MIL, a Monadic Intermediate Language for Implementing Functional Languages

Mark P. Jones
Portland State University
Portland, Oregon, USA
mpj@pdx.edu

Justin Bailey
Portland, Oregon, USA
jgbailey@gmail.com

Theodore R. Cooper
Portland State University
Portland, Oregon, USA
ted.r.cooper@gmail.com

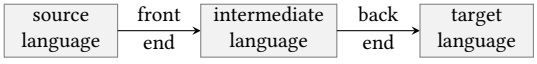
ABSTRACT

This paper describes MIL, a "monadic intermediate language" that is designed for use in optimizing compilers for strict, strongly typed functional languages. By using a notation that exposes the construction and use of closures and algebraic datatype values, for example, the MIL optimizer is able to detect and eliminate many unnecessary uses of these structures prior to code generation. One feature that distinguishes MIL from other intermediate languages in this area is the use of a typed, parameterized notion for basic blocks. This both enables new optimization techniques, such as the ability to create specialized versions of basic blocks, and leads to a new approach for implementing changes in data representation.

ACM Reference Format:
Mark P. Jones, Justin Bailey, and Theodore R. Cooper. 2019. MIL, a Monadic Intermediate Language for Implementing Functional Languages. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern compilers rely on intermediate languages to simplify the task of translating programs in a rich, high-level source language to equivalent, efficient implementations in a low-level target. For example, a compiler may use two distinct compilation steps, with a *front end* that translates source programs to an intermediate language (IL) and a *back end* that translates from IL to the target:



This splits the compilation process in to smaller, conceptually simpler components, each of which is easier to implement and maintain. Moreover, with a well-defined IL, it is possible for the front and back end components to be developed independently and to be reused in other compilers with the same source or target.

The biggest challenge in realizing these benefits is in identifying a suitable intermediate language. A good IL, for example, should retain some high-level elements, allowing a relatively simple translation from source to IL, and avoiding a premature commitment to low-level details such as data representation and register allocation. At the same time, it should also have some low-level characteristics, exposing implementation details that are hidden in the original source language as opportunities for optimization, but ultimately also enabling a relatively simple translation to the target.

In this paper, we describe MIL, a "monadic intermediate language", inspired by the monadic metalanguage of Moggi [12] and the monad comprehensions of Wadler [16], that has been specifically designed for use in implementations of functional programming languages. MIL bridges the tension between the high-level and low-level perspectives, for example, by including constructs for constructing, defining, and entering closures. These objects, and associated operations, are important in the implementation of higher-order functions, but the details of their use are typically not explicit in higher-level languages. At the same time, MIL does not fix a specific machine-level representation for closures, and does not specify register-level protocols for entering closures or for accessing their fields, delegating these decisions instead to individual back ends. This combination of design choices enables an optimizer for MIL, for example, to detect and eliminate many unnecessary uses of closures before the resulting code is passed to the back end.

MIL was originally developed as a platform for experimenting with the implementation and optimization of functional languages [2]. Since then, it has evolved in significant ways, both in the language itself—which now supports a type system with multi-value returns, for example—and its implementation—including an aggressive whole-program optimizer and novel representation transformation and specialization components. The implementation can be used as a standalone tool, but it can also be integrated in to larger systems. We currently use MIL, for example, as one of several intermediate languages in a compiler for the Habit programming language [15] with the following overall structure:

In the initial stage of this compiler, Habit programs are desugared to LC, a simple functional language that includes lambda and case constructs (LC is short for "LambdaCase" and is named for those two features) but lacks the richer syntax and semantics of Habit. LC programs are then translated in to MIL, processed by the tools described in this paper, and used to generate code for LLVM [9, 10], which is a popular, lower-level IL. There is, for example, no builtin notion of closures in LLVM, so it would be difficult to implement analogs of MIL closure optimizations at this stage of the compiler. However, the LLVM tools do implement many important, lower-level optimizations and can ultimately be used to generate assembly code ("asm" in the diagram) for an executable program.

IFL'18, August 2019, Lowell, MA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Outline and Contributions

We begin the remaining sections of this paper with a detailed overview of the syntax and features of MIL (Section 2). To evaluate its suitability as an intermediate language, we have built a practical implementation that uses MIL: as the target language for a simple front end (Section 3); as a framework for optimization (Section 5), and as a source language for an LLVM back end (Section 6). Along the way, we describe new global program transformations techniques for implementing changes in data representation (Section 4).

MIL has much in common with numerous other ILs for functional languages, including approaches based on CPS [1, 8], monads [12, 16], and other functional representations [5, 11]; these references necessarily only sample a very small portion of the literature. Our work makes some new contributions by exploring, for example, a parameterized form of basic block, and the new optimizations for "derived blocks" that this enables (Section 5.5), as well as new techniques for representation transformations (Section 4). An additional contribution of our work is the combination of many small design and engineering decisions that, together, result in a coherent and effective IL design, scaling to a practical system that can be used either as a standalone tool or as part of a larger compiler.

2 AN OVERVIEW OF MIL

We begin with an informal introduction to MIL, highlighting fundamental concepts, introducing terminology, and illustrating the concrete syntax for programs. We follow Haskell conventions [14] for basic syntax—from commenting and use of layout, to lexical details, such as the syntax of identifiers. That said, while we expect some familiarity with languages like Haskell, we will not assume deep knowledge of any specific syntax or features.

2.1 Types in MIL

MIL is a strongly typed language and all values that are manipulated in MIL must have a valid type. MIL provides several builtin types, including `Word` (machine words) and `Flag` (single-bit booleans). MIL also supports first class functions with types of the form $[d_1,\ldots,d_n]\twoheadrightarrow[r_1,\ldots,r_m]$ that map a tuple of input values with types d_i to a tuple of results with types r_j. (The prefixes `d` and `r` were chosen as mnemonics for domain and range, respectively.) Curried functions can also be described by using multiple $\twoheadrightarrow$ types. For example, the type `[Word]↠[[Word]↠[Flag]]` could be used for a curried function that compares `Word` values.

MIL programs can also include definitions for parameterized algebraic datatypes, as illustrated by the following familiar examples:

```
data List a   = Nil | Cons a (List a)
data Pair a b = MkPair a b
```

Each definition introduces a new type name (`List` and `Pair` in these examples) and one or more *data constructors* (`Nil`, `Cons`, and `MkPair`). Each data constructor has an *arity*, which is the number of fields it stores. For example, `Nil` has arity 0 and `Cons` has arity 2.

MIL also supports `bitdata` types [4] that specify a bit-level representation for all values of the type. In other respects, however, values of these types can be used in the same way as values of algebraic datatypes. With the following definitions, for example: Every value of type `Bool` is either `False` or `True`, with either option being represented by a single bit; and a `PCI` address can be described by a 16 bit value with subfields of width 8, 5, and 3:

```
bitdata Bool = False [B0] | True [B1]
bitdata PCI  = PCI [bus::Bit 8 | dev::Bit 5 | fun::Bit 3]
```

Bitdata types allow programmers to match the bit-level representation of data structures that are used in low-level systems applications, which is a key domain for Habit. Ultimately, however, `bitdata` values are represented by sequences of `Word` values (Section 4.2). For completeness, we mention that MIL also supports several other builtin types for systems programming, including the `Bit n` types seen here that represent bit vectors of width `n`.

2.2 MIL programs

The core of a MIL program is a sequence of block, closure, and top-level definitions, as suggested by the following grammar:

```
prog ::=  def1 ... defn                      -- program
def  ::=  b[v1,...,vn] = c                   -- block
       |  k{v1,...,vn} [u1,...,um] = c       -- closure
       |  [v1,...,vn] ← t                    -- top-level
       |  ...                                -- other
```

The last line here uses "..." as a placeholder for other forms of definition, such as those introducing new types (see the previous section). The full details, however, are not central to this paper.

2.3 Blocks

Blocks in MIL are like basic blocks in traditional optimizing compilers except that each block also has a list of zero or more parameters. Intuitively, these can be thought of as naming the "registers" that must be loaded with appropriate values before the block is executed. The syntax of block definitions is reminiscent of Haskell's monadic notation (without the `do` keyword), as in the following schematic example for a block `b` with arguments $u_1,\ldots,u_n$:

```
b[u1,...,un] = v1 ← t1  -- run t1, capture result in v1
               ...
               vn ← tn  -- run tn, capture result in vn
               e
```

The body of this block is executed by running the statements $v_i \leftarrow t_i$ in sequence, capturing the result of each computation t_i in a variable v_i so that it can be referenced in later steps. The expression `e` on the last line may be either a tail expression (such as a `return` or an unconditional jump to another block) or a `case` or `if` construct that performs a conditional jump.

MIL relies on type checking to ensure basic consistency properties (for example, to detect calls with the wrong number or types of arguments). The code fragment above assumes that each t_i returns exactly one result but it is possible to use computations that return zero or more results at the same time, as determined by the type of t_i. In these cases, the binding for t_i must specify a corresponding list of variables, enclosed in brackets, as in $[w_1,\ldots,w_m] \leftarrow t_i$.

2.4 Tail Expressions

MIL allows several different forms of "tail" expressions (each of which may be used in place of the expressions $t_1,\ldots,t_n$ in the preceding discussion), as summarized in the following grammar.

```
t  ::=  return [a1,...,an]  -- monadic return
    |   b[a1,...,an]        -- basic block call
    |   p((a1,...,an))      -- primitive call
    |   C(a1,...,an)        -- allocate data value
    |   C i a               -- select component
    |   k{a1,...,an}        -- allocate closure
    |   f @ [a1,...,an]     -- enter closure
```

Every tail expression represents a computation that (potentially) returns zero or more results:

- `return [a1,...,an]` returns the values $a_1, \ldots, a_n$ immediately with no further action. The most frequently occurring form produces a single result, `a`, and may be written without brackets as `return a`.
- `b[a1,...,an]` is a call to a block `b` with the given arguments. When a block call occurs at the end of a code sequence, it can potentially be implemented as a tail call (i.e., a jump). This is why we refer to expressions in the grammar for `t` as *tail* expressions or sometimes *generalized tail calls*.
- `p((a1,...,an))` is a call to a primitive `p` with the given arguments. Primitives typically correspond to basic machine instructions (such as `add`, `and`, or `shr`), but can also be used to link to external functions. Primitive calls are written with double parentheses around the argument list so that they are visibly distinguished from block and constructor calls.
- `C(a1,...,an)` allocates a data value with constructor `C` and components $a_1, \ldots, a_n$, where `n` is the arity of `C`.
- `C i a` is a selector that returns the `i`th component of the value in `a`, assuming that it was constructed with `C`. For example, having executed `v ← C(a1,...,an)`, a subsequent `C i v` call (with $0 \le i < n$) will return a_{i+1}. (Note that `i` must be a constant and that the first component has index `0`.)
- `k{a1,...,an}` allocates a closure (i.e., the representation of a function value) with a code pointer specified by `k` and arguments $a_1, \ldots, a_n$. This is only valid in a program that includes a corresponding definition for `k`. Closures are an important feature of MIL and are the topic of further discussion throughout this paper.
- `f @ [a1,...,an]` is a call to an unknown function, `f`, with arguments $a_1, \ldots, a_n$. For this to be valid, `f` must hold a closure for a function that expects exactly `n` arguments. The process of executing an expression of this form is often referred to as *entering* a closure. Mirroring the syntax for `return`, we write `f @ a` in the case where there is exactly one argument. As another special case, an unknown function `f` can be applied to an empty list of arguments, as in `f @ []`; this can be used, for example, to invoke a (monadic) thunk.

2.5 Atoms

In the preceding examples, the symbols $a, a_1, \ldots, a_n$, and `f` represent arbitrary *atoms* that are either numeric constants (integer literals) or variable names. The latter correspond either to *temporaries* (i.e., local variables holding parameter values or intermediate results) or to variables defined at the top level (see Section 2.9):

```
a  ::=  i  -- a constant (i.e., an integer literal)
    |   v  -- a variable name
```

Atoms are the only form of expression that can be used as arguments in tail expressions. If a program requires the evaluation of a more complex expression, then it must be computed in steps using temporaries to capture intermediate results. For example, the following block calculates the sum of the squares of its inputs using the `mul` and `add` primitives for basic arithmetic:

```
sumSqrs[x,y] = u ← mul((x,x))  -- compute x*x in u
               v ← mul((y,y))  -- compute y*y in v
               add((u,v))      -- return the sum
```

2.6 Block Types

Blocks are not first class values in MIL, so they cannot be stored in variables, passed as inputs to a block, or returned as results. Nevertheless, it is still useful to have a notion of types for describing the inputs and results of each block, and we use the notation `[d1,...,dn] >>= [r1,...,rm]` for this where d_i and r_j are the types of the block's inputs and results, respectively. For example, the type of `sumSqrs` can be declared explicitly as follows:[1]

```
sumSqrs :: [Word,Word] >>= [Word]
```

Blocks may have polymorphic types, as in the following examples, the second of which also illustrates a block with multiple results:

```
idBlock    :: forall (a::type). [a] >>= [a]
idBlock[x]  = return x

dupBlock   :: forall (a::type). [a] >>= [a, a]
dupBlock[x] = return [x, x]
```

The explicit `forall` quantifiers here give a precise way to document polymorphic types but they are not required; those details can be calculated automatically instead using kind and type inference.

It is important to distinguish block types, formed with `>>=`, from the first-class function types using `->>` that were introduced in Section 2.1. The notations are similar because both represent a kind of function, but there are also fundamental differences. In particular, blocks are executed by jumping to a known address and not by entering a closure, as is done with a first-class function.

2.7 Code Sequences

The syntax for the code sequences on the right of each block definition is captured in the following grammar:

```
c    ::=  [v1,...,vn] ← t; c     -- monadic bind
      |   assert a C; c          -- data assertion
      |   t                      -- tail call
      |   if v then bc1 else bc2 -- conditional
      |   case v of alts         -- case construct
alts ::=  {alt1;...;altn}        -- alternatives
alt  ::=  C → bc                 -- match against C
      |   _ → bc                 -- default branch
bc   ::=  b[a1,...,an]           -- block call
```

As illustrated previously, any given code sequence begins with zero or more *monadic bind* steps (or *statements*) of the form `vs ← t` (where `vs` is either a single temporary or a bracketed list). Each such step describes a computation that begins by executing the tail

[1] The same block type is also used for the `mul` and `add` primitives.

t, binding any results to the temporaries in vs, and then continuing with the entries in vs now in scope. The grammar also includes an assert construct; its purpose will be illustrated later.

Where space permits, we write the steps in a code sequence using a vertical layout, relying on indentation to reflect its structure. However, we will also use a more compact notation, matching the preceding grammar with multiple statements on a single line:

```
v₁ ← t₁; v₂ ← t₂; ...; vₙ ← tₙ; e
```

However they are written, every code sequence is ultimately terminated by an expression e that is either an unconditional jump (i.e., a tail expression, t); or a conditional jump (i.e., an if or case construct). Note that, like a traditional basic block, there are no labels in the middle of code sequences. As a result, we can only enter a code sequence at the beginning and, once entered, we cannot leave until we reach the end expression e.

2.8 Conditional Jumps

An if construct ends a code sequence with a conditional jump; it requires an atom, v, of type Flag as the test expression and block calls for each of the two branches. Beyond these details, if constructs work in the usual way. For example, the following two blocks can be used to calculate the maximum of two values u and v:

```
b1[u,v] = t ← primGt((u,v))  -- primitive for u>v
          if t then b2[u] else b2[v]
b2[x]   = return x
```

Other conditional jumps can be described with a case v of alts construct, using the value of v—the "discriminant"—to choose between the alternatives in alts. Specifically, a case construct is executed by scanning the list of alternatives and executing the block call bc for the first one whose constructor, C, matches v. The following code sequence corresponds to an if-then-else construct for a variable v of type Bool: the code will execute b1[x,y] if v was built using True(), or b2[z] if v was built using False():

```
case v of { True → b1[x,y]; False → b2[z] }
```

We can also write the previous expression using a default branch (signaled by an underscore) instead of the case for False:

```
case v of { True → b1[x,y]; _ → b2[z] }
```

Once again, we often write examples like this with a vertical layout, eliding the punctuation of semicolons and braces:

```
case v of
  True → b1[x,y]
  _    → b2[z]
```

Each alternative in a case only tests the outermost constructor, C, of the discriminant, v. If the constructor matches but subsequent computation requires access to components of v, then they must be referenced explicitly using C i v selectors, where i is the appropriate field number. The following blocks, for example, can be used to compute the length of a list:

```
length[list] = loop[0,list]
loop[n,list] = case list of
                 Nil  → done[n]
                 Cons → step[n,list]
done[n]      = return n
step[n,list] = assert list Cons
               tail ← Cons 1 list
               m ← add(n,1)
               loop[m,tail]
```

The length block passes the given list to loop, with an initial (accumulating) parameter 0. The loop block uses a case to examine the list. A Nil value represents an empty list, in which case we branch to done and immediately return the current value of n. Otherwise, list must be a Cons, and we jump to step, which uses a Cons 1 list selector to find the tail of the list. After calculating an incremented count in m, step jumps back to examine the rest of the list. The assert here indicates that list is known to be a Cons node; this information ensures that the Cons selector in the next line is valid, and can also be leveraged during optimization.

2.9 Top-level Definitions

MIL programs can use definitions of the form [v₁,...,vₙ] ← t to introduce top-level bindings for variables called v₁,...,vₙ. Variables defined in this way are initialized to the values that are produced by executing the tail expression t. In the common case where a single variable is defined, we omit the brackets on the left of the ← symbol and write v₁ ← t. Note that variables introduced in a definition like this are not the same as *global variables* in an imperative programming language because their values cannot be changed by a subsequent assignment operation.

Top-level definitions like this are typically used to provide names for constants or data structures (including closures, as shown below). For example, the following top-level definitions, each with a data constructor on the right hand side of the ← symbol, will construct a static list data structure with two elements:

```
list1 ← Cons(1, list2)
list2 ← Cons(2, nil)
nil   ← Nil()
```

It is possible to specify types for variables introduced in a top-level definition using declarations of the form v₁, ..., vₙ :: t. For example, we might declare types for the list variables defined previously by writing:

```
list1, list2, nil :: List Word
```

However, there is no requirement to provide types for top-level variables because they can also be inferred in the usual way.

2.10 Closures and Closure Definitions

First-class functions in MIL are represented by *closure* values that are constructed using tail expressions of the form k{a₁,...,aₙ}. Here, a₁,...,aₙ are atoms that will be stored in the closure and accessed when the closure is entered (i.e., applied to an argument) and the code identified by k is executed. For each different k that is used in a given program, there must be a corresponding *closure definition* of the form k{v₁,...,vₙ} [u₁,...,uₘ] = t. Here, the vᵢ are variables representing the values stored in the closure; the uᵢ are variables representing the arguments that are required when the closure is entered; and *t* is a tail expression that may involve any of these variables. Conceptually, each such closure definition corresponds to the code that must be executed when a closure is entered, loading stored values and arguments as necessary in to

registers before branching to the code as described by `t`. Following our usual pattern, in the case where there is just one argument, u_1, the brackets may be omitted and the definition can be written `k{`v_1`,...,`v_n`} `u_1` = t`. The ability to pass multiple (or zero) arguments u_i to a closure, however, is important because it allows us to work with values whose representation may be spread across multiple components; we will return to this in Section 4.2.

As an example, given the definition `k{n} x = add((n,x))`, we can use a closure with code pointer `k` and a stored free variable `n` to represent the function (`λx → n + x`) that will return the value `n+x` whenever it is called with an argument `x`.

The type of any closure can be written in the form:

```
{t1,...,tn} [d1,...,dm] ↠ [r1,...,rp]
```

where the t_i are the types of the stored components, the d_j are the types of the inputs (the domain), and the r_k are the types of the results (the range). This is actually a special case of an *allocator type* `{t1,...,tn} t`, which corresponds to a value of type `t` whose representation stores values of the types listed in the braces. For example, the type of the closure `k` defined above is `{Word} [Word] ↠ [Word]`, while the type of the `Cons` constructor for lists can be written `{a, List a} List a`.

2.11 Example: Implementing map in MIL

In this section, we illustrate how the components of MIL described previously can be used together to provide an implementation for the familiar `map` function. In a standard functional language, we might define this function using the following two equations:

```
map f Nil         = Nil
map f (Cons y ys) = Cons (f y) (map f ys)
```

Using only naive techniques, this function can be implemented by the following MIL code (presented here in two columns) with one top-level, two closure, and three block definitions:

```
map      ← k0{}              b2[f,xs] = assert xs Cons
k0{}  f  = k1{f}                      y  ← Cons 0 xs
k1{f} xs = b0[f,xs]                   ys ← Cons 1 xs
b0[f, xs] = case xs of                z  ← f @ y
              Nil  → b1[]             m  ← map @ f
              Cons → b2[f,xs]         zs ← m @ ys
b1[]     = Nil()                      Cons(z,zs)
```

This implementation starts with a top-level definition for `map`, binding it to a freshly constructed closure `k0{}`. When the latter is entered with an argument `f`, it captures that argument in a new closure structure `k1{f}`. No further work is possible until this second closure is entered with an argument `xs`, which results in a branch to the block `b0`, and a test to determine whether the list value is either a `Nil` or `Cons` node. In the first case, the `map` operation is completed by returning `Nil` in block `b1`. Otherwise, we execute the code in `b2`, extracting the head, `y`, and tail, `ys`, from the argument list. This is followed by three closure entries: the first calculates the value of `f y`, while the second and third calculate `map f ys`, with one closure entry for each argument. The results are then combined using `Cons(z,zs)` to produce the final result for the `map` call.

The MIL definition of `map` reveals concrete implementation details, such as the construction of closures, that are not immediately visible in the original code. For an intermediate language, this is exactly what we need to facilitate optimization, and we will revisit this example in Section 5.4, showing how the code in `b2` can be rewritten to avoid unnecessary construction of closures.

2.12 Notes on Formalization of MIL

Although we do not have space here for many details, we have developed both a formal type system and an abstract machine for MIL. The former has served as a guide in the implementation of the MIL typechecker, which is useful in practice for detecting errors in MIL source programs (and, occasionally, for detecting bugs in our implementation). Perhaps the most interesting detail here is the appearance of a type constructor, M—representing the underlying monad in which MIL computations take place—in rules like the following (for type checking bindings in code sequences):

$$\frac{A \vdash \mathtt{t} : M\,[r_1, \ldots, r_n] \quad A, \mathtt{v}_1 : r_1, \ldots, \mathtt{v}_n : r_n \vdash \mathtt{c} : M\,a}{A \vdash (\mathtt{[v_1, \ldots, v_n]} \leftarrow \mathtt{t;\ c}) : M\,a}$$

An interesting direction for future work is to allow the fixed `M` to be replaced with a type variable, `m`, and to perform a monadic effects analysis by collecting constraints on `m`.

The design of an abstract machine for MIL has also had practical impact, guiding the implementation of a bytecode interpreter that is useful for testing. In the future, the abstract machine may also provide a formal semantics for verifying the program rewrites used in the MIL optimizer (Section 5).

3 COMPILING LC TO MIL

In this section, we discuss the work involved in compiling programs written in LC—a simple, high-level functional language—into MIL. This serves two important practical goals: First, by using MIL as the target language, we demonstrate that it has sufficient features to serve as an intermediate language for a functional language with higher-order functions, pattern matching, and monadic operations. Second, in support of testing, it is easier to write programs in LC and compile them to MIL than to write MIL code directly.[2]

3.1 Translating LC Types to MIL

The task of translating LC types to corresponding MIL types is almost trivial: the two languages have the same set of primitive types and use the same mechanisms for defining new data and bitdata types. The only complication is in dealing with function types in LC, which map a single input to a single result, instead of the tuples of inputs and results that are used in MIL. To bridge this gap, we define the LC function type, written using a conventional infix → symbol, as an algebraic datatype:

```
data d → r = Func ([d] ↠ [r])
```

With this definition, we now have three different notions of function types for MIL: ↠ and → describe first-class function values while >>= is for block types (Section 2.6). The following definitions show how all of these function types can be used together in a MIL implementation of the LC identity function, (`λx → x`):

[2]Our implementation actually accepts combinations of MIL and LC source files as input; this allows users to mix higher-level LC code that is translated automatically to MIL with handwritten MIL code. The latter is useful in practice for implementing libraries of low-level primitives that cannot be expressed directly in LC.

```
k    :: {} [a] ↠ [a]  -- a closure definition for
k{} x = return x      -- the identity function

b  :: [] >>= [a → a]  -- create a closure value and
b[] = c ← k{}         -- package it as a → function
      Func(c)

id :: a → a           -- set top-level name id to the
id ← b[]              -- value produced by b[]
```

A legitimate concern here is that every use of an LC function (→) requires extra steps to wrap or unwrap the `Func` constructor around a MIL function (↠). Fortunately, we will see that it is possible to eliminate these overheads (Section 4.1).

3.2 Translating LC Code to MIL

There is a large body of existing work on compilation of functional languages, much of which can be easily adapted to the task of translating LC source programs in to MIL. As a simple example, to compile a lambda expression like `λx → e` with free variables `fvs`, we just need to: pick a new closure name, `k`; add a definition `k{fvs} x = e'` to the MIL program (where `e'` is compiled code for `e`); and then use `k{fvs}` in place of the original lambda expression. Beyond these general comments, we highlight the following details from our implementation:

- To simplify code generation, we use a lambda lifting transformation [6] to move locally defined recursive definitions to the top-level (possibly with added parameters).
- Our code generator is based on compilation schemes for "compiling with continuations" [8]. As a rough outline (in pseudo-Haskell notation), it can be described as a function:

  ```
  compile :: Expr → (Tail → CM Code) → CM Code
  ```

  The first argument is the LC expression, `exp`, that we want to compile. The second argument is a continuation that takes a MIL tail expression, `t`, corresponding to `exp`, and embeds it in a MIL code sequence for the rest of the program. For instance, if `exp` is the lambda expression `λx → e` in the example at the start of this section, then `t` will be the closure allocator `k{fvs}`. Note that `CM` in the type above represents a "compilation monad", with operations for generating new temporaries, and for adding new block, closure, or top-level definitions to the MIL program as compilation proceeds.
- Continuation based techniques require special care with conditionals like `if c then t else f`). In particular, we need to ensure that the `(Tail → CM Code)` continuation is not applied separately to each of the tail expressions for `t` and `f`, which could lead to duplicated code. Fortunately, it is easy to avoid this in MIL by placing the continuation code in a new block, serving as a "join point" [11], and then having the code in each branch end with a jump to that block.

4 REPRESENTATION TRANSFORMATIONS

Although MIL is more broadly applicable, our current implementation assumes a whole-program compilation model. One benefit of this is that we can consider transformations that require changes across many parts of input programs. In the following subsections, we describe three specific transformations of this kind, all implemented in our toolset, that change the representation of data values in MIL programs. Each of these typically requires modifications in both code and type definitions. For example, if a program uses values of type `T` that can be more efficiently represented as values of type `T'`, then implementing a change of representation will require updates, not only to code that manipulates values of type `T`, but also to any type definitions that mention `T`. Our tools allow these transformations to be applied in any order or combination, running the MIL type checker after each pass as a sanity check (although the original types are not preserved, each transformation is expected to preserve typeability). In addition, because these transformations can create new opportunities for optimization, it is also generally useful to (re)run the MIL optimizer (Section 5) after each pass.

4.1 Eliminating Single Constructor Types

Our first application for representation transformations deals with 'single constructor', non-recursive algebraic datatypes with definitions of the form: `data T` a_1 `...` a_n `= C t'`. Types like this are often used to create wrappers that avoid type confusion errors. For example, by defining `data Time = Millis Word`, we ensure that values of type `Word` are not used accidentally where `Time` values are actually required. And, as discussed in Section 3.1, we also use a type of this form to map between the → and ↠ function types when compiling LC to MIL. These types are useful because they can enforce correct usage in source programs. But for compilation purposes, the extra constructors—like `Millis` and `Func`—add unnecessary runtime overhead, and can block opportunities for optimization. To avoid this, we can rewrite the MIL program to replace every tail of the form `C 0 a` or `C(a)` with `return a` (effectively treating selection and construction as the identity function) and every `case v of C → bc` that 'matches' on `C` with a direct block call `bc`. In addition, the original definition of `T` can be discarded, but every remaining type of the form `T` t_1 `...` t_n must be replaced with the corresponding instance $[t_1/a_1,\ldots,t_n/a_n]$`t'` of `t'`.

4.2 Representation Vectors

Our second application was initially prompted by the use of `bitdata` types in MIL (see Section 2.1) but is also useful with some algebraic datatypes. In the early stages of compilation, we manipulate values of bitdata types like `Bool` or `PCI` with the same pattern matching and constructor mechanisms as algebraic datatypes. At some point, however, the compiler must transition to the bit-level representation that is specified for each of these types. This means, for example, that values of type `Bool` should be represented by values of type `Flag`. Similarly, `PCI` values might be represented using `Word` values, with the associated selectors for `bus`, `dev`, and `fun` replaced by appropriate bit-twiddling logic using shift and mask operations.

To specify representation changes like this, we define a function that maps every MIL type `t` to a suitable *representation vector*: a list of zero or more types that, together, provide a representation for `t`. We use a vector, rather than a single type, to accommodate types that require multi-word representations. A `Bit 64` value, for example, cannot be stored in one 32 bit word, but can be supported by using a representation vector `[Word,Word]` with two `Word` values. Similarly, a zero length vector, `[]`, can be used for `Bit 0` and,

indeed, for any other singleton type, such as the standard unit type, `()`. This reflects the fact that, if a type only has one possible value, then it does not require a runtime representation.

The resulting representation vectors can be used to guide a program transformation that replaces each variable `v` of a type `t` with a list of zero or more variables $v_1, \ldots, v_n$, where `n` is the length of the representation vector for `t`. This was the primary technical motivation for using tuples of values throughout MIL because it allows us to rewrite tails like `f @ a` and top-level definitions like `v ← t`, for example, as `f @ [a1,a2]` or `[v1,v2] ← t` when `a` and `v` are each represented by two words. Of course, additional rewrites are needed for selectors and primitive calls that use values whose representation is changed. A convenient way to manage this is to replace the operations in question with a calls to new blocks that will be inlined and optimized with the surrounding code.

Support for the representation transformation described here is a distinguishing features of our toolset: to our knowledge, no other current system implements a transformation of this kind.

4.3 Specializing Polymorphic Definitions

Our third application, included to satisfy a requirement of the LLVM backend (Section 6), is a transformation that eliminates polymorphic definitions and generates specialized code for each monomorphic instance that is needed in a given program. One problem is that this transformation cannot be used in some programs that rely on polymorphic recursion [13]. Another concern is that specialization has the potential to cause an explosion in code size. In practice, however, specialization has proved to be an effective tool in domain-specific [3] and general-purpose functional language compilers [17], and even in implementations of overloading [7].

Representation transformation is relevant here when we consider the task of generating code for specific monomorphic instances of polymorphic functions. As part of this process, our implementation of specialization also eliminates all uses of parameterized datatypes. A program that uses a value of type `Pair PCI PCI`, for example, might be transformed in to code that uses a value of a new type `data` $\texttt{Pair}_1$ `= MkPair1 PCI PCI`, that is generated by the specializer. This provides an interesting opportunity for using type-specific representations. For example, the $\texttt{Pair}_1$ type described here should hold two 16-bit `PCI` values, so it could easily be represented using a single, 32-bit `Word` with no need for heap allocation. We have already started to explore the possibility of inferring bit-level representations for a wide-range of types like this, and expect to include support for this in a future release of our toolset.

5 OPTIMIZATION

A common goal in the design of an intermediate language is to support optimization: program transformations that preserve program behavior but improve performance, reduce memory usage, etc. For MIL programs, we have identified a collection of rules that describe how some sections of code can be rewritten to obtain better performance. A small but representative set of these rules is presented in Figure 1. The full set has been used to build an optimizer for MIL that works by repeatedly applying rewrites to input programs. Individual rewrites typically have limited impact, but using many rewrites in sequence can yield significant improvements by reducing code size, eliminating unnecessary operations, and replacing expensive operations with more efficient equivalents.

The table in Figure 1 has three columns that provide, for each rule: a short name; a rewrite; and, in several cases, a set of side conditions that must be satisfied in order to use the rule. The rewrites are written in the form `e` $\Longrightarrow$ `e'` indicating that an expression matching `e` should be replaced by the corresponding `e'`. To avoid ambiguity, we use two variants of this notation with $\Longrightarrow_c$ for rewrites on code sequences and $\Longrightarrow_t$ for rewrites on tails. In the rest of this section, we will walk through each of the rewrites in Figure 1 in more detail (Sections 5.1–5.5), describe additional optimizations that are supported by our implementation (Section 5.6), and reflect on the overall effectiveness of our optimizer for MIL (Section 5.7).

5.1 Using the Monad Laws

The first group of rules are by standard laws for monads (as described, for example, by Wadler [16]), but are also recognizable as traditional compiler optimizations. Rule (1), for example, implements a form of *copy* or *constant propagation*, depending on whether the atom `a` is a variable or a constant. The notation `[a/x]c` here represents the substitution of `a` for all free occurrences of `x` in the code sequence `c`; concretely, instead of creating an extra variable, `x`, to hold the value of `a`, we can just use that value directly. Rule (2) corresponds to the right monad law, which can also be seen as eliminating an unnecessary `return` at the end of a code sequence and potentially as creating a new opportunity for a tail call. Finally, Rules (3) and (4) are based on the associativity law for monads, but can also be understood as descriptions of function inlining rules; we use the terms *prefix* and *suffix* to distinguish between cases where the block being inlined appears at the beginning or the end of the code sequence. To better understand the relationship with the associativity law, note that a naive attempt to inline the block `b` in the code sequence `v ← b[x]; c`, using the definition of `b` in the figure, would produce `v ← (`v_0 `←` t_0`;` t_1`); c`. With the previous grammar for MIL code sequences, this is not actually a valid expression. Using associativity, however, it can be flattened/rewritten as the code sequence on the right hand side of the rule. As is always the case, unrestricted use of inlining can lead to an explosion in code size with little or no benefit in performance. To avoid such problems, our implementation uses a typical set of heuristics—limiting inlining to small blocks or to blocks that are only used once, for example—to strike a good balance between the benefits and potential risks of an aggressive inlining strategy.

5.2 Eliminating Unnecessary Code

The second group of rewrites in Figure 1 provide ways to simplify MIL programs by trimming unnecessary code. Rule (5), for example, uses the results of a simple analysis to detect tail expressions `t` that do not return (e.g., because they call a primitive that terminates the program, or enter an infinite loop). In these situations, any code that follows `t` can be deleted without a change in semantics.

Rules (6) and (7) allow us to detect, and then, respectively, to eliminate code that has no effect. Rule (6), sets the result variable name for a statement to underscore to flag situations where the original variable is not used in the following code. A subsequent

Name	Rewrite			Conditions
Using the monad laws, where block b is defined by b[x] = $v_0 \leftarrow t_0$; t_1				
1) Left monad law (constant propagation)	x ← return a; c	$\Longrightarrow_c$	[a/x]c	–
2) Right monad law (tail call introduction)	x ← t; return x	$\Longrightarrow_c$	t	–
3) Prefix inlining	v ← b[x]; c	$\Longrightarrow_c$	$v_0 \leftarrow t_0$; v ← t_1; c	–
4) Suffix inlining	v ← t; b[x]	$\Longrightarrow_c$	v ← t; $v_0 \leftarrow t_0$; t_1	–
Eliminating unnecessary code				
5) Unreachable code elimination	v ← t; c	$\Longrightarrow_c$	t	t does not return
6) Wildcard introduction	v ← t; c	$\Longrightarrow_c$	_ ← t; c	v is not free in c
7) Dead tail elimination	_ ← t; c	$\Longrightarrow_c$	c	t is pure
8) Common subexpression elimination	t	$\Longrightarrow_t$	return v	{v=t}
Using algebraic identities (focusing here on bitwise and and writing M, N, and P for arbitrary integer constants)				
9) Identity laws	and((v,0))	$\Longrightarrow_t$	return 0	–
10) Idempotence	and((v,v))	$\Longrightarrow_t$	return v	–
11) Constant folding	and((M,N))	$\Longrightarrow_t$	return (M&N)	–
12) Commutativity	and((M,v))	$\Longrightarrow_t$	and((v,M))	–
13) Associative folding (u&M)&N = u&(M&N)	and((v,N))	$\Longrightarrow_t$	and((u,M&N))	{v=and((u,M))}
14) Distributive folding (1) (u|M)&N = (u&N)|(M&N)	and((v,N))	$\Longrightarrow_c$	v' ← and((u,N)); or((v',M&N))	{v=or((u,M))}
15) Distributive folding (2) ((u|M)&N)|P = (u&N)|((M&N)|P)	or((w,P))	$\Longrightarrow_c$	v' ← and((u,N)); or((v',(M&N)|P))	{v=or((u,M)), w=and((v,N))}
Known structures, where closure k is defined by k{x} y = t				
16) Known constructor	case v of ...; C → b'[...]; ...	$\Longrightarrow_c$	b'[...]	{c=C(...)}
17) Known closure	f @ y	$\Longrightarrow_t$	t	{f=k{x}}
Derived blocks, where block b is defined by b[x] = $v_0 \leftarrow t_0$; t_1				
18) Known structure	b[v] where b'[y] = x ← C(y); $v_0 \leftarrow t_0$; t_1	$\Longrightarrow_t$	b'[y]	{v=C(y)}
19) Trailing enter	f ← b[x]; f @ a where b'[x,a] = $v_0 \leftarrow t_0$; f ← t_1; f @ a	$\Longrightarrow_c$	b'[x,a]	–
20) Trailing case	v ← b[x]; case v of ... where b'[x,...] = $v_0 \leftarrow t_0$; v ← t_1; case v of ...	$\Longrightarrow_c$	b'[x,...]	–

Figure 1: A representative set of rewrite rules for MIL optimization

use of Rule (7) will then eliminate the statement altogether if the associated tail expression t has no externally visible effects (for example, if t is a call to a pure primitive function like add or mul, or if t is a closure or data allocator). Our presentation of this process using two separate rules reflects the fact that our optimizer actually applies these two rules in separate passes over the abstract syntax.

Rule (8) uses a local dataflow analysis to detect situations where the result of a previous computation can be reused. To implement this, our optimizer calculates a set of "facts", each of which is a statement of the form v=t, as it traverses the statements in each code sequence. Starting with an empty set, the optimizer will add (or "generate") a new fact v=t for every statement v ← t that it encounters with a pure t. At the same time, for each statement v ← t, it will also remove (or "kill") any facts that mention v because the variable that they reference will no longer be in scope. The rightmost column in Figure 1 documents the facts that are required to apply each rewrite. In this case, given v=t, we can avoid recalculating t and just return the value v that it produced previously. (In practice, the return introduced here will often be eliminated later using Rules (1) or (2).)

5.3 Using Algebraic Identities

The next group (Rules (9)–(15)) take advantage of (mostly) well-known algebraic identities to simplify programs using the builtin primitives for arithmetic, logic, and comparison operations. We only show a small subset of the (more than 100) rewrites of this kind that are used in our implementation, all of which involve the bitwise and primitive. In combination with additional rewrites involving bitwise or and shift operations, these rules are very effective in simplifying the bit-twiddling code that is generated when manipulating or constructing Habit-style bitdata types [4, 15].

Rules (9), (10), and (11), for example, each eliminate a use of and in a familiar special case. Rule (12) is not useful as an optimization by itself, but instead is a first step in rewriting expressions in to

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← map @ f
           zs ← m @ ys
           Cons(z,zs)
```
(a)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← k1{f}
           zs ← m @ ys
           Cons(z,zs)
```
(b)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           m  ← k1{f}
           zs ← b0[f,ys]
           Cons(z,zs)
```
(c)

```
b2[f,xs] = assert xs Cons
           y  ← Cons 0 xs
           ys ← Cons 1 xs
           z  ← f @ y
           zs ← b0[f,ys]
           Cons(z,zs)
```
(d)

Figure 2: An example illustrating the optimization of known closures (Rule (17))

a canonical form that enables subsequent optimizations. In this case, the rewrite ensures that, for an and with one constant and one unknown argument, the constant is always the second argument. This explains, for example, why we do not need a variant of Rule (9) for tails of the form `and((0,v))`, and also why we do not need four distinct variations of the pattern in Rule (13) where an unknown `u` is combined via bitwise ands with two constants `M` and `N`. In this case, we rely on facts produced by the dataflow analysis to determine that the value of `v` in `and((v,N))` was calculated using a prior `and((u,M))` call. The rewrite here replaces one and with another, so it may not result in an immediate program optimization. However, there are two ways in which this *might* open up opportunities for subsequent rewrites: One possibility is that `M&N` may be zero, in which case we will be able to eliminate the and using Rule (9). Another possibility is that, by rewriting the call to and in terms of `u`, we may eliminate all references to `v` and can then eliminate the statement defining `v` as dead code using Rules (6) and (7).

In a similar way, Rules (14) and (15) take advantage of standard distributivity laws to implement more complex rewrites. By using these rules in combination, we can rewrite any expression involving bitwise ands and ors of an unknown `u` with an arbitrary sequence of constants into an expression of the form `(u&M)|N` for some constants `M` and `N`, with exactly one use of each primitive. These rewrites are potentially dangerous because they increase the size of the MIL code, replacing one primitive call on the left of the rewrite with two on the right. In practice, however, these rules often turn the definitions of the variables `v` and `w` that they reference in to dead code that can be eliminated by subsequent rewrites.

5.4 Known Structures

MIL provides case and @ constructs that work with arbitrary data values and closures, respectively. But the general operations are not needed in situations where we are working with known structures. Rule (16), for example, eliminates a conditional jump if the constructor, `C`, that was used to build `v` is already known: we can just make a direct jump using the alternative for `C` (or the default branch if there is no such alternative), and delete all other parts of the original case construct.

In a similar way, Rule (17) can be used to avoid a general closure entry operation when the specific closure is known. To see how this works in practice, consider the example in Figure 2, with the original code for b_2 in our implementation of map (Section 2.11) in Column (a). From the other definitions in this program, we know that map is a reference to the closure $k_0\{\}$, and that $k_0\{\}\ f = k_1\{f\}$. By allowing our dataflow analysis to derive the fact map=$k_0\{\}$ from its top-level definition, we can use Rule (17) to rewrite the tail defining m, as shown in Column (b). After this transformation, the local dataflow analysis will reach the definition of zs with a list of facts that includes m=$k_1\{f\}$, and so we can apply Rule (17) again to obtain the code in Column (c). This removes the only reference to m, and allows subsequent uses of Rules (6) and (7) will eliminate its definition, producing the code definition in Column (d). This example shows that the original implementation of map would have allocated a fresh closure, $k_1\{f\}$, for every element of the input list. The transformations we have applied here, however, eliminate this overhead and substitute a more efficient, direct recursive call.

5.5 Derived Blocks

Optimizing compilers often use collections of basic blocks, connected together in control flow graphs, as a representation for programs. Most of the rules that we have described so far are traditionally considered local optimizations because they only consider the code within a single block. While much can be accomplished using local optimizations, it is also useful to take a more global view of the program, and to use optimizations that span multiple blocks. In other words, in addition to the *content* of individual blocks, we would also like to account for the *context* in which they appear.

One interesting way that we have been able to handle this in MIL is by using the code of existing blocks to generate new versions—which we refer to as *derived blocks*—that are specialized for use in a particular context. The final group of rewrites in Figure 1 corresponds to different strategies for generating derived blocks that we have found to be effective in the optimization of MIL programs. Of course, adding new blocks to a program increases program size and does not immediately provide an optimization. But, in practice, the addition of new derived blocks often opens new opportunities for optimization—by bringing the construction and matching of a data value into the same block, for example—and the original source block often becomes dead code that will be removed from the program once any specialized versions have been generated.

Rule (18) illustrates one way of using derived blocks to take advantage of information produced by our local dataflow analysis and to obtain results that typically require a global analysis. The techniques that we describe here can be applied very broadly, but, for this paper, we restrict ourselves to a special case: a call to a block `b` that has just one parameter and a very simple definition.

In this situation, if the argument, `v`, to `b` is known to have been constructed using a tail `C(y)`, then we can replace the call `b[v]` with a call of the form `b'[y]`. Here, `b'` is a new block that begins with a statement that recomputes `v` ← `C(y)` and then proceeds in the same way as the original block `b`. In theory, this could result in an 'optimized' program that actually allocates twice as many `C(y)` objects as necessary; that would obviously not be a good outcome. In practice, however, this transformation often enables subsequent optimizations both in the place where the original `b[v]` call appeared (for example, the statement that initialized `v` may now be dead code) and in the new block, the latter resulting from a new fact, `v=C(y)`, that can be propagated through the code for `b'`. Note that Rule (18) also extends naturally to calls with multiple parameters and to cases where one or more of those parameters is a known closure; in that case the process of generating a new derived block has much the same effect as specializing a higher-order function to a known function argument.

Rules (19) and (20) deal with situations where a block is called and its result is immediately used as a closure or data value, respectively. The code on the left side of these rewrites essentially *forces* the allocation of a closure or data object in `b`, just so that value can be returned and then, most likely, discarded after one use. The right sides deal with this by introducing a tail call in the caller and then turning the use of whatever value is produced in to a 'trailing' action in the new block. As in other examples, this does not produce an immediate optimization. However, these rules generally lead to useful improvements in practice, enabling new optimizations by bringing the construction and use of `v` in to the same context.

5.6 Additional Optimizations

Beyond the rewrites kind described in previous sections, our optimizer implements several other program transformations that help to improve code quality. One of the most important of these in practice—because it also performs a strongly-connected components analysis on the program to prioritize the order in which rewrites are applied—is a "tree-shaking" analysis. This automatically removes definitions from a program if they are not reachable from the program's entry points. In addition to sections of library code that are not used in a given application, this also helps to clean up after other optimizations by eliminating single-use blocks whose definitions have been inlined or blocks that have been replaced with new derived versions. The optimizer also attempts to recognize and merge duplicated definitions, and to rewrite block definitions (and all corresponding uses) to eliminate unused parameters. Unused stored fields in closure definitions can also be eliminated in this way, but we cannot remove arguments in closure definitions: even if they are not used in the code for a particular closure, they must be retained for compatibility with other closures of the same type.

One other detail in our implementation is that we run the MIL type checker after every use of the optimizer. This has two practical benefits: (1) All of our optimizations are required to preserve typing, so running the type checker provides a quick sanity check and may help to detect errors in the optimizer. (2) The type checker will automatically reconstruct type information for each part of the program, so we do not need to deal with those details in the implementations of individual rewrites.

5.7 Reflections on Optimization

The design of an optimizer compiler inevitably requires some judicious compromises. After all, the problem that it is trying to solve—to generate truly optimal versions of any input program—is uncomputable, and so, at some point, it must rely instead on heuristics and incomplete strategies. Even if an optimizer delivers good results on a large set of programs, there is still a possibility that it will perform poorly on others. Subtle interactions between different optimization techniques may prevent the use of key transformations in some situations and instead lead to expanded code size or degraded performance. With those caveats in mind, we have, so far, been very satisfied with the performance of our optimizer for MIL.

As a concrete example, the diagrams in Figure 3 outline the structure of a MIL implementation of the Habit "prioset" example [15, Section 5]. Part (a) here is for the original MIL implementation, generated directly from 56 lines of LC code; it is too small to be readable, but does convey that the original program—910 lines of MIL code—is quite complex. Part (b) shows the result obtained after 1,217 separate rewrite steps in the MIL optimizer, resulting in 140 lines of MIL code. This version of the program still has non-trivial control flow, but its overall structure is much simpler and we can start to see details such as loop structures and a distinction between the blue nodes (representing blocks) and the red nodes (representing closure definitions). By comparison, there are red and blue nodes scattered throughout the diagram in Part (a), which suggests that our optimizations have been effective in eliminating many uses of closures in the original program. (The remaining red nodes in Part (b) are only there because the program exports the definitions of two functions, `insertPriority` and `removePriority` as first-class values; in a program that only uses fully-applied calls to these functions, even those nodes would be eliminated.)

The results that we see with this example are representative of our experience across a range of test programs, and suggest that the MIL optimizer can work will in practice. That said, we plan to do more extensive studies, including performance benchmarking, and to use those results to further tune and refine our implementation. (As a new language, we do not currently have a pre-existing set of benchmarks, but we do hope to grow such a library as our implementation matures and gains users.)

One particularly effective aspect of our implementation that is already clear is the decision to perform optimization at multiple levels throughout the compiler pipeline. An initial use of the MIL optimizer takes care of relatively high-level rewrites, such as inlining of higher-order functions to eliminate the costs of constructing closures. It would be harder to apply this kind of optimization at later stages once the operations for function application and closure construction have been decomposed in to sequences of lower level instructions. A subsequent use of the optimizer, after representation transformations have been applied, enables the compiler to find newly exposed opportunities for optimization, and to further simplify the generated MIL code before it is translated in to LLVM, as described in the next section. Finally, the use of LLVM itself allows us to exploit the considerable effort that has been invested in that platform to perform lower-level optimizations, and—although our focus to date has been on IA32-based systems—also provides a path for targeting other architectures.

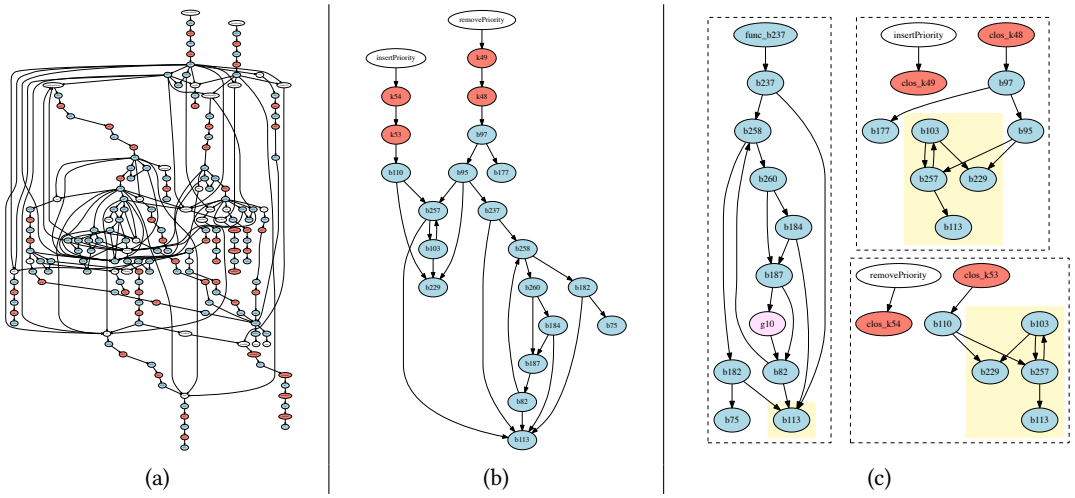


Figure 3: Control flow graph examples.

6 COMPILING MIL TO LLVM

In this section, we explain how MIL programs can be translated in to corresponding LLVM programs, which can then be subjected to further optimization and used to generate executable binaries. Beyond the specific practical role that it serves in our Habit compiler, this also provides another test for MIL's suitability as an intermediate language: it is important, not only that we are able to generate executable programs from our IL, but also that we are able to do so without introducing overhead or undoing any of the improvements that were made as a result of optimizations on the IL.

6.1 Translating MIL Types to LLVM

Before generating LLVM code, we use representation transformations to provide `Word`-based implementations for bitdata types (Section 4.2) and to eliminate polymorphism and parameterized datatypes (Section 4.3). In the resulting programs, MIL types like `Word` and `Flag` are easily mapped to LLVM types such as `i32` and `i1`. The only types that require special attention are for functions (which we cover in this section) and algebraic datatypes (which are handled in a similar manner).

Every MIL value of type $[d_1, \ldots, d_m] \twoheadrightarrow [r_1, \ldots, r_n]$ will be a closure that can be represented by a block of memory that includes a code pointer and provides space, as needed, for stored fields:

entry	···

We can describe structures of this form using three LLVM types with mutually recursive definitions:

```
%clo = type { %fun }
%fun = type {r1, ..., rn} (%ptr, d1, ..., dm)
%ptr = type %clo*
```

Here, `%clo` is a structure type that describes the layout of the closure. Its only component is the code pointer of type `%fun`: no additional components are listed because the number and type of fields is a property of individual closure definitions, not the associated function type. In the generated code, functions of this type will be represented by pointers of type `%ptr`. Given such a pointer, the implementation can read the code pointer from the start of the closure and invoke the function, passing in the closure pointer and the argument values corresponding to the domain types d_i. If that function needs access to stored fields, then it can cast the `%ptr` value to a more specific type that reflects the full layout for that specific type of closure. Finally, the function can return a new structure containing values for each of the range types r_j. (If there is only one result, then it can be returned directly, without a structure; if there are no results at all, then we can use a void function.)

The techniques described here are standard, but there are still many details to account for. Among other things, this reinforces the importance of performing closure optimizations in MIL, rather than generating LLVM code directly and then hoping, unrealistically, that the LLVM tools will be able to detect the same opportunities for improvement. Instead, we divide the responsibilities for optimization between MIL and LLVM, with each part making important contributions to the overall quality of generated code.

6.2 Translating MIL Code to LLVM

The process of translating MIL statements to LLVM instructions is relatively straightforward. As examples: an `and` primitive in MIL maps directly to the (identically named) `and` instruction in LLVM; a closure allocation in MIL is implemented by a call to a runtime library function to allocate space for the closure, followed by a sequence of `store` instructions to initialize its fields; and so on. As such, we will not discuss the fine details of this translation here.

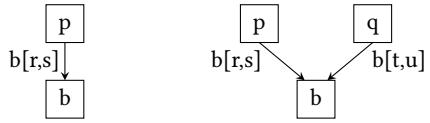
There are, however, some key, higher-level structural mismatches between MIL and LLVM—specifically, parameterization and sharing of blocks—that do need to be addressed. As we have seen, MIL programs are collections of (parameterized) basic blocks that are connected together either by regular block calls or tail calls (in the middle, or at the end, respectively, of a code sequence). By contrast, LLVM programs consist of a collection of (parameterized) functions, each of which has a *control flow graph* (CFG) comprising a single (parameterized) entry point, and a body that is made up from a

collection of (unparameterized) basic blocks. How then should we approach the translation of an arbitrary MIL programs into LLVM?

One approach would be to generate an separate LLVM function for each MIL block (and closure definition). Although LLVM does provide support for tail calls, those features are difficult to use and it might be difficult to ensure that the compiled loops, encoded as tail recursive blocks in MIL, run in constant space.

Our strategy instead is to compile mutually tail recursive blocks directly into loops, accepting that there will be some (small) duplication of blocks in the process. In Figure 3(b), for example, there are several blocks that are reachable from either of the two distinct entry points at the top of the diagram. Our code generator uses some simple heuristics to generate a set of LLVM CFG structures from MIL programs with the following properties: (1) There must be a distinct CFG for every closure definition and for every block that is a program entry point or the target of a non-tail call; (2) If one block is included in a CFG, then all other blocks in the same strongly connected component (SCC) should also be included in the same CFG (this ensures that tail calls can be compiled to jumps); (3) If a single block has multiple entry points from outside its SCC, then it is a candidate entry point for a new CFG (this attempts to reduce duplication of blocks). The result of applying our algorithm to this particular example is shown in Figure 3(c) and is typical of the behavior we see in general: there is some duplication of blocks (the portions highlighted with a yellow background) but the amount of duplicated code is small and has not been a concern in practice.

Our second challenge is in dealing with the mapping from parameterized blocks in MIL to unparameterized blocks in LLVM. It turns out that the number of predecessors is key in determining how to generate code for the body of each block. To understand this, consider the following two diagrams:



In both diagrams, we assume a block `b` defined by `b[x,y] = c` for some code sequence `c`. For the diagram on the left, there is exactly one predecessor, `p`, which ends with a call to `b[r,s]`. In this situation, there is actually no need for the parameters to `b` because we already know what values they will take at the only point where `b` is called. All that it needed is to apply a substitution, replacing the formal parameters, `x` and `y`, with the actual parameters `r` and `s`, respectively, so that we use the code sequence `[r/x, s/y]c` for the body of `b`. (Of course, we also need to account for this substitution on any edges from `b` to its successors.) For the diagram on the right, there are two predecessors, each of which ends by calling `b` with (potentially distinct) parameters. In this case, the phi functions that are part of LLVM's SSA representation provide exactly the functionality that we need to 'merge' the incoming parameters and we can generate code of the following form for `b`:

```
x = phi [r,p], [t,q]
y = phi [s,p], [u,q]
... LLVM code for c goes here ...
```

Generating code in SSA form is sometimes considered to be a tricky or complicated step in the construction of an optimizing compiler. It is fortunate that the translation is relatively straightforward and that we are able to take advantage of phi functions—a key characteristic of the SSA form—quite so directly.

7 CONCLUSIONS

In this paper, we have described the MIL language and toolset, demonstrating (1) that it has the fundamental characteristics needed to qualify as an effective intermediate language for the compilation of functional languages; and (2) that it can enable new techniques for choosing efficient data representations. The design and implementation of any new intermediate language requires considerable engineering effort. As we continue to develop the MIL system ourselves—for example, to explore its potential for compilation of lazy languages—we hope that it will also serve as useful infrastructure for other functional language implementors.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful feedback. This work was supported in part by funding from the National Science Foundation, Award No. CNS-1422979.

REFERENCES

[1] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA.
[2] Justin Bailey. 2012. *Using Dataflow Optimization Techniques with a Monadic Intermediate Language.* Master's thesis. Department of Computer Science, Portland State University, Portland, OR.
[3] Adam Chlipala. 2015. An Optimizing Compiler for a Purely Functional Web-application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015).* ACM, New York, NY, USA.
[4] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. 2005. High-level views on low-level representations. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005.* ACM, 168–179.
[5] Matthew Fluet and Stephen Weeks. 2001. Contification Using Dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01).* ACM, New York, NY, USA, 2–13.
[6] Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture (Lecture Notes in Computer Science, 201).* Springer-Verlag, 190–203.
[7] Mark P. Jones. 1994. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94).*
[8] Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).* ACM, New York, NY, USA, 177–190.
[9] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[10] LLVM 2018. The LLVM Compiler Infrastructure. http://llvm.org.
[11] Luke Maurer, Zena Ariola, Paul Downen, and Simon Peyton Jones. 2017. Compiling without continuations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'17).* ACM, 482–494.
[12] E. Moggi. 1989. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science.* IEEE Press, Piscataway, NJ, USA, 14–23.
[13] Alan Mycroft. 1984. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming.* Springer-Verlag, London, UK, UK, 217–228.
[14] Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press.
[15] The Hasp Project. 2010. The Habit Programming Language: The Revised Preliminary Report. http://github.com/habit-lang/language-report.
[16] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90).* 61–78.
[17] Stephen Weeks. 2006. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML '06).* ACM, New York, NY, USA.