MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age

ROLAND LEISSA, University of Mannheim, Germany
MARCEL ULLRICH, Saarland University, Germany
JOACHIM MEYER, Saarland University, Germany
SEBASTIAN HACK, Saarland University, Germany

Traditional compilers, designed for optimizing low-level code, fall short when dealing with modern, computation-heavy applications like image processing, machine learning, or numerical simulations. Optimizations should understand the primitive operations of the specific application domain and thus happen on that level.

Domain-specific languages (DSLs) fulfill these requirements. However, DSL compilers reinvent the wheel over and over again as standard optimizations, code generators, and general infrastructure & boilerplate code must be reimplemented for each DSL compiler.

This paper presents MimIR, an extensible, higher-order intermediate representation. At its core, MimIR is a pure type system and, hence, a form of a typed lambda calculus. Developers can declare the signatures of new (domain-specific) operations, called *axioms*. An axiom can be the declaration of a function, a type constructor, or any other entity with a possibly polymorphic, polytypic, and/or dependent type. This way, developers can extend MimIR at any low or high level and bundle them in a *plugin*. Plugins extend the compiler and take care of optimizing and lowering the plugins' axioms.

We show the expressiveness and effectiveness of MimIR in three case studies: Low-level plugins that operate at the same level of abstraction as LLVM, a regular-expression matching plugin, and plugins for linear algebra and automatic differentiation. We show that in all three studies, MimIR produces code that has state-of-the-art performance.

CCS Concepts: • **Software and its engineering → Compilers**; *Semantics*; *Domain specific languages*.

Additional Key Words and Phrases: compiler intermediate representations, lambda-calculus, dependent types

ACM Reference Format:
Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2025. MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age. *Proc. ACM Program. Lang.* 9, POPL, Article 4 (January 2025), 31 pages. https://doi.org/10.1145/3704840

1 Introduction

Dennard scaling enabled the continuous growth of single-thread performance of legacy code for decades. After its decline in the early 2000s, domain-specific languages (DSLs) have received a lot of attention in the effort to harvest as much performance as possible in a productive way. DSLs offer specialized abstractions tailored to specific problem domains enhancing the programmer's productivity on one side and enabling the generation of high-performance code on the other side. To generate actual code, DSL compilers typically resort to existing compiler frameworks such as

Authors' Contact Information: Roland Leißa, University of Mannheim, Mannheim, Germany, leissa@uni-mannheim.de; Marcel Ullrich, Saarland University, Saarbrücken, Germany, ullrich@cs.uni-saarland.de; Joachim Meyer, Saarland University, Saarbrücken, Germany, jmeyer@cs.uni-saarland.de; Sebastian Hack, Saarland University, Saarbrücken, Germany, hack@cs.uni-saarland.de.

ACM 2475-1421/2025/1-ART4
https://doi.org/10.1145/3704840

LLVM [32]. However, there is a significant gap in the abstractions DSLs provide and the low-level nature of these compiler frameworks which often leads to the situation that DSL compilers have an additional, more high-level intermediate representation (IR) that bridges the gap between the DSL and the back-end compiler.

This more high-level IR is often designed and implemented for each new DSL from scratch. Current research in high-level IRs, most notably MLIR [33], seeks to provide a least common denominator for the basis of such an IR to facilitate the reuse of core data structures and algorithms across different DSLs but not much more. While MLIR provides extensibility to host domain-specific *dialects*, it is still low-level in several aspects. First, it relies on control-flow graphs (CFGs), thereby relying on multiple concepts to represent control flow including functions, basic blocks, instructions, and so-called regions to delineate code for high-level transformations. Second, MLIR does not provide a type system that is expressive enough to host the type systems of DSLs. It relies on the dialects to implement their own type systems in C++. This causes more implementation effort for the DSL implementer and an unclear notion of well-typedness when multiple dialects interact. Finally, because MLIR is designed with "little builtin, everything customizable" [33, p. 3], it is *by design* impossible to give a formal account of its static and dynamic semantics that would encompass all DSLs.

In this paper, we want to go one step further and present MimIR, a higher-order IR that provides a type system that sets out to be expressive enough to host a wide range of DSLs. Consider the following DSL interface for a *tensor* plugin in MimIR's *surface language* Mim—the textual interface of MimIR:

```
axm %tensor.Mat: {n m: Nat, T: *} → *; // n × m matrix with element type T
axm %tensor.zip: {n m: Nat, T: *} [f: [T, T] → T] [a b: %tensor.Mat (n, m, T)]
                                              → %tensor.Mat (n, m, T);
```

The so-called *axiom* `%tensor.Mat` is a type constructor that expects two dimensions `n` and `m` and an element type `T` to construct a matrix. The `%tensor.zip` axiom is polymorphic in the arguments' dimensions and element type (whose values will be inferred at a call-site as the curly braces mark this parameter group as *implicit*) and computes a new matrix by applying `f` in pairs to all elements of both inputs. Axioms are declarations of entities that do not have an implementation in MimIR per se. Instead, DSL designers define a *plugin* that consists of an interface (see above) and a C++ implementation that provides domain-specific program transformations and code generators to lower the axioms into the language of lower-level MimIR plugins or to generate target code directly.

IRs like LLVM are not flexible enough to express types like `%tensor.Mat` at all. In MLIR, the designers of plugins (there called dialects) need to manually implement the type system in C++ for each dialect operation. But even this has many severe limitations as MLIR does neither directly support higher-order functions nor polymorphism. For example, `vector.reduction` from the MLIR *vector* dialect hard-codes a set of predefined reduction operations. In MimIR we can define a `reduce` function that works on any array and appropriate function. For example, here we create a matrix addition with `%tensor.zip elem_add` that we use to reduce an `array_of_matrices` to a single `res_mat`rix:

```
let res_mat = reduce (%tensor.zip elem_add) (init, array_of_matrices);
```

Note that `%tensor.Mat`'s type is an ordinary function type and, yet, it is a type operator. Furthermore, note that the types of `%tensor.zip`'s arguments `a` and `b` depend on the type variable `T` (polymorphism) and the term variables `n` and `m` which makes `%tensor.Mat (n, m, T)` a *dependent type*. This is possible because at its core MimIR is a minimal, typed λ-calculus that is based upon the λ-cube [7], the Calculus of Constructions (CC) [16], and pure type system (PTS) [8]. This makes MimIR not only very expressive but also simplifies the compiler design in many ways because

MimIR only knows a single syntactic category: expressions. However, Mim supports syntactic sugar that Mim translates into its minimal, graph-based representation MimIR.

Contributions. In summary, this paper makes the following contributions:

- We introduce the intermediate representation MimIR that allows, through its plugin architecture, to modularly extend the MimIR compiler with custom, domain-specific operations, types, and type operators, as well as domain-specific code transformations—in particular so-called *normalizations* that are eagerly applied by MimIR (Section 2).
- We formally introduce MimIR's semantics and its sound type system which allows for a type-safe composition of plugins. MimIR's expressive type system is rooted in the Calculus of Constructions and, hence, features dependent-types. These are mainly used in proof assistants and associated with writing complex proof terms. MimIR, however, features full recursive functions and dependent types without the hassle of writing complex proof expressions. This is possible because MimIR's type checker tightly interacts with normalizations and a partial evaluator which allows for a novel way to type-check dependent types (Section 3).
- Code transformations are implemented in C++ by manipulating MimIR's program graph. This is a "sea of nodes"-style [13] IR for a higher-order, PTS-style language. As MimIR only knows expressions, the program graph is also very simple: Each node represents an expression and has outgoing edges to each of its operands and the type it inhabits—which is again an expression. Besides an internal hash set that hash-conses all nodes, MimIR does *not* need any other auxiliary data structures such as instruction lists, basic blocks, CFGs, or special regions (Section 4).
- MimIR performs type checking, inference, and normalization on-the-fly *during program construction*. This way, a plugin/compiler developer can resort to type inference, normalization, and (partial) evaluation upon creating an expression—regardless of whether they use Mim or MimIR's C++ interface. MimIR will immediately report any typing errors (Section 5).
- Section 6 presents case studies that show MimIR's versatility and ability to host DSLs and generate high-performance code. These include a set of low-level, LLVM-like plugins, that we show to perform just as well as if directly using LLVM, a plugin for matching regular expressions (RegExes) which outperforms other popular RegEx engines, a plugin for tensor computations, and a plugin for automatic differentiation with state-of-the-art performance at a tenth of the complexity.

2 Overview

The core of MimIR is a minimal, typed λ-calculus with strong semantics based on the λ-cube, PTS, and CC. We describe the core calculus' syntax and semantics in more detail in Section 3. What makes MimIR an attractive target for DSLs is its extensibility through plugins. These plugins can define intrinsic operations (in MimIR called "axioms") on various abstraction levels. Furthermore, plugins can provide transformations and lowering passes to perform (domain-specific) optimizations on each abstraction level and to convert between the levels. There are several ways how a DSL or a general-purpose language can target MimIR. The most important ones are: an embedded DSL or a DSL compiler may use MimIR's API to construct the IR. Alternatively, a compiler can communicate with MimIR textually through its surface language Mim. MimIR itself comes with a backend that emits textual LLVM IR. This IR can be further processed by LLVM tools to obtain an executable. Thereby, MimIR currently targets any CPU supported by LLVM. Since plugins are essential to MimIR's design, we will use the example of a plugin for matching RegExes to present MimIR's components.

```
let Char             = I8;                                                            1
lam Str (n: Nat): * = %mem.Ptr «n; Char»;                                            2
lam Res (n: Nat): * = [%mem.M, Bool, Idx n];                                         3
let RegEx            = {n: Nat} [%mem.M, Str n, Idx n] → Res n;                      4
                                                                                      5
axm %regex.any:                                RegEx;                                6
axm %regex.conj:      {i: Nat} [«i; RegEx»] → RegEx, normalize_conj,   2;            7
axm %regex.disj:      {i: Nat} [«i; RegEx»] → RegEx, normalize_disj,   2;            8
axm %regex.range:                  «2; Char» → RegEx, normalize_range, 1;            9
axm %regex.not:                       RegEx → RegEx, normalize_not,    1;            10
axm %regex.quant(optional,star,plus): RegEx → RegEx, normalize_quant, 1;             11
                                                                                      12
lam %regex.lit(val: Char) = %regex.range (val, val);                                 13
let %regex.cls.d          = %regex.range ('0', '9'); // similar: %regex.cls.w, %regex.cls.W, 14
let %regex.cls.D          = %regex.not %regex.cls.d; //          %regex.cls.s, %regex.cls.S  15
```

Listing 1. RegEx plugin declaration file `regex.mim`

```
plugin regex; // parse "regex.mim" and load "libmim.so"
let pattern = %regex.conj (%regex.quant.plus %regex.cls.w,          // '\w+\.[a-z]+'
                           %regex.lit '.', %regex.quant.plus (%regex.range ('a','z')));
```

Listing 2. MIM code that constructs RegEx pattern `^\w+\.[a-z]+$` (simple top-level domains)

```
if (auto star_outer = match(regex::quant::star, r))
    if (auto star_inner = match(regex::quant::star, star_outer->arg())) return star_inner;
```

Listing 3. C++ code that matches $r{*}{*}$ and yields $r{*}$

```
world.call<regex::range>(Defs{a, b}, Defs{mem, str, pos})
```

Listing 4. C++ code that constructs a curried call `%regex.range (a, b) (mem, str, pos)`

2.1 Plugin Architecture

A plugin consists of two parts: A `*.mim` file that contains declarations in MIM of what the plugin exports and code transformations implemented in C++. The `%regex` plugin declares its operations in `regex.mim` (Listing 1). All names prefixed with `%regex` are called *annexes*. Before building the plugin's C++ sources, MIMIR *bootstraps* the plugin by parsing `regex.mim` and generating a C++ header file that declares all annexes as C++ **`enum`**s. This process does *not* involve a "compilation" of the annexes to C++ in any way. The purpose of this is that the C++ part of the plugin can reference an annex via a C++ name instead of a string (see below). Then, the build system compiles the plugin's sources with the help of the generated header into a shared object `libmim_regex.so`. Now, other MIMIR code can use the plugin (see Listing 2): The **`plugin`** `regex` directive instructs MIMIR to parse `regex.mim`. In doing so, MIMIR will know the names of all `%regex` annexes and their types. In addition, MIMIR will dynamically load `libmim_regex.so` to incorporate the plugin's code transformations into the MIMIR compiler. Now that the plugin has been loaded, the **`let`**-expression binds the variable `pattern` to a MIMIR expression that represents the RegEx `^\w+\.[a-z]+$` (which matches simple top-level domains).

2.2 Plugin Declaration

The `%regex` plugin declares the so-called *axioms* `%regexconj`, `%regex.disj`, etc. Line 11 declares the RegEx quantifiers `%regex.quant.optional`, `%regex.quant.star`, `%regex.quant.plus`. In addition to axioms, plugins can provide supplementary definitions (variables, functions, ...) that are entirely defined in MIM. For example, the digit character class `%regex.cls.d` is just **`let`**-bound to `%regex.range ('0', '9')`. Axioms that inhabit a function type, are usable like ordinary functions. However, axioms do *not* provide an implementation in MIMIR per se. Their purpose is to denote domain-specific language constructs that the plugin's code transformations refine. For example, the C++ code in Listing 3 matches `%regex.quant.star (%regex.quant.star regex)` and peels off the superfluous outer quantifier. Note that `regex::quant::star` stems from the auto-generated header and references `%regex.quant.star` from `regex.mim`. Similarly, plugin developers can access or call annexes from C++ (Listing 4).

2.3 Types

One of MimIR's most prominent features is that it does *not* have a special syntactic category for types. Instead, types are also expressions. This has several advantages as we will outline in the following.

First, MimIR does not need special constructs to declare type aliases or type-level functions. An ordinary **`let`**-binding (line 1 in Listing 1) defines the alias `Char` for the 8-bit-wide integer type **`I8`**. `Str` is also a normal function. It expects a size `n` of type **`Nat`** and returns `*`: the type of all types. Thus, the type of `Str` is **`Nat`** `→ *`. Here, `Str` yields a pointer to an array of size `n` and element type `Char`. The pointer type constructor is an axiom from the `%mem` plugin: `axm %mem.Ptr: * → *` (Section 6.1.3). The *expression* «e_n; T» introduces an array *type* while both e_n and `T` are again expressions. We just use the metavariable `T` to suggest that this is a type expression. In particular, the size e_n can be an arbitrarily complex expression and may not necessarily be a compile-time constant as opposed to `n` in C++'s `std::array<T, n>`. For this reason, «e_n; T» is called a *dependent type*, as the type depends on a value. A common misconception is to think of «e_n; T» as a pair consisting of the size e_n and the "actual" array. This is *not* the case. It is just an array which "tracks" its size e_n. Dependent types are mostly known from theorem provers such as Coq or Lean. However, in MimIR we are not concerned about proofs. We are using dependent types to abstract from the size of integer operations or arrays which in turn allows for type-safe variadic functions and polymorphism over array rank while tracking this dependency in the type system (Section 6).

The expression $(e_0, \ldots, e_{n-1})$ forms a tuple *value* while $[T_0, \ldots, T_{n-1}]$ forms a tuple *type*. Hence, the function `Res` returns a tuple *type* which models the result type of a RegEx match. The first element type `%mem.M` stems again from the `%mem` plugin and abstracts from the machine state; any operation that potentially has a side-effect such as memory accesses consumes a machine state, i.e. a value of type `%mem.M`, and produces a new one. This is similar to the `IO` monad in Haskell. The second element type **`Bool`** indicates the success or failure of a match. The last element type **`Idx`** `n` is an integer within the range $0_n, \ldots, (n-1)_n$ and keeps track of the current position within the string to match. Note that this index type guarantees to access the string in bounds.

The type `RegEx` of a RegEx matcher is a *dependent* function type. Mim allows for convenient specification of curried function types with complex dependencies via `d ··· d → T`: Each domain `d` constitutes a curried domain and may as well as the final codomain `T` depend on the *value* of the preceding domains. Here, the second domain and the codomain `Res n` depend on `n` which is a *value* of the first domain **`Nat`**. This dependency makes the first domain deducible when calling a function of type `RegEx`. This is why first the domain is put within curly braces which marks it as *implicit*: MimIR will automatically infer this argument when calling a function of type `RegEx`. The second domain constitutes the "actual" parameters of the matcher: a machine state, a string, and the current position within this string.

The `RegEx` constructors yield matchers of type `RegEx` while potentially composing other matchers. For example, `%regex.not` expects another matcher to negate and `%regex.range` a range given by two `Char`s as showcased in Listing 4: It creates a range pattern and matches it on a `str`ing. Note that the size argument `n` is implicit and inferred in both MimIR's textual representation as well as the C++ code. The junctions `%regex.conj` and `%regex.disj` demonstrate how dependent arrays allow for variadic functions as they expect `i`-many `RegEx` matchers as inputs.

The `%regex` plugin does not have to provide any kind of additional validators as would be necessary in MLIR. All declared annexes have a proper type and MimIR type-checks expressions containing annexes just like any other expression.

2.4 Normalization

Whenever MimIR creates an expression, it is immediately *normalized* (Section 3.1.1). For example, the tuple extraction `(0, 1, 2)#2`$_3$ is right away resolved to `2`. In addition, MimIR consistently removes 1-tuples and 1-tuple types. Hence, it does not matter whether a function is specified as **`lam`** `Str(n:` **`Nat`**`)` or **`lam`** `Str n:` **`Nat`** and whether it is invoked with `Str n` or `Str (n)`.

Normalizations are the backbone of MimIR's optimizer but also influence type checking. MimIR handles tuples and arrays in a uniform way by normalizing a tuple type `[`**`Nat`**`,` **`Nat`**`]` to an array `«2;` **`Nat`**`»`. This is why MimIR does not need different introduction and elimination constructs for tuples and arrays: The type of `(0, 1)` is `«2;` **`Nat`**`»` and of `(0,` **`ff`**`)` is `[`**`Nat`**`,` **`Bool`**`]`; extraction `e#e`$_i$ works regardless of whether `e` is an array or a tuple (Section 3.1.3). Thus, `%regexconj (re1, re2, re3)` types fine: MimIR infers `3` for `i`, types the argument as `«3; RegEx»`, and removes superfluous 1-tuple types from `%regex.conj`'s domains. Additionally, less syntax also implies fewer patterns to match when writing program analyses.

Finally, all expressions are *hash-consed*: Whenever an expression is created, MimIR first checks whether a syntactically equal expression already exists. If this is the case, MimIR will reuse this existing expression. This has the effect that in the C++ implementation two pointers to MimIR expressions enjoy pointer equality if they are syntactically equal.

Axioms can provide their own *normalizers*: local transformations that are considered generally useful. The `%regex` plugin, for instance, merges quantifiers—`r*?`, `r?*`, `r+?`, `r?+`, `r*+`, `r+*` all normalize to `r*`—and removes idempotence—`r??` results in `r?`, ditto for `*` and `+`. As the axiom declaration indicates (line 11), the C++ function `normalize_quant`, which is part of `libmim_regex.so`, is the *normalizer* of this axiom and implements this logic. The actual implementation consists of ~20 lines of C++ code and involves a few matches and building new calls similar to Listing 3 and 4. The `%regex` plugin implements similar normalizations for the other axioms to compute a (pseudo) normal form for RegExes.

By default, MimIR fires the specified normalizer when the last curried argument is applied to an axiom and this is in most cases the desired behavior. However, plugin designers can override this behavior and specify when exactly normalization should happen. The `%regex` plugin is an exception to the default behavior as it wants to normalize, for example, a quantifier as soon as its matcher is applied: `%regex.quant.star regex`. If we waited until all curried arguments were passed, we would entirely miss normalization in some instances or be too late for others:

```
%regex.quant.star (%regex.quant.star regex /*miss*/) (mem, str, pos) /*too late*/
```

To this end, the axiom demands normalization when the first argument is passed to the axiom (last part of line 11). The same counting mechanism applies when `match`ing curried axiom calls in C++ and is the reason why Listing 3 works as intended. Other plugins implement constant folding and various peephole optimizations such as `x+0` ▹ `x` via normalizers.

2.5 Lowering

Since axioms are opaque entities without implementation, plugins must somehow provide an implementation. The `%mem` plugin contains an LLVM backend that additionally understands the `%core` plugin (for integer operations) and `%math` plugin (for floating-point operations). Unless other plugins want to ship their own or extend the existing LLVM backend, they need a phase that substitutes axioms unknown to the LLVM backend to low-level code known to the backend (i.e., only using operations from `%mem`, `%core`, and `%math`). What is more, many plugins want to apply domain-specific transformations on the code in addition to normalizations before lowering its domain-specific axioms. To this end, MimIR provides a sophisticated optimizer and a flexible, modular pass manager. Even compilation phases are exposed as axioms with the help of the

`%compile` plugin. This allows users to compose their own compilation pipeline as a MimIR program. However, the details of the optimizer are beyond the scope of this paper.

As outlined above, the `%regex` plugin applies various normalizations to given RegExes. Additionally, the plugin provides a pass written in C++ that constructs a nondeterministic finite automaton (NFA) from a pattern, makes it deterministic, and minimizes it. Finally, the pass will generate low-level code that implements the minimized deterministic finite automaton (DFA) and replaces the axiom calls with it. This code is amenable to the LLVM backend. We discuss the performance of the plugin in Section 6.2.

2.6 Discussion: MimIR vs. Thorin and MLIR

MimIR's code base was initially derived from Thorin [35] but MimIR is now entirely different from its predecessor: Thorin uses continuation-passing style (CPS) for *all* of its functions. MimIR on the other hand, supports both direct style and CPS by giving a continuation the type $T \to \bot$ (see Section 3.1.5). MimIR adds (higher-order) polymorphism and dependent types with local type inference. Thorin has a set of hard-coded, second-class built-ins in direct style (much like instructions in LLVM) that MimIR completely removes. Instead, MimIR provides a plugin system that allows developers to declare not only custom operations but also custom types and type constructors through first-class axioms with a possibly polymorphic, polytypic, and/or dependent type. In addition, MimIR introduces the Mim language, which allows developers to specify the plugin interface or complete programs directly in Mim. MimIR's core calculus is now so different from Thorin's, and MimIR's sources hardly contain any Thorin-derived code anymore, that we have renamed the project rather than bumping Thorin's version number.

While MLIR [33] and MimIR are both extensible compiler frameworks and pursue similar goals, MLIR only provides a basic infrastructure for hosting languages that has "little builtin, everything customizable" [33, p. 3]. MimIR, on the other hand, aims to provide a general base language with an expressive type system. Similarly to MimIR's plugins, MLIR offers extensibility through so-called *dialects*. Dialects enhance MLIR's parser and type checker because MLIR lacks polymorphism or dependent types. This raises the question of how the different type-systems of the individual dialects interact and integrate. There are even dialects such as CIRCT [69] that violate basic SSA invariants (no cycles in the data dependence graph).

Additionally, MLIR has only limited support for higher-order functions: Ops in MLIR do not have a function type and cannot be passed to other functions as first-class citizens. Custom functions cannot have free variables—like in C. Nor are regions true functions as regions do not even have a type per se. They can be syntactically passed to Ops that are specifically designed to expect regions. Then, the C++ validator checks whether the passed region is used correctly in this specific spot. But you cannot capture this closure as value and pass it around in MLIR out of the box. The `lp` dialect [9] uses this feature to implement full closure support but `lp` works on a type-erased representation. For example, the types of all higher-order arguments have been erased to `!lp.t`—a boxed heap value.

3 Semantics

In the following, we first present the formal definition of MimIR including its syntax, semantics, and normalization rules (Figure 1). Mim supports syntactic sugar, which Mim translates into the core syntax (Figure 2). In order to keep this presentation as concise as possible, we leave out a few details and full recursion which we will discuss afterward. Then, we introduce MimIR's partial evaluator that tightly interacts with type checking and normalization. Finally, we present MimIR's type safety.

3.1 Syntax, Typing & Normalization

Preliminaries. Since MimIR is based on PTS, it uses the same syntax for terms, types, and kinds. However, we usually use the metavariable `e` to evoke a term expression and `T` or `U` to evoke a type expression. MimIR uses a stratified, countably infinite hierarchy of sorts which are organized in a predicative way (Rule Sort). This avoids well-known paradoxes (like Girard's paradox) associated with self-referential definitions. As syntactic sugar we write `*` for `Sort 0`: the type of all types. Type constructors such as `* → *` are of type `Sort 1` etc.

MimIR knows several *binders*. These are expressions that introduce a variable of the form `x:e`. We follow *Barendregt's convention*: No variable is both free and bound; every bound variable is bound *exactly once*. While Mim supports lexical scoping, MimIR's graph representation (Section 4) actually ensures Barendregt's convention.

We call a binder *parametric*, if the introduced variable occurs free in any subsequent expression of the binder. Otherwise, we call the binder *non-parametric*. There is syntactic sugar for non-parametric binders available which omits the variable altogether (Figure 2).

3.1.1 Normalization. Whenever MimIR builds an expression, it will immediately *normalize* it according to $\underline{\texttt{e}} \triangleright \texttt{e}$. Normalization rules play not only an important role in MimIR's optimizer but also in its type checker. First, types themselves are normalized. Second, a normalized expression may appear as an argument to another type constructor. In particular in the case of dependent types, checking for type equality requires checking for program equivalence (see also Section 3.3).

Example 3.1. Consider the function:

```
λ (a: «%core.nat.add (0, n); T»): U = body
```

The `%core` plugin (Section 6.1) normalizes the addition and, hence, simplifies the function to:

```
λ (a: «n; T»): U = body
```

This allows a caller to pass a value of type `«n; T»` to this function. This would be ill-typed without normalization.

Most dependently typed languages suffer from artifacts such as `f (n + 0)` types in some context whereas `f (0 + n)` does not. MimIR's extensible normalization framework is able to mitigate such issues.

Example 3.2. The `%core` plugin normalizes both

`f (%core.nat.add (0, n))` and `f (%core.nat.add (n, 0))` to `f n`.

We *underline* an expression $\underline{\texttt{e}}$ to denote that it *might not* be *normalized*. A *non-underlined* expression `e` denotes that it *is* already *normalized*. Non-normalized expressions only exist in Mim which is compiled into MimIR's internal graph form early in the compilation pipeline. Mim translates an expression to MimIR by *first* recursively translating all subexpressions into the desugared, normalized MimIR graph representation, and *then* desugaring the current expression and assembling it via the ▷-relation. Therefore, only normalized expressions exist within MimIR itself. For this reason, the normalization rules do not include premises to normalize subexpressions as MimIR will only need to assemble expressions from subexpressions that *are* already normalized. Mim only performs rudimentary semantic checks such as name analysis. Type checking happens on MimIR's normalized graph representation.

Example 3.3. MimIR will never need to normalize `((e))`. As soon as Mim translates `(e)` to its internal graph, Rule N-Tup$_1$ fires and yields `e`. Then, the outer parentheses form `(e)` again. This will fire N-Tup$_1$ once more and yield `e`.

$$
\begin{array}{rcll}
\Gamma & ::= & \cdot \mid \Gamma, x : T & \textit{Typing Environment} \\
\underline{e}, \underline{T}, \underline{U} & ::= & \textbf{Sort}\ s \mid \bot \mid \textbf{Nat} \mid \textbf{Idx} \mid n \mid i_n & \textit{Sort / Bottom / Nat / Idx / Literal} \\
& \mid & x \mid \textbf{let}\ x = \underline{e};\ \underline{e} \mid \textbf{axm}\ x{:}\underline{T};\ \underline{e} & \textit{Var / Let / Axiom} \qquad i, n, s \in \mathbb{N} \\
& \mid & [x{:}\underline{T}] \to \underline{U} \mid \lambda x{:}\underline{T}@\underline{e}{:}\underline{U} = \underline{e} \mid \underline{e}\ \underline{e} & \textit{Pi / Lam / App} \qquad i < n \\
& \mid & [x{:}\underline{T}] \mid (\underline{e}) \mid «x{:}\underline{e};\ \underline{T}» \mid ‹x{:}\underline{e};\ \underline{e}› \mid \underline{e}\#\underline{e} & \textit{Sigma / Tuple / Array / Pack / Extract}
\end{array}
$$

$$\boxed{e \to e} \qquad \textsc{Cong}\ \frac{e \to e'}{\mathcal{E}[e] \to \mathcal{E}[e']} \qquad \textsc{Beta}\ \frac{}{(\lambda\ x{:}T@e{:}U = e_b)\ e_a \to e_b[e_a/x]}$$

$$
\begin{array}{rcll}
\mathcal{E}[\cdot] & ::= & \textbf{axm}\ x{:}T;\ \cdot \mid \textbf{axm}\ x{:}\cdot;\ e \mid [\ldots,\ \cdot,\ \ldots] \mid (\ldots,\ \cdot,\ \ldots) \mid «x{:}\cdot;\ e» \mid «x{:}e;\ \cdot» \mid ‹x{:}\cdot;\ e› \mid ‹x{:}e;\ \cdot› & \textit{Evaluation} \\
& \mid & \cdot\ e \mid e\ \cdot \mid \cdot\#e \mid e\#\cdot \mid [x{:}\cdot] \to U \mid [x{:}T] \to \cdot \mid \lambda x{:}\cdot@e{:}U = e \mid \lambda x{:}T@e{:}\cdot = e \mid \lambda x{:}T@e{:}U = \cdot & \textit{Context}
\end{array}
$$

$$\boxed{\Gamma \vdash e : T} \qquad \textsc{Sort}\ \frac{n' = n + 1}{\Gamma \vdash \textbf{Sort}\ n : \textbf{Sort}\ n'} \qquad \textsc{Bot}\ \frac{}{\Gamma \vdash \bot : *} \qquad \textsc{Nat}\ \frac{}{\Gamma \vdash \textbf{Nat} : *} \qquad \textsc{Idx}\ \frac{}{\Gamma \vdash \textbf{Idx} : \textbf{Nat} \to *}$$

$$\textsc{Lit-N}\ \frac{}{\Gamma \vdash n : \textbf{Nat}} \qquad \textsc{Lit-I}\ \frac{i < n}{\Gamma \vdash i_n : \textbf{Idx}\ n} \qquad \textsc{Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{Ax}\ \frac{\Gamma \vdash T : \textbf{Sort}\ s \qquad \Gamma, x : T \vdash e : U}{\Gamma \vdash \textbf{axm}\ x\ :\ T;\ e : U}$$

$$\textsc{Pi}\ \frac{\Gamma \vdash T : \textbf{Sort}\ s_t \qquad \Gamma, x : T \vdash U : \textbf{Sort}\ s_u}{\Gamma \vdash [x{:}T] \to U : \textbf{Sort}\ \max\{s_t, s_u\}} \qquad \textsc{Lam}\ \frac{\Gamma, x : T \vdash e_f : \textbf{Bool} \qquad \Gamma, x : T \vdash U \leftarrow e \qquad \Gamma \vdash [x{:}T] \to U : \textbf{Sort}\ s}{\Gamma \vdash \lambda\ x{:}T@e_f{:}\ U = e : [x{:}T] \to U}$$

$$\textsc{App}\ \frac{\Gamma \vdash e : [x{:}T] \to U \qquad \Gamma \vdash T \leftarrow e_T}{\Gamma \vdash e\ e_T : U[e_T/x]} \qquad \textsc{Sig}\ \frac{\Gamma \vdash T_0 : \textbf{Sort}\ s_0 \quad \cdots \quad \Gamma, x_0 : T_0, \ldots, x_{n-2} : T_{n-2} \vdash T_{n-1} : \textbf{Sort}\ s_{n-1}}{\Gamma \vdash [x_0{:}T_0, \ldots, x_{n-1}{:}T_{n-1}] : \textbf{Sort}\ \max\{s_0, \ldots, s_{n-1}\}}$$

$$\textsc{Tup}\ \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \quad \cdots \quad \Gamma \vdash e_{n-1} : T_{n-1} \\ [T_0, \ldots, T_{n-1}] \triangleright T \qquad \Gamma \vdash T : \textbf{Sort}\ s\end{array}}{\Gamma \vdash (e_0, \ldots, e_{n-1}) : T} \qquad \textsc{Arr}\ \frac{\begin{array}{c}\Gamma \vdash e_n : \textbf{Nat} \\ \Gamma, x : \textbf{Idx}\ e_n \vdash T : \textbf{Sort}\ s\end{array}}{\Gamma \vdash «x{:}\ e_n;\ T» : \textbf{Sort}\ s} \qquad \textsc{Pack}\ \frac{\begin{array}{c}\Gamma \vdash e_n : \textbf{Nat} \qquad \Gamma, x : \textbf{Idx}\ e_n \vdash e : T \\ «x{:}\ e_n;\ T» \triangleright U \qquad \Gamma \vdash U : \textbf{Sort}\ s\end{array}}{\Gamma \vdash ‹x{:}\ e_n;\ e› : U}$$

$$\textsc{Ex-S}_L\ \frac{\Gamma \vdash e : [x_0{:}T_0, \ldots, x_{n-1}{:}T_{n-1}] \qquad i < n}{\Gamma \vdash e\#i_n : T_i[e\#0_n/x_0] \cdots [e\#(i\text{-}1)_n/x_{i-1}]}$$

$$\textsc{Ex-S}_i\ \frac{\begin{array}{c}\Gamma \vdash e_i : \textbf{Idx}\ n \qquad \Gamma \vdash e : [x_0{:}T_0, \ldots, x_{n-1}{:}T_{n-1}] \qquad \forall_{1 \le i < n}.\Gamma \vdash T_i : \textbf{Sort}\ s \\ T_j' = T_j[e\#0_n/x_0] \cdots [e\#(j\text{-}1)_n/x_{j-1}] \qquad (T_0', \ldots, T_{n-1}') \triangleright T \qquad T\#e_i \triangleright U\end{array}}{\Gamma \vdash e\#e_i : U} \qquad \textsc{Ex-A}\ \frac{\begin{array}{c}\Gamma \vdash e : «x{:}e_n;T» \\ \Gamma \vdash e_i : \textbf{Idx}\ e_n\end{array}}{\Gamma \vdash e\#e_i : T[e_i/x]}$$

$$\boxed{\Gamma \vdash T \leftarrow e} \qquad \textsc{A-T}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash T \leftarrow e} \qquad \textsc{A-Tup}\ \frac{\Gamma \vdash T_0 \leftarrow e\#0_n \qquad \forall_{1 \le i < n}.\Gamma \vdash T_i[e\#0_n/x_0] \cdots [e\#(i\text{-}1)_n/x_{i-1}] \leftarrow e\#i_n}{\Gamma \vdash [x_0{:}T_0, \ldots, x_{n-1}{:}T_{n-1}] \leftarrow e}$$

$$\boxed{\underline{e} \triangleright e} \qquad \textsc{N-Let}\ \frac{}{\textbf{let}\ x = e;\ e' \triangleright e'[e/x]} \qquad \textsc{N-Ex}_1\ \frac{}{e\#0_1 \triangleright e} \qquad \textsc{N-Tup}_1\ \frac{}{(e) \triangleright e} \qquad \textsc{N-Sig}_1\ \frac{}{[T] \triangleright T} \qquad \textsc{N-Pack}_1\ \frac{}{‹1;\ e› \triangleright e}$$

$$\textsc{N-Arr}_1\ \frac{}{«1;T» \triangleright T} \qquad \textsc{N-Tup}_\beta\ \frac{}{(e_0, \ldots, e_{n-1})\#i_n \triangleright e_i} \qquad \textsc{N-Pack}_\beta\ \frac{}{‹n;\ e›\#e_i \triangleright e} \qquad \textsc{N-Tup}_\eta\ \frac{}{(e\#0_n, \ldots, e\#(n\text{-}1)_n) \triangleright e}$$

$$\textsc{N-PackTup}\ \frac{n > 1}{\underbrace{(e, \ldots, e)}_{n\text{ times}} \triangleright ‹n;e›} \qquad \textsc{N-ArrSig}\ \frac{n > 1}{\underbrace{[T, \ldots, T]}_{n\text{ times}} \triangleright «n;T»} \qquad \textsc{N-}\beta\ \frac{e_f[e_a/x] \equiv \textbf{tt}}{(\lambda\ x{:}T@e_f{:}U = e_b)\ e_a \triangleright e_b[e_a/x]}$$

$$\textsc{N-TupPack}\ \frac{n \in \mathbb{N} \qquad x \in FV\{e\}}{‹x{:}n;e› \triangleright (e[0_n/x], \ldots, e[(n\text{-}1)_n/x])} \qquad \textsc{N-SigArr}\ \frac{n \in \mathbb{N} \qquad x \in FV\{e\}}{«x{:}n;T» \triangleright [T[0_n/x], \ldots, T[(n\text{-}1)_n/x]]} \qquad \textsc{N-Id}\ \frac{\text{otherwise}}{\underline{e} \triangleright e}$$

Fig. 1. Syntax, β-Reduction, Typing, & Normalization. $\underline{e}$ *might not* be normalized; e *is* normalized.

Sort/Integer	Bool	non-parametric binders	function/continuation type
`*` ≔ **Sort** `0`	**Bool** ≔ **Idx** `2`	`[…,T,…]` ≔ `[…,_:T,…]`	`T → U` ≔ `[_:T] → U`
I8 ≔ **Idx** `0x100`	**ff** ≔ 0_2	`«`e_n`;T»` ≔ `«_:`e_n`;T»`	**Cn** `T` ≔ `T → ⊥`
I16 ≔ **Idx** `0x10000` etc.	**tt** ≔ 1_2	`‹`e_n`;e›` ≔ `‹_:`e_n`;e›`	**Fn** `T → U` ≔ **Cn** `[T,` **Cn** `U]`

where
`e` **where let** `x1 = …; ⋯` **let** `xn = …;` **end** ≔ **let** `x1 = …; ⋯` **let** `xn = …; e`

anonymous function/continuation	named function/continuation
`λ x:T :U = e` ≔ `λ x:T@`**tt**`:U = e`	**lam** `f x:T :U = e` ≔ **let** `f = λ x:T :U = e`
cn `x:T = e` ≔ `λ x:T@`**ff**`:⊥ = e`	**con** `f x:T = e` ≔ **let** `f =` **cn** `x:T = e`
fn `x:T :U = e` ≔ **cn** `(x:T, return:`**Cn** `U) = e`	**fun** `f x:T :U = e` ≔ **let** `f =` **fn** `x:T :U = e`

(a) Simple syntactic sugar

```
λ  (T: *) ((x y: T), return: T → ⊥): ⊥ = return x
cn (T: *) ((x y: T), return: Cn T)      = return x
fn (T: *)   (x y: T): T                 = return x
λ T: * @tt: [_: [[T, T], [_: T] → ⊥]] → ⊥ =
    λ xyr: [[T, T], [_: T] → ⊥]@ff: ⊥ = xyr#1₂ xyr#0₂#0₂
```

(b) Curried functions/continuations

```
   [T: *] [[T, T], T → ⊥] → ⊥
Cn [T: *] [[T, T], Cn T]
Fn [T: *]  [T, T] → T

[T:*] → [_:[[T, T], [_:T] → ⊥]] → ⊥
```

(c) Curried function/continuation types

Fig. 2. Syntactic sugar (excerpt). All functions in 1b are equivalent. The type can be expressed by any expression in 1c—they are equivalent, too. The last respective item depicts the completely desugared version. Similar sugar is available for **lam**, **con**, **fun**. In addition, **lam**/**con**/**fun** allow for recursion (Section 3.2).

Example 3.4. Consider `let x = 3; x`. Rule N-Let will immediately normalize this expression to `3`. In fact, `let`-expressions *only* exist in Mim as N-Let will eliminate them outright. For this reason, there is neither a typing nor an evaluation rule for a `let`-expression. Note that all expressions are hash-consed (Section 4) and, thus, appear exactly once in the program graph. In fact, the implementation does not really perform a substitution here. The front-end just memorizes that `x` refers to a certain subgraph and all uses of `x` will be wired to that subgraph.

Plugins have the opportunity to extend normalizations (see Section 2.4) by adding rules of the form: `%myaxmiom` $e_1 \cdots e_n \triangleright e_{something}$. Typical examples include constant folding or various identities like `x+0` ▷ `x`.

Normalizations must be deterministic and cycle-free. Cyclic rules such as `x + x` ▷ `2 · x` and `2 · x` ▷ `x + x` can cause MimIR to diverge as it may endlessly oscillate between these two rules. Determinism is achieved automatically since rules are implemented in C++ which is executed deterministically. However, in the future, we would like to permit the plugin designer to directly specify rewrite rules in Mim. This would allow plugin designers to specify nondeterministic rules but MimIR could also issue warnings or errors in the case of potentially nondeterministic or cyclic rules.

Substitution. Substitution $e[e_b/e_a] = e_s$ where e_a is recursively replaced with e_b within `e` is defined in the usual manner with the following extension: *All expressions created along the way are also normalized.* Thus, the resulting expression e_s is again normalized. Note that `e`, e_b, and e_a are also normalized.

Example 3.5. The substitution $e\#x[0_1/x]$ directly yields `e` as N-Ex1 removes the superfluous extraction when the substitution assembles all substituted subexpressions.

3.1.2 Nat & Idx. MimIR has a builtin type **`Nat`** inhabited by `0`, `1`, `2`, ... Given `e` of type **`Nat`**, the type **`Idx`** `e` represents integers within the range $0_e, \ldots, (e-1)_e$ (Nat/Idx/Lit-N/Lit-I). For example, type **`Idx`** `3` has three inhabitants: 0_3, 1_3, 2_3. For convenience, **`Bool`** is an alias for **`Idx`** `2`, as this type has exactly two inhabitants, for whom appropriate aliases are available, too: **`ff`** $\coloneqq 0_2$ (false) and **`tt`** $\coloneqq 1_2$ (true). In addition, there is **`I8`**, **`I16`**, ... available for **`Idx`** `0x100`, **`Idx`** `0x10000`, ... We use these types for bootstrapping axioms, but they also play an important role within MimIR itself: The *arity*, i.e. the number of elements of a tuple/array, is a value of type **`Nat`**, whereas a value of type **`Idx`** `e` addresses a specific element of a tuple/array with arity `e`.

3.1.3 Tuples, Packs, Arrays & Σ-Types. Most languages distinguish between array and tuple terms as well as their types. Albeit MimIR does have different syntax, this distinction does not matter much because MimIR normalizes between both representations.

MimIR generalizes dependent pair types to n-ary dependent tuple types (Σ-types) where the *type* of an element may depend on the *value* of *any preceding* element (Sig). A dependent pair—denoted by `Σ x:T.U` in literature—is written as `[x: T, U]` in MimIR. However, MimIR allows for more complex dependent Σ-types:

```
let Num = [T: *, add: [T, T] → T, mul: [T, T] → T, _0:  T, _1: T];
```

The expression $«x{:}e_n;\ T»$ forms an *array type* with e_n-many elements and element type `T`. The *arity* e_n must be of type **`Nat`** and may introduce a variable `x` whose type is **`Idx`** e_n and may be used inside the *body* `T` (Arr). Rule N-ArrSig expands *parametric* arrays of *constant* arity to tuple types whereas Rule N-SigArr compresses *homogeneous* tuple types to arrays. This compression is not strictly necessary but makes the implementation more efficient. In particular, huge arrays like `«1000000; T»` are not expanded at all.

The term $(e_0,\ \dots,\ e_{n-1})$ introduces a *tuple* while $\langle x{:}e_n;\ e\rangle$ introduces a *pack*. Think of a pack as a compressed tuple. Both terms either inhabit a Σ-type or an array. Tuples and packs work analogously to tuple types and arrays but on term level (Tup/Pack/N-TupPack/N-PackTup).

Example 3.6. Due to normalization MimIR considers both the non-normalized as well as the normalized expressions as equal:

$$[\textbf{Nat},\ \textbf{Nat}] \triangleright «2;\textbf{Nat}» \qquad «i{:}2;\ F\ i» \triangleright [F\ 0_2,\ F\ 1_2]$$
$$(0,\ 0) \triangleright \langle 2;0\rangle \qquad \langle i{:}2;\ f\ i\rangle \triangleright (f\ 0_2,\ f\ 1_2)$$

The term $e\#e_i$ *extracts* from e the element with index e_i. This index must type as $\textbf{Idx}\ e_n$. If e types as array with e_n-many elements, the type of the extraction is the body of the array while substituting the array's variable with the given index e_i (Ex-A). If e types as Σ-type with e_n-many elements, there are two subcases. Ex-S_L picks the i^{th} element type (while substituting all preceding variables x_j with $e\#j_n$), if the index is a literal i_n.

Example 3.7. Suppose nmx has type $[n{:}\ \text{Nat},\ m{:}\ \text{Nat},\ x{:}\ F\ n\ m]$. Then, $nmx\#2_3$ has type $F\ nmx\#0_3\ nmx\#1_3$.

If the index is not a literal, Ex-S_i types the extraction as another extraction from a tuple "one level up". Each element of this tuple contains the corresponding element type of the Σ-type (while substituting all preceding variables x_j with $e\#j_n$). This is only allowed, if all involved element types agree on the same sort (see Example 3.8).

Table 1. Typing examples

Expression	Type
(0, 1, 2)	«3; **Nat**»
(0, 1, 2)#i	**Nat**
(0, **tt**)	[**Nat**, **Bool**]
(0, **tt**)#0_2	**Nat**
(0, **tt**)#i	(**Nat**, **Bool**)#i
(**Nat**, **Bool**)#i	‹2; *›#i ▷ *
(0, **Bool**)#i	↯

Rule N-β eliminates a tuple extraction with a known index while N-Pack$_\beta$ eliminates an extraction—no matter the index—from a non-parametric pack. Rule N-Tup$_\eta$ resolves a tuple comprised of a sequence of extractions from the same entity with increasing indices. Finally, MimIR consistently removes 1-tuples/packs (N-Tup$_1$, N-Pack$_1$), 1-tuple types/arrays (N-Sig$_1$, N-Arr$_1$), and extractions with 0_1 (N-Ex$_1$).

Example 3.8. Note that most languages need different syntax to introduce a tuple or an array term such as (0, 1, 2) vs. [0, 1, 2] in Rust. MimIR does not need this distinction. As Table 1 showcases, tuples with homogeneous element types are typed as array. This makes them amendable for extractions with an unknown index. However, tuples with inhomogeneous element types are typed as Σ-type. Extraction with a known index yields the corresponding element type as expected. However, extraction with an unknown index yields a type computation as another extraction (third last row). This again yields a type computation as another extraction (second last row). If Rule Ex-S_i would not prohibit extraction with an unknown index for a tuple whose elements do not agree on their sort, the type of the expression in the last row would be (**Nat**, **Sort** 0)#i of type (**Sort** 0, **Sort** 1)#i etc., leading to an infinite typing derivation.

Example 6.1 and Section 6.3.1 demonstrate the interplay between tuples/packs/Σ-types/arrays, their normalizations, and dependent types.

MimIR also provides an **insert** $(e_t,\ e_i,\ e_v)$ operation, that we elided in the formal presentation. It non-destructively creates a new tuple where the element at index e_i has been replaced with e_v. From a semantic point of view, insertion is a mix of term elimination and introduction as, for instance, **insert** $(e_t,\ 1_3,\ e_v)$ is the same as $(e_t\#0_3,\ e_v,\ e_t\#2_3)$.

3.1.4 Assignable. Consider (**Nat**, %core.ncmp.l) of type $[*,\ [\textbf{Nat},\ \textbf{Nat}] \to \textbf{Bool}]$. However, we can also type it as

```
let Cmp = [T: *, [T, T] → Bool]
```

which is similar to a trait in Rust or type class in Haskell. Most systems featuring existential or Σ-types require tuples to be *ascribed* such as `(Nat, %core.ncmp.l):Cmp`. In MimIR, tuples are *not* ascribed. Instead, whenever an expression `e` is *assigned* to a variable `x:T`, MimIR checks via $\Gamma \vdash$ `e` $\leftarrow$ `T` whether this assignment actually makes sense. This is trivially the case, if `e`'s type is `T` (A-T). Otherwise, A-Tup recursively checks whether all elements of a tuple are assignable while successively resolving the dependencies the Σ-type `T` may introduce (similar to Ex-S_L/Ex-S_i discussed above).

Example 3.9. In the following code the type checker allows passing `(Nat, %core.ncmp.l)` to `f`, although it expects an instance of `Cmp`: Rule Lam asks Rule A-Tup whether the given pair is assignable to `Cmp`, which it is in fact.

```
lam f(T: *, less: [T, T] → Bool)(x: T): Bool = less (x, x);
f (Nat, %core.ncmp.l) 23
```

3.1.5 Functions. A function `λx:T@`e_f`:U = e` has a dependent function type `[x:T] → U` (Lam)—written as `Π x:T.U` in literature. This means that the *type* of the function's codomain `U` may depend on the *value* of the argument `x` which in turn must inhabit the domain `T`. The callee `e` of an application `e` e_T must type as a Π-type and the given argument e_T must be *assignable* (see above) to the domain; the type of the application is resolved by substituting `x` with the argument in the codomain `U` (App). The Boolean term `@`e_f is the so-called *filter* and is used for partial evaluation (Section 3.3).

CPS. Continuations are functions that never return. MimIR models them as functions whose codomain is ⊥ (Bot)—a type without inhabitants—and Mim provides syntactic sugar (Figure 1a) for continuations (**`cn`**/**`con`**) and their types (**`Cn`**). Continuations bridge the gap between CFGs and the λ-calculus, as continuations are akin to basic blocks.

Example 3.10. Consider the "if diamond" in Figure 3b. The expression `(F, T)#cond ()` first selects either the `F` or `T` continuation. This works because `cond` is of type **`Bool`**—an alias for **`Idx`** `2`. The result of the extraction is of type **`Cn`** `[]` (see Ex-S_i and N-Pack$_\beta$) which allows us to apply the selected continuation to `()` to continue execution there. The `T` case invokes `N 42` while the `F` case invokes `N 23`. In static single assignment (SSA) form, `f`, `F`, `T`, and `N` would be basic blocks and `N` would need a ϕ-function to select either `23` or `42`. In CPS the ϕ-function becomes the parameter `phi` of continuation `N` while the operands of the ϕ-function become arguments to the appropriate continuation call [5, 28]. See also Example 4.2.

Continuations may be used (mutually) recursive (Section 3.2) to model arbitrary, unstructured control flow. Note that `f` in Figure 3b is a continuation and the `return` point is explicit by passing it to `f` as another continuation. This idiom is so common that Mim introduces **`fn`**/**`fun`**/**`Fn`** as sugar (Figure 1a). For example, in Figure 3b we can instead write:

```
fun f(cond: Bool): Nat = /*...*/;
```

Finally, Mim provides syntactic sugar for *curried* functions & continuations (Figure 1b-c). Note that all curried function groups except the last one are in direct style in the case of curried continuations.

3.2 Full Recursion

Named functions (Figure 1a) are in fact more powerful than ordinary **`let`** bindings, as they allow for (mutual) recursion and, hence, are more like `letrec` in other languages.

```
lam forever(x: Nat): Nat = forever x;
```

This is an extension of the calculus presented so far. Internally, the recursive function `forever` is represented as a cyclic graph (Section 4). MimIR allows recursion in both direct style (Example 3.12) and CPS:

Example 3.11. The following `loop` counts from `0` to `42`. If its parameter `i` (the "ϕ-function") is less than `42`, `i` is incremented and `loop` recurses. Otherwise, it `exit`s.

```
con loop(i: Nat) =                                  // i = ϕ(0, i + 1)
    (exit, body)#(%core.ncmp.l (i, 42)) () where    // if i < 42 then body() else exit()
        con body() = loop (%core.nat.add (i, 1)); // recurse
        con exit() = next i;                        // pass last instance of i = 42 to next
    end;
loop 0;                                             // loop entry
```

3.3 Partial Evaluation & β-Equivalence

The so-called *filter* `@e`$_f$ of a function is used for partial evaluation and is a Boolean expression that may depend on the function's parameter (N-β). For each call-site, MimIR instantiates the filter by substituting the function's variable with the call-site's argument. Remember that substitution also normalizes. If this syntactically yields **`tt`**, MimIR will β-reduce this call-site—potentially recursively inlining more function calls. This mechanism allows for compile-time specialization.

Since MimIR has a dependent type system featuring full recursion (Section 3.2), type-checking becomes undecidable in general since checking for β-equivalence is. This is arguably the most concerning issue for picking up dependent types in more mainstream programming languages. MimIR tackles this issue by (partially) evaluating functions according to the filters during normalization.

Example 3.12. The following recursive function `pow`, will be recursively β-reduced, if the exponent `b` is a compile-time constant. Note that `f`'s parameter `x` has a dependent type.

```
lam pow(a b: Nat)@%core.pe.known b: Nat =
    (%core.nat.mul (a, pow(a, %core.nat.sub (b, 1))), 1)#(%core.ncmp.e (b, 0));

lam f(n: Nat, x: «%core.nat.mul (n, %core.nat.mul (n, n)); Nat»): [] = ();
lam g(m: Nat, y: «pow (m, 3); Nat»): [] = f (m, y);
```

As `g` calls `f` with `(m, y)`, MimIR has to check whether `y`'s type is assignable to `g`'s domain (Pi). Rule N-β determines that `pow`'s filter evaluates to **`tt`** with the given argument `(m, 3)`. This causes MimIR to immediately β-reduce `pow (m, 3)` which will result in a call-site `pow (m, 2)`. Rule N-β determines again that the filter yields **`tt`**, causing another β-reduction. The type checker now sees `y`'s type as

```
«%core.nat.mul (m, %core.nat.mul (m, m)); Nat»
```

which is assignable to `g`'s domain. Using a **`ff`** filter for `pow` would result in a type error, as `x`'s type is different from `«pow (m, 3); Nat»`. Using a **`tt`** filter for `pow` is dangerous: While the example above would still work, a call-site like `pow (i, j)`, where `j` is not a compile-time constant, would cause MimIR to diverge, as it would endlessly β-reduce new calls to `pow`. However, this termination behavior is not random and completely transparent to the programmer. It depends on the specified filters as well as how they are used—*as stated by the programmer.*

Coupling partial evaluation with normalization that in turn interacts with type checking is a novel approach to dependent type checking. Moreover, partial evaluation filters clearly determine which parts of a program are evaluated at compile time and which ones remain in the compiled program. This distinction is somewhat unclear in many dependently typed languages—Idris 2 is an exception (see Section 7).

Note that the rules in Figure 1 do *not* contain a conversion rule that states that two terms are equal modulo β-equivalence. The typing rules are syntax-directed and deterministic similar to

Pollack and Poll [52]. The normalization rules unambiguously state where normalization and, in particular, β-reduction happens during type checking.

Default Filter Policy. Since our goal is high-performance code, we want to specialize type variables at compile time [63]—similar to C++ templates, Rust generics, polymorphism in MLton [56] but unlike Java generics. For this reason, Mim defaults to `tt` for elided partial evaluation filters except for the final continuation in a (curried) `cn`/`con`/`fn`/`fun` function. The filter of this final continuation defaults to `ff` (Figure 2). This has the desired effect that the actual computations are deferred to runtime while other abstractions such as type abstractions are specialized. However, programmers can override the default behavior by explicitly specifying a filter as in Example 3.12.

3.4 Type Safety

In order to argue about MimIR's soundness, we present a nondeterministic β-reduction relation `e` $\longrightarrow$ `e` that *steps* a normalized expression—possibly by descending into a subexpression via Cong—into another normalized expression by performing a single β-reduction (Beta). Rule Cong introduces nondeterminism as the evaluation context $\mathcal{E}$ comprises all possible evaluations for expressions. This provides the flexibility during optimization, for instance, to either descend into the callee or the argument of an application—or directly apply Beta, if possible.

We have formalized the rule system in the proof assistant Coq and the following lemmas establish MimIR's type safety (recall once again that all expressions are normalized):

Lemma 3.13 (Progress). *If* $\Gamma \vdash$ `e : T`, *then* `e` *is a value or there exists an* `e'` *such that* `e` $\longrightarrow$ `e'`.

Lemma 3.14 (Preservation). *If* $\Gamma \vdash_\beta$ `e : T`, *and* `e` $\longrightarrow$ `e'`, *then* $\Gamma \vdash_\beta$ `e' : T`.

Since dependent types may depend on variables introduced by lambdas, a Beta-step may change the type of an expression. For this reason, MimIR does not guarantee *strong* preservation, as the resulting type after a step may differ from the original one. However, the preservation is stronger than *weak* preservation since the type of the expression after a step is not arbitrary but evolves from the old one by zero or more $\longrightarrow$-steps. To model this, we introduce β-equivalence on type level as indicated in the typing rule $\vdash_\beta$. This is a technical addition to prove preservation as seen in other models of CC.

For a proper evaluation semantics you can choose any deterministic subset of $\longrightarrow$ such as a strict, left-to-right evaluation order as in MimIR's LLVM backend. However, the nondeterministic relation $\longrightarrow$ also establishes type safety for reductions that happen a priori. For example, many optimizations such as copy propagation or scalarization are implemented on top of β-reductions.

3.4.1 Composition of Plugins.

Custom axioms/normalizers, however, could violate progress or preservation. Violation of preservation is a bug in the plugin that will trigger a type error as soon as the erroneous normalization fires. For example, the nonsensical normalization `3 + 5` ▹ `()` changes the type from `Nat` to `[]`. MimIR will catch such errors after applying a non-type-preserving rewrite, as this will lead to an ill-typed program. Specifying rewrite rules directly in Mim (see end of Section 3.1.1) would allow us to directly type-check rules for preservation even before firing them.

In addition, an axiom may violate progress, if the function it represents is not total. This means that the axiom's normalizer (or the translation of the axiom into another program) cannot cope with all possible inputs. For example, `%mem.load`ing from a dangling pointer or `%core.bitcast`ing 4_5 to `Idx 4` are such unsafe instances and lead to undefined behavior. Violating progress is not necessarily bad per se. For example, if you want to compile an unsafe language like C, you obviously need an unsafe language to model this behavior. If you want your axioms to be type safe, you must guarantee that they do not violate progress.

Even though axioms do not provide an implementation by themselves, they are treated and, in particular, type-checked like any other value in MimIR. As long as the axioms' implementations obey preservation and progress, Lemma 3.13 and 3.14 guarantee a type-safe composition which in turn enables a type-safe composition of plugins. Even if an axiom violates progress, it is merely a problem inherent to that axiom, and does not arise from interactions with other axioms. In addition, axioms are in contrast to instructions in MLIR, LLVM, or Thorin *first-class* citizens. This means, for instance, that an axiom or a partial application of it can be passed as an argument to another function or higher-order axiom. See `%tensor.reduce` in Section 6.3.1 for an example and Section 6.4 for our practical experience with composing plugins.

3.4.2 Type-Safe Code Transformations. Oftentimes, code transformations are expressible as specializations. For example, suppose we want to create several unrolled loops—each time with a different body and differently typed induction variables. Mim allows us to directly specify a polymorphic loop and partially evaluate it to generate the unrolled, specialized version. The following function `iter` invokes `n` times body and jumps to `exit` afterwards. The parameter body expects another continuation as argument which is the `continue` point within `iter`. In order to allow for arbitrary induction variables, `iter` is polymorphic in its accumulator type `A` that the other continuations use to communicate between different iterations.

```
con iter {A: *} (n: Nat) (body: Cn [Cn A][A]) (exit: Cn A) (acc: A)@tt =
    head(0, acc) where
        con head (i: Nat, acc: A)@%core.pe.known n =
            let cond = %core.ncmp.l (i, n);
            (f, t)#cond () where
                con t() = body continue acc where
                    con continue(acc: A) = head ((%core.nat.add (i, 1)), acc);
                end
                con f() = exit acc;
            end;
    end;
```

Now, if we invoke

```
iter n body exit (0, 1)
where
    con body (yield: Cn [Nat, Nat]) (a b: Nat) = yield (b, %core.nat.add (a, b));
    con exit (a _: Nat) = return a;
end
```

we essentially construct a `while` loop that uses in addition to the loop counter `i` two **`Nat`**s as accumulator (the loop's "ϕ"-functions—see Example 3.10) and computes the nth Fibonacci number. If we change the call to use 12 instead of n, the filter of head recursively becomes **`tt`** which causes a complete unrolling of the loop and results in a cascade of 12 continuations. This will also happen, if calling `iter` from C++ (see Section 5.3). After a standard set of optimizations the result is just `return 144`—the 12th Fibonacci number.

Note that in IRs like MLIR or LLVM we would write C++ code, instead, that *generates* an unrolled loop specialized for the desired type (two **`Nat`**s in our example). This C++ "generator"-code is significantly more complex because writing code that emits code is inherently more verbose. In addition, we have to "look through" the API calls and picture in our minds how the generated program will look like. Furthermore, this generator code is also significantly more error-prone because we may very well accidentally construct ill-typed IR. This cannot happen, if our starting point is already well-typed Mim as above.

4 Graph Representation

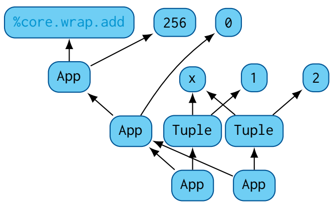
MimIR's implementation is based upon the "sea of nodes" concept [13]. This means that MimIR's internal representation is a data dependence graph where each node in the graph represents an

```
let a = %core.wrap.add 0 (x,1I8);
let b = %core.wrap.add 0 (x,2I8);
```

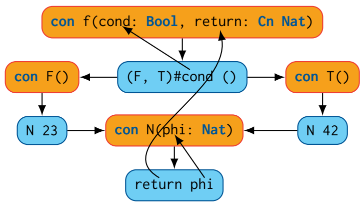
(a) Two additions in Mim

```
con f(cond: Bool, ret: Cn Nat) =
  (F,T)#cond () where
    con F() = N 23;
    con T() = N 42;
    con N(phi: Nat) = ret phi;
  end;
```

(b) "If diamond" w/ "phi" in Mim



(c) MimIR graph for 3a



(d) Condensed MimIR graph for 3b

Fig. 3. Mim vs. MimIR. Type edges are elided. Immutables are in blue, mutables in orange.

expression. If e_s is a subexpression/operand of an expression e, the graph contains an edge $e \rightarrow e_s$. For example, an *App* has two operands—the callee and argument—and, hence, two outgoing edges while a Σ-type with three element types has three operands and, hence, three outgoing edges. In addition, every expression e has exactly one type T it inhabits. This relation is modeled with another edge $e \multimap T$ in the program graph. Note once again that T is an expression. Hence, there is only *one* graph that contains both terms and types as well as any dependencies between them via edges; in particular, we may have types that depend on terms due to dependent types. Furthermore, this graph is *complete*. This means that this graph comprises the whole semantics of a MimIR program and—apart from an internal hash set to hash-cons all nodes (Section 2.4)—MimIR does *not* rely on any other auxiliary data structures like instruction lists, basic blocks, special regions, or CFGs. **let**-expressions and even explicit function nesting only exist in Mim. Thus, a MimIR graph *solely* consists of a large number of nodes that "float" in a complex network, resembling a sea—hence the term "sea of nodes".

Example 4.1. Figure 3c depicts the MimIR graph of the Mim program in Figure 3a. Note that the **let**-expressions in Mim are absent in MimIR and the curried call `%core.wrap.add /*256*/ 0`, where the inferred implicit argument is now explicit, is shared by both additions. In fact, all 8-bit wide additions with mode `0` (Section 6.1.1) will reuse this subgraph. Similarly, `x` is shared by both argument tuples of the additions. What is not shown in Figure 3c are the type edges. For example, the bottom Apps and the literals `1` and `2` point to a subgraph that constitutes **Idx** `256`.

Immutables vs. Mutables. So far, we have discussed expressions that we create by first building their operands, and then the actual nodes. MimIR calls these expressions *immutable*: Once constructed, they cannot be changed later on. Hence, immutables form a directed acyclic graph (DAG). In order to allow for recursion, we have to somehow form a cyclic graph. For this reason, there are also *mutable* expressions (Table 2). For mutables, we first build the node and set its operands later on. This enables us to form a cyclic graph and therefore enables recursion. Mutables also allow changing their operands later on—hence their name. This also means that a mutable will not be hash-consed as opposed to immutables. But MimIR will check mutables for α-equivalence, if necessary. Finally, all expressions (both on term and type level) that introduce variables are also modeled as mutables.

Table 2. Immutable vs. mutable nodes

Immutable	Mutable
build operands first, then the actual node	build the node first, then set the operands
operands form a DAG	operands may be cyclic
hash-consed	each new entity is fresh
non-parametric	may be parametric

Example 4.2. Consider the MimIR graph in Figure 3d of Figure 3b. Note how this graph resembles a classic CFG. In order to access `f`'s and `N`'s variables, these continuations are mutables. In fact, most of the time we build functions as mutables even though `F` and `T` could be modeled as immutables in this particular case, as they are neither used recursively nor are their variables accessed.

5 Type Checking, Inference, and Normalization On-The-Fly

MimIR's implementation performs normalization, type checking, and inference of implicits as soon as a new expression is constructed. This eagerness has the advantage that a compiler/plugin developer will immediately notice, if they incorrectly plugged together some expressions that are ill-typed.

5.1 α-Equivalence

This means that type checking, inference, and normalization must also work on open terms, i.e., in the presence of free variables. The implementation boils down to checking for α-equivalence modulo free variables inside of the *assignable* relation. Here we know that *either* two expressions *must* be α-equivalent *or* there is a type error. For this reason, MimIR optimistically assumes during *type checking* that any free variables are α-equivalent. Eventually, all terms will be closed where any remaining issues will be found.

Example 5.1. During type checking MimIR considers the following expressions as α-equivalent:

`λ(a: Nat): Nat = b` and `λ(x: Nat): Nat = y`.

Example 5.2. For *normalization*, however, this assumption is unsound. N-PackTup cannot simply normalize the following expression as b and y may be bound differently:

`(λ(a: Nat): Nat = b, λ(x: Nat): Nat = y) ⋫‹2; λ(a: Nat): Nat = b›`

As discussed in Section 4, pointer equality of two MimIR expressions implies normalized, syntactic equivalence. Both expressions refer to the same object. However, pointer equality does not necessarily imply α-equivalence when free variables are involved. For this reason, MimIR only resorts to pointer equality when checking for α-equivalence in the absence of free variables.

Example 5.3. Although the bodies of the following functions enjoy pointer equality, these functions are *not* α-equivalent as x is bound in the first function and free in the second one:

`λ(x: Nat): Nat = x` and `λ(y: Nat): Nat = x`

5.2 Type Inference

Whenever a function with an implicit argument is invoked, the function will first be applied with a placeholder.

Example 5.4. Consider the annex `%core.minus` that computes unary minus of its s-sized `Idx` argument a (Figure 4a). Note that s is implicit. Furthermore, the operation expects a so-called mode of type `Nat` (see Section 6.1 for details). As soon as we apply the first explicit argument mode to `%core.minus`, MimIR will insert a fresh placeholder—let us say ?5—as implicit argument in between:

```
%core.minus ?5 mode
```

This yields the type `Idx ?5 → Idx ?5` (see App). When we apply 42_{256} as third argument, App triggers the *assignable* relation. Rules A-T and A-Tup additionally match placeholders with the provided argument and fill out any gaps. So here, we find: ?5 = 256:

```
%core.minus /*256*/ mode 42₂₅₆
```

Due to the implicit `tt` filter, MimIR will β-reduce the application to:

```
%core.wrap.sub /*256*/ mode (%core.idx 256 mode 0, 42₂₅₆)
```

This in turn will be normalized to 214_{256} (two's complement of -42).

5.3 C++ Interface

Type checking, inference, and normalization happens *regardless* of whether MimIR is used through Mim or its C++ interface.

Example 5.5. To better bring the point across how MimIR behaves when controlling from C++, reconsider Example 5.4. We can construct the same call using the C++-API:

```
const Def* res = world.call<core::minus>(mode, world.lit_idx(256, 42));
```

Now, type checking, inference, and normalization will happen as discussed above. Thus, the C++ variable `res` (of static type `const Def*` and dynamic type `const Lit*`) will point to a literal that represents 244 of type `Idx 256`. In other words, creating MimIR expressions from C++ triggers the usual normalization that may in turn cause a Turing-complete (partial) evaluation of the expression.

Example 5.6. When trying to construct an ill-typed MimIR program from C++, MimIR will throw a C++ exception as in the following code that erroneously passes a `Nat` instead of an `Idx` literal to `%core.minus`:

```
const Def* res = world.call<core::minus>(mode, world.lit_nat(42)); // %core.minus mode 42
```

6 Case Studies & Evaluation

Our case studies show dependent types as a useful tool in regards to safe guards, expressiveness, and efficiency. Informative types prevent unwanted behavior and allow for more aggressive optimizations. This section showcases MimIR's flexibility by discussing some of the plugins we have already developed and evaluating their performance. In particular, we want to show that MimIR is able to achieve the same performance as other low-level approaches like C/LLVM, is able to fully remove carefully crafted abstractions, and that designing code analyses/transformations in MimIR is no more complicated than in traditional compiler IRs like LLVM.

If not mentioned otherwise, we ran all tests using a single thread on an AMD Ryzen 7 3700X supported by 64GB of DDR4@2133MT/s RAM. We used LLVM/Clang 15.0.7 for the C sources as well as the LLVM code emitted by MimIR. All MimIR programs compiled within a few seconds at most.

6.1 Low-Level Plugins

In this section, we discuss the design of three low-level plugins, which are deliberately modeled on LLVM: First of all, LLVM is a well-thought-out low-level IR; second, it allows for straightforward mapping of axioms defined in these plugins to LLVM instructions in MimIR's LLVM backend. This backend also lives within a plugin. Other more high-level plugins eventually lower their axioms to those of these low-level plugins. The `%core` plugin introduces integer operations, the `%math` plugin a floating-point type operator and operations for it, and the `%mem` plugin side-effects, memory operations, and pointer arithmetic.

6.1.1 The core Plugin. This plugin (Figure 4a) defines arithmetic operations (`%core.nat`) and comparisons (`%core.ncmp`) for `Nat` and integer operations on values of type `Idx s`. All integer operations abstract over the size `s`. Like in LLVM, it is up to the operation to decide whether a specific `Idx s`-typed value is signed or unsigned. For example, the right-shift `%core.shr` comes in two flavors: `arith`metic (with sign extension), and `logical` (without sign extension). Integer operations like `%core.wrap.add` take an overflow mode `m`. This mode dictates how overflow is handled (wrap-around or undefined behavior for both signed and unsigned overflow). Along the same lines, `%core` introduces all 4 unary (`%core.bit1`), all 16 binary bitwise (`%core.bit2`), as well as the usual signed/unsigned comparison (`%core.icmp`) operations. The `%core.conv` axiom

```
// Create literal of type Idx s from l while obeying mode m
axm %core.idx: [s: Nat] [m: Nat] [l: Nat] → Idx s;

axm %core.nat(add, sub, mul): [a b: Nat] → Nat;
axm %core.ncmp(/*...*/):      [a b: Nat] → Bool;

axm %core.bit1(f, neg, id, t):      {s: Nat} [m: Nat] [a  : Idx s] → Idx s;
axm %core.bit2(/*...*/):            {s: Nat} [m: Nat] [a b: Idx s] → Idx s;
axm %core.wrap(add, sub, mul, shl): {s: Nat} [m: Nat] [a b: Idx s] → Idx s;
axm %core.shr(a, l):                {s: Nat}          [a b: Idx s] → Idx s;
axm %core.icmp(/*...*/):            {s: Nat}          [a b: Idx s] → Bool;

axm %core.conv(s, u): {ss: Nat} [ds: Nat] [Idx ss] → Idx ds;
axm %core.bitcast:    {S: *} [D: *] [S] → D;

lam %core.minus {s: Nat} (m: Nat) (a: Idx s): Idx s = %core.wrap.sub m (%core.idx s m 0, a);
```

(a) The %core plugin

```
axm %math.F: [p e: Nat] → *;
let %math.f64 = (52, 11);          // similar: f16, f32, ...
let %math.F64 = %math.F %math.f64; //          F16, F32, ...

axm %math.arith(add, sub, mul, div, rem):
                        {p e: Nat} [m: Nat] [a b: %math.F (p, e)] → %math.F (p, e);
axm %math.tri(/*...*/): {p e: Nat} [m: Nat] [a  : %math.F (p, e)] → %math.F (p, e);
axm %math.cmp(/*...*/): {p e: Nat} [m: Nat] [a b: %math.F (p, e)] → Bool;

lam %math.minus .(p e: Nat) (m: Nat) (a: %math.F (p, e)): %math.F (p, e) =
    %math.arith.sub m (0:(%math.F pe), a);
```

(b) The %math plugin

```
axm %mem.M:     *;
axm %mem.Ptr:   * → *;
axm %mem.alloc: [T: *] [%mem.M] → [%mem.M, %mem.Ptr T];
axm %mem.free:  {T: *} [%mem.M, %mem.Ptr T] → %mem.M;
axm %mem.load:  {T: *} [%mem.M, %mem.Ptr T] → [%mem.M, T];
axm %mem.store: {T: *} [%mem.M, %mem.Ptr T, T] → %mem.M;
axm %mem.slot:  [T: *] [%mem.M, id: Nat] → [%mem.M, %mem.Ptr T];
axm %mem.lea:   {n: Nat, Ts: «n; *»} [%mem.Ptr «j: n; Ts#j», i: Idx n] → %mem.Ptr Ts#i;
```

(c) The %mem plugin

```
axm %autodiff.AD:           * → *;               // marks type-level transformation
axm %autodiff.ad: [T: *] [T] → %autodiff.AD T; // marks term-level transformation
```

(d) The %autodiff plugin

Fig. 4. Selection of plugins we implemented (excerpts). Normalizer information has been elided.

converts between different-sized signed or unsigend values via truncation, zero-, or sign-extension whereas `%core.bitcast` allows for arbitrary (potentially unsafe) casts.

For convenience, there is a function `%core.minus` available that builds a unary minus by subtracting the given operand `a` from `0` (see also Section 5.2-5.3). This function will always be inlined due to the default **`tt`** filter (Section 3.3). Note that in LLVM, MLIR, and similar IRs such helpers are usually implemented as C++ code that directly emit the desired code snippet as their type systems cannot express polymorphic/dependently typed functions. The MimIR function `%core.minus` on the other hand can be called and passed around just like any other axiom.

6.1.2 The math Plugin. This plugin (Figure 4b) introduces a type operator `%math.F` that expects the number of significant precision bits `p` and exponent bits `e` over which all `%math` operations abstract. Additionally, most operations expect a mode `m` that fine-adjusts how strictly they should obey the IEEE-754 standard for floating-point transformations. For example, you may choose to allow reassociation of floating-point operations or ignore `NaN`s or infinity. The axioms for the actual operations such as `%math.arith`metic, `%math.tri`gonometric, or floating-point comparisons (`%math.cmp`) and convenience wrappers like `%math.minus` are straightforward.

6.1.3 The mem Plugin. This plugin (Figure 4c) introduces a type `%mem.M` to abstract from the machine state (Section 2.3). We expose axioms to allocate memory, load from, and store values into previously allocated pointers, and to index into pointers that point to compound data types. Most axioms expect a machine state and additional arguments like the pointer to operate on, and return a memory instance together with the produced results like the loaded value. The `%mem.lea`[1] axiom performs pointer arithmetic. Since this instruction does not have side effects as it does not directly interact with memory, no machine state is needed. The `%mem.lea` axiom takes a pointer to a tuple or array and the offset `i` in which it wants to index. The result is the pointer to the `i`-th element.

Table 3. Original C version vs. our Impala implementation that uses MimIR (lower is better)

Benchmark	Input size	C [s]	Impala [s]
aobench	–	0.667 ± 0.014	0.659 ± 0.004
fannkuch	12	27.709 ± 0.210	26.301 ± 0.186
fasta	25,000,000	0.711 ± 0.011	0.814 ± 0.013
mandelbrot	5,000	1.393 ± 0.011	1.390 ± 0.010
meteor	2,098	0.036 ± 0.001	0.036 ± 0.004
nbody	50,000,000	3.621 ± 0.031	2.825 ± 0.021
pidigits	10,000	0.371 ± 0.005	0.370 ± 0.004
spectral	5,500	1.387 ± 0.011	1.409 ± 0.014
regex	–	4.101 ± 0.037	4.128 ± 0.044
reverse	–	0.744 ± 0.025	0.714 ± 0.029
geom. speedup	–	1	1.020

Example 6.1. In the following listing, `p1` points to a 3-tuple and `%mem.lea` indexes into the second element (1_3). Then, `%mem.lea` computes the pointer to the `i`th element of an array of size `n`. Both `i` and `n` are statically unknown. In particular, note how `%mem.lea`'s dependent type computes with the help of the normalization rules the result type of the element pointers.

```
let p1/*: %mem.Ptr [Nat, I16, Nat]*/ = /*...*/;
let q1/*: %mem.Ptr I16              */ = %mem.lea /*(3, (Nat, I16, Nat)*/ (p1, 1_3);
let i /*: Idx n                     */ = /*...*/;
let p2/*: %mem.Ptr «n; Nat»       */   = /*...*/;
let q2/*: %mem.Ptr Nat              */ = %mem.lea /*(3, ‹n; Nat›)*/ (p2, i);
```

6.1.4 Evaluation. In this experiment, we want to confirm that the low-level plugins perform as well as directly using LLVM. To this end, we ported the code generator of the research language Impala [35] to MimIR 2. We ported the fastest available C implementations that were neither manually vectorized nor parallelized from *The Computer Language Benchmarks Game* [62] to Impala. In addition, we ported the publicly available `aobench` [22]. Table 3 shows that the performance is as expected nearly identical to the original C versions (with two slight outliers—one for the better, the other one for the worse). Note that the `regex` benchmark does *not* use the `%regex` plugin.

6.2 Regular Expressions

We have already discussed the `%regex` plugin in Section 2: The plugin defines a set of axioms representing ranges of literals, consecutive elements (conjunction), alternative elements (disjunction), negation, and quantifiers. These are sufficient to define a useful set of compile-time regular expressions. While we only test this plugin with Mim, one could easily integrate MimIR with this plugin as a code generator into a RegEx parser and either generate code for a RegEx at compile time or just-in-time (JIT)-compile at run time.

This showcase demonstrates that MimIR provides a powerful core whose extensibility indeed makes implementing a DSL with intrinsic, normalizable expressions and domain-specific optimizations very straightforward. Our `%regex` plugin provides a legalization pass that is written in C++

[1]The name is inspired by the x86 assembly instruction and its semantics is similar to `getelementptr` in LLVM but arguably more streamlined.

and receives the normalized RegEx pattern in its opaque form. This allows the pass to use the exhaustive understanding of the RegEx to generate an optimized matcher using finite automatons. To do so, the pass first translates the RegEx into a NFA, further converts this to a DFA [67, 2.2], and then minimizes it [27]. Finally, the minimal DFA is translated into low-level control flow in MimIR's IR that is purely based on integer comparisons as well as jumps between state continuations. The generated code is put into action by the pass as it replaces the pattern application with a call to the newly generated matcher function.

In total, the RegEx plugin only encompasses 919 lines of C++ code and 22 lines of MimIR code. We compare our RegEx implementation in MimIR with Compile Time Regular Expression (CTRE) [19], `std::regex` [21], as well as Perl-compatible Regular Expressions 2 (PCRE2) [26] in an interpreted and a JIT compiled variant. From the lines of code (LoC) numbers in Table 4 we observe that MimIR's implementation is at least one order of magnitude less complex. Despite the rather low complexity, our engine outperforms state-of-the-art RegEx engines. It is even 28% faster than a manually written, low-level matcher that we implemented in C and matches only the specific pattern below. Most likely, our C implementation contains some redundant checks, but spotting them is very hard in such low-level code that comes from manually writing a complicated matcher.

Table 4. Comparison of RegEx engines

RegEx Engine	LoC	Compile time [ms]	Match time [µs]
CTRE	4,153	1,677	4,736
std::regex	4,874	2,207	10,151
pcre2	85,879	67	3,882
pcre2-jit			1,308
hand-written C	102	45	886
MimIR	941	145	640

The listed LoC include actual code lines exclusively.[2] "Compile time" is the execution time of `clang++` and for MimIR the MimIR frontend and optimizer. All benchmarks test the following RegEx on 10,215 E-Mail addresses that are matched by this pattern and 450 that are not [53]:

```
^[a-zA-Z0-9](?:[a-zA-Z0-9]*[._\-]+[a-zA-Z0-9])*[a-zA-Z0-9]*@[a-zA-Z0-9](?:[a-zA-Z0-9]*[_\-]+[a-zA-Z0-9])*[a-
        zA-Z0-9]*\.(?:(?:[a-zA-Z0-9]*[_\-]+[a-zA-Z0-9])*[a-zA-Z0-9]+\.)*[a-zA-Z][a-zA-Z]+$
```

We have also tested other regular expressions with similar results.

6.3 Machine Learning

6.3.1 Tensor Plugin. The `%tensor` plugin provides operations that abstract over the tensors' rank, shape, and element type. The rank and shape of involved tensors is tracked in their types across operations. For instance, the matrix product takes two matrices of size $l \times m$ and $m \times n$ returning a matrix of size $l \times n$. Note that l, n, and m do not necessarily have to be constants—a common restriction in many other programming idioms/systems including XLA [24] or C++ templates. Operations like getting the shape of a tensor are statically optimized away by directly accessing the information of the type. Additionally, MimIR's type system guarantees that read and write accesses occur within bounds, eliminating the need for dynamic bound checks.

Using MimIR's expressive type system, the `%tensor` plugin defines a general `map_reduce` function in the spirit of Numpy's `einsum` function. The plugin eventually lowers all other tensor operations into a `map_reduce` call. This significantly cuts the number of cases to check in optimizations. In particular, many peephole optimizations like collapsing two consecutive transpositions into the identity are handled automatically. Furthermore, the usage of high-level axioms also benefits other plugins. For example, the `%autodiff` plugin (see below) directly operates on the high-level axioms of the `%tensor` plugin.

As a non-trivial example that showcases many of MimIR's features, consider the following axiom in the spirit of XLA's variadic reduction:

[2]LoC as reported by cloc, treating MimIR code as Rust

```
fun (x: T, y: T): T =
  let z = x + y;
  return (x * z);
```

```
fun ((x, x*): [T, T → T], (y, y*): [T, T → T]): [T, T → T] =
  let z, z* = (x + y, fn (s: T): T = return (x*(s) + y*(s)));
  return (x * z, fn (s: T): T = return (x*(z*s) + z*(x*s)));
```

Fig. 5. Each computation of the original program (left) is augmented with a backpropagator marked with $\square^*$ (colored in blue, right). The differentiated function returns $x \cdot (x+y), \lambda s.s \cdot (2x+y, x)$. MimIR-like pseudocode.

```
axm %tensor.reduce: {r: Nat, s: «r; Nat», ni: Nat, Is: «ni; *»}
                    [f: «2; «i: ni; Is#i» » → «i: ni; Is#i»]
                    [is: «i: ni; «s; Is#i» », init: «i: ni; Is#i», dims: «r; Bool»]
                  → «i: ni; « ‹j: r; (s#j, 1)#(dims#j)›; Is#i» »;
```

This operation is polymorphic in the number of inputs `ni`, the rank `r` and the shape `s` of the involved tensors, as well as their element types `Is`. The reduction function `f` expects a pair of `ni`-tuples whose elements correspond to `Is` and yields again such an `ni`-tuple. The `ni`-tuple `init` comprises the initial values of the reduction. The Boolean mask vector `dims` of size `r` selects for each dimension whether it should be reduced. For this reason, the reduction yields `ni` arrays of rank `r` whose shape is computed by either using the original dimension or `1`, if the corresponding element in the `dims` vector is **`tt`**: Since dimensions of arity `1` will collapse during normalization, the resulting array will only exhibit dimensions that were not selected with `dims`. For example, suppose `a`, `b`, and `c` are $n \cdot 4 \times n$ matrices with element types **`I8`**, **`I16`**, and **`I32`**, respectively. Then, the following expression

```
lam f (x y: [I8, I16, I32]): [I8, I16, I32] = /*...*/;
let n4  = %core.nat.mul (n, 4);
%tensor.reduce /*(2, (n4,n), 3, (I8,I16,I32))*/ f ((a, b, c) (0I32, 0I16, 0I32) (ff, tt))
```

has type `[«n4; I8», «n4; I16» ,«n4; I32»]`. Note that although the matrix dimensions are *not* compile-time constants, the dependent types keep track of the information that the rows of the input matrices as well as the lengths of the output vectors are of size `n4`. This information may be useful during vectorization, for instance, to remove edge cases. You can match applications of this axiom from within C++ for further analyses/optimizations:

```
if (auto reduce = match<tensor::reduce>(e)) { auto [is, init, dims] = reduce->args(); /*...*/ }
```

6.3.2 Autodiff Plugin. The `%autodiff` plugin implements reverse-mode automatic differentiation (AD)—a prominent technique to compute the derivatives of code. These derivatives are used to optimize parameters using gradient descent methods in, for example, machine learning frameworks.

A popular way to implement reverse-mode AD in functional languages is to use *backpropagators* [49]. Each function f is augmented to return a function f^*, the backpropagator, in addition to its original result (see Figure 5 for an example): $D\,f\,(x, x^*) \coloneqq (f\,(x), \lambda a.x^*\,(f'\,(x) \cdot a))$.

By this technique, AD effectively builds a list of backpropagators: The backpropagator of f uses x^* which is the backpropagator of the function that was used to compute the argument of f. In turn, f's backpropagator is passed to all functions that take the value f returned as an argument. The derivative is then computed by invoking the backpropagator of the end result with 1.

The `%autodiff` plugin (Figure 4d) expresses this transformation as a set of local rewrite rules that are applied in a bottom-up fashion to transform the original function into its derivative. Besides simple functions, the `%autodiff` plugin handles a rich set of features including non-scalar types, pointers, and higher-order, recursive functions. The advantage of the local-rewrite approach to AD is that it is simple to implement and modular in the sense that AD of a compound language element translates to differentiating its constituents. This modularity allows us to extend the implementation at a high level by just specifying the derivative of another operation. For example, the `%tensor` plugin specifies derivatives for common matrix operations without tainting the `%autodiff` plugin with dependencies on the `%tensor` plugin. By doing so, the plugin computes the derivative of

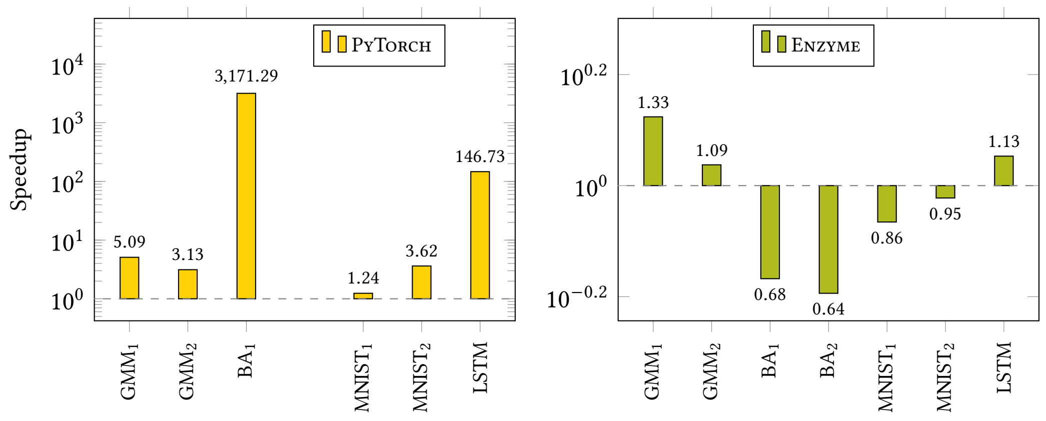


Fig. 6. Speedup t/t_{mim} of MimIR vs. Enzyme, and PyTorch

a matrix product using matrix products instead of low-level for-loops. The use of optimal code generation for these high-level operations results in a significant speedup of the resulting program—as demonstrated by Peng and Dubach [50].

The use of higher-order functions in the derivative computation makes it more challenging for the compiler to produce efficient code. However, MimIR's optimizer and partial evaluator are able to remove this overhead entirely in the benchmarks that we considered as our evaluation below shows. In particular, MimIR's design resolves many practical implementation problems basically for free: Most notably, hash-consing merges identical backpropagator invocations, and partially evaluating backpropagators removes most of the boilerplate that is introduced by the newly introduced nested function calls.

To show that MimIR is able to succinctly optimize the code that results from applying backpropagator-based AD, we compare it against the state-of-the-art AD frameworks PyTorch [48] and Enzyme [43]. PyTorch builds dynamic computation graphs via operator overloading in Python. Enzyme differentiates LLVM code during compilation.

Setup. We evaluate the frameworks on Microsoft's ADBench suite [57]. We use the Gaussian mixture model (GMM), bounded analysis (BA), and a long short-term memory neural network (LSTM) from the ADBench suite. Additionally, we compare the approaches on a network for the MNIST classification task [17]. For a fair comparison to Enzyme, we instruct the `%tensor` plugin to generate low-level, straightforward loop nests. Alternatively, the plugin could also employ highly tuned BLAS [10] routines, instead. But we are interested in how well the `%autodiff` plugin copes with low-level loop nests as this abstraction layer corresponds to the knowledge Enzyme obtains via LLVM's scalar evolution analyses.

6.3.3 Running Time. We ran the benchmarks using a single thread on an AMD Ryzen 7 5800X with 32 GB of DDR4@2400MT/s RAM. Figure 6 compares the speedup/slowdown of Enzyme/PyTorch compared to `%autodiff` as baseline. For the MimIR implementations, we write the benchmarks in Impala (Section 6.1.4) that compiles to MimIR and uses the `%autodiff` AD compiler pass. Finally, MimIR emits an LLVM file that we feed to Clang to generate an executable. Enzyme is an LLVM pass written in C++ while PyTorch is written in Python.

Our evaluation shows that Enzyme and `%autodiff` are comparable in performance. In some cases like bounded analysis, Enzyme has a better caching/recomputation balance resulting in lower runtime. In other cases like the GMM or LSTM benchmark, `%autodiff` is faster due to more caching. The main performance difference is due to caching or recomputation of intermediate results. Enzyme

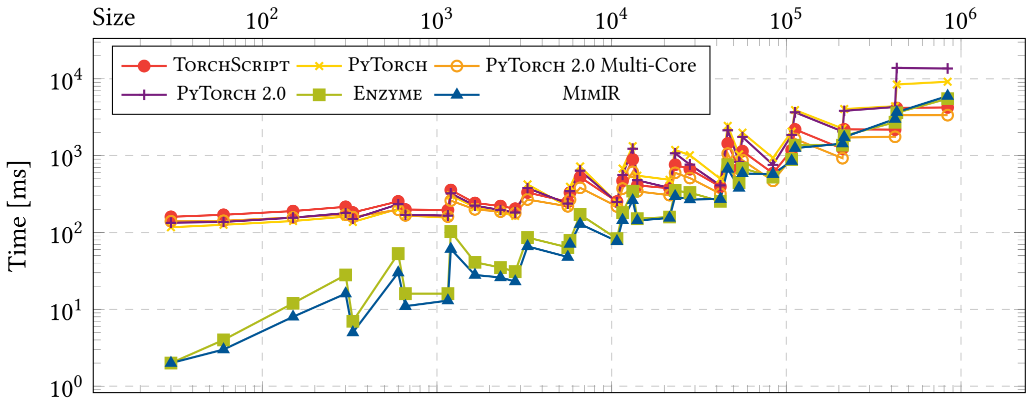


Fig. 7. Runtime of the Gaussian Mixture Model (GMM) algorithm on different input sizes (lower is better)

manages to better detect induction variables and recomputes them in the backward pass instead of storing them. On the other hand, it sometimes stores additional unnecessary intermediate results in the forward pass.

In our evaluation, PyTorch is slower than Enzyme and `%autodiff`. This is partly due to the overhead of constructing the backward graph at runtime, the overhead of the Python interpreter, more memory usage, and missing optimizations like inlining. PyTorch is inefficient in the bounded analysis benchmark resulting in a significantly longer running time. This slowdown is caused by the deeply nested function calls and loops that contain conditionals in the benchmark source code. The BA_2 test timed out, and thus, is not listed in Figure 6.

Table 5. LoC and cycl. complexity measure (lower is better). We compute the measure per function; *total* cyclomatic complexity refers to the sum over all files.

			Cyclomatic complexity	
		LoC [k]	total	avg.
Enzyme	w/	58.9	13,793	10.9
	w/o	49.4	11,857	12.8
`%autodiff`	w/	4.7	704	1.6
	w/o	1.3	141	1.8

w/: with extensions & plugins w/o: without extensions & plugins

TorchScript improves upon PyTorch by compiling parts of the code. The resulting speedup is especially visible for very large input sizes where TorchScript becomes one of the best implementations as shown in Figure 7. PyTorch 2.0 further refines the TorchScript approach of ahead-of-time compilation using TorchDynamo [3], TorchInductor, and AOTAutoGrad. For small inputs, the remaining overhead of Python, not compiled functions, and communication overhead still causes PyTorch 2.0 to be slower than Enzyme and `%autodiff`. One advantage of PyTorch is its support for utilizing multiple cores. In Figure 7, the PyTorch 2.0 Multi-Core variant shows how this notably helps PyTorch's performance on large problem sizes. Enzyme and `%autodiff` both do not use multiple cores yet.

6.3.4 Code Complexity. One major contribution of our approach to AD is its simplicity. In order to verify this claim, we estimate the code complexity of `%autodiff` and Enzyme using code complexity metrics. Table 5 summarizes the LoC and the *cyclomatic complexity* [39]. For Enzyme we measured the source folder. For better comparability, we also measured the metrics for Enzyme without the Clang plugin, the MLIR code, and without the different scalar evolution expander versions. The numbers without extensions refer to the plugin without the special casing of pointer arguments. Enzyme has roughly 10× more code than `%autodiff`. This does not necessarily give an indication of which code is simpler. But as a rule of thumb the larger a code base gets, the harder it becomes to debug and maintain. As a more profound code complexity metric, we also look at the *cyclomatic* *complexity* metrics[3] of both implementations. Enzyme's cyclomatic complexity is roughly an order of magnitude larger than `%autodiff`'s.

6.4 Other Plugins, Composition of Plugins, and Practical Experience

In addition, we have also created the following plugins: `%affine` to represent affine loop nests, `%clos` which implements a typed closure conversion [41], `%direct` which allows to invoke direct-style functions as continuations and vice versa, `%refly` which allows for introspection & reflection, `%compile` exposes MimIR's optimization framework as axioms, and `%opt` which implements MimIR's standard optimization pipeline as a Mim program.

We have already described the interaction of the `%autodiff` and the `%tensor` plugin in Section 6.3.2. In addition, the `%autodiff` plugin uses the `%affine` plugin for loops, the `%clos` plugin to introduce and eliminate closures, and the `%direct` plugin to seamlessly switch between direct style and CPS. The `%core`, `%math`, and `%mem` plugins use each other mutually and are used by all other plugins that need to generate low-level code including the `%regex`, `%tensor`, and `%autodiff` plugins.

During development, MimIR's type system was instrumental in identifying and addressing typical bugs such as incorrect wiring of expressions early in the implementation of our plugins (see Section 5.3). In addition, MimIR's support for polymorphism enabled a shift from C++ code generating MimIR to directly working with polymorphic, type-safe Mim (see Section 3.4.2). For example, many plugins implement small wrappers and glue code directly in Mim.

7 Related & Future Work

MimIR lies at the intersection of higher-order, dependent type theory and low-level compiler IRs and is to the best of our knowledge the first of its kind in this regard. We have already discussed MimIR's relationship to Thorin and MLIR in Section 2.6. This section discusses work that influenced MimIR and other related approaches.

Partial Evaluation & AnyDSL. Partial Evaluation filters were first introduced by SCHISM [15] and are also used by the modern partial evaluation framework AnyDSL [34]. AnyDSL has been successfully applied for high-performance applications such as sequence alignment [44, 45] or ray tracing [51]. AnyDSL's IR is Thorin [35] (see Section 2.6). AnyDSL relies on *shallow* DSL embedding and partial evaluation to remove the overhead DSL interfaces impose. Like AnyDSL, MimIR supports shallow DSL embedding but adds the possibility of *deep embeddings* via plugins (Section 2). Furthermore, MimIR is to the best of our knowledge the first system that employs filter-based partial evaluation to resolve β-equivalence during type-checking a dependently typed language.

Sea of Nodes. While the idea of a "sea of nodes" goes back to Click and Paleczny [13], Thorin pioneered this concept for higher-order languages. MimIR inherits this representation and simplifies the graph even further: Due to MimIR's roots in PTS, types—which are also just expressions—are part of the normal program graph as well.

Since the arrival of graph-based IRs, a debate has arisen among compiler engineers as to whether graph-based IRs or instruction lists are superior. Graph-based IRs are used, for example, in the HotSpot JVM [47], GraalVM [18], and Google's TurboFan compiler (part of V8). Graph-based IRs enable several optimizations directly during IR construction, such as constant folding, various

[3]We used metrix++ for the measurement.

arithmetic simplifications, or semi-global value numbering[4] through hash-consing. With graph-based IRs, operations track dependencies solely through data dependencies. This makes the graphs invariant to code motion and allows analyses to avoid dead code by only following data dependence edges. MimIR leverages these features for its normalization framework. Instruction lists, in contrast, are simpler to understand and straightforward to traverse. Visualizing a graph-based IR (especially if it is broken) requires external graph tools and specialized debugging infrastructure. In contrast, dumping an instruction list is straightforward—even if the program is incomplete or contains errors. When the original order of instructions is crucial (e.g., when handling side effects), graph-based IRs must make these dependencies explicit. This can feel cumbersome, but it also more accurate (see Section 2.3 and 6.1.3).

λ-Calculi & SSA. As a λ-calculus, MimIR takes inspiration from and shares similarities with many other λ-calculi—most notably: CC, λ-cube [7], PTS, Henk [40], and the zip calculus [65]. Since MimIR is based upon a predicative flavor of CC it also subsumes a predicative flavor of System F [23, 54] and System F_ω. Rossberg et al. [55] have shown that ML's module system can be encoded in System F_ω. Apart from impredicative idioms, MimIR can thus encode ML modules as well.

MimIR is also flexible enough to mimic various SSA flavors. We have already discussed in Section 3.2 how CPS makes MimIR akin to an SSA representation [5, 28]. MimIR can also model various extensions to SSA form such as *(thin) gated SSA* [25, 64], loop-closed SSA (LCSSA), or static single information (SSI) form [2]. It just depends on where additional variables are placed.

Maziarz et al. [38] present an algorithm to hash expressions modulo α-equivalence. This technique does not work for MimIR, however, as MimIR's program graph is mutable at very specific spots (Section 4). Moreover, the *assignable* relation checks for compatible tuple types (Section 3.1.4) and resolves implicits (Section 5.2) anyway. By doing this, the relation additionally checks for α-equivalence.

CPS vs. Direct Style. In the community for compilers of functional languages, there is a decades-old debate about whether to use CPS [4, 29, 31] or direct style [20, 37]. On the one hand, many optimizations such as simple rewrites like $x + 0 \triangleright x$ are much easier to implement in direct-style. On the other hand, we need CPS to model unstructured control flow and at the end of the day, a compiled program consists of basic blocks with machine instructions that jump to each other—which *is* CPS. Thus, we second the opinion of Cong et al. [14] that the question of whether to compile with or without continuations is more a matter of what a compiler engineer/language designer wants to achieve and the specific stage in the compilation pipeline. MimIR itself does not prioritize the use of CPS or direct style, offering syntactic sugar to accommodate both styles. The MimIR plugin `%direct` enables the invocation of direct-style functions as continuations and certain continuations as direct-style functions. For example, given `f` of type `Fn T → U`, the expression `%direct.cps2ds f` has type `T → U`. Here we noted similarities to negation in the work of Ostermann et al. [46]. In addition, the `%direct` plugin CPS-converts direct-style function to CPS because MimIR's LLVM backend expects CPS. Cong et al. [14] present a calculus that differentiates between first- and second-class continuations and a type system that ensures proper use. MimIR's `%clos` plugin performs a similar classification via static analysis, instead. This plugin transforms escaping continuations via typed closure conversion [41] but is limited to non-dependent function types. Bowman and Ahmed [11] present a technique to closure-convert dependently typed functions in CC.

Typed Assembly Language. Previous work on low-level types [42] enriched imperative assembly languages with a type system. Building up on this work, Dependently Typed Assembly Language

[4]This is *global* in the sense that instructions float beyond basic block boundaries but *local* as hashing stops at ϕ-functions.

(DTAL) [68] adds dependent types that are limited to linear constraints on integers while MimIR features full dependent types. DTAL uses an integer linear programming (ILP) solver to solve these constraints whereas MimIR uses normalization and partial evaluation filters to decide dependent type checking. These strategies are orthogonal to each other and in MimIR we could define a `%dtal` plugin that captures DTAL-like constraints in an axiomatized dependent type and connect to an ILP solver during normalization to find more program equivalences and, hence, emulate DTAL constraints.

Dependent Types. Dependent types give more freedom in program construction and allow to express semantic properties in the types of expressions. Although they have been mainly used in proof assistants such as Coq, Agda, or Lean in the past, they have received some attention beyond proof assistants in recent years: Dependent Haskell [66] is an extension of Haskell that allows combining types and expressions while preserving backward compatibility with Haskell. Scala 3 is based upon dependent object types (DOT) [1] which features *path-dependent types*. These are a specific kind of dependent type where the dependent-upon value is a path.

Idris focuses on type-driven development but, if desired, properties of program behavior can be formally stated and proven. MimIR similarly expresses semantics on the type-level using dependent-types but differs in the focus on optimization instead of verification. Whereas Idris is a high-level programming language, MimIR as an IR is much closer to the hardware. Idris 2 [12] introduces Quantitative Type Theory (QTT) [6]. This not only allows to specify protocols as types but also to tag which types should be erased to clearly state which parts of a program materializes at runtime. MimIR on the other hand, uses partial evaluation filters to resolve β-equivalence and make this distinction.

F* [60] combines automated theorem proving and interactive proving. The automation also aids when using refinement types as utilized in length-annotated arrays. MimIR's user experience is similar as the normalization approach handles dependent types without further complications. However, MimIR does not aim to be a theorem prover. Furthermore, MimIR uses general dependent types instead of refinement types. This approach allows for more freedom as is exemplified in the AD plugin that computes the result type at the type level. Furthermore, the developer can extend the normalization approach to handle more complex cases. One possible rule could optimize a double reversion `rev (rev xs)` directly to `xs`; a use case, many automatic systems have problems with.

Future Work. In the future, we want to make it possible to specify normalizations directly as rewrite rules in Mim. This further reduces boilerplate C++ code. Moreover, this opens the door for additional sanity checks and nondeterministic rule sets that we could explore with rewrite engines like equality saturation [61]. For example, map/reduce-style rewrite rules have already been used with equality saturation to produce high-performance code [30]. We are also working on support for singleton, union, and intersection types. Singleton and union types are useful in many settings, e.g., we could replace the type of the `mode` arguments in the `%core` and `%math` dialect with a proper union type instead of a **`Nat`**. Intersection types allow us to combine trait-like Σ-types such as `Cmp` ∩ `Num`. We also want to add linear types or full QTT in order to validate by the type checker whether values that are supposed to be used linearly are used correctly. For example, values of type `%mem.M`—which track side-effects—must be used linearly, but this is right now not enforced by the type system. Furthermore, we want to enable parallelization and accelerators, such as GPUs, to enhance higher-level DSLs with these mechanisms. Finally, we want to reimplement existing DSLs such as Lift [59] or RISE/Shine [58] in MimIR.

8 Conclusion

In this paper, we presented MimIR, an extensible, higher-order intermediate representation. MimIR's extensibility allows for expressing and optimizing programs at any level of abstraction. MimIR's foundation in the Calculus of Constructions provides a general and common type system for domain-specific languages to nest in. DSL authors benefit from reusing MimIR's type system, normalization, and optimization framework without having to provide manual type-checkers and reimplementing standard optimizations for their custom operations. We have shown that using MimIR, we can generate code with state-of-the-art performance for high-level, domain-specific applications such as a RegEx matcher, automatic differentiation, as well as for low-level, imperative code.

Data Availability

MimIR is available as open source on GitHub.[5] All data used in Section 6 is available at Zenodo [36].

Acknowledgments

The authors would like to thank the anonymous reviewers for their concise and constructive feedback, which has greatly contributed to the improvement of this paper.

References

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

[2] C. Scott Ananian. 1999. *The Static Single Information Form.* Ph. D. Dissertation. Princeton University.

[3] Jason Ansel. 2022. *TorchDynamo.* https://github.com/pytorch/torchdynamo

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

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

[6] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

[7] Henk Barendregt. 1991. Introduction to Generalized Type Systems. *J. Funct. Program.* 1, 2 (1991), 125–154. https://doi.org/10.1017/s0956796800020025

[8] H. P. Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science.* Vol. 2. Oxford University Press. http://citeseer.ist.psu.edu/barendregt92lambda.htmlandhttp://www.cs.ru.nl/~henk/PersonalWebpage http://citeseer.ist.psu.edu/barendregt92lambda.htmlElectronic Edition.

[9] Siddharth Bhat and Tobias Grosser. 2022. Lambda the Ultimate SSA: Optimizing Functional Programs in SSA. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 1–11. https://doi.org/10.1109/CGO53902.2022.9741279

[10] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[11] William J. Bowman and Amal Ahmed. 2018. Typed closure conversion for the calculus of constructions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 797–811. https://doi.org/10.1145/3192366.3192372

[12] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPICS.ECOOP.2021.9

[5] see https://anydsl.github.io/MimIR/

[13] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*. 35–49. https://doi.org/10.1145/202529.202534

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

[15] Charles Consel. 1988. New Insights into Partial Evaluation: the SCHISM Experiment. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings (Lecture Notes in Computer Science, Vol. 300)*, Harald Ganzinger (Ed.). Springer, 236–246. https://doi.org/10.1007/3-540-19027-9_16

[16] Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[17] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[18] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, Indianapolis, IN, USA, 28 October 2013*, Christoph Bockisch, Michael Haupt, Steve Blackburn, Hridesh Rajan, and Joseph Gil (Eds.). ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[19] Hana Dusíková. 2023. *Compile Time Regular Expression in C++*. https://github.com/hanickadot/compile-time-regular-expressions.

[20] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[21] Free Software Foundation. 2023. `std::regex` *from libstdc++ version 13.1.1*. https://gcc.gnu.org/onlinedocs/gcc-13.1.0/libstdc++/api/a07327.html.

[22] Syoyo Fujita. 2014. *aobench*. https://code.google.com/archive/p/aobench/.

[23] Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63–92. https://doi.org/10.1016/S0049-237X(08)70843-7

[24] Google Brain. 2023. *XLA: Optimizing Compiler for Machine Learning*. https://www.tensorflow.org/xla.

[25] Paul Havlak. 1993. Construction of Thinned Gated Single-Assignment Form. In *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 768)*, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua (Eds.). Springer, 477–499. https://doi.org/10.1007/3-540-57659-2_28

[26] Philip Hazel. 2021. *Perl-compatible Regular Expressions v2*. http://www.pcre.org/current/doc/html/pcre2.html.

[27] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*. Elsevier, 189–196.

[28] Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, Michael D. Ernst (Ed.). ACM, 13–23. https://doi.org/10.1145/202529.202532

[29] Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 177–190. https://doi.org/10.1145/1291151.1291179

[30] Thomas Koehler, Phil Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. *CoRR* abs/2111.13040 (2021). arXiv:2111.13040 https://arxiv.org/abs/2111.13040

[31] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. ORBIT: an optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, Richard L. Wexelblat (Ed.). ACM, 219–233. https://doi.org/10.1145/12276.13333

[32] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[33] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https:

//doi.org/10.1109/CGO51591.2021.9370308

[34] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: a partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 119:1–119:30. https://doi.org/10.1145/3276489

[35] Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015.* 202–212. https://doi.org/10.1109/CGO.2015.7054200

[36] Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2024. *MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age.* https://doi.org/10.5281/zenodo.13952579

[37] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. https://doi.org/10.1145/3062341.3062380

[38] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew W. Fitzgibbon, and Simon Peyton Jones. 2021. Hashing modulo alpha-equivalence. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 960–973. https://doi.org/10.1145/3453483.3454088

[39] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[40] E Meijer and Simon Peyton Jones. 1997. *Henk: a typed intermediate language* (types in compilation ed.). https://www.microsoft.com/en-us/research/publication/henk-a-typed-intermediate-language/

[41] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. 1996. Typed Closure Conversion. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996.* 271–283. https://doi.org/10.1145/237721.237791

[42] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1998. From System F to Typed Assembly Language. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 85–97. https://doi.org/10.1145/268946.268954

[43] William Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. *Advances in neural information processing systems* 33 (2020), 12472–12485.

[44] André Müller, Bertil Schmidt, Andreas Hildebrandt, Richard Membarth, Roland Leißa, Matthis Kruse, and Sebastian Hack. 2020. AnySeq: A High Performance Sequence Alignment Library based on Partial Evaluation. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020.* IEEE, 1030–1040. https://doi.org/10.1109/IPDPS47924.2020.00109

[45] André Müller, Bertil Schmidt, Richard Membarth, Roland Leißa, and Sebastian Hack. 2022. AnySeq/GPU: a novel approach for faster sequence alignment on GPUs. In *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, June 28 - 30, 2022*, Lawrence Rauchwerger, Kirk W. Cameron, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos (Eds.). ACM, 20:1–20:11. https://doi.org/10.1145/3524059.3532376

[46] Klaus Ostermann, David Binder, Ingo Skupin, Tim Sü berkrüb, and Paul Downen. 2022. Introduction and elimination, left and right. *Proc. ACM Program. Lang.* 6, ICFP (2022), 438–465. https://doi.org/10.1145/3547637

[47] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, Saul Wold (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[49] Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–36.

[50] Mai Jacob Peng and Christophe Dubach. 2023. LAGrad: Statically Optimized Differentiable Programming in MLIR. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction.* 228–238.

[51] Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: generating renderers without writing a generator. *ACM Trans. Graph.* 38, 4 (2019), 40:1–40:12. https://doi.org/10.1145/3306346.3322955

[52] Randy Pollack and Eric Poll. 1992. Typechecking in Pure Type Systems. In *Informal proceedings of Logical Frameworks'92.* 271–288.
[53] Dragomir Radev. 2008. *CLAIR collection of fraud email.*
[54] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard J. Robinet (Ed.). Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148
[55] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264
[56] Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. 2021. Real-time MLton: A Standard ML runtime for real-time functional programs. *J. Funct. Program.* 31 (2021), e19. https://doi.org/10.1017/S0956796821000174
[57] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. 2018. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software* 33, 4-6 (2018), 889–906.
[58] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. *CoRR* abs/2201.03611 (2022). arXiv:2201.03611 https://arxiv.org/abs/2201.03611
[59] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 74–85. http://dl.acm.org/citation.cfm?id=3049841
[60] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655
[61] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 264–276. https://doi.org/10.1145/1480881.1480915
[62] Benchmarksgame Team. 2023. *The Computer Language Benchmarks Game.* https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
[63] 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
[64] Peng Tu and David A. Padua. 1995. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, David W. Wall (Ed.). ACM, 47–55. https://doi.org/10.1145/207110.207115
[65] Mark Tullsen. 2000. The Zip Calculus. In *Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings.* 28–44. https://doi.org/10.1007/10722010_3
[66] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP (2017), 31:1–31:29. https://doi.org/10.1145/3110275
[67] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. 2013. *Compiler Design - Syntactic and Semantic Analysis.* Springer. https://doi.org/10.1007/978-3-642-17540-4
[68] Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 169–180. https://doi.org/10.1145/507635.507657
[69] Jianhong Zhao, Jinhui Kang, and Yongwang Zhao. 2024. K-CIRCT: A Layered, Composable, and Executable Formal Semantics for CIRCT Hardware IRs. *CoRR* abs/2404.18756 (2024). https://doi.org/10.48550/ARXIV.2404.18756 arXiv:2404.18756

Received 2024-07-10; accepted 2024-11-07