-NORMALIZATION OPTIMIZES THE STACK (KONTINUATION)). *If $e \longrightarrow^*_A M$ where $\mathcal{D}_B : (e;\boldsymbol{mt} \to^*_{bnf} v;\boldsymbol{mt})$ and $\mathcal{D}_A : (M;\boldsymbol{mt} \to^*_{anf} v;\boldsymbol{mt})$, then trace $\mathcal{D}_A$ uses no more frames than $\mathcal{D}_B$. That is, MAX-STACK$[\![\mathcal{D}_B]\!] \geq$ MAX-STACK$[\![\mathcal{D}_A]\!]$. Furthermore, there exists a program C for which a trace $\mathcal{D}_A$ uses strictly fewer frames.*

PROOF. The proof is straightforward by induction on the trace $\mathcal{D}_B$. When $\mathcal{D}_B$ takes a BIND step, C must take an *A*-reduction, and the *A*-reduction on the C will either not increase or will decrease the MAX-STACK$[\![_]\!]$. □

442
443
444
445
446
447
448
449

$$\begin{array}{lcl} r & \in & \textit{Regions} \\ e & ::= & .... \mid (\textbf{letregion}\ r\ e) \mid (@\ r\ e) \\ E & ::= & .... \mid (@\ r\ E) \\ F & ::= & .... \mid (\textbf{free}\ r) \mid (@\ r\ \cdot) \end{array} \qquad \begin{array}{lcl} o & \in & \textit{Offset} \\ a & ::= & (r\ .\ o) \\ S & ::= & \emptyset \mid (S, a \mapsto v) \end{array}$$

Fig. 7. λ^r Syntax

450 This is more than a theoretical effect. The CertiCoq compiler has had two backends, a CPS and an
451 ANF backend, and ANF outperforms CPS [16–18]. Paraskevopoulou [16] attributes this to additional
452 heap allocation in CPS, noting that "sophisticated techniques to reduce heap allocation" could be
453 used in the CPS backend. CPS can result in additional heap allocation by allocating a continuation
454 for local control. Each frame in the CK machine corresponds to a continuation, and could cause
455 heap allocation of a closure in CPS. The ANF backend, by contrast, avoids this allocation, since
456 these frames are made explicit syntactically and never allocate. This is no more than we should
457 *expect*; *A*-normalization was partially derived from eliminating administrative redexes introduced
458 by CPS translation. This suggests that merely using ANF simplifies control-related optimization
459 compared to CPS, at least in the context of a fully mechanically verified compiler.

460

461 ### 3.2 A Region Calculus and Machine

462 Unfortunately, *A*-normalization is an unsafe optimization in some settings. In general, *A*-normalization
463 is *unsafe for scope*: lexical scope is extended, changing variable extent, lifetimes, and other effects re-
464 lated to lexical scope. Extensionally, such as in a pure calculus when reasoning up to α-equivalence,
465 this is irrelevant. However, it matters when reasoning about intensional properties or effectful
466 calculi. We consider a version of the scoped regions of Tofte and Talpin [20].
467 We first extend our λ-calculus with lexically scoped regions and region allocation; the syntax is
468 given in Figure 7. The expression (**letregion** r e) runs e with a new region r allocated in the store
469 S[3]. After e returns, the region r is freed. In this calculus, all values must be explicitly allocated in
470 a region in the store. The form (**@** r e) evaluates to a reference, by evaluating e to a value that is
471 stored in region r. All primitive operations and value must be explicitly allocated; the result of a
472 function must also be allocated before it returns. We also extend evaluation contexts, for *A*- and
473 *B*-normalization of this syntax, and add a new frames for the machine semantics. The new frame
474 (**free** r) is unusual; it is not a frame of an evaluation context, but instead directs the machine to
475 free region r.
476 In Figure 8, we give a CSK machine [5–7] for λ^r. The machine includes a straightforward
477 extension of the transitions from the λ CK machine in Figure 5. The machine is unrealistic in that
478 all values are passed by references, whereas a realistic machine might avoid boxing word-sized
479 data. However, this detail is irrelevant for our purposes: some ANF terms bind non-word sized data,
480 so *any* extension of a region lifetime is unsafe for space.
481 The machine is essentially similar to the λ CK machine, with two main differences: the represen-
482 tation of run-time values, and the new transitions for regions. The new region-related transitions
483 are given below the line. Run-time values are addresses, a pair $(r\ .\ o)$ of a region and an offset into
484 that region, so all computations dereference an address a in the store S, written $S(a)$. Syntactic
485 values must be allocated in a region, indicated by the frame, in the ALLOC transition. (**letregion** r e)

486

487 [3]The allocation pattern forms a (data) stack (of regions), as explained by Tofte and Talpin [20]. The stack of regions is
488 irrelevant to us, while the control stack (kontinuation) K *is* important, so we refer to allocation as occurring in the heap,
489 and use "stack" to refer to the control stack.

490

$$\boxed{C;S;K \rightarrow_{\lambda r} C;S;K}$$

$$\begin{array}{rcllr}
(a_1\ a_2);S;K & \rightarrow_{\lambda r} & e[x := a_2];S;K & \textsc{call} & \text{where } (\boldsymbol{\lambda}\ (x)\ e) = S(a_1) \\
(\textbf{let}\ (x\ a)\ e);S;K & \rightarrow_{\lambda r} & e[x := a];S;K & \textsc{move} & \\
(\textbf{if0}\ a\ e_1\ e_2);S;K & \rightarrow_{\lambda r} & e_1;S;K & \textsc{ifz} & \text{where } 0 = S(a) \\
(\textbf{if0}\ a\ e_1\ e_2);S;K & \rightarrow_{\lambda r} & e_2;S;K & \textsc{ifnz} & \text{where } 0 \neq S(a) \\
(op\ \vec{a});S;K & \rightarrow_{\lambda r} & [\![op\ \overrightarrow{S(a)}]\!];S;K & \textsc{primop} & \\
F[e];S;K & \rightarrow_{\lambda r} & e;S;(F :: K) & \textsc{push} & \text{where } e \neq a \\
a;S;(F :: K) & \rightarrow_{\lambda r} & F[S(a)];S;K & \textsc{pop} & \\
\hline
(\textbf{letregion}\ r\ e);S;K & \rightarrow_{\lambda r} & e;S;((\textbf{free}\ r) :: K) & \textsc{ralloc} & \\
a;S;((\textbf{free}\ r) :: K) & \rightarrow_{\lambda r} & a;\textsc{free}[\![S,r]\!];K & \textsc{rfree} & \\
v;S;((@\ r\ \bullet) :: K) & \rightarrow_{\lambda r} & (r\ .\ o);(S[(r.o) := v];K & \textsc{alloc} & \text{fresh } o
\end{array}$$

Fig. 8. λ^r CSK Machine

(implicitly) allocates a new region, and adds a **free** frame to the stack. The metafunction FREE$[\![S,r]\!]$ deallocates all addresses with the region r in S.

Consider the following example term's evaluation in this machine. This example is an λ^r equivalent of (* (* 1 2) (* 3 4)), with region sizes and lifetimes minimized. r_0 is an initial region in which the final result is allocated.

$$\begin{array}{ll|l|l}
 & (\textbf{letregion}\ r_2 & \emptyset & \textbf{mt} \\
 & \quad (@\ r_0\ (*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2)))) & & \\
 & \qquad\qquad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4))))))) & & \\
\rightarrow_{\lambda r} & (@\ r_0\ (*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2)))) & \emptyset & (\textbf{free}\ r_2) :: \textbf{mt} \\
 & \qquad\qquad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4)))))) & & \\
\rightarrow_{\lambda r} & (*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2)))) & \emptyset & (@\ r_0\ \bullet)::(\textbf{free}\ r_2):: \textbf{mt} \\
 & \quad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4))))) & & \\
\rightarrow_{\lambda r} & \ldots & &
\end{array}$$

$$\begin{array}{ll|l|l}
\rightarrow_{\lambda r} & (*\ (r_2.o_1)\ (r_2.o_2)) & [(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12] & (@\ r_0\ \bullet)::(\textbf{free}\ r_2)::\textbf{mt} \\
\rightarrow_{\lambda r} & 24 & [(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12] & (@\ r_0\ \bullet)::(\textbf{free}\ r_2)::\textbf{mt} \\
\rightarrow_{\lambda r} & (r_0.o_3) & [(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12][(r_0.o_3) \mapsto 24] & (\textbf{free}\ r_2)::\textbf{mt} \\
\rightarrow_{\lambda r} & (r_0.o_3) & [(r_0.o_3) \mapsto 24] & \textbf{mt}
\end{array}$$

Using this abstract machine, we can measure some simple allocation behaviour, such as:

(1) What is the maximum number of regions live at once (MAX-REGIONS$[\![_]\!]$)?
(2) What is the maximum number of live addresses, in all regions, at once (MAX-MEMORY$[\![_]\!]$)?

We omit their formal definitions, but these metafunctions are essentially similar to definition of MAX-STACK$[\![_]\!]$: they take a trace and measuring the maximum value in the trace. For the above example, the MAX-REGIONS$[\![_]\!]$ of the trace is 3 and the MAX-MEMORY$[\![_]\!]$ is 4.

We extend A-normalization to λ^r with the following two A-reductions.

$$\begin{array}{rcll}
 & \ldots\ldots & & \\
E[(\textbf{letregion}\ r\ e)] & \longrightarrow_A & (\textbf{letregion}\ r\ E[e]) & A_4 \\
 & & \text{where } E \neq \cdot & \\
E[(@\ r\ N)] & \longrightarrow_A & (\textbf{let}\ (x'\ (@\ r\ N))\ E[x']) & A_5 \\
 & & \text{where } E \neq \cdot, E \neq E'[(\textbf{let}\ (x\ \bullet)\ e')], E \neq E'[(@\ r\ \bullet)], \text{and fresh } x', &
\end{array}$$

The side condition $E \neq E'[(@\ r\ \bullet)]$ must also be added to the A_3 reduction. The reduction system is fully implemented in the anonymous supplementary materials.

A_4 extends the lifetime of region r, but this is necessary for A_3 from Figure 1 to lift an intermediate computation. Concretely, A-normalizing the running example, we get the term in Listing 15, which runs with MAX-REGIONS⟦_⟧ 4 and a MAX-MEMORY⟦_⟧ of 7, compared to the original 3 and 4, respectively.

```
(letregion r2
 (letregion r1
  (let (x (@ r1 1))
   (let (x1 (@ r1 2))
    (let (x2 (@ r2 (* x x1)))
     (letregion r3
      (let (x3 (@ r3 3))
       (let (x4 (@ r3 4))
        (let (x5 (@ r2 (* x3 x4)))
         (@ r0 (* x2 x5))))))))))
```

Listing 15. A-normalization of λ^r Example

```
(letregion r2
 (let (x4 (letregion r1
            (let (x2 (@ r1 1))
             (let (x3 (@ r1 2))
              (@ r2 (* x2 x3))))))
  (let (x5 (letregion r3
             (let (x (@ r3 3))
              (let (x1 (@ r3 4))
               (@ r2 (* x x1))))))
   (@ r0 (* x4 x5)))))
```

Listing 16. B-normalization of λ^r Example

Theorem 3.2 (A-normalization is Unsafe for Scope).
If $C \longrightarrow_A^* M$, $\mathcal{D}_B : (C; \emptyset; \boldsymbol{mt} \rightarrow_{\lambda r}^* a; S'; \boldsymbol{mt})$ and $\mathcal{D}_A : (M; \emptyset; \boldsymbol{mt} \rightarrow_{\lambda r}^* a; S''; \boldsymbol{mt})$, then MAX-REGIONS$[\![\mathcal{D}_A]\!] \geq$ MAX-REGIONS$[\![\mathcal{D}_B]\!]$, and MAX-MEMORY$[\![\mathcal{D}_A]\!] \geq$ MAX-MEMORY$[\![\mathcal{D}_B]\!]$. Furthermore, there exists C and $\mathcal{D}_A$ such that MAX-MEMORY$[\![D_A]\!] >$ MAX-MEMORY$[\![D_B]\!]$ and MAX-REGIONS$[\![D_A]\!] >$ MAX-REGIONS$[\![D_B]\!]$.

By contrast, we can extend B-normalization to λ^r with the following reductions.

$$
\begin{array}{rcll}
 & \cdots & & \\
E^b[(\textbf{letregion}\ r\ e)] & \longrightarrow_B & (\textbf{let}\ (x\ (\textbf{letregion}\ r\ e))\ \ E^b[x]) & B_4 \\
 & & E^b \neq \cdot & \\
E^b[(@\ r\ N)] & \longrightarrow_B & (\textbf{let}\ (x'\ (@\ r\ N))\ E^b[x']) & B_5 \\
 & & E^b \neq \cdot, E^b \neq E_1^b[(\textbf{let}\ (x\ \cdot)\ e')], E^b \neq E_2^b[(@\ r\ \bullet)], \text{fresh } x' &
\end{array}
$$

Since **letregion** is effectful, B_4 binds it instead of lifting it, preserving the original lifetime. The B-normalization of our running example is given in Listing 16.

Theorem 3.3 (B-normalization is Safe-for-Scope).
If $e \longrightarrow_B^* C$, $\mathcal{D}_\lambda : (e; \emptyset; \boldsymbol{mt} \rightarrow_{\lambda r}^* a; S'; \boldsymbol{mt})$ and $\mathcal{D}_B : (C; \emptyset; \boldsymbol{mt} \rightarrow_{\lambda r}^* a; S''; \boldsymbol{mt})$, then MAX-REGIONS$[\![\mathcal{D}_\lambda]\!] =$ MAX-REGIONS$[\![\mathcal{D}_B]\!]$, and MAX-MEMORY$[\![\mathcal{D}_\lambda]\!] =$ MAX-MEMORY$[\![\mathcal{D}_B]\!]$.

4 IMPERATIVE A-NORMALIZATION

The problems with normalizing commuting conversion are merely with the syntax of ANF, and not with (machine) semantics of commuting conversion. By designing a syntax based on the ANF abstract machine, we get an imperative language very similar to ANF. Working backwards, un-normalizing commuting conversions, we design an imperative monadic language in which A-normalization is safe, neither extending scope, nor duplicating continuations.

Recall from Figure 6 that each transition in the monadic CK machine uniquely maps to an A-normal form expression. In Figure 9, we present an imperative A-normal form designed from the transitions in the abstract machine, with **begin** for sequencing imperative statements.

$$
\begin{array}{lcl}
t & ::= & (\textbf{begin}\ \vec{s}\ t) \mid v \mid (\textbf{if0}\ v\ t\ t) \mid (\textbf{call}\ v\ \vec{v}) \mid (op\ \vec{v}) \\
s & ::= & (\textbf{begin}\ \vec{s}) \mid (\textbf{set!}\ x\ v) \mid (\textbf{set!}\ x\ (op\ \vec{v})) \mid (\textbf{set!}\ x\ (\textbf{call}\ v\ \vec{v})) \\
v & ::= & (\boldsymbol{\lambda}\ (x)\ t) \mid \iota
\end{array}
$$

Fig. 9. Imperative ANF Syntax (AB-normal form)

Intuitively, a program is a sequence of statements s followed by a tail-position operation (tail) t producing the final value. We interpret the ANF **let** as an effectful **set**! operation; rather than performing substitution, we set a variable in the machine state. Tails t include operations for machine transitions that correspond to all expression in M position in ANF. For example, the TAIL-CALL transition is realized by the **call** statement in tail position. All non-tail expressions in ANF must appear on the right-hand side of a **let** in ANF, and therefore appear on the right-hand side of **set**! in imperative ANF. These transitions—CALL, MOVE, and PRIMOP—are realized by statements s. The result is similar to a register-transfer language, although one with an infinite set of registers x, higher-order functions, and primitive call and return.

The translation from ANF into this syntax is straightforward; we give a definition later in Figure 18a. But for now, we want to work backwards from this syntax to imperative monadic form, and demonstrate that an imperative interpretation of A-normalization within imperative monadic form is safe and easy. So what must imperative monadic form be?

If **set**! is **let**, then we also need to support (**set**! x t), that is, a tail computation on the right-hand side an assignment. This corresponds to (**let** (x C) C) in monadic form. We see this also by doing code generation of Equation Associativity and Equation Commute, which would look like the following.

$$(\textbf{begin}\ (\textbf{set!}\ y\ (\textbf{begin}\ (\textbf{set!}\ x\ t)\ t))\ t) \equiv (\textbf{begin}\ (\textbf{set!}\ x\ t)\ (\textbf{set!}\ y\ t)\ t) \qquad \text{(Imp. Associativity)}$$

$$(\textbf{begin}\ (\textbf{set!}\ y\ (\textbf{if}\ v\ t_1\ t_2))\ t) \equiv (\textbf{if0}\ v\ (\textbf{begin}\ (\textbf{set!}\ y\ t_1)\ t)\ (\textbf{begin}\ (\textbf{set!}\ y\ t_2)\ t)) \qquad \text{(Imp. Commute (1))}$$

To support both sides of the equations, we must allow **begin** and **if0** on the right-hand side of a **set**!. This requires one new statement.

$$
\begin{array}{lcl}
t & ::= & (\textbf{begin}\ \vec{s}\ t) \mid v \mid (\textbf{if0}\ v\ t\ t) \mid (\textbf{call}\ v\ \vec{v}) \mid (op\ \vec{v}) \\
s & ::= & (\textbf{begin}\ \vec{s}) \mid (\textbf{set!}\ x\ v) \mid \boxed{(\textbf{set!}\ x\ t)} \mid (\textbf{set!}\ x\ (op\ \vec{v})) \mid (\textbf{set!}\ x\ (\textbf{call}\ v\ \vec{v})) \\
v & ::= & (\boldsymbol{\lambda}\ (x)\ t) \mid \iota
\end{array}
$$

Equation Imp. Commute (1) still duplicates the continuation t, but only because the syntax supports conditional *expressions*, but not conditional *statements*. If we add a conditional statement, we could rephrase the equation as the following.

$$(\textbf{begin}\ (\textbf{set!}\ y\ (\textbf{if}\ v\ t_1\ t_2))\ t) \equiv (\textbf{begin}\ (\textbf{if0}\ v\ (\textbf{set!}\ y\ t_1)\ (\textbf{set!}\ y\ t_2))\ t) \qquad \text{(Imp. Commute (2))}$$

We perform the commuting conversion by duplicating the assignment to x rather than duplicating the context in which x is bound. This isn't possible in (high-level) ANF, since lexical binding cannot export x, but it is possible when all lexical binding has been transformed into imperative statements.

Adding the conditional statement and **set**! with a complex right-hand side, we get the final version of the imperative monadic form is in Figure 10.

In this imperative monadic syntax, we can easily perform the imperative equivalent of A-normalization into imperative ANF by taking the congruent closure of the following set of reductions

638
639
640

$$
\begin{array}{lcl}
t & ::= & (\textbf{begin}\ \vec{s}\ t) \mid v \mid (\textbf{if0}\ v\ t\ t) \mid (\textbf{call}\ v\ v) \mid (op\ \vec{v}) \\
s & ::= & (\textbf{begin}\ \vec{s}) \mid (\textbf{set!}\ x\ v) \mid \boxed{(\textbf{if0}\ v\ s\ s)} \mid \boxed{(\textbf{set!}\ x\ t)} \mid (\textbf{set!}\ x\ (op\ \vec{v})) \mid (\textbf{set!}\ x\ (\textbf{call}\ v\ v)) \\
v & ::= & (\boldsymbol{\lambda}\ (x)\ t) \mid \iota
\end{array}
$$

641
642 Fig. 10. Imperative Monadic Syntax
643
644
645
646
647

$$
\begin{array}{lcl}
hv & ::= & \iota \mid (\textbf{closure}\ \Sigma\ (\boldsymbol{\lambda}\ (x)\ t)) \\
wv & ::= & x \mid \iota
\end{array}
\qquad
\begin{array}{lcl}
\Sigma & ::= & \emptyset \mid \Sigma[x \mapsto hv] \\
F & ::= & (\textbf{begin}\ (\textbf{set!}\ x\ \cdot)\ \vec{s}\ t) \\
K & ::= & \textbf{mt} \mid F :: K
\end{array}
$$

648
649 Fig. 11. CEK Machine Syntax
650
651 $AB = \{AB_1, AB_2\}$, yielding AB-normal form[4].
652
653

$$
\begin{array}{rcll}
(\textbf{set!}\ x\ (\textbf{if0}\ v\ t_1\ t_2)) & \longrightarrow_{AB} & (\textbf{if0}\ v\ (\textbf{set!}\ x\ t_1)\ (\textbf{set!}\ x\ t_2)) & AB_1 \\
(\textbf{set!}\ x\ (\textbf{begin}\ \vec{s}\ t)) & \longrightarrow_{AB} & (\textbf{begin}\ \vec{s}\ (\textbf{set!}\ x\ t)) & AB_2
\end{array}
$$

654
655 *AB*-normalization is straightforward. Unlike *A*-normalization and *B*-normalization, it does not
656 shuffle the evaluation contexts at all. It is a completely local transformation. We never need to deal
657 with join points, or CPS'd compilers, or code duplication. We gain all the stack optimization effect
658 on *A*-normalization (as we formalize next), the advantages of a register-transfer syntax for code
659 generation and machine implementation, etc.
660 *The key idea* in *AB*-normalization is that lexical expressions must be elaborated into imperative
661 statements before *A*-normalization.
662
663 ### 4.1 Machine Semantics of AB-normal form

664 We use a CEK machine [5–7] to define a machine semantics for imperative monadic form in
665 Figure 11 and Figure 12. We then show that *AB*-normalization optimizes the stack in the same was
666 as *A*-normalization.
667 The environment Σ (to distinguish it from evaluation contexts) represents the register file,
668 mapping variables to their values. We separate word values *wv*, which represent run-time values
669 that can be eliminated, and heap values *hv*, which must be bound to a name and intuitively would
670 be allocated in memory. We use the syntax $\Sigma(wv)$ get the value of wv, either dereferencing wv in
671 Σ if wv is a variable, or returning wv if not. The code continues to represent the program counter,
672 and the kontinuation kontinues to represent the call stack. W.l.o.g., we assume that all **λ**s appear
673 on the right-hand side of a **set!**, to simplify implementing closures. This can be implemented by
674 another *B*-reduction that binds **λ**, treating it as a computation rather than a value (which it is, in
675 this machine; it performs allocation).
676 This machine is essentially similar to the monadic CK machine in Figure 6. The main difference
677 is the use of an environment rather than substitution, becomes necessary with imperative inter-
678 pretation of **let** as **set!**. We also require two administrative reductions, ADMIN1 and ADMIN2, to
679 enable composing internally with **begin**, although these could be normalized to simplify the final
680 machine semantics.
681 There is one key difference: the AB transition, which cannot occur in ABNF but can in imperative
682 monadic form. This rule, and CALL for a non-tail call, both push a frame (**begin** (**set!** $x\ \cdot$) t). After
683
684 [4]We choose the name *AB* for two reasons: (1) this normalization yields the best of both *A*- and *B*-normalization (2) *AB*-
685 normalization follows *B*-normalization, which was defined second after *A*-normalization, so *AB* is 3... in big-endian binary,
anyway.
686

$$\boxed{t;\Sigma;K \longrightarrow_{CEK} t;\Sigma;K}$$

$$\begin{array}{rcll}
(\textbf{begin}\ (\textbf{set!}\ x\ wv)\ \vec{s}\ t);\Sigma;K & \longrightarrow_{CEK} & t;\Sigma[x := wv];K & \text{MOVE}\\
(\textbf{begin}\ (\textbf{set!}\ x\ (\boldsymbol{\lambda}\ (x)\ t))\ \vec{s}\ t);\Sigma;K & \longrightarrow_{CEK} & t;\Sigma[x :=(\textbf{closure}\ \Sigma\ (\boldsymbol{\lambda}\ (x)\ t))];K & \text{CLOSURE}\\
(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{call}\ wv_1\ wv_2))\ \vec{s}\ t);\Sigma;K & \longrightarrow_{CEK} & t;\Sigma'[y := wv_2];(\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ \vec{s}\ t) :: K & \text{CALL}\\
 & & \text{where } \Sigma(wv_1) = (\textbf{closure}\ \Sigma'\ (\boldsymbol{\lambda}\ (y)\ t)) & \\
v;\Sigma;F :: K & \longrightarrow_{CEK} & F[v];\Sigma;K & \text{RETURN}\\
(\textbf{begin}\ (\textbf{if0}\ wv\ s_1\ s_2)\ t);\Sigma;K & \longrightarrow_{CEK} & (\textbf{begin}\ s_1\ t);\Sigma;K & \text{IFZ-S}\\
 & & \text{where } \Sigma(wv) = 0 & \\
(\textbf{begin}\ (\textbf{if0}\ wv\ s_1\ s_2)\ t);\Sigma;K & \longrightarrow_{CEK} & (\textbf{begin}\ s_2\ t);\Sigma;K & \text{IFNZ-S}\\
 & & \text{where } \Sigma(wv) \neq 0 & \\
(\textbf{begin}\ (\textbf{set!}\ x\ (op\ \overrightarrow{wv}))\ t);\Sigma;K & \longrightarrow_{CEK} & t;\Sigma[x := [\![op\ \overrightarrow{\Sigma(sv)}]\!]];K & \text{PRIMOP}\\
(op\ \overrightarrow{wv});\Sigma;K & \longrightarrow_{CEK} & [\![op\ \overrightarrow{\Sigma(wv)}]\!];\Sigma;K & \text{TAIL-PRIMOP}\\
(\textbf{if0}\ wv\ t_1\ t_2);\Sigma;K & \longrightarrow_{CEK} & t_1;\Sigma;K & \text{IFZ-T}\\
 & & \text{where } \Sigma(wv) = 0 & \\
(\textbf{if0}\ wv\ t_1\ t_2);\Sigma;K & \longrightarrow_{CEK} & t_2;\Sigma;K & \text{IFNZ-T}\\
 & & \text{where } \Sigma(wv) \neq 0 & \\
(\textbf{begin}\ (\textbf{set!}\ x\ t_1)\ \vec{s}\ t_2);\Sigma;K & \longrightarrow_{CEK} & t_1;\Sigma;(\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ \vec{s}\ t_2) :: K & \text{AB}\\
(\textbf{begin}\ (\textbf{begin}\ \vec{s_1})\ \vec{s_2}\ t);\Sigma;K & \longrightarrow_{CEK} & (\textbf{begin}\ \vec{s_1}\ \vec{s_2}\ t);\Sigma;K & \text{ADMIN1}\\
(\textbf{begin}\ t);\Sigma;K & \longrightarrow_{CEK} & t;\Sigma;K & \text{ADMIN2}
\end{array}$$

Fig. 12. CEK Machine for Imperative Monadic Form

AB-normalization, the AB rule cannot occur, so we regain the useful property that only non-tail calls push a stack frame.

It's straightforward to show that AB-normalization is an optimization of the stack, just like A-normalization.

Theorem 4.1 (AB-normalization Optimizes the Stack). *If* $t \longrightarrow^*_{AB} t'$ *where* $\mathcal{D}_B : (t;\emptyset;\boldsymbol{mt} \rightarrow^*_{CEK} v;S;K)$ *and* $\mathcal{D}_A : (t';\emptyset;\boldsymbol{mt} \rightarrow^*_{CEK} v';S';K')$*, then trace* $\mathcal{D}_A$ *uses no more frames than* $\mathcal{D}_B$*. That is,* $\text{MAX-STACK}[\![\mathcal{D}_B]\!] \geq \text{MAX-STACK}[\![\mathcal{D}_a]\!]$*. Furthermore, there exists a program t for which a trace* $\text{MAX-STACK}[\![\mathcal{D}_B]\!] > \text{MAX-STACK}[\![\mathcal{D}_A]\!]$*.*

Proof. The proof proceeds by induction on the trace $\mathcal{D}_B$. The two interesting cases correspond to the AB reductions; all other cases preserve max stack size. We sketch the case for $t =$ $(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{begin}\ s\ t_1))\ t_2)$ as a diagram; the other case is similar.

$$\begin{array}{ccc}
(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{begin}\ \vec{s}\ t_1))\ t_2);\Sigma;K & \xrightarrow[CEK*]{} & (\textbf{begin}\ \vec{s}\ t_1);\Sigma;(\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ t_2) :: K\\
\Big\downarrow AB_2 & & \Big\downarrow CEK\text{-AB}\\
 & & t_1;\Sigma';K' + (\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ t_2) :: K\\
 & & \vdots\ {>}\text{max-stack}\\
(\textbf{begin}\ \vec{s}\ (\textbf{set!}\ x\ t_1)\ t_2);\Sigma;K & \xrightarrow[CEK*]{\qquad\qquad} & (\textbf{begin}\ (\textbf{set!}\ x\ t_1)\ t_2);\Sigma;K' + K
\end{array}$$

Before AB-normalization, a program of this shape must push the stack frame $F = (\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ t_2)$, since the right-hand side of the instruction is a complex tail. But AB-normalization, via the AB_2 rule, eliminates that one stack frame by reassociating the **set!**. In the machine state after evaluating the AB-normalized program, the instructions $\vec{s}$ evaluate in a subtrace, producing a stack $K' + K$ (where $+$ is the append operation on stacks). The resulting stack is one frame smaller than $K' + F :: K$. The

$$
\begin{array}{rcl}
t & ::= & (\textbf{begin}\ \vec{s}\ t) \mid rv \mid \boxed{(\textbf{alloc}\ r\ v)} \mid (\textbf{if0}\ rv\ t\ t) \mid (\textbf{call}\ rv\ rv) \mid \boxed{(\textbf{alloc}\ r\ (op\ \vec{rv}))} \\
s & ::= & (\textbf{begin}\ \vec{s}) \mid (\textbf{if0}\ rv\ s\ s) \mid (\textbf{set!}\ x\ \boxed{(\textbf{alloc}\ r\ v)}) \mid (\textbf{set!}\ x\ rv) \mid (\textbf{set!}\ x\ t) \\
 & & \mid (\textbf{set!}\ x\ \boxed{(\textbf{alloc}\ r\ (op\ \vec{rv}))}) \mid (\textbf{set!}\ x\ (\textbf{call}\ rv\ rv)) \mid \boxed{(\textbf{ralloc}\ r)} \mid \boxed{(\textbf{rfree}\ r)}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
v & ::= & \iota \mid x \mid a \mid (\lambda\ (x)\ t) & hv & ::= & (\textbf{closure}\ \Sigma\ (\lambda\ (x)\ t)) \mid wv \\
a & ::= & (r.o) & F & ::= & (\textbf{begin}\ (\textbf{set!}\ x\ \cdot)\ \vec{s}\ t) \\
rv & ::= & a \mid x & S & ::= & \emptyset \mid S[a \mapsto hv] \\
wv & ::= & \iota \mid a & \Sigma & ::= & \emptyset \mid \Sigma[x := a]
\end{array}
$$

Fig. 13. Imperative Monadic w/ Regions Syntax

result follows by the induction hypotheses, which guarantees that the subtraces for $\vec{s}$, t_1, and t_2 do not increase the stack size. □

4.2 AB-normalization and Regions

Recall our key idea claims that we must elaborate lexical expressions into imperative statements to solve A-normalization. Let us test this idea against lexically scoped regions.

Unlike **let**[5], **letregion** has an effect at the beginning and end of its scope. We can compile lexical regions using something like the following code generator.

$$
\begin{array}{rcl}
\textsc{cg}[\![_]\!] & : & C \to t \\
\textsc{cg}[\![(\textbf{letregion}\ r\ C)]\!] & = & (\textbf{begin}\ (\textbf{ralloc}\ r)\ (\textbf{set!}\ x\ \textsc{cg}[\![C]\!])\ (\textbf{rfree}\ r)\ x) \\
\textsc{cg}[\![(@\ r\ N)]\!] & = & (\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{alloc}\ r\ \textsc{cg}[\![N]\!]))\ x)
\end{array}
$$

This syntax begins to look suspiciously like monadic regions [9, 10].

We extend the CEK machine into a CESK machine [5–7], using the store to model regions. The machine essentially smashes together the previous two machines. We give only the key rules, for brevity, but a complete implementation is available in the anonymous supplementary materials.

We extend our imperative monadic syntax with imperative regions in Figure 13. There are two main differences. First, the additional instructions for **ralloc** and **rfree**, which correspond to the RALLOC and RFREE transitions of the CSK machine for the monadic region calculus. Second, as all values must be passed by reference, all the prior value positions and primitive operators are wrapped in **alloc**, which corresponds to the ALLOC transition of the CSK machine in Figure 8. To ensure all values are passed by reference, all value operands are now register values *rv*—either a variable *x* or an address. Heap values are now explicitly allocated in the store.

We define the key rules CESK machine in Figure 14. The MOVE transition dereferences a register value *rv*, either reading the value from a register *x* or if *rv* is an *a* using it directly. Then the register file is updated to map *x* to that value. The ALLOC transition allocates a word value *wv* in memory to a fresh address in region *r*, updating the regsiter file with *x* mapped to that address. There are analogous instructions for allocating closures and allocating the result of a primop, as well as analogous instructions for allocating in tail position. The RALLOC transition allocates the region *r*, which in this machine is a no-op. The RFREE transition frees a region. These two corresponds to the transitions of the same name in the CSK machine, but as they are now instructions, they do not use the stack. The only transitions that push a stack frame are the non-tail CALL transition, and the AB transition for not *AB*-normal terms.

[5]Strictly, there is an effect at the end of **let**'s scope: the variable is dead. We could introduce a marker for the end-of-lifetime of a variable into the imperative language. But this sort of information is simple to infer via liveness analysis.

$$\boxed{t; \Sigma; S; K \rightarrow_{CESK} t; \Sigma; S; K}$$

$$\cdots$$

$(\textbf{begin}\ (\textbf{set!}\ x\ rv)\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$(\textbf{begin}\ \vec{s}\ t); \Sigma[x := \Sigma(rv)]; S; K$	MOVE
$(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{alloc}\ r\ wv))\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$(\textbf{begin}\ \vec{s}\ t); \Sigma[x := (r.o)]; S[(r.o) := v]; K$ fresh o	ALLOC
$(\textbf{begin}\ (\textbf{ralloc}\ r)\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$(\textbf{begin}\ \vec{s}\ t); \Sigma; S; K$	RALLOC
$(\textbf{begin}\ (\textbf{rfree}\ r)\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$(\textbf{begin}\ \vec{s}\ t); \Sigma; \mathrm{free}(S, r); K$	RFREE
$(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{call}\ rv_1\ rv_2))\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$t_1; \Sigma_1[y := rv_2]; S; ((\textbf{begin}\ (\textbf{set!}\ x\ \cdot)\ \vec{s}\ t) :: K)$ where $(\textbf{closure}\ \Sigma_1\ (\boldsymbol{\lambda}\ (y)\ t_1)) = S(\Sigma(rv_1))$	CALL
$(\textbf{begin}\ (\textbf{set!}\ x\ t_1)\ \vec{s}\ t); \Sigma; S; K$	$\rightarrow_{CESK}$	$t_1; \Sigma; S; K$	AB

Fig. 14. CESK Machine for Imperative Monadic w/ Regions

With this machine, we can now *AB*-normalize and run our earlier region example *without* extending lifetimes, but *with* optimized stack usage. Recall from Listing 15 that *A*-normalization increased MAX-REGIONS⟦_⟧ and MAX-MEMORY⟦_⟧. In Listing 17, we perform code generation followed by *AB*-normalization on the monadic example from Listing 16. In Listing 17, we perform code generation of the ANF example from Listing 15.

```
1  (begin
2   (ralloc r2)
3   (ralloc r1)
4   (set! x2 (alloc r1 1))
5   (set! x3 (alloc r1 2))
6   (set! xt1 (alloc r2 (* x2 x3)))
7   (rfree r1)
8   (set! x4 xt1)
9   (ralloc r3)
10  (set! x (alloc r3 3))
11  (set! x1 (alloc r3 4))
12  (set! xt2 (alloc r2 (* x x1)))
13  (rfree r3)
14  (set! x5 xt2)
15  (set! xt3 (alloc r0 (* x4 x5)))
16  (rfree r2)
17  xt3)
```

Listing 17. AB-Normalized · Code Gen · Monadic

```
1  (begin
2   (ralloc r2)
3   (ralloc r1)
4   (set! x (alloc r1 1))
5   (set! x1 (alloc r1 2))
6   (set! x2 (alloc r2 (* x x1)))
7   (ralloc r3)
8   (set! x3 (alloc r3 3))
9   (set! x4 (alloc r3 4))
10  (set! x5 (alloc r2 (* x3 x4)))
11  (set! xt1 (alloc r0 (* x2 x5)))
12  (rfree r3)
13  (set! xt2 xt1)
14  (rfree r1)
15  (set! xt3 xt2)
16  (rfree r2)
17  xt3)
```

Listing 17. Code Generation · ANF

The lifetime of *r1* is extended by *A*-normalization, but not *B*-normalization or *AB*-normalization, and *AB*-normalization has eliminated all the stack usage of monadic form. The formal proof is similar to that of Theorem 3.2; note that *AB*-normalization does not change the order of instructions.

THEOREM 4.2 (*AB*-NORMALIZATION IS SAFE-FOR-SCOPE).
If $t_M \longrightarrow^*_{AB} t_{AB}$, $\mathcal{D}_M : (t_M; \emptyset; \emptyset; \boldsymbol{mt} \rightarrow^*_{CESK} a; \Sigma; S; \boldsymbol{mt})$ *and* $\mathcal{D}_{AB} : (t_{AB}; \emptyset; \emptyset; \boldsymbol{mt} \rightarrow^*_{CESK} a; \Sigma'; S'; \boldsymbol{mt})$, *then* MAX-REGIONS$[\![\mathcal{D}_M]\!]$ = MAX-REGIONS$[\![\mathcal{D}_{AB}]\!]$, *and* MAX-MEMORY$[\![\mathcal{D}_M]\!]$ = MAX-MEMORY$[\![\mathcal{D}_{AB}]\!]$, *and*

834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882

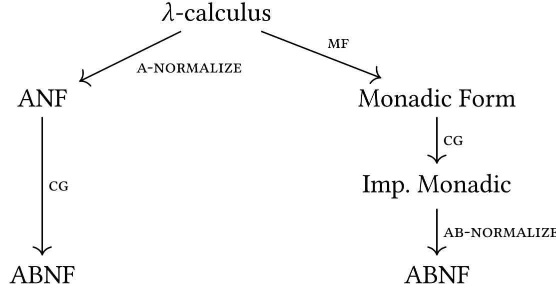

Fig. 15. Compiler Architecture

MAX-STACK$[\![\mathcal{D}_M]\!] \geq$ *MAX-STACK*$[\![\mathcal{D}_{AB}]\!]$. *Furthermore, there exists a program t with trace $\mathcal{D}_{AB}$ such that* *MAX-STACK*$[\![D_M]\!] >$ *MAX-STACK*$[\![D_{AB}]\!]$.

5 AN AB-NORMAL COMPILER

Formalizing these normal forms as reduction systems is useful for metatheory, allowing us to prove generic properties of any compiler that targets these normal forms, but the formalism ignores important details about how to implement a compiler using these normal forms. In this section, we define two compilers: the ABnormal compiler, and the ANF compiler. Their pipelines are summarized in Figure 15. The ABnormal compiler targets *AB*-normal form via monadic form and *AB*-normalization, while the ANF compiler target *AB*-normal form via *A*-normalization. We show that the compiler design is simplified using *AB*-normalization compared to *A*-normalization.

5.1 ANF Compiler

There are pragmatic problems to implementing an effective compiler into ANF. This is one of the major disadvantages of target ANF directly: normalizing commuting conversions while also sequencing computations introduces (unnecessary) complexity.

We define ANF translation in Figure 16. This implementation requires managing some complexity.

The main issue in the definition is that the result of ANF translation is an M, which cannot be composed internally with another M. Some other external technique is needed to compose two Ms. This is a result of the requirement that we normalize Equation Associativity and Equation Commute. We use the original technique of Flanagan et al. [8], in which the compiler reifies the evaluation context of the reduction system as a meta-language continuation that builds target language terms. The compiler is indexed by this continuation, and builds up the translation in the continuation, rather than directly returning the translated term. The continuation has type $V \longrightarrow M$. The original Flanagan et al. [8] version did not actually produce ANF terms, but monadic terms, to avoid code duplication. The typical solution in the literature to generating ANF without code duplication is to introduce a *join point* [1, 3, 11, 14]. This implementation technique has been formalized and verified to be correct with respect to typing, whole program compilation, and separate compilation by Koronkevich et al. [13] in the context of an ANF compiler for dependent types.

The first bit of complexity is that we write the compiler in CPS. This adds some complexity to the implementation, and may negatively impact compile-time performance. The compiler ANF$[\![e]\!]\kappa$ takes a source term e and an ANF evaluation context (continuation) κ. When e is a value v, it's translated by calling the continuation with the value $\kappa[v]$, forming a complete ANF program. Otherwise, the subterms are translated, calling the compiler in CPS: a subterm is translated with a

$$\kappa \quad ::= \quad \bullet \mid (\textbf{let}\ (x\ \bullet)\ M)$$

$$\boxed{\textsc{anf}[\![e]\!]\kappa = M}$$

$$
\begin{array}{lcl}
\textsc{anf}[\![e]\!] & = & \textsc{anf}[\![e]\!]\bullet \\
\textsc{anf}[\![\iota]\!]\kappa & = & \kappa[\iota] \\
\textsc{anf}[\![x]\!]\kappa & = & \kappa[x] \\
\textsc{anf}[\![(\boldsymbol{\lambda}\ (x)\ e)]\!]\kappa & = & \kappa[(\boldsymbol{\lambda}\ (x)\ \textsc{anf}[\![e]\!])] \\
\textsc{anf}[\![(e_1\ e_2)]\!]\kappa & = & \textsc{anf}[\![e_1]\!](\textbf{let}\ (x_1\ \bullet)\ \textsc{anf}[\![e_2]\!](\textbf{let}\ (x_2\ \bullet)\kappa[(x_1\ x_2)])) \\
\textsc{anf}[\![(\textbf{if0}\ e\ e_1\ e_2)]\!]\kappa & = & \textsc{anf}[\![e]\!](\textbf{let}\ (f\ (\boldsymbol{\lambda}\ (x)\ \kappa[x])) \\
& & \qquad (\textbf{let}\ (x\ \bullet)\ (\textbf{if0}\ x\ \ \textsc{anf}[\![e_1]\!](\textbf{let}\ (x_1\ \bullet)\ (f\ x_1)) \\
& & \qquad\qquad \textsc{anf}[\![e_2]\!](\textbf{let}\ (x_2\ \bullet)\ (f\ x_2))))) \\
\textsc{anf}[\![(op\ \vec{e})]\!]\kappa & = & \textsc{anf}[\![e_i]\!](\textbf{let}\ (x_i\ \bullet)\ \textsc{anf}[\![e_{i1}]\!]\ (\textbf{let}\ (x_{i1}\ \bullet)\ ...\kappa[(op\ \ \vec{x})])) \\
\textsc{anf}[\![(\textbf{let}\ (x\ e_1)\ e_2)]\!]\kappa & = & \textsc{anf}[\![e_1]\!](\textbf{let}\ (x\ \bullet)\textsc{anf}[\![e_2]\!]\kappa)
\end{array}
$$

Fig. 16. λ-calculus to ANF Compiler with Join Points

new continuation, which when called with a ANF value, produces an ANF term. Compiling n-ary operators requires a fold over the list of operands, which is slightly informally specified.

It's possible to avoid this CPSed compiler, but empirically this approach appears to be easier to reason about. An alternative involves returning two values, the tail of the computation M and the list of introduced bindings, and eventually merging them[6]. Bowman [2] attempted a proof of compiler correctness for ANF using a technique that relied on reasoning about lists of introduced bindings in this way, but the formalism did not scale to join points or branching constructs in general. By contrast, Koronkevich et al. [13] provided an extension of the ANF translation, with join points and branching, proving correctness entirely by a dependent typing of the compiler's continuation. The history with compiler verification suggests that if we care about reasoning, the CPSed compiler is the right approach, as it leads to a scalable, compositional, verifiable compiler.

The second bit of complexity is the representation of the join point in the target language. If we just use $\boldsymbol{\lambda}$, as we do in the above translation, we introduce a procedure call for every branch. Worse, it's a closure, since there are free variables in those branches, introducing allocation and memory indirects. Instead, we need an intermediate language with some notion of continuation, ideally one that introduces no allocation, allowing registers and frames to be shared between the caller and caller. This is possible—for example, Kennedy [11] and Tolmach and Oliva [21] gives intermediate languages with constructs for introducing a local continuation, which could be compiled separately, and contrast this with ANF. Maurer et al. [14] add a primitive join point form and equations for reasoning about them, citing problems with ANF. Cong et al. [3] add control operators, and types to distinguish different kinds of continuations, to enable optimizing using either or both ANF or CPS. All these choices increases the complexity of the IL, with new abstractions for procedures and continuations.

While managing all this complexity and using ANF successfully is possible, it is unnecessary. Our ANF compiler is more complex and unsafe for scope.

5.2 Monadic Compiler

We define the monadic form translation in Figure 17. The monadic form compiler merely sequences intermediate computation, but does not normalize commuting conversions.

[6]http://siek.blogspot.com/2012/07/my-new-favorite-abstract-machine-ecd-on.html

$$\boxed{\text{MF}[\![e]\!] = C}$$

$$\begin{array}{lcl}
\text{MF}[\![\iota]\!] & = & \iota \\
\text{MF}[\![x]\!] & = & x \\
\text{MF}[\![(\boldsymbol{\lambda}\ (x)\ e)]\!] & = & (\boldsymbol{\lambda}\ (x)\ \text{MF}[\![e]\!]) \\
\text{MF}[\![(e_1\ e_2)]\!] & = & (\textbf{let}\ (x_1\ \text{MF}[\![e_1]\!])\ (\textbf{let}\ (x_2\ \text{MF}[\![e_2]\!])\ (x_1\ x_2))) \\
\text{MF}[\![(\textbf{if0}\ e\ e_1\ e_2)]\!] & = & (\textbf{let}\ (x\ \text{MF}[\![e]\!])\ (\textbf{if0}\ x\ \text{MF}[\![e_1]\!]\ \text{MF}[\![e_2]\!])) \\
\text{MF}[\![(op\ \vec{e_i})]\!] & = & (\textbf{let}\ \overrightarrow{(x_i\ \text{MF}[\![e_i]\!])}\ (op\ \vec{x_i})) \\
\text{MF}[\![(\textbf{let}\ (x\ e_1)\ e_2)]\!] & = & (\textbf{let}\ (x\ \text{MF}[\![e_1]\!])\ \ \text{MF}[\![e_2]\!])
\end{array}$$

Fig. 17. λ-calculus to Monadic Form Compiler

The monadic form translation is straightforward; every expression can be locally transformed to sequence computation. The simplicity is because the compiler return a C, and a C can be composed with any other C using **let**. That is, the simplicity relies on *not* normalizing Equation Associativity and Equation Commute.

This monadic form compiler does not produce the *same* B-normal form that the B-reductions produce. Instead, they're equivalent up to Equation Associativity. However, they AB-normalize to equal terms.

5.3 Code Generation

We define two versions of code generation: one for ANF and one for monadic form. Performing code generation from monadic form relies the ability to generate (**set**! x t), while the ANF code generator should never generate this instruction. Otherwise, the output of the ANF compiler would need to be AB-normalized, defeating the purpose of A-normalizing in the first place.

The code generator for ANF is given in Figure 18a. Code generation from ANF is straightforward, since all the complexity is in getting into ANF in the first place. We give the types of each translation function, although they are imprecise. The function $\text{ACG}[\![_]\!]_N$ does not return an arbitrary t, but only a value, call, or primitive operation, which are also valid in non-tail position. The definitions could be collapsed, but since we must be careful to ensure well-formedness of **set**! to avoid extra stack frames, we separate the traversal of different syntactic categories.

The code generator for monadic form is defined in Figure 18b. It is simpler than the ANF code generator, since it can freely compose all ts in (**set**! x t).

5.4 AB-normalizer

Finally, we define the AB-normalizer Figure 19. The entire compiler is essentially searching for a (**set**! x t) form, then performing the AB-reductions, recursively. There is nothing complex in the definition; all rules but the last two are just traversals. The compiler assumes all $\boldsymbol{\lambda}$s are bound to variable with **set**!, which is necessary in the region calculi; this also simplifies the compilation of operands.

The compiler AB-NORMAL-COMPILE = ABNF·MCG·MF is simpler than the ANF-COMPILE = ACG·ANF compiler. AB-NORMAL-COMPILE does not require CPS to implement, does not require implementing (or managing the implementation of) join points, and it achieves the same or better performance characteristics as ANF-COMPILE, by Theorem 4.2.

$$
\begin{aligned}
\textsc{acg}\llbracket _ \rrbracket &: M \to t \\
\textsc{acg}\llbracket (\textbf{let}\ (x\ N)\ M) \rrbracket &= (\textbf{begin}\ (\textbf{set!}\ x\ \textsc{acg}\llbracket N \rrbracket_N) \\
&\qquad \textsc{acg}\llbracket M \rrbracket) \\
\textsc{acg}\llbracket (\textbf{if0}\ V\ M_1\ M_2) \rrbracket &= (\textbf{if0}\ \textsc{acg}\llbracket V \rrbracket_V\ \textsc{acg}\llbracket M_1 \rrbracket \\
&\qquad \textsc{acg}\llbracket M_2 \rrbracket) \\
\textsc{acg}\llbracket V \rrbracket &= \textsc{acg}\llbracket V \rrbracket_V \\
\textsc{acg}\llbracket N \rrbracket &= \textsc{acg}\llbracket N \rrbracket_N \\
\textsc{acg}\llbracket _ \rrbracket_N &: N \to t \\
\textsc{acg}\llbracket (O\ \vec{V}) \rrbracket_N &= (O\ \overrightarrow{\textsc{acg}\llbracket V \rrbracket_V}) \\
\textsc{acg}\llbracket V \rrbracket_N &= \textsc{acg}\llbracket V \rrbracket_V \\
\textsc{acg}\llbracket _ \rrbracket_V &: V \to t \\
\textsc{acg}\llbracket x \rrbracket_V &= x \\
\textsc{acg}\llbracket \iota \rrbracket_V &= \iota \\
\textsc{acg}\llbracket (\boldsymbol{\lambda}\ (x)\ M) \rrbracket_V &= (\boldsymbol{\lambda}\ (x)\ \textsc{acg}\llbracket M \rrbracket)
\end{aligned}
$$

(a) From ANF

$$
\begin{aligned}
\textsc{mcg}\llbracket _ \rrbracket &: C \to t \\
\textsc{mcg}\llbracket (\textbf{let}\ (x\ C_1)\ C_2) \rrbracket &= (\textbf{begin}\ (\textbf{set!}\ x\ \textsc{mcg}\llbracket C_1 \rrbracket) \\
&\qquad \textsc{mcg}\llbracket C_2 \rrbracket) \\
\textsc{mcg}\llbracket (\textbf{if0}\ U\ C_1\ C_2) \rrbracket &= (\textbf{if0}\ \textsc{mcg}\llbracket U \rrbracket\ \textsc{mcg}\llbracket C_1 \rrbracket \\
&\qquad \textsc{mcg}\llbracket C_2 \rrbracket) \\
\textsc{mcg}\llbracket (O\ \vec{U}) \rrbracket &= (O\ \overrightarrow{\textsc{mcg}\llbracket U \rrbracket}) \\
\textsc{mcg}\llbracket x \rrbracket &= x \\
\textsc{mcg}\llbracket \iota \rrbracket &= \iota \\
\textsc{mcg}\llbracket (\boldsymbol{\lambda}\ (x)\ C) \rrbracket &= (\boldsymbol{\lambda}\ (x)\ \textsc{mcg}\llbracket C \rrbracket)
\end{aligned}
$$

(b) From Monadic Form

Fig. 18. Code Generation

$$
\begin{aligned}
\textsc{abnf}\llbracket _ \rrbracket_t &: t \to t \\
\textsc{abnf}\llbracket (\textbf{begin}\ \vec{s}\ t) \rrbracket_t &= (\textbf{begin}\ \overrightarrow{\textsc{abnf}\llbracket s \rrbracket_s}\ \textsc{abnf}\llbracket t \rrbracket) \\
\textsc{abnf}\llbracket (\textbf{if0}\ v\ t_1\ t_2) \rrbracket_t &= (\textbf{if0}\ v\ \textsc{abnf}\llbracket t_1 \rrbracket_t\ \textsc{abnf}\llbracket t_2 \rrbracket_t) \\
\textsc{abnf}\llbracket (op\ \vec{v}) \rrbracket_t &= (op\ \vec{v}) \\
\textsc{abnf}\llbracket (\textbf{call}\ v_1\ v_2) \rrbracket_t &= (\textbf{call}\ v_1\ v_2) \\
\textsc{abnf}\llbracket _ \rrbracket &: v \to v \\
\textsc{abnf}\llbracket x \rrbracket_v &= x \\
\textsc{abnf}\llbracket \iota \rrbracket_v &= \iota \\
\textsc{abnf}\llbracket (\boldsymbol{\lambda}\ (x)\ t) \rrbracket_v &= (\boldsymbol{\lambda}\ (x)\ \textsc{abnf}\llbracket t \rrbracket_t) \\
\textsc{abnf}\llbracket _ \rrbracket_s &: s \to s \\
\textsc{abnf}\llbracket (\textbf{begin}\ \vec{s}) \rrbracket_s &= (\textbf{begin}\ \overrightarrow{\textsc{abnf}\llbracket s \rrbracket_s}) \\
\textsc{abnf}\llbracket (\textbf{set!}\ x\ v) \rrbracket_s &= (\textbf{set!}\ x\ \textsc{abnf}\llbracket v \rrbracket_v) \\
\textsc{abnf}\llbracket (\textbf{set!}\ x\ (op\ \vec{v})) \rrbracket_s &= (\textbf{set!}\ x\ (op\ \vec{v})) \\
\textsc{abnf}\llbracket (\textbf{set!}\ x\ (\textbf{call}\ v_1\ v_2)) \rrbracket_s &= (\textbf{set!}\ x\ (\textbf{call}\ v_1\ v_2)) \\
\textsc{abnf}\llbracket (\textbf{set!}\ x\ (\textbf{begin}\ \vec{s}\ t)) \rrbracket_s &= (\textbf{begin}\ \textsc{abnf}\llbracket \vec{s} \rrbracket_s\ \textsc{abnf}\llbracket (\textbf{set!}\ x\ t) \rrbracket_s) \\
\textsc{abnf}\llbracket (\textbf{set!}\ x\ (\textbf{if0}\ v\ t_1\ t_2)) \rrbracket_s &= (\textbf{if0}\ v\ \textsc{abnf}\llbracket (\textbf{set!}\ x\ t_1) \rrbracket_s\ \textsc{abnf}\llbracket (\textbf{set!}\ x\ t_2) \rrbracket_s)
\end{aligned}
$$

Fig. 19. AB-normalization

6 CONCLUSIONS

6.1 What about CPS

We ignore CPS throughout this paper because, in our view, CPS solves a different problem than ANF.

A-normalization, and our AB-normalization, solve the problem of *local* control: how to explicate the data and control flow of subexpressions without control effects. But neither address the problem

1030 of *non-local* control, such as returning from a function call (a control effect). The problems with
1031 ANF begin when conflating these two, using a solution for non-local control (continuations) to
1032 solve a local control problem (commuting conversions).
1033 Non-local control is still important. In this paper, we never notice the non-local control problem
1034 since we never compile CALL and RETURN. To compile these, we need something like a join point
1035 or a control operator–*i.e.*, we need object-language continuations by any other name. Whatever
1036 the technique, it should reify the continuation introduced by the non-tail CALL instruction, and
1037 used by the RETURN instruction, into data, which is stored in the heap, and optimized by some set
1038 of equations. This would let us transform our CESK machine into a CES machine, which more
1039 closely models a real machine, with a code pointer, a register file, and memory. For example, if we
1040 add a **let/cc** *instruction* of the form (**let/cc** k t) to our ABNF, we can compile CALL and RETURN as
1041 something like the following.
1042
1043
1044

$$\begin{aligned} \textsc{mcg}[\![(\textbf{call}\ v_1\ v_2)]\!]_C &= (\textbf{let/cc}\ k\ (\textbf{call}\ \textsc{mcg}[\![v_1]\!]\ \textsc{mcg}[\![v_2]\!]\ \ k)) \\ \textsc{mcg}[\![v]\!]_C &= (\textbf{call}\ k\ \textsc{mcg}[\![v]\!]_V) \end{aligned}$$

1045 That is, when we find a non-tail call (a call in C position), we capture the current continuation
1046 and generate a tail-call (since the body of **let/cc** accepts a tail) that explicitly passes the captured
1047 continuation as its return label. A value in C position is being returned, so we transform it into a
1048 tail-call to the continuation. The **let/cc** operation can be compiled to a labelled instruction that
1049 pushes and pops caller saved variables around the call, avoiding allocating continuations altogether.
1050 This calculus starts to look, morally, like Tolmach and Oliva [21]'s SIL or Cong et al. [3]'s IL.
1051 Chez Scheme implements this transformation, suggesting it works well in practice (Subsection 6.4).
1052 We could probably perform a rational reconstruction of this pattern from the machine semantics,
1053 as we did with ANF and monadic form.
1054

1055
6.2 Case-of-Case

1056 The AB-normal compiler fails to optimize case-of-case commuting conversions. Consider the
1057 example (**if0** (**if0** e 1 0) 5 6). From this, the ANF compiler generates (**if0** e (**if0** 1 5 6) (**if0** 0 5 6)).
1058 We can see the branches have been duplicated, which looks like a problem. However, now a
1059 simple partial evaluator can optimize this to (**if0** e 6 5). By contrast, the *AB*-normal compiler
1060 produces first (**let** (x (**if0** e 1 0)) (**if0** x 5 6)) then (**begin** (**if0** e (**set**! x 1) (**set**! x 0)) (**if0** x 5 6)). No
1061 code is duplicated, but we cannot easily perform the same optimization. The join point calculus of
1062 Maurer et al. [14] handles this example very well, although at the cost of introducing join points.
1063 We believe this can be supported without join points by contextually separating values and
1064 boolean position. For example, Danvy [4] present a standard monadic translation (such as we define
1065 in Figure 17) extended with the following rules (more or less).
1066
1067
1068
1069
1070

$$\begin{aligned} \textsc{mf}[\![(\textbf{if0}\ e\ e_1\ e_2)]\!] &= (\textbf{if}\ [\![e]\!]_P\ \textsc{mf}[\![e_1]\!]\ \textsc{mf}[\![e_2]\!]) \\ [\![_]\!]_P &: C \to \mathbb{B} \\ [\![(\textbf{if0}\ e\ e_1\ e_2)]\!]_{\mathbb{B}} &= (\textbf{if}\ \ [\![e]\!]_{\mathbb{B}}\ \ [\![e_1]\!]_{\mathbb{B}}\ \ [\![e_2]\!]_{\mathbb{B}}) \\ [\![v]\!]_{\mathbb{B}} &= (\textbf{equal}?\ \textsc{mf}[\![v]\!]\ 0) \end{aligned}$$

1071
1072 This compiles pattern matching to a sublanguage of boolean expressions. With this translation,
1073 the example is transformed to (**if** (**if** (**equal**? e 0) (**equal**? 1 0) (**equal**? 0 0)) 5 6), which a simple
1074 boolean optimizer could simplify as (**if** (**if** (**equal**? e 0) **false true**) 5 6), then (**if** (**equal**? e 0) 6 5).
1075 It's not clear that this completely solves the case-of-case problem that Maurer et al. [14] investi-
1076 gate, but it solves their simple example, and does so without join points. Chez Scheme implements
1077 this transformation, suggesting it works well in practice (Subsection 6.4).
1078

6.3 AB-normalizing with Monadic Effects

We work in an imperative language, but that's a choice based on our thinking in machines. AB-normalization could be first performed in a high-level monadic language by transforming lexical binding into a state monad to normalize commuting conversions. For example, perhaps an A-normalizing monadic compiler would normalize with respect to the following rule.

$$E[(\mathbf{if}\ v\ e_1\ e_2)] \longrightarrow_B \begin{array}{l}(\mathbf{do}\ (\mathbf{if}\ v\ (\mathbf{do}\ (v_1 \leftarrow e_1)\ (\mathbf{set}\ x\ v_1)) \\ \qquad\qquad (\mathbf{do}\ (v_2 \leftarrow e_2)\ (\mathbf{set}\ x\ v_2))) \\ \quad E[(\mathbf{get}\ x)])\end{array} \qquad B_A$$

We use the **do** notation to interact with the state monad in the target language. This lets us to express an **if** statement in our high-level language. If **do** is compiled to **begin**, **set** to **set!**, and **get** to variable reference, then this ends up in the same AB-normal form as our compiler.

Similarly, regions could first be elaborated into monadic regions [9, 10] rather than directly to low-level machine instructions.

Of course, if we consider **let** to be the bind of partiality monad, for explicating local control, and **do** to be the bind for the state monad, we may run into problems with composing these two monads. The problem is made worse if we throw in a region monad. Perhaps we need a single "compilation effects" monad; or perhaps not—what's wrong with an imperative language that behaves monadicly? We leave this question to future work.

6.4 AB-normalization in Practice

The AB-reductions are the essence of a transformation done in Chez Scheme. The can be seen, indirectly, in the difference between two of Chez's language definitions: L10[7], and L7[8]. L7 includes the expression (**set**! *lvalue e*), which is our (**set**! *x e*), and L6 generates this from a **let** expression. L10 rewrite this into (**set**! *lvalue rhs*), where *rhs* is a heavily restricted set of operations very similar to our final ABNF. In fact, Chez uses both of the transformations mentioned above: transforming boolean position into a predicate sublanguage, to optimize case-of-case transformations[9], and compiling non-tail calls to tail calls with **let/cc** (called **return–point** in Chez)[10].

The Chez compiler is renowned for its performance both at compile-time and in the run-time code it generates. Our paper, in part, is an attempt to understand why and how Chez avoids CPS, ANF, and monadic form—the standard choices in the published literature—in favour of an apriori strange imperative intermediate representation. The answer is that they choose ABNF before it had a name.

ACKNOWLEDGMENTS

We gratefully acknowledge the feedback of the many reviewers of this work. Also some other people.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-9999-9999. Cette recherche a été financée par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-9999-9999.

[7]https://github.com/cisco/ChezScheme/blob/9576b83dd757cf1494933c9fbc80cb6aff022295/s/np-languages.ss#L836
[8]https://github.com/cisco/ChezScheme/blob/9576b83dd757cf1494933c9fbc80cb6aff022295/s/np-languages.ss#L561
[9]https://github.com/cisco/ChezScheme/blob/9576b83dd757cf1494933c9fbc80cb6aff022295/s/np-languages.ss#L901
[10]https://github.com/cisco/ChezScheme/blob/9576b83dd757cf1494933c9fbc80cb6aff022295/s/np-languages.ss#L1055

1128
DATA AVAILABILITY STATEMENT

The software artifact with mechanized models, containing the full definitions and details, is available in anonymous supplementary materials and will be made publically available.

REFERENCES

[1] Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling Standard ML to Java Bytecodes. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/289423.289435

[2] William J. Bowman. 2018. *Compiling with Dependent Types*. Ph. D. Dissertation. Northeastern University. https://doi.org/10.17760/D20316239

[3] Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? whatever. *PACMPL* 3, ICFP (2019), 79:1–79:28. https://doi.org/10.1145/3341643

[4] Olivier Danvy. 2003. A New One-Pass Transformation into Monadic Normal Form. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 77–89. https://doi.org/10.1007/3-540-36579-6_6

[5] Matthias Felleisen. 1987. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. phdthesis. https://www2.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf

[6] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. https://mitpress.mit.edu/books/semantics-engineering-plt-redex

[7] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, Martin Wirsing (Ed.). North-Holland, 193–222. https://legacy.cs.indiana.edu/ftp/techreports/TR197.pdf

[8] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/155090.155113

[9] Matthew Fluet and Greg Morrisett. 2006. Monadic regions. *J. Funct. Program.* 16, 4-5 (2006), 485–545. https://doi.org/10.1017/S095679680600596X

[10] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming (ESOP)*. Springer. https://doi.org/10.1007/11693024_2

[11] Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1291220.1291179

[12] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Symposium on Principles of Programming Languages (POPL)*. ACM. https://doi.org/10.1145/2103656.2103691

[13] Paulette Koronkevich, Ramon Rakow, Amal Ahmed, and William J. Bowman. 2022. ANF Preserves Dependent Types up to Extensional Equality. *Journal of Functional Programming (JFP)* 32 (2022). https://doi.org/10.1017/s0956796822000090

[14] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *International Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3062341.3062380

[15] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[16] Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages*. Ph. D. Dissertation. Princeton University. https://zoep.github.io/thesis_final.pdf

[17] Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with continuations, correctly. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485491

[18] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473591

[19] Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, Robert R. Kessler (Ed.). ACM, 150–161. https://doi.org/10.1145/182409.156783

[20] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Symposium on Principles of Programming Languages (POPL)*. ACM Press. https://doi.org/10.1145/174675.177855

[21] Andrew P. Tolmach and Dino Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *J. Funct. Program.* 8, 4 (1998), 367–412. https://doi.org/10.1017/S0956796898003086