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, difficult to extend with new lexically scoped constructs such as scoped region-based allocation. The problem, as has often been observed, is that normalization of commuting conversions is hard.

This traditional view of ANF that normalizing commuting conversions is hard, found in formal models and informed by high-level calculi, 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 easily extended with new lexically scoped effects. 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 for regions. We prove that any such compiler preserves, or optimizes, stack and memory behaviour compared to ANF. Our formalization reconstructs and systematizes pragmatic choices found in practice, including current production-ready compilers.

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. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 277 (October 2024), 27 pages. https://doi.org/10.1145/3689717

1 Introduction

Intermediate representations (IRs), 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 [4, 9, 14, 16, 17, 20]. 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

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

ACM 2475-1421/2024/10-ART277
https://doi.org/10.1145/3689717

call) explicit in the syntax, simplifying analysis, optimization, and compilation. For example, the following two terms are equivalent, but Listing 2 is the A-normalization of Listing 1.

```
(+ (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\n **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 an ANF abstract machine can run with a smaller stack; a machine for the λ term must evaluate the first operand of + to a value, after pushing the frame (+ · 6), while an ANF machine need not push any frames.

ANF has several known disadvantages. We discuss these in detail later, 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 difficult for lexically scoped effects. 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 [25], 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

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. To target ANF safely (as in safe-for-space [22]), we must either make the allocation and free operations explicit or resort to explicit continuations.

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 [18] 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 produced by the usual translation of Listing 1.)

```
(+ (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 $(\mathbf{let}\ (x\ (\mathbf{if}\ v\ e_1\ e_2))\ e_3) \equiv (\mathbf{if}\ v\ (\mathbf{let}\ (x\ e_1)\ e_3)\ (\mathbf{let}\ (x\ e_2)\ e_3)$. Duplicating e_3 is undesired, so typically this is abstracted into a continuation, called a *join point*, as in $(\mathbf{let}\ (x\ (\mathbf{if}\ v\ e_1\ e_2))\ e_3) \equiv (\mathbf{let}\ (j\ (\boldsymbol{\lambda}\ (y)\ e_3))\ (\mathbf{if}\ v\ (\mathbf{let}\ (x\ e_1)\ (j\ x))\ (\mathbf{let}\ (x\ e_2)\ (j\ x))))$.

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 [14] observes that normalizing commuting conversions causes all the well known problems with ANF, before resorting to CPS. Maurer et al. [17] 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 intensionally 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 series of IRs, normal forms, and transformation to make precise what practioniers have long known: ANF, as typically studied, is not a good IR. Instead, normalizing commuting conversions should happen later in the compiler pipeline, and monadic form should be used until then. By fully understanding the intensional aspects of commuting conversions, we can gain the benefits of ANF with none of the drawbacks. We make these intensional aspects formal using abstract machines. To contrast to Maurer et al. [17], our motto is: *work in monadic form, but think in abstract machines*. If we think of $(\mathbf{let}\ (x\ (\mathbf{if}\ v\ e_1\ e_2))\ e_3)$ as its computation in an abstract machine, we might render it as $(\mathbf{begin}\ (\mathbf{set!}\ x\ (\mathbf{if}\ v\ e_1\ e_2))\ e_3)$, which by (a low-level interpretation of) associativity is the same as $(\mathbf{begin}\ (\mathbf{if}\ v\ (\mathbf{set!}\ x\ e_1)\ (\mathbf{set!}\ x\ e_2))\ e_3)$. This avoids the need to deal with join points explicitly, and still avoids code duplication. This idea is understood by some compiler writers; for example, the high-performance Chez Scheme compiler uses a similar transformation (Subsection 6.4). Our work formalizes and rationally reconstructions design choices that some compiler writers have made in practice.

Concretely, our contributions are:

(1) A formalization of monadic form as normalizing a subset of the A-reductions, which is important for clarifying the distinction between ANF and monadic form, and identifying the source of problems with ANF (Section 2).
(2) An analysis of the abstract machine of high-level ANF and monadic form, with which we formalize how ANF optimizes stack usage (Section 3).
(3) A formalization of a counterexample to the safety of direct-style A-normal form with respect to scoped regions, which is important for understanding limitations of ANF in compilation (Subsection 3.2).
(4) A formalization of imperative variants of monadic form and ANF derived from the machine semantics and admissible equations, and an algorithm for normalizing commuting conversions without join points or duplication. These are important as normal forms for compilation,

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

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

Fig. 1. A-normalization for λ-calculus

optimization, and analysis as they enable the advantages of ANF but suffers none of the known disadvantages (Section 4). These IRs are essentially similar to some that appear in practice, including several IRs used in the Chez Scheme compiler (Subsection 6.4), although our reconstruction of them is novel.

(5) An analysis of the abstract machine for imperative monadic form, with which we prove that using monadic form followed by imperative A-normalization has the same or better performance characteristics as ANF. In particular, we show that any such compiler (1) avoids code duplication and join point introduced by commuting conversion in high-level ANF; (2) preserves the stack behaviour of ANF; and (3) preserves the memory usage of scoped regions, unlike the ANF compiler (Section 4).

(6) A model compiler designed to use monadic form as a high-level intermediate language, and imperative ANF as a low-level IL. We argue that monadic form followed by imperative A-normalization simplifies compiler implementation compared to ANF (Section 5).

All machines, reduction systems, languages, compilers, examples, and counterexample are implemented in a PLT Redex [7, 15]; an artifact is publicly available [2].

2 A-normal and Monadic Form, Formally

2.1 A-normal Form

A-normal form (ANF) is often called "administrative normal form" or sometimes "administrative form", but it is important and useful to think about ANF not as a vague form related to administrative reductions, but formally and precisely as a normal form with respect to a set of reductions, as it was originally formalized [9].

Formally, A-normal form was introduced as the form normal with respect to the set of reductions $A = \{A_1, A_2, A_3\}$, defined in Figure 1. A term e represents an arbitrary λ-calculus expression. An operator op is some n-ary primitive operator, such as addition, and must appear in operator position. An ι is some ground value, such as a natural number, and a value v is any syntactic value. We use the slight abuse of notation $(O\ \vec{e})$ to mean an application—of either a function or an operator, and restrict the function position to a value as necessary. The A-reductions use the call-by-value 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.

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

Fig. 2. A-normal Form

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. 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.

```
(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

Rather than trying to patch ANF, we could question the premise: why did we choose to normalize that equation when it causes so many problems? Wouldn't it be more pragmatic to simply *not* normalize the commuting conversion in A_2, and allow $(\textbf{let}\ (x\ (\textbf{if0}\ V\ M_1\ M_2))\ M)$, which after all is essentially just $(\textbf{begin}\ (\textbf{if0}\ V\ (\textbf{set!}\ x\ M_1)\ (\textbf{set!}\ x\ M_2))\ M)$ after compilation. This pragmatic choice is

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

Fig. 3. Monadic Form Syntax

appealing, and practioners have made it before—we return to this in Subsection 6.4. But formalisms have dogmatically adherred to this problematic interpretation of ANF. We show how to formalize these pragmatic choices. We start by deriving monadic form from ANF, and show the derived normal form does not normalize this problematic equation.

2.2 Monadic Form

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.

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.

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

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

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

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

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

Both sides of the equations are in monadic form.

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.

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

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

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.

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**.

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

[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.

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

Fig. 4. B-normalizations for λ-calculus

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 (**let** $(x\ \cdot)\ e$). 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 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.

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

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

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

Fig. 5. λ-calculus CK Machine

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) direct-style ANF is unsafe for scoped regions, making extending ANF to scoped effects difficult. We show that A-normalization optimizes stack usage—there are fewer frames in use after A-normalization compared to monadic form, in general. However, we then introduce a region calculus with scoped regions, and show that direct-style A-normalization of the region calculus results in more regions and longer region lifetimes than in monadic form, in general.

3.1 ANF vs λ Machines

We start with a CK machine [6–8] 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 simple; 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 as follows.

$$
\begin{array}{rll}
 & (+\ (+\ 2\ 2)\ (\textbf{let}\ (x\ 1)\ (f\ x))); & \textbf{mt} \\
\rightarrow_\lambda & (+\ 2\ 2); & ((+\ \bullet\ (\textbf{let}\ (x\ 1)\ (f\ x)))::\textbf{mt}) \quad \textsc{push} \\
\rightarrow_\lambda & 4; & ((+\ \bullet\ (\textbf{let}\ (x\ 1)\ (f\ x)))::\textbf{mt}) \quad \textsc{primop} \\
\rightarrow_\lambda & (+\ 4\ (\textbf{let}\ (x\ 1)\ (f\ x))); & \textbf{mt} \quad \textsc{pop} \\
\rightarrow_\lambda & (\textbf{let}\ (x\ 1)\ (f\ x)); & ((+\ 4\ \bullet)::\textbf{mt}) \quad \textsc{push} \\
\rightarrow_\lambda & (f\ 1); & (+\ 4\ \bullet)::\textbf{mt} \quad \zeta
\end{array}
$$

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

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

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

$$
\begin{array}{rcll}
(\textbf{let}\ (x'\ ((\boldsymbol{\lambda}\ (x)\ M)\ V))\ M');K & \rightarrow_{\_nf} & M[x := V];(\textbf{let}\ (x'\ \bullet)\ M') :: K & \textsc{call} \\
V;(F :: K) & \rightarrow_{\_nf} & F[V];K & \textsc{return} \\
((\boldsymbol{\lambda}\ (x)\ M)\ V);K & \rightarrow_{\_nf} & M[x := V];K & \textsc{tail-call} \\
(\textbf{let}\ (x\ V)\ M);K & \rightarrow_{\_nf} & M[x := V];K & \textsc{move} \\
(\textbf{if0}\ 0\ M_1\ M_2);K & \rightarrow_{\_nf} & M_1;K & \textsc{ifz} \\
(\textbf{if0}\ V\ M_1\ M_2);K & \rightarrow_{\_nf} & M_2;K & \textsc{ifnz}\quad \text{where } V \neq 0 \\
(\textbf{let}\ (x\ (op\ \vec{V}))\ M);K & \rightarrow_{\_nf} & M[x := \delta[\![op\ \vec{V}]\!]];K & \textsc{primop} \\
(op\ \vec{V});K & \rightarrow_{\_nf} & \delta[\![op\ \vec{V}]\!];K & \textsc{tail-primop} \\
\hline
(\textbf{let}\ (x\ C_1)\ C_2);K & \rightarrow_{bnf} & C_1;(\textbf{let}\ (x\ \bullet)\ C_2) :: K & \textsc{bind}\quad \text{where } C_1 \neq N
\end{array}
$$

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

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 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 using explicit environments, 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.

$$
\begin{array}{rll}
 & (\textbf{let}\ (x_1\ (+\ 2\ 2))\ (\textbf{let}\ (x\ 1)\ (\textbf{let}\ (x_2\ (f\ x))\ (+\ x_1\ x_2)))); & \textbf{mt} \\
\rightarrow_{anf} & (\textbf{let}\ (x\ 1)\ (\textbf{let}\ (x_2\ (f\ x))\ (+\ 4\ x_2))); & \textbf{mt}\quad \textsc{primop} \\
\rightarrow_{anf} & (\textbf{let}\ (x_2\ (f\ 1))\ (+\ 4\ x_2)); & \textbf{mt}\quad \textsc{move}
\end{array}
$$

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 $(\textbf{let}\ (x\ (\textbf{if0}\ 0\ 0\ 1))\ C)$ (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.

$$
\begin{array}{ll}
\textsc{max-stack}[\![\_]\!] : (\mathcal{D} : (C;K \rightarrow^*_{\_nf} C;K)) \rightarrow \mathbb{N} & \\
\textsc{max-stack}[\![C;K \rightarrow^0_{\_nf} C;K]\!] & = 0 \\
\textsc{max-stack}[\![C;K \rightarrow_{\_nf} C';K' \rightarrow^*_{\_nf} C'';K'']\!] & = \textsc{max}[\![\textsc{len}[\![K]\!], \textsc{max-stack}[\![C';K' \rightarrow^*_{\_nf} C'';K'']\!]]\!]
\end{array}
$$

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

Fig. 7. λ^r Syntax

Theorem 3.1 (*A*-normalization Optimizes the Stack (Kontinuation)). *If $e \longrightarrow_A^* M$ where $\mathcal{D}_B : (e;\boldsymbol{mt} \rightarrow_{bnf}^* v;\boldsymbol{mt})$ and $\mathcal{D}_A : (M;\boldsymbol{mt} \rightarrow_{anf}^* v;\boldsymbol{mt})$, 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$[\![_]\!]$. □

This is more than a theoretical effect. The CertiCoq compiler has had two backends, a CPS and an ANF backend, and ANF outperforms CPS [19–21]. Paraskevopoulou [19] attributes this to additional heap allocation in CPS, noting that "sophisticated techniques to reduce heap allocation" could be used in the CPS backend. CPS can result in additional heap allocation by allocating a continuation for local control. Each frame in the CK machine corresponds to a continuation, and could cause heap allocation of a closure in CPS. The ANF backend, by contrast, avoids this allocation, since these frames are made explicit syntactically and never allocate. This is no more than we should *expect*; *A*-normalization was partially derived from eliminating administrative redexes introduced by CPS translation. This suggests that merely using ANF simplifies control-related optimization compared to CPS, at least in the context of a fully mechanically verified compiler.

3.2 A Region Calculus and Machine

Unfortunately, *A*-normalization is an unsafe optimization in some settings. In general, direct-style *A*-normalization is extends lexical scope, changing variable extent, lifetimes, and other effects related to lexical scope. Extensionally, such as in a pure calculus when reasoning up to α-equivalence, this is irrelevant. However, it matters when reasoning about intensional properties or effectful calculi. We consider a version of the scoped regions of Tofte and Talpin [25].

We first extend our λ-calculus with lexically scoped regions and region allocation; the syntax is given in Figure 7. The expression (**letregion** r e) runs e with a new region r allocated in the store S[3]. After e returns, the region r is freed. In this calculus, all values must be explicitly allocated in a region in the store. The form (@ r e) evaluates to a reference, by evaluating e to a value that is stored in region r. All primitive operations and value must be explicitly allocated; the result of a function must also be allocated before it returns. We also extend evaluation contexts, for *A*- and *B*-normalization of this syntax, and add a new frames for the machine semantics. The new frame (**free** r) is unusual; it is not a frame of an evaluation context, but instead directs the machine to free region r.

In Figure 8, we give a CSK machine [6–8] for λ^r. The machine includes a straightforward extension of the transitions from the λ CK machine in Figure 5. The machine is unrealistic in that all values are passed by references, whereas a realistic machine might avoid boxing word-sized

[3]The allocation pattern forms a (data) stack (of regions), as explained by Tofte and Talpin [25]. The stack of regions is irrelevant to us, while the control stack (kontinuation) K *is* important, so we refer to allocation as occurring in the heap, and use "stack" to refer to the control stack.

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

$(a_1\ a_2); S; K$	$\to_{\lambda r}$	$e[x := a_2]; S; K$	CALL	where $(\boldsymbol{\lambda}\ (x)\ e) = S(a_1)$
$(\textbf{let}\ (x\ a)\ e); S; K$	$\to_{\lambda r}$	$e[x := a]; S; K$	MOVE	
$(\textbf{if0}\ a\ e_1\ e_2); S; K$	$\to_{\lambda r}$	$e_1; S; K$	IFZ	where $0 = S(a)$
$(\textbf{if0}\ a\ e_1\ e_2); S; K$	$\to_{\lambda r}$	$e_2; S; K$	IFNZ	where $0 \neq S(a)$
$(op\ \overrightarrow{a}); S; K$	$\to_{\lambda r}$	$[\![op\ \overrightarrow{S(a)}]\!]; S; K$	PRIMOP	
$F[e]; S; K$	$\to_{\lambda r}$	$e; S; (F :: K)$	PUSH	where $e \neq a$
$a; S; (F :: K)$	$\to_{\lambda r}$	$F[S(a)]; S; K$	POP	
$(\textbf{letregion}\ r\ e); S; K$	$\to_{\lambda r}$	$e; S; ((\textbf{free}\ r) :: K)$	RALLOC	
$a; S; ((\textbf{free}\ r) :: K)$	$\to_{\lambda r}$	$a; \text{FREE}[\![S, r]\!]; K$	RFREE	
$v; S; ((@\ r\ \bullet) :: K)$	$\to_{\lambda r}$	$(r\ .\ o); (S[(r.o) := v]; K$	ALLOC	fresh o

Fig. 8. λ^r CSK Machine

data. However, this detail is irrelevant for our purposes: some ANF terms bind non-word sized data, so *any* extension of a region lifetime is unsafe.

The machine is essentially similar to the λ CK machine, with two main differences: the representation of run-time values, and the new transitions for regions. The new region-related transitions are given below the line. Run-time values are addresses, a pair $(r\ .\ o)$ of a region and an offset into that region, so all computations dereference an address a in the store S, written $S(a)$. Syntactic values must be allocated in a region, indicated by the frame, in the ALLOC transition. (**letregion** r e) (implicitly) allocates a new region, and adds a **free** frame to the stack. The metafunction $\text{FREE}[\![S, r]\!]$ 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.

	$(\textbf{letregion}\ r_2$ $\quad (@\ r_0\ (*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2))))$ $\qquad\qquad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4)))))))$	$\emptyset$	**mt**
$\to_{\lambda r}$	$(@\ r_0\ (*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2))))$ $\qquad\qquad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4))))))$	$\emptyset$	$(\textbf{free}\ r_2) :: \textbf{mt}$
$\to_{\lambda r}$	$(*\ (\textbf{letregion}\ r_1\ (@\ r_2\ (*\ (@\ r_1\ 1)\ (@\ r_1\ 2))))$ $\quad (\textbf{letregion}\ r_3\ (@\ r_2\ (*\ (@\ r_3\ 3)\ (@\ r_3\ 4)))))$	$\emptyset$	$(@\ r_0\ \bullet) :: (\textbf{free}\ r_2) :: \textbf{mt}$
$\to_{\lambda r}$	...		

$\to_{\lambda r}\ (*\ (r_2.o_1)\ (r_2.o_2))$	$[(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12]$	$(@\ r_0\ \bullet) :: (\textbf{free}\ r_2) :: \textbf{mt}$
$\to_{\lambda r}\ 24$	$[(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12]$	$(@\ r_0\ \bullet) :: (\textbf{free}\ r_2) :: \textbf{mt}$
$\to_{\lambda r}\ (r_0.o_3)$	$[(r_2.o_2) \mapsto 2][(r_2.o_1) \mapsto 12][(r_0.o_3) \mapsto 24]$	$(\textbf{free}\ r_2) :: \textbf{mt}$
$\to_{\lambda r}\ (r_0.o_3)$	$[(r_0.o_3) \mapsto 24]$	**mt**

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.

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

The side condition $E \neq E'[(@\ r\ \cdot)]$ must also be added to the A_3 reduction. The reduction system is fully implemented in the artifact [2].

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

These equations aren't the only way to normalize regions, but they are the direct-style A-normalization rules. We could alternatively bound the scope of r using an explicit continuation as in the following rule.

$$E[(\textbf{letregion}\ r\ e)] \quad \longrightarrow_A \quad (\textbf{let}\ (k\ (\boldsymbol{\lambda}\ ()\ (\textbf{letregion}\ r\ e)))\ E[(k)])$$

But this is not in direct style, and introduces a local continuation similar to join points. In our opinion, this is resorting to using CPS, and not in the spirit of ANF.

THEOREM 3.2 (DIRECT-STYLE A-NORMALIZATION IS REGION-UNSAFE).
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⟦$\mathcal{D}_A$⟧ ≥ MAX-REGIONS⟦$\mathcal{D}_B$⟧, *and* MAX-MEMORY⟦$\mathcal{D}_A$⟧ ≥ 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 easily B-normalization to λ^r with the following reductions, maintaining direct style without extending scope.

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

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

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

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 Region-Safe).
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.

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 19a. 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.

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

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

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

Fig. 10. Imperative Monadic Syntax

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

Fig. 11. AB Reduction

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 set of reductions $AB = \{AB_1, AB_2\}$ in Figure 11, yielding AB-normal form[4].

AB-normalization is straightforward. Unlike A-normalization and B-normalization, it does not shuffle the evaluation contexts at all. It is a completely local transformation. We never need to deal with join points, or CPS'd compilers, or code duplication. We gain all the stack optimization effect of A-normalization (as we formalize next), the advantages of a register-transfer syntax for code generation and machine implementation, etc.

The key idea in AB-normalization is that lexical expressions must be elaborated into imperative statements before A-normalization.

4.1 Machine Semantics of AB-normal Form

We use a CEK machine [6–8] to define a machine semantics for imperative monadic form in Figure 12 and Figure 13. We then show that AB-normalization optimizes the stack in the same way as A-normalization.

The environment Σ (to distinguish it from evaluation contexts) represents the register file, mapping variables to their values. We separate word values wv, which represent run-time values

[4]We choose the name AB for two reasons: (1) this normalization yields the best of both A- and B-normalization (2) AB-normalization follows B-normalization, which was defined second after A-normalization, so AB is 3... in big-endian binary, anyway.

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

Fig. 12. CEK Machine Syntax

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

$$
\begin{array}{rll}
(\textbf{begin}\ (\textbf{set!}\ x\ wv)\ \overrightarrow{s}\ t); \Sigma; K \rightarrow_{CEK} t; \Sigma[x := wv]; K & & \textsc{move} \\
(\textbf{begin}\ (\textbf{set!}\ x\ (\boldsymbol{\lambda}\ (x)\ t))\ \overrightarrow{s}\ t); \Sigma; K \rightarrow_{CEK} t; \Sigma[x := (\textbf{closure}\ \Sigma\ (\boldsymbol{\lambda}\ (x)\ t))]; K & & \textsc{closure} \\
(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{call}\ wv_1\ wv_2))\ \overrightarrow{s}\ t); \Sigma; K \rightarrow_{CEK} t; \Sigma'[y := wv_2]; (\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ \overrightarrow{s}\ t) :: K & & \textsc{call} \\
\quad \text{where } \Sigma(wv_1) = (\textbf{closure}\ \Sigma'\ (\boldsymbol{\lambda}\ (y)\ t)) & & \\
v; \Sigma; F :: K \rightarrow_{CEK} F[v]; \Sigma; K & & \textsc{return} \\
(\textbf{begin}\ (\textbf{if0}\ wv\ s_1\ s_2)\ t); \Sigma; K \rightarrow_{CEK} (\textbf{begin}\ s_1\ t); \Sigma; K & & \textsc{ifz-s} \\
\quad \text{where } \Sigma(wv) = 0 & & \\
(\textbf{begin}\ (\textbf{if0}\ wv\ s_1\ s_2)\ t); \Sigma; K \rightarrow_{CEK} (\textbf{begin}\ s_2\ t); \Sigma; K & & \textsc{ifnz-s} \\
\quad \text{where } \Sigma(wv) \neq 0 & & \\
(\textbf{begin}\ (\textbf{set!}\ x\ (op\ \overrightarrow{wv}))\ t); \Sigma; K \rightarrow_{CEK} t; \Sigma[x := [\![op\ \overrightarrow{\Sigma(sv)}]\!]]; K & & \textsc{primop} \\
(op\ \overrightarrow{wv}); \Sigma; K \rightarrow_{CEK} [\![op\ \overrightarrow{\Sigma(wv)}]\!]; \Sigma; K & & \textsc{tail-primop} \\
(\textbf{if0}\ wv\ t_1\ t_2); \Sigma; K \rightarrow_{CEK} t_1; \Sigma; K & & \textsc{ifz-t} \\
\quad \text{where } \Sigma(wv) = 0 & & \\
(\textbf{if0}\ wv\ t_1\ t_2); \Sigma; K \rightarrow_{CEK} t_2; \Sigma; K & & \textsc{ifnz-t} \\
\quad \text{where } \Sigma(wv) \neq 0 & & \\
(\textbf{begin}\ (\textbf{set!}\ x\ t_1)\ \overrightarrow{s}\ t_2); \Sigma; K \rightarrow_{CEK} t_1; \Sigma; (\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ \overrightarrow{s}\ t_2) :: K & & \textsc{ab} \\
(\textbf{begin}\ (\textbf{begin}\ \overrightarrow{s_1})\ \overrightarrow{s_2}\ t); \Sigma; K \rightarrow_{CEK} (\textbf{begin}\ \overrightarrow{s_1}\ \overrightarrow{s_2}\ t); \Sigma; K & & \textsc{admin1} \\
(\textbf{begin}\ t); \Sigma; K \rightarrow_{CEK} t; \Sigma; K & & \textsc{admin2}
\end{array}
$$

Fig. 13. CEK Machine for Imperative Monadic Form

that can be eliminated, and heap values hv, which must be bound to a name and intuitively would be allocated in memory. We use the syntax $\Sigma(wv)$ get the value of wv, either dereferencing wv in Σ if wv is a variable, or returning wv if not. The code continues to represent the program counter, and the kontinuation kontinues to represent the call stack. W.l.o.g., we assume that all $\boldsymbol{\lambda}$s appear on the right-hand side of a **set!**, to simplify implementing closures. This can be implemented by another B-reduction that binds $\boldsymbol{\lambda}$, treating it as a computation rather than a value (which it is, in this machine; it performs allocation).

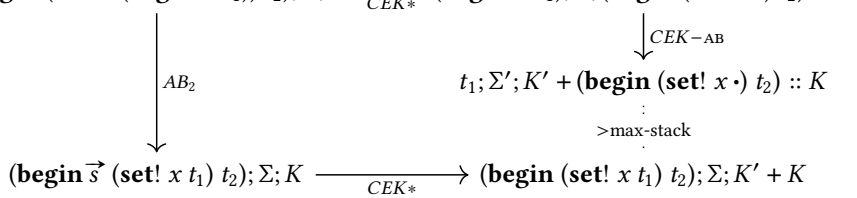
This machine is essentially similar to the monadic CK machine in Figure 6, but using an environment rather than substitution. The environment is necessary with the imperative interpretation of **let** as **set!**. We also require two administrative reductions, ADMIN1 and ADMIN2, to enable composing s internally using **begin**, although these could be normalized to simplify the final machine semantics.

There is one key difference: the AB transition, which cannot occur in ABNF but can in imperative monadic form. This rule, and CALL for a non-tail call, both push a frame $(\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ t)$. After 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* $\mathcal{D}_B : (t; \emptyset; \mathbf{mt} \rightarrow^*_{CEK} v; S; K)$ *and* $\mathcal{D}_A : (t'; \emptyset; \mathbf{mt} \rightarrow^*_{CEK} v'; S'; K')$*, 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* t *for which a trace* max-stack$[\![\mathcal{D}_B]\!] >$ 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 $t =$ (**begin** (**set**! x (**begin** s t_1)) t_2) as a diagram; the other case is similar.



Before *AB*-normalization, a program of this shape must push the frame $F =$ (**begin** (**set**! x •) 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 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.

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

This syntax begins to look suspiciously like monadic regions [11, 12].

We extend the CEK machine into a CESK machine [6–8], 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 artifact [2].

We extend our imperative monadic syntax with imperative regions in Figure 14. 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

[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.

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

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

Fig. 14. Imperative Monadic w/ Regions Syntax

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

$$\ldots$$

MOVE

$$(\textbf{begin}\ (\textbf{set!}\ x\ rv)\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} (\textbf{begin}\ \overrightarrow{s}\ t); \Sigma[x := \Sigma(rv)]; S; K$$

ALLOC

$$(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{alloc}\ r\ wv))\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} (\textbf{begin}\ \overrightarrow{s}\ t); \Sigma[x := (r.o)]; S[(r.o) := v]; K \quad \text{fresh } o$$

RALLOC

$$(\textbf{begin}\ (\textbf{ralloc}\ r)\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} (\textbf{begin}\ \overrightarrow{s}\ t); \Sigma; S; K$$

RFREE

$$(\textbf{begin}\ (\textbf{rfree}\ r)\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} (\textbf{begin}\ \overrightarrow{s}\ t); \Sigma; \text{free}(S, r); K$$

CALL

$$(\textbf{begin}\ (\textbf{set!}\ x\ (\textbf{call}\ rv_1\ rv_2))\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} t_1; \Sigma_1[y := rv_2]; S; ((\textbf{begin}\ (\textbf{set!}\ x\ \bullet)\ \overrightarrow{s}\ t) :: K) \quad \text{where } (\textbf{closure}\ \Sigma_1\ (\lambda\ (y)\ t_1)) = S(\Sigma(rv_1))$$

AB

$$(\textbf{begin}\ (\textbf{set!}\ x\ t_1)\ \overrightarrow{s}\ t); \Sigma; S; K \rightarrow_{CESK} t_1; \Sigma; S; K$$

Fig. 15. CESK Machine for Imperative Monadic w/ Regions

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 of the CESK machine in Figure 15. 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.

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 18, 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)
18
```

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)
18
```

Listing 18. 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 REGION-SAFE).
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* MAX-STACK⟦$\mathcal{D}_M$⟧ ≥ 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 16. The ABnormal compiler targets *AB*-normal form via monadic form and *AB*-normalization, while the ANF compiler targets *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 targeting ANF directly: normalizing commuting conversions while also sequencing computations introduces (unnecessary) complexity.

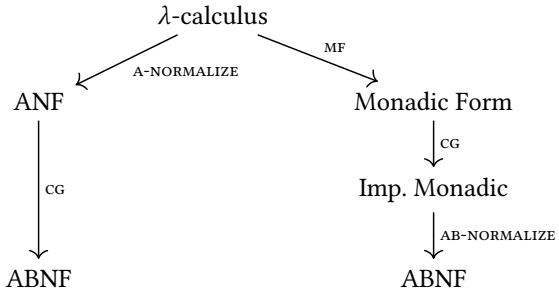


Fig. 16. Compiler Architecture

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

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

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

Fig. 17. λ-calculus to ANF Compiler with Join Points

We define ANF translation in Figure 17. 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. [9], 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. [9] 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, 4, 14, 17]. 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. [16] 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 $\textsc{anf}[\![e]\!]\kappa$ 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

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

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

Fig. 18. λ-calculus to Monadic Form Compiler

new continuation, which when called with an 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 (see *e.g.*, Siek [23]). Bowman [3] 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. [16] 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 callee. This is possible—for example, Kennedy [14] and Tolmach and Oliva [26] gives intermediate languages with constructs for introducing a local continuation, which could be compiled separately, and contrast this with ANF. Maurer et al. [17] add a primitive join point form and equations for reasoning about them, citing problems with ANF. Cong et al. [4] 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 18. The monadic form compiler merely sequences intermediate computation, but does not normalize commuting conversions.

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.

$$
\begin{aligned}
\textsc{acg}[\![\_]\!] &: M \to t \\
\textsc{acg}[\![(\textbf{let}\ (x\ N)\ M)]\!] &= (\textbf{begin}\ (\textbf{set!}\ x\ \textsc{acg}[\![N]\!]_N)\ \textsc{acg}[\![M]\!]) \\
\textsc{acg}[\![(\textbf{if0}\ V\ M_1\ M_2)]\!] &= (\textbf{if0}\ \textsc{acg}[\![V]\!]_V\ \textsc{acg}[\![M_1]\!]\ \textsc{acg}[\![M_2]\!]) \\
\textsc{acg}[\![V]\!] &= \textsc{acg}[\![V]\!]_V \\
\textsc{acg}[\![N]\!] &= \textsc{acg}[\![N]\!]_N \\
\textsc{acg}[\![\_]\!]_N &: N \to t \\
\textsc{acg}[\![(O\ \vec{V})]\!]_N &= (O\ \overrightarrow{\textsc{acg}[\![V]\!]_V}) \\
\textsc{acg}[\![V]\!]_N &= \textsc{acg}[\![V]\!]_V \\
\textsc{acg}[\![\_]\!]_V &: V \to t \\
\textsc{acg}[\![x]\!]_V &= x \\
\textsc{acg}[\![\iota]\!]_V &= \iota \\
\textsc{acg}[\![(\boldsymbol{\lambda}\ (x)\ M)]\!]_V &= (\boldsymbol{\lambda}\ (x)\ \textsc{acg}[\![M]\!])
\end{aligned}
$$

(a) From ANF

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

(b) From Monadic Form

Fig. 19. Code Generation

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 19a. 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 $\textsc{acg}[\![\_]\!]_N$ 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 19b. 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 20. 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

$$
\begin{array}{rcl}
\textsc{abnf}[\![\_]\!]_t & : & t \to t \\
\textsc{abnf}[\![(\textbf{begin}\ \overrightarrow{s}\ t)]\!]_t & = & (\textbf{begin}\ \overrightarrow{\textsc{abnf}[\![s]\!]_s}\ \textsc{abnf}[\![t]\!]) \\
\textsc{abnf}[\![(\textbf{if0}\ v\ t_1\ t_2)]\!]_t & = & (\textbf{if0}\ v\ \textsc{abnf}[\![t_1]\!]_t\ \textsc{abnf}[\![t_2]\!]_t) \\
\textsc{abnf}[\![(op\ \overrightarrow{v})]\!]_t & = & (op\ \overrightarrow{v}) \\
\textsc{abnf}[\![(\textbf{call}\ v_1\ v_2)]\!]_t & = & (\textbf{call}\ v_1\ v_2) \\
\\
\textsc{abnf}[\![\_]\!] & : & v \to v \\
\textsc{abnf}[\![x]\!]_v & = & x \\
\textsc{abnf}[\![\iota]\!]_v & = & \iota \\
\textsc{abnf}[\![(\boldsymbol{\lambda}\ (x)\ t)]\!]_v & = & (\boldsymbol{\lambda}\ (x)\ \textsc{abnf}[\![t]\!]_t) \\
\\
\textsc{abnf}[\![\_]\!]_s & : & s \to s \\
\textsc{abnf}[\![(\textbf{begin}\ \overrightarrow{s})]\!]_s & = & (\textbf{begin}\ \overrightarrow{\textsc{abnf}[\![s]\!]_s}) \\
\textsc{abnf}[\![(\textbf{set!}\ x\ v)]\!]_s & = & (\textbf{set!}\ x\ \textsc{abnf}[\![v]\!]_v) \\
\textsc{abnf}[\![(\textbf{set!}\ x\ (op\ \overrightarrow{v}))]\!]_s & = & (\textbf{set!}\ x\ (op\ \overrightarrow{v})) \\
\textsc{abnf}[\![(\textbf{set!}\ x\ (\textbf{call}\ v_1\ v_2))]\!]_s & = & (\textbf{set!}\ x\ (\textbf{call}\ v_1\ v_2)) \\
\textsc{abnf}[\![(\textbf{set!}\ x\ (\textbf{begin}\ \overrightarrow{s}\ t))]\!]_s & = & (\textbf{begin}\ \overrightarrow{\textsc{abnf}[\![s]\!]_s}\ \textsc{abnf}[\![(\textbf{set!}\ x\ t)]\!]_s) \\
\textsc{abnf}[\![(\textbf{set!}\ x\ (\textbf{if0}\ v\ t_1\ t_2))]\!]_s & = & (\textbf{if0}\ v\ \textsc{abnf}[\![(\textbf{set!}\ x\ t_1)]\!]_s\ \textsc{abnf}[\![(\textbf{set!}\ x\ t_2)]\!]_s)
\end{array}
$$

Fig. 20. AB-normalization

(or managing the implementation of) join points, and it achieves the same or better performance characteristics as ANF-compile, by Theorem 4.2.

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 of *non-local* control, such as returning from a function call (a control effect). The problems with ANF begin when conflating these two, using a solution for non-local control (continuations) to solve a local control problem (commuting conversions).

Non-local control is still important. In this paper, we never notice the non-local control problem since we never compile call and return. To compile these, we need something like a join point or a control operator–*i.e.*, we need object-language continuations by any other name. Whatever the technique, it should reify the continuation introduced by the non-tail call instruction, and used by the return instruction, into data, which is stored in the heap, and optimized by some set of equations. This would let us transform our CESK machine into a CES machine, which more closely models a real machine, with a code pointer, a register file, and memory. For example, if we add a **let/cc** *instruction* of the form (**let/cc** $k\ t$) to our ABNF, we can compile call and return as something like the following.

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

That is, when we find a non-tail call (a call in C position), we capture the current continuation and generate a tail-call (since the body of **let/cc** accepts a tail) that explicitly passes the captured

continuation as its return label. A value in C position is being returned, so we transform it into a tail-call to the continuation. The **let/cc** operation can be compiled to a labelled instruction that pushes and pops caller saved variables around the call, avoiding allocating continuations altogether[6].

This calculus starts to look, morally, like Tolmach and Oliva [26]'s SIL or Cong et al. [4]'s IL. Chez Scheme implements this transformation, suggesting it works well in practice; IL language L13 removes calls from non-tail positions, and replaces them with *goto* and a **return–point** construct (whose semantics is similar to **let/cc**)[7]. We could probably perform a rational reconstruction of this pattern from the machine semantics, as we did with ANF and monadic form.

6.2 Case-of-Case

The AB-normal compiler fails to optimize case-of-case commuting conversions. Consider the example (**if0** (**if0** e 1 0) 5 6). From this, the ANF compiler generates (**if0** e (**if0** 1 5 6) (**if0** 0 5 6)). We can see the branches have been duplicated, which looks like a problem. However, now a simple partial evaluator can optimize this to (**if0** e 6 5). By contrast, the *AB*-normal compiler 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 code is duplicated, but we cannot easily perform the same optimization. The join point calculus of Maurer et al. [17] handles this example very well, although at the cost of introducing join points.

We believe this can be supported without join points by contextually separating values and boolean position. For example, Danvy [5] present a standard monadic translation (such as we define in Figure 18) extended with the following rules (more or less).

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

This compiles pattern matching to a sublanguage of boolean expressions. With this translation, the example is transformed to (**if** (**if** (**equal**? e 0) (**equal**? 1 0) (**equal**? 0 0)) 5 6), which a simple boolean optimizer could simplify as (**if** (**if** (**equal**? e 0) **false true**) 5 6), then (**if** (**equal**? e 0) 6 5).

It's not clear that this completely solves the case-of-case problem that Maurer et al. [17] investigate, but it solves their simple example, and does so without join points. Chez Scheme implements this transformation, suggesting it works well in practice (Subsection 6.4).

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.

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

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.

[6]For a description of this in Chez, see Flatt [10, Lines 447 and 484].
[7]https://github.com/cisco/ChezScheme/blob/9576b83dd757cf1494933c9fbc80cb6aff022295/s/np-languages.ss#L1055

Similarly, regions could first be elaborated into monadic regions [11, 12] 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 if we wish to use a pure IR.

6.4 AB-normalization in Practice

The AB-reductions are the essence of a transformation found in several high-performance compilers for function languages.

AB-normalization can be seen in Chez Scheme's current implementation. Chez normalizes both the let/let and let/if commuting conversions in one step in an imperative IR, as we formalize. The can be seen, indirectly, in the difference between two of Chez's language definitions: L10[8] and L7[9].

```
(Lvalue (lvalue)
 x (mref x x imm type))
(Rhs (rhs)
 (+ lvalue literal (immediate imm) (label–ref l offset) (call ....) (alloc t) ....)
(Expr (e)
 (– .... (set! lvalue e) (let ([x e] ...) e)) ; (if e e e) is still in e
 (+ rhs (set! lvalue rhs) (values t ...))
```

Listing 19. Excerpt of Chez's L10 (Simplified)

A simplification of Chez's L10 language is presented in Listing 19. The language is implemented the Nanopass DSL [13], which enables defining a language in a BNF-like syntax with difference annotations. The *lvalue* non-terminal defines two terminal productions, either *x* or (**mref** *x x imm type*). The – meta-production removes productions from the non-terminal compared to the previous language, while the + meta-production adds new productions to the non-terminal. We've elided irrelevant productions and inlined some definition for clarity.

Listing 19 shows that **let** expressions are compiled into **set!**, and both AB_1 and AB_2 from Figure 11 are normalized. The expression syntax is almost entirely removed and replaced by three productions: the primitives forms (*rhs*), return values (**values** *t* ...), and **set!** with a primitive right-hand side; however, the (**if** *e e e*) syntax is not removed. Particularly the expressions (**set!** *lvalue e*) and (**let** ([*x e*] ...) *e*) are replaced by only (**set!** *lvalue rhs*), where algebraic expressions *e* have been unnested, as in our *s* non-terminal for AB-normal form Figure 9 (although Chez has more primitive computations, including allocation, memory references, more base values, *etc.*). Since the right-hand side of **set!** is also normalized, any **if** on the right-hand side of a **let** or **set!** must also be normalized using the AB_1 equation of Figure 11.

A key difference in Chez compared to our pipeline is that Chez does not use monadic form early in the pipeline. Chez still allows the expression (**if** (**if** e_1 e_2 e_3) e_4 e_5) in L10, just not on the right-hand side of a **set!**. Instead of using the monadic form normalization of **if**, Chez uses the case-of-case transformation presented in Subsection 6.2, transforming boolean position into a predicate sublanguage[10]. A simplification of Chez's L11 is presented in Listing 20. In the process, Chez also introduces the monadic form distinction we introduced earlier in the pipeline, resulting in essentially AB-normal form.

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

```
(Expr (e)
 (– ....)) ; entirely removed
(Tail (t)
 (+ rhs (if p t t) (seq e t) (values t ...) (goto l)))
(Pred (p)
 (true) (false) (if p p p) (seq e p) ....)
(Effect (e)
 (+ (set! lvalue rhs) (if p e e) (seq e e) ....))
```

Listing 20. Excerpt of Chez's L11 (Simplified)

The Chez compiler is still widely used, renowned for its performance, and the Nanopass definition makes comparison easy, but other high-performance compilers for functional languages have adopted similar IRs.

The TIL compiler for ML [24] uses a monadic form IR called B-form. The authors claim it is similar to A-normal form, but B-form does not normalize let/if commuting conversions. All switch and case expressions branch on a value, as do most computations, but declarations including **let** can bind switch and case expressions. The key syntax is essentially the following.

$$\begin{array}{lcl} e & ::= & v \mid .... \mid (\textbf{switch}\ v\ (f\ ...)\ d) \mid (\textbf{listcase}\ v\ (\textbf{nil} : d)\ (\textbf{cons} : f)) \\ f & ::= & (\boldsymbol{\lambda}\ (x\ ...)\ d) \\ d & ::= & v \mid (\textbf{let}\ (x\ e)\ d) \mid .... \end{array}$$

Notably, **let** cannot be let-bound (so let/let commuting conversions are normalized), but **switch** and **listcase** can appear let-bound (so let/if is not normalized). This form is preserved until code generation. The translation into machine code informally describes the AB-reductions: "Translate each arm [of the **switch**] so the the result of evaluating the arm is stored in r, the result variable for the switch. ... Note that it is easy to get the declaractions for the arms to store their results into r. We simply pass r to [the declaractions] when generating code for each arm.". This mirrors our AB_1 from Figure 11, and ABNF translation of (**set**! x (**if0** $v\ t_1\ t_2$)) from Figure 20.

Similarly, the ML to Ada compiler of Tolmach and Oliva [26] use an IR called SIL. The authors describe SIL as "it permits the result of a case to be let-bound, unlike A-normal form, and even permits the result of a let expression to be let-bound, unlike both A-normal form and TIL's B-form". Their motivation is the same as our initial motivation for AB-normal form: to transform case expressions without duplicating code or introducing continuation functions. SIL is essentially our monadic form. SIL is later compiled into MIL, a sequential IR that has the same restrictions as our AB-normal form, although MIL and the translation into MIL are only describe informally.

Our paper, in part, is an attempt to understand why and how these compilers avoids CPS and ANF—the standard choices in published formal models—in favour of an apriori strange imperative intermediate representation. The answer is that they choose ABNF before it had a name.

Acknowledgments

I'm gratefully for the feedback of the reviewers of this work, whose feedback significantly improved this paper. I'm also personally grateful to the following people: Jonathan Brachthäuser, who asked me an interesting question about ANF and regions; R. Kent Dybvig, who taught me a ton about compilation and about some of the inner workings of Chez Scheme; Conor McBride, for advice they may not know they gave me; Paulette Koronkevich, who advised me on one of the proofs.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2019-04207. Cette recherche a été financée

par le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), numéro de référence RGPIN-2019-04207.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. NN66001-22-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or NIWC Pacific.

Data Availability Statement

The software artifact with mechanized models, containing the full definitions and implementations, is publicly available [2].

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 Bowman. 2024. *A low-level look at A-normal form (artifact)*. https://doi.org/10.5281/zenodo.13376916

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

[4] 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

[5] Olivier Danvy. 2003. A New One-Pass Transformation into Monadic Normal Form. In *International Conference on Compiler Construction*. https://doi.org/10.1007/3-540-36579-6_6

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

[7] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press. https://mitpress.mit.edu/9780262062756/

[8] 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

[9] 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

[10] Matthew Flatt. 2020. Functions and Calls. In *ChezScheme Implementation Overview*. https://github.com/cisco/ChezScheme/blob/1524f8065d8e731f6f8be2caaf36d296f4b91a32/IMPLEMENTATION.md?plain=1#L447 Accessed on Aug. 20, 2024.

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

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

[13] Andrew W. Keep and R. Kent Dybvig. 2013. A nanopass framework for commercial compiler development. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2500365.2500618

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

[15] 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

[16] 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

[17] 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

[18] 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

[19] Zoe Paraskevopoulou. 2020. *Verified Optimizations for Functional Languages.* Ph. D. Dissertation. Princeton University. https://www.cs.princeton.edu/techreports/2020/006.pdf
[20] 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
[21] 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
[22] Zhong Shao and Andrew W. Appel. 1994. Space-Efficient Closure Representations. In *Conference on LISP and Functional Programming (LFP).* https://doi.org/10.1145/182409.156783
[23] Jeremy Siek. 2012. My new favorite abstract machine: ECD on ANF. http://siek.blogspot.com/2012/07/my-new-favorite-abstract-machine-ecd-on.html Accessed on Aug. 20, 2024.
[24] David Tarditi. 1996. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* Ph. D. Dissertation. Carnegie Mellon University. https://csd.cmu.edu/sites/default/files/phd-thesis/CMU-CS-97-108.pdf
[25] 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).* https://doi.org/10.1145/174675.177855
[26] 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

Received 2024-04-02; accepted 2024-08-18