The Design and Implementation of the Wolfram Language Compiler

Abdul Dakkak
University of Illinois,
Urbana-Champaign
Champaign, Illinois, USA
dakkak@illinois.edu

Tom Wickham-Jones
Wolfram Research
Oxford, UK
twj@wolfram.com

Wen-mei Hwu
University of Illinois,
Urbana-Champaign
Champaign, Illinois, USA
w-hwu@illinois.edu

Abstract

The popularity of data- and scientific-oriented applications, abundance of on-demand compute resources, and scarcity of domain expert programmers have given rise to high-level scripting languages. These high-level scripting languages offer a fast way to translate ideas into code, but tend to incur a heavy performance overhead. In order to alleviate the performance penalty, each implementation of these languages often offers a compilation path to a subset of the language.

In this paper we present the design and implementation of the Wolfram Language compiler, the production compiler for the Wolfram Language. We show how popular language features and runtime behavior, expected by Wolfram Language developers, are efficiently implemented within the compiler. We then show how the compiler provides a friction-less path to migrate programs from the interpreter to the compiler. We evaluate the compiler and show that compiled code matches the performance of highly tuned hand-written C code. The compiler has been released as a prominent feature of Wolfram Engine v12 and is readily available to developers.

CCS Concepts. • **Software and its engineering** Compilers; *Software performance*; • **Mathematics of computing** Mathematical software.

Keywords. Wolfram Compiler, Mathematica, Type Inference

ACM Reference Format:
Abdul Dakkak, Tom Wickham-Jones, and Wen-mei Hwu. 2020. The Design and Implementation of the Wolfram Language Compiler. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3368826.3377913

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377913

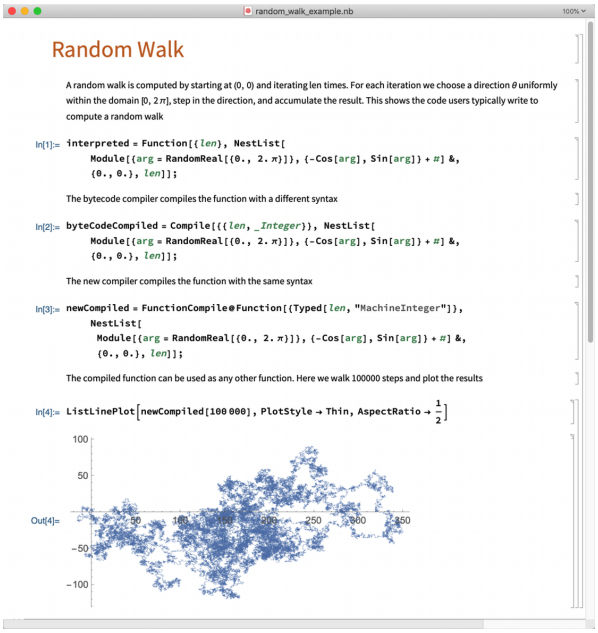

Figure 1. A typical Wolfram Notebook session. Users intermix documentation with code to develop computation Notebooks. The Notebook is interactively evaluated. This Notebook defines a random walk function, described in Section 2.1, using the Wolfram interpreter (In[1]), bytecode compiler (In[2]), and the new compiler (In[3]). The results are then plotted (In[4]).

1 Introduction

The abundance of on-demand computer resources and the shortage of programmers have given rise to modern scripting languages such as Python, R, and the Wolfram Language. These scripting languages are interpreted and generally favor ease of use and productivity over performance, allowing fast translation from ideas to prototype implementations. However, because of the interpreted nature of these languages, performance is sacrificed. Code that worked well solving small scale problems tends to break down as the problem sizes grow. This is mainly because of interpreter overhead [43, 72]. Therefore, it is common for these scripting language to have a mechanism which omits certain language features to enable them to be compiled or to be interpreted

with reduced overhead (e.g. Cython [7, 66] or Numba [49] for Python, RCPP [26] for R, and Compile [84] for the Wolfram Language).

The Wolfram Language [8, 48, 88, 102], first released in 1988 [90, 97], is an interpreted language which is widely used within both research and industry to solve engineering, mathematical, and data science problems. While initially targeted towards mathematicians and physicists as a computer algebra system, it has matured into a fully fledged language which supports computational domains such as: algebra, number theory, machine learning, graph theory, statistics, computer vision, differential equations, etc. Programmers tend to use Wolfram Notebooks [99, 103] as their programming interface to develop and evaluate code (shown in Figure 1). The language design [101] is unique in that it has wide coverage of interoperable computational domains, blends symbolic and numerical methods, offers many high-level programming constructs and supports a wide range of visualization modalities. For example, a program which generates a 10×10 matrix of random variates sampled from a normal distribution and sums each row is written[1] as `Total[RandomVariate[NormalDistribution[], {10, 10}]]`. Using the Wolfram Language to write programs makes development easier, natural, succinct, and efficient, boosting programmers productivity.

To decrease the interpreter overhead, a lightweight bytecode compiler was introduced in the Wolfram Language V2 [87]. This bytecode compiler supports around 200 commonly used functions (mainly numerical computation for linear algebra, differential equations, and optimization). The generated bytecode is run within a stack-based [68] Wolfram Virtual Machine (*WVM*).

There are two modes of operation for the bytecode compiler. The first is when a user explicitly invokes the compiler using the `Compile` keyword. Figure 1 shows that minor modification are needed to explicitly call the compiler (In[2]) on an interpreted function (In[1]). The bytecode compiled function achieves a 2× speedup over the interpreted function when the input length is 100, 000. The second mode of operation is when a high level function invoke the compiler internally. This compilation is implicit to the user and we call this *auto compilation*. Many numerical functions such as `NMinimize`, `NDSolve`, and `FindRoot` perform auto compilation implicitly to accelerate the evaluation of function calls. Code such as `FindRoot[Sin[x] + E^x, x, 0]` which finds the root of the equation $sin(x) + e^x$ around 0 performs auto compilation of the input function (i.e. `Sin[x] + E^x`) and achieves a 1.6× speedup over an uncompiled version. Because of the performance boost, the bytecode compiler is popular among users.

[1]Whenever possible, we use full Wolfram programs as examples. As the function definitions and syntax may not be familiar to the reader, all text colored `blue` are linked to the corresponding symbol's documentation page.

While the bytecode compiler has proved to be popular, it has some inherent limitations in terms of: **L1 Expressiveness** — many functions within the Wolfram Language cannot be compiled; e.g. functions operating on strings, symbolic expressions, etc. **L2 Extensibility** — users are unable to extend the compiler by adding datatypes, function definitions, or optimizations. **L3 Performance** — the performance of the compiled code is limited by bytecode interpretation/JIT cost. Furthermore, compile time and bytecode execution time degrade as the number of supported bytecode instructions increases. Aside from the design issues, implementation choices has hindered: **L4 Development** — the compiler is written in C and therefore is inaccessible to Wolfram Language developers. **L5 Distribution** — the compiler is implemented as part of the Wolfram Engine and cannot be distributed independently, which means that each update of the compiler requires an update release of the entire Wolfram Engine. These design and implementation choices limit the bytecode compiler's wider adoptions and slow down its development.

As a result, a new compiler was proposed to address these design and implementation issues. During the new compiler design, we identified useful features and behaviors that we want to carry over from the bytecode compiler. The bytecode compiler is **F1** hosted within the Wolfram Language interpreter which allows Wolfram programs to **intermix interpreted and compiled code**. The bytecode compiler has a **F2 soft runtime failure** mode; runtime errors (such as numerical overflows) can be handled by re-evaluating the code using the interpreter as a fallback. Bytecode compiled functions are **F3 abortable** by users and do not exit the Wolfram Engine process, and do not pollute the interpreter's state. The bytecode compiler **F4** supports **multiple backends**, such as a C backend, which makes it possible to export compiled functions for usage within other applications.

This paper describes our experience in designing a new Wolfram Language compiler as shipped within Wolfram Engine *V*12 [86, 96, 98]. The new design addresses limitations **L1–5**, maintains features **F1–4**, and supports additional capabilities (shown in Table 1) that would otherwise not be possible within the bytecode compiler. We designed the compiler as a staged pipeline [50, 64] and implemented it using the Wolfram programming language as a distributable package. Developing within the compiler is simple: as users can write pipeline stages using the Wolfram Language and inject them into any stage within the pipeline at runtime. To allow for more expressiveness, a type system [17, 52, 53] with wider support for datatypes has been introduced. To increase the generality of user defined functions, functions are defined polymorphically and can be type qualified [44, 45]. Users can compile a larger subset of the Wolfram Language which includes operations on strings and symbolic expression. Finally, the compiler JITs the compiled functions into native assembly and thus avoids degradation of performance

due to bytecode interpretation. While the compiler targets the Wolfram Language, the design of the compiler and its solutions to these challenges are applicable to other languages and compilers (see Section 4.8).

This paper describes key components of the new Wolfram Language compiler and is structured as follows. Section 2 first overviews the Wolfram Language and describe how the language and programming environment impacts the compiler's design. Section 3 describes the objectives and constraints of the new compiler. Section 4 proposes a new compiler which addresses these challenges. Sections 5 and 6 describe the implementation and evaluation of the new compiler. Section 7 details previous work and we conclude in Section 8.

2 Background and Motivation

This section will give a taste of the Wolfram Language and describe how it impacts the new compiler's design. We will then describe the previous bytecode compiler and Wolfram Notebook programming environment.

2.1 Language Overview

The Wolfram Language is a dynamic, untyped language that was first released in 1988. The language [89] is interpreted within the Wolfram Engine and provides an ecosystem for developing applications that leverage knowledge bases, visualization, symbolic reasoning, and numerical computation. The language is well documented [1, 81, 83, 89] and is used globally by educators, students, researchers, scientists, financial analysts, and mathematicians. The breadth of its features makes it a powerful and wide reaching tool for solving many technical and computation problems. To aid in understanding the rest of the paper, we will describe some basics of the Wolfram Language. Consider the Fibonacci function which can be defined[2] recursively by:

```
fib = Function[{n}, If[n < 1, 1, fib[n-1]+fib[n-2]]]
```

This statement defines a variable `fib` and binds it to function with a single argument `n`. When the function is called, e.g. `fib[10]` (or using the function application `fib@10` shorthand), the body of the function (the `If` statement) is executed, which causes further recursive executions of the `fib` function. A slightly more complicated example is the random walk function shown in Figure 1 (In[1]) and repeatedly here:

```
interpreted = Function[{len},
  NestList[
    Module[{arg = RandomReal[{0, 2π}]},
      {-Cos[arg], Sin[arg]} + #
    ]&,
    {0, 0},
    len
  ]
];
```

[2]A function f can also be defined as a list of `DownValues` rules that transform its arguments.

This function defines a new variable `interpreted` and binds it to a function with one argument. The function body invokes `NestList` which is a high level primitive (similar to fold in other languages) which takes an input function (first argument) and iteratively constructs a list by applying it repeatedly to the initial input (second argument) n-times (third argument) — returning a new list of length $n + 1$. In each iteration, the output of the function in the previous iteration becomes the input of the function for current iteration; `NestList[f,x,3]` evaluates to `{x,f[x],f[f[x]],f[f[f[x]]]}`. In this example, the input function to `NestList` is defined using the shorthand notation for `Function` (`&`) and takes an argument which is substituted into the `Slot` (`#`). Within this inner function, `Module` is used to introduce a new scoped variable `arg`. The `arg` variable is set to a random real value uniformly selected between 0 and 2π. The inner function uses the scoped variable to construct the list `{-Cos[arg], Sin[arg]}` and add it to the inner function's input.

The examples demonstrate important principles of the language. We highlight these along with other language features:

Untyped, Functional, and High-Level – The language requires no type annotations and is functional. The language provides high-level primitives and users tend to steer away from writing generic `For` loops and instead use high-level constructs such as: `NestList`, `FixedPoint`, `Map`, `Select`, `Fold`, and `Table`.

Symbolic Computation – Unlike traditional languages, the Wolfram Language supports symbolic computation. A program such as `Sin[x]` is a valid symbolic expression; even if `x` is never defined. Wolfram programs leverages the combination of symbolic and numeric support to enable novel design decisions and use cases. For example, to find the root of the equation $sin(x) + e^x$ around 0 one writes: `FindRoot[Sin[x] + E^x, {x, 0}]`. The root solver symbolically computes the derivative of the input equation and uses Newton's method to find the root at $x \approx -0.588533$ numerically.

Infinite Evaluation – Expressions are evaluated infinitely[85, 92] until either a fixed point or recursion limit is reached. Consider the program `y=x;x=1;y`. The value of `y` in another language (such as Python or R) would be undefined or an error, but in the Wolfram Language the result is `1`, since `y` evaluates to `x`, then `x` evaluates to `1`, then `1` evaluates to itself — reaching a fixed point. As a consequence `x=x+1` results in an infinite loop if `x` is undefined, since the following `x→x+1→(x+1)+1→((x+1)+1)+1→...` rewrite occurs.

2.2 ByteCode Compiler

A lightweight bytecode compiler [93] has been bundled with the Wolfram Engine since Version 2. Although limited, the bytecode compiler has proved to be very popular — being

used by users to improve the performance of their code. This section describes the bytecode compiler.

Modes of Operations of the ByteCode Compiler – The bytecode compiler targets numerical code, and translates a subset of the language (around 200 functions) into bytecode instructions which are then run within the Wolfram Virtual Machine. As described in the introduction, the bytecode compiler is used either implicitly or explicitly. The bytecode compiler is explicitly invoked using the `Compile` keyword. For example, Figure 1 (In[2]) shows that structural modifications are needed to explicitly compile the function using the bytecode compiler — namely changing `Function[{len}, ...]` into `Compile[{{len, _Integer}}, ...]`. The `Compile` inputs can be typed, otherwise they are assumed to be `Real`. The bytecode compiler is also implicitly used by users. Numeric functions such as `FindRoot[Sin[x] + E^x, x, 0]` automatically invoke the bytecode compiler to compile the input equation (as `Compile[{{x, _Real}}, Sin[x]+E^x]`) along with its derivative (`Compile[{{x,_Real}}, Cos[x]+E^x]`). The bytecode compiled version of these functions are then internally used by these numerical methods.

Internals and Flow – By default, the bytecode compiler targets the Wolfram Virtual Machine. When a function is explicitly compiled, e.g.: `cf = Compile[{{x, _Real}}, Sin[x]+E^x]`, the bytecode compiler first performs optimizations on the AST, such as common sub-expression elimination [3, 20]. The optimized expression is then traversed in depth-first order to construct the bytecode instructions. If an expression is not supported by the compiler, then the compiler inserts a statement which invokes the interpreter at runtime to evaluate that expression. Along the way, the compiler propagates the types of intermediate variables and any unknown type is assumed to be a `Real`. Finally, register allocation is performed to reduce the total number of virtual machine registers required. A developer can examine the internal full serialized representation of the `cf` compiled function by looking at its `InputForm`:

```
Out[●]//InputForm=
CompiledFunction[
  {11, 12, 5468},(* Compiler, Engine Version, and Compile Flags *)
  {_Real},                        (* Input Arguments *)
  {{3, 0, 0}, {3, 0, 1}},         (* Init Instructions *)
  {},                             (* Constant Variables *)
  {0, 0, 3, 0, 0},                (* Register Allocations *)
  {
   {40, 1, 3, 0, 0, 3, 0, 1},     (* Sin Op *)
   {40, 32, 3, 0, 0, 3, 0, 2},    (* Exp Op *)
   {13, 1, 2, 1},                 (* Plus Op *)
   {1},                           (* Return *)
  },
  Function[{x}, Sin[x] + E^x],    (* Input Function *)
  Evaluate
]
```

The `CompiledFunction`'s compiler and engine versions are checked when the compiled functions is run. If the versions do not match the current environment, then code is recompiled using the input function. Functions that fail to compile, or produce a runtime errors, are run using the interpreter.

Design Limitations of the ByteCode Compiler – The bytecode compiler is designed as a single forward monolithic transformation with fixed optimizations and datatypes. The monolithic design [6, 77] means that adding optimizations requires working knowledge of the compiler's internals. The compiler only supports a handful of datatypes that cannot be modified or extended by users and are: machine integers (`int32` on 32-bit systems and `int64` on 64-bit systems), reals, complex numbers, tensor representations of these scalars, and booleans. This means that performance is impacted due to wasted memory when trying to use compiled function on unsupported, small datatypes (`int8`, `float16`, etc.). The lack of expressive datatypes is also partially responsible for compilable functions being mostly confined to numerical functions. These design decisions has contributed to the bytecode compiler's limitations in **L1 Expressiveness**, **L2 Extensibility**, and **L3 Performance**.

Implementation Limitations of the ByteCode Compiler – Aside from the design issues, there are notable implementation issues with the bytecode compiler. The compiler is written in `C` and is inaccessible to Wolfram Language developers. The compiler is also heavily intertwined with the Wolfram Engine. Therefore it is not possible to distribute the bytecode compiler independently, and updates must coincide with updates to the Wolfram Engine (typically yearly). These implementation choices have slowed down rapid **L4 Development** and **L5 Distribution** of the compiler.

2.3 Programming Environment

There are two ways of programming with the Wolfram Language. IDEs and text editors [55, 65, 104] are used when developing large or non-interactive code. The Wolfram Notebook is targeted towards interactive scripting and prototyping where users write sharable computational documents containing a mixture of text, programs, data, and visualizations. The Wolfram Notebook offers a non-traditional programming interface and presents unique challenges to the compiler design such as: sessions cannot crash, code must be abortable, etc.

A typical Wolfram Notebook is shown in Figure 1. The notebook is divided into multiple cells. Code cells are labeled with In[n] where n represents the sequence of evaluation of the cells. Cells do not need to be evaluated sequentially. A Code cell can have a corresponding Output cell, labeled Out[n]. Figure 1 shows that the output of the In[4] (the `ListLinePlot[...]` statement) is Out[4] (a graphics plot). The other cells shown in the figure are Text cells which are used to document the surrounding code.

Table 1. Features and objectives of the new compiler. The ⋆ entries denote limited or inefficient support in the bytecode compiler.

	Objectives	New Wolfram Compiler	Bytecode Compiler
F1	Integration with Interpreter	✓	✓
F2	Soft Failure Mode	✓	✓
F3	Abortable Evaluation	✓	✓
F4	Backends Support	✓	⋆
F5	Mutability Semantics	✓	⋆
F6	Extensible User Types	✓	✗
F7	Memory Management	✓	⋆
F8	Symbolic Compute	✓	✗
F9	Gradual Compilation	✓	✗
F10	Standalone Export	✓	⋆

3 Design Objectives and Constraints

The compiler is designed to maintain features found within the bytecode compiler and solve its design and implementation limitations (See Table 1).

F1 ***Integration with Interpreter*** – Like the bytecode compiler, the new compiler must provide a seamless integration when used in the Wolfram interpreter. This means that the new compiler must conform to the same programming experience offered by the interpreter and compiled functions can be used as if they were any other Wolfram Language function.

F2 ***Soft Failure Mode*** – More than anything, Wolfram code should evaluate correctly. For example, in the event of numerical expression evaluation overflow at run time, the Wolfram interpreter automatically switches to arbitrary precision arithmetic [70]. The Wolfram Virtual Machine mimics that, by detecting runtime errors and reverting to the interpreter's for arbitrary precision arithmetic. The new compiler is expected to have the same behavior.

F3 ***Abortable Evaluation*** – The Wolfram Notebook is a prevalent way of writing Wolfram code (see Section 2.3). It provides a novel interactive programming session where code is developed and evaluated iteratively. Inevitably, however, evaluating an expression might take a long time; either because the code has bad complexity or is an infinite loop. To abort an "infinite" evaluation, quitting the notebook is not an option, since it means losing the accumulated session state. Therefore, both the interpreter and the bytecode compiler have a mechanism to abort evaluation without exiting the notebook. To see that, consider evaluating the infinite loop `i=0;While[True,If[i>3,i--,i++]]`. If a user issues an abort, execution of the `While` expression should be aborted and the interpreter prompt returned to the user. The returned session state must be usable but it may be mutated by the aborted computation; e.g. the variable `i` in the previous program. Internally, a user triggers an abort interrupt which causes the interpreter to abort the evaluation of the current expression. The new compiler should maintain this behavior.

F4 ***Backend Support*** – The current bytecode compiler is only able to target the Wolfram Virtual Machine or `C` language. Since Wolfram code can be used within the cloud [99], HPC systems [94], or micro-controllers [95], the new compiler is expected to support diverse backends. E.g. The compiler should support the option of compiling cloud deployed Wolfram Language functions into JavaScript or WebAssembly and run the deployed function on the client side rather than the server side.

F5 ***Maintain Mutability Semantics*** – All Wolfram Language expressions are immutable; except for symbols. For example, operations that modify expressions operate on a copy. Consider the following function application: `({#,StringReplace[#,"foo"→"grok"]}&)["foobar"]`. The result, `{"foobar", "grokbar"}`, is a copy and the original input string (`"foobar"`) is not mutated. Symbols are mutable however. E.g. `a="foo";a="bar"`. Data pointed to by the symbol can also be mutated. E.g. `a={1,2,3};a[[3]]=-20;a` would output `{1,2,-20}`. Note that this does not affect any other references to the data. E.g. `a={1,2,3};b=a;a[[3]]=-20;b` would output `{1,2,3}`. The Wolfram interpreter uses a reference counting mechanism to determine if copying is needed and the bytecode compiler performs copying on read. Too much copying can be a major performance limiting factor for bytecode compiled functions. The new compiler should maintain the mutability semantics in an efficient way.

F6 ***Extensibility of Types*** – As described in Section 2.2, the bytecode compiler has limited datatype. These fixed datatypes make it impossible to express functional programs. For example, the following cannot be compiled using the bytecode compiler `Function[{i,v},Module[{f=If[i≡0,Sin,Cos]},f[v]]]`, since it has no way to represent function types. Furthermore, there is no mechanism within the bytecode compiler for users to extend these datatypes. To alleviate such limitation, the new compiler design is expected to support extensible types and allow users to define their own.

F7 ***Automatic Memory Management*** – The Wolfram Language uses an automatic memory management scheme to manage the lifetime of expressions. Internally, expressions maintain a reference count and are freed when their count reaches 0. Since expressions are passed in and out of the compiler, it is important for the reference count and lifetime to be maintained consistently. The new compiler design is expected to maintain equivalent memory management semantics.

F8 ***Symbolic Computation*** – Some operations are more natural to be performed symbolically. Because the bytecode compiler lacks support of expression datatypes, it does not support symbolic computation. There is an urging need to support it however, since it allows one to represent equations (e.g. for use within numerical solvers) or to manipulate expressions.

F9 ***Gradual Compilation*** – Because the Wolfram Language's long history, users have amassed a large amount of code. Therefore it is not realistic to expect users to switch all their code to utilize the compiler. Both compiled and interpreted code thus are expected to intermingle. The new compiler must provide a bridge between interpreted and compiled code where compiled functions can invoke the interpreter to interpret parts of the code. This feature is analogous to gradual typing [69, 75].

F10 ***Standalone Export*** – The bytecode compiler provides a pathway to export bytecode compiled code into a standalone `C` program. The `C` program can then be linked with external applications without requiring the Wolfram Engine. This feature is popular, since it means that users do not have to rewrite their Wolfram Language code and avoids dependence on the Wolfram Engine. The new compiler design must support this feature and extend it to support other export targets.

4 Compiler Design and Implementation

The new compiler is designed to address the limitations of the existing bytecode compiler, allow for easy extensibility, and facilitate the compilation of large programs. As a result, the Wolfram compiler is written as a package within the Wolfram Language. It follows a traditional compiler pipeline design: MExpr → WIR → TWIR → code generation. Where *MExpr* is the Wolfram Language AST, *WIR* is an untyped Wolfram compiler IR, and *TWIR* is a typed-annotated Wolfram compiler IR. All analysis and transforms are expressed either as AST or IR passes. We highlight important analysis and transformation passes and how they solve the objectives laid out is Section 3.

4.1 Wolfram Language Extensions

Since the Wolfram Language is untyped, the first challenge is retrofiting [52, 53, 76] type annotation into the Wolfram Language. The bytecode compiler has a design, which relies on patterns, for specifying types. This pattern-based approach introduces ambiguity for polymorphic types and does not support user types. Instead of patterns, we introduce a `Typed` construct to the language along with an unambiguous type specification (see Section 4.4).

Another language extension is to enable users to explicitly invoke the new compiler. The bytecode compiler is explicitly invoked by using `Compile` in place of `Function`. This is not an ideal design, since it requires users to rewrite their code and introduces a new function constructor which does not have full feature parity with the existing `Function` constructor (e.g. `Compile` has no syntax for representing pure functions). Instead of proposing a new function constructor, the new compiler opts to maintain the use of `Function` and performs explicit compilation when a function is wrapped with `FunctionCompile`. For example, the `fib` (defined in Section 2.1) can be compiled using:

```
In[1]:= cfib = FunctionCompile[
          Function[{Typed[n, "MachineInteger"]},
            If[n < 1,
              1,
              cfib[n-1] + cfib[n-2]
            ]
          ]
        ]
```

4.2 Compiler AST Representation and Passes

`FunctionCompile` first represents the input program as an AST datastructure called *MExpr*. This is not difficult in the Wolfram Language, since programs can already be captured as inert expressions. We designed the MExpr datastructure for the compiler. MExprs are constructed from Wolfram inert expressions. Similar to Wolfram Language expressions, MExpr is either an atomic leaf node (representing a literal or `Symbol`) or a tree node (representing a `Normal` Wolfram expression) and can be serialized and deserialized. Arbitrary metadata can be set on any node within the AST.

Transformations and analyses are carried out on the AST using either the macro system or the MExpr visitor API. We highlight how the macro pattern-based substitution system simplifies desugaring and optimizations of ASTs, and how MExpr visitor API are used to analyze scoped variables.

Macro Based Desguaring and Optimization – While compiler developers can write AST transformations manually, using pattern-based macro substitution is more succinct and natural to Wolfram developers [91]. The macro system mimics the existing pattern-based expression substitution system within the Wolfram Engine with the key distinction being that substitution is hygienic [2, 19, 23, 24, 31]. Macro substitution has two aims: to desugar high-level constructs to their primitive forms and perform some "always-safe" AST-level optimizations. Macros are evaluated in depth-first order and terminate when a fixed point is reached. To demonstrate the macro system, consider the desugaring of the *n*-ary `And` (`&&`) function as implemented within the compiler:

```
RegisterMacro[macroEnv, And,
  (*1*) And[x_] → x≡True,
  (*2*) And[False, _] → False,
  (*3*) And[_, False] → False,
  (*4*) And[True, rest__] → And[rest],
  (*5*) And[x_, y_] → If[x≡True, y≡True, False],
  (*6*) And[x_, y_, rest__] → And[And[x,y], rest]
]
```

As shown, macros are registered within an environment (a default environment bundled by the compiler). The `And` macro has multiple rules that are matched based on the rules' pattern specificity and adhere to the Wolfram pattern ordering [91]. The first `(*1*)` macro rule desugars a unary `And` function into an equality check. The second and third rules are AST-level optimizations to replace the `And` with

`False` if one of its arguments is `False`. The fourth rule skips the first argument if it is `True`. The fifth rule performs short circuiting on the `And` function by desugaring it into an `If` condition. The sixth rule desugars an *n*-ary `And` into a nested binary `And`.

MExpr Visitor API: Binding Analysis – An important AST analysis is the handling of scoped variables [9, 30, 61]. The Wolfram Language has various scoping constructs (`Module`, `Block`, `With`, etc.), each of which has slightly different semantics. The binding analysis uses the MExpr visitor API to traverse all scoping constructs within the MExpr. It then adds metadata to each variable and links it to its binding expression. Along the way, the MExpr is mutated and all scoping constructs are desugared, nested scopes are flattened out, and variables are renamed to avoid shadowing. For example code such as `Module[{a=1,b=1},a+b+Module[{a=3},a]]` is rewritten to `Module[{a=1,b=1,a1=3},a+b+a1]`. By flattening the scopes, subsequent analysis and transformations on the MExpr are greatly simplified. Escape analysis [18] is also performed as part of the binding analysis. Escaped variables are annotated and are used during closure conversion [35, 73].

4.3 Untyped Wolfram IR Representation and Passes

The second phase of compilation is lowering the MExpr into untyped Wolfram Intermediate Representation (*WIR*). At the onset, a decision was made to develop a new IR. The rational is three fold: ① we want the IR to have a symbolic Wolfram representation, ② we want the IR to represent both typed and untyped code, and ③ we want to be able to attach arbitrary metadata to each IR node. Existing IRs would either be limiting or introduce external dependencies — complicating the design and implementation.

The WIR structure is inspired by the LLVM IR [50]. A sequence of instructions form a basic block, a DAG of basic blocks represent a function module, and a collection of function modules form a program module. Other elements such as: global program metadata, global variables, constant definitions, and type declarations of external functions are placed into the program module.

Lowering MExpr to WIR – Unlike LLVM Clang, which lowers all local variables into stack loads and stores — relying on an additional pass to promote variables from the stack to virtual registers [22] —, the compiler lowers MExprs directly into SSA form [15]. During lowering, if an IR node (variable, instruction, or function) has direct correspondence to an MExpr, then the MExpr is added as a property to the IR node. The MExpr properties are used during error reporting and, later on, to generate debug symbols.

Going straight to SSA form simplifies analysis, and WIR passes maintain the SSA property[3]. Absent type information, not all optimizations are safe on the WIR. For example, it is not generally valid to perform copy propagation, since it may affect the language's mutability semantics if the propagated copy is an expression. Still, there are many optimizations and analysis that are valid on the WIR. Optimizations on the control flow graph (dead-branch deletion, basic block fusion, etc.) or properties/analysis (dominator [21], loop-nesting [13, 62], liveness [12], etc.) are safe to perform on the WIR.

[3] An IR linter exists to check if the SSA property is maintained when writing passes.

4.4 Compiler Type System and Inference

The Wolfram Language is untyped, but type information is needed to generate fast code. Extensions to the language syntax were made to retrofit a type specification within the existing Wolfram Language (see Section 4.1). While every expression within a Wolfram program can be type annotated, this is not idiomatic of Wolfram code and places a significant burden on developers. Therefore, the compiler is designed to only require the minimal number of type annotations and infers the rest. The compiler stores all types and function definitions within a type environment and allow for F6 user-defined datatypes. Type classes [36, 40] are used to group types implementing the same methods (`"Integral"`, `"Ordered"`, `"Reals"`, `"Indexed"`, `"MemoryManaged"`, etc.).

Type System Specification – Any expression within a Wolfram program can be type annotated. A type can be specified on an expression either through the `Typed[expr, "ty"]` syntax or an expression decorator `Typed["ty"][expr]` syntax (or the `Typed["ty"]@expr` shorthand notation). The type specification (`TypeSpecifier`) supports both monomorphic and polymorphic types. The type classes can be used as qualifiers [44] for the polymorphic types. A `TypeSpecifier` can be:

- **Atomic Constructor**: `"Integer8"`, `"Real64"`, etc.
- **Compound Constructor**: e.g. `"Tensor"["Integer64", 2]` – a rank 2 tensor with `"Integer64"` element type.
- **Literal**: e.g. `TypeLiteral[1, "Integer64"]` – a type-level constant of value 1 and of type `"Integer64"`.
- **Function**: e.g. `{"Integer32", "Integer32"}→"Real64"` – a function with two `"Integer32"` arguments and a `"Real64"` return.
- **Polymorphic Function**: e.g. `TypeForAll[{"α"}, {"α"}→"Real64"]` – a parametric function with any `"α"` type as argument which returns a `"Real64"`.
- **Qualified Polymorphic Function**: e.g. `TypeForAll[{"α"}, {"α"∈"Integral"},{"α"}→"Real64"]` – a parametric function with any `"α"` argument type which implements the `"Integral"` type class and returns a `"Real64"`.

Types can be arbitrarily nested. For example, the type `TypeSpecifier[TypeForAll[{"α","β"},{{"α", "β"}→"β","Tensor"["α",1]}→"Tensor"["β",1]]]` is valid, and in fact is one of the definitions of `Map`. The compiler also supports other

type constructs such as `TypeProduct` and `TypeProjection`, which are used to handle structural types.

Type Inference – The compiler leverages type inference [56] to minimize the number of type annotations that needs to be explicitly written by users. In fact, it is enough to specify the input type arguments to a function. The types of all other variables within the function are inferred. Type inference occurs as a two-phase process [5, 37, 38]. In the first phase, the IR is traversed to generate a system of constraints handling rules [32, 33]. There are only a handful of constraints including:

- `EqualityConstraint[`α`,`β`]` constraints the type α to be equal to β. For example, the types α and β must unify.
- `AlternativeConstraint[`α`,{`$\beta 1$`,`$\beta 2$`,...}]` constraints α to be equal to any one βi.
- `InstantiateConstraint[`τ`,`ρ`,m]` constraints τ to be an instance of the polymorphic type ρ with respect to the monomorphic set `m`.
- `GeneralizeConstraint[`σ`,`τ`,m]` constraints σ to be a generalization of τ with respect to the monomorphic set `m`.

The system of constraints are then solved. The solver creates a graph where every node in the graph is a constraint. An edge exists between two constraints in the graph if their free variable sets overlap. This graph is maintained throughout the constraint solving process. The graph is also used during variable substitution, which is applied iteratively only on the strongly connected components of the graph which contain the variables. If multiple types are matched by the `AlternativeConstraint`, then an ordering [58, 74] is computed between the matched types. Lack of ordering is an ambiguity and the compiler raises an error.

Expressivness and Extensibility of the Type System – Functions are defined within a type environment. Function definitions can be overloaded by type, arity, and return type. This is different from some other languages [41] which do not allow for arity-based overloading. Multiple type environments can be resident within the compiler; a default builtin type environment is provided. Users can extend the type environment and specify which type environment to use at `FunctionCompile` time[4]. As an example, the definition for the `Min` is shown here:

```
tyEnv["declareFunction", Min,
  Typed[TypeForAll[{"α"}, {"α" ∈"Ordered"},
        {"α","α"} → "α"]
  ]@
  Function[{e1, e2},
    If[e1 < e2, e1, e2]
  ]
];
```

[4]We use this to perform compilation for different architectures. For example the CUDA type system is different from the one for the Wolfram Language, since CUDA has different semantics.

This definition is defined polymorphically on any scalar argument which can be ordered (e.g. integer and reals, but not complex). The type system offers a balance between generality (through polymorphism) and constraint (through qualifiers). We can use this scalar definition to define `Min` on any container of arbitrary rank:

```
tyEnv["declareFunction", Min,
  Typed[TypeForAll[{"α","c","ρ"},
        {"α" ∈"Ordered","c" ∈"Container"},
        {"c"["α","ρ"]}⟶"α"]
  ]@
  Function[{arry}, Fold[Min, arry]]
];
```

4.5 Typed Wolfram IR Representation and Passes

WIR variables are annotated with the types inferred by the type inference pass. A WIR where all the variables have been type annotated is called a Typed Wolfram Intermediate Representation (*TWIR*). Having the same representation means that transformations can introduce untyped instructions (changing a TWIR back to a WIR). When this occurs, the type inference is rerun and a TWIR representation is reconstructed. Traditional compiler optimizations such as: sparse conditional constant propagation [79], common subexpression elimination [20], dead code elimination [47], etc. are implemented within the compiler and are safe to perform on the TWIR. This section highlights some analysis, transformations, optimizations used in the compiler and describe how they solve the objectives and challenges introduced in Section 3.

Function Resolution – The first transformation performed on the TWIR is to resolve all function implementations within the program. For each call instruction, a lookup into the type environment is performed. A lookup can either return a monomorphic or polymorphic function implementation. It can also find no matching implementation; reporting an error. If a function has a monomorphic implementation, then it is inserted into the TWIR. If the function exists polymorphically within the type environment, then it is instantiated with the appropriate type, the function is inserted into the TWIR, and the call instruction is rewritten to the mangled name of the function. A function is inlined at this stage if it has been marked by users to be forcibly inlined.

Abortable Evaluation – The generated code must be F3 abortable. While a valid solution of handling aborts is by inserting a check after each TWIR instruction, this would inhibit many optimizations. Instead, the compiler performs analysis to compute the loops [13, 62] and then inserts an abort check at the head of each loop. Since functions can be recursive; e.g. the `cfib` in Section 4.1 function. The compiler also inserts an abort check in each function's prologue. The abort checks if a user initiated abort signal has been issued to the Wolfram Engine and, if so, throws a hardware exception.

The compiler generates the code needed to free resources if the exception is thrown.

Maintaining Mutability Semantics – The compiler relies on a few analysis and transformations to perserve the F5 mutability semantics. Given a program such as `x={...};...; y[[1]]=3`, a copy of `x` is only needed if `y` aliases `x` and if `x` is used in subsequent statements. Both alias [57] and live [12] analysis are performed to determine the above conditions. A copy is performed if the above conditions are satisfied.

Automatic Memory Management – The Wolfram Language is F7 memory managed. The compiler computes the live intervals [82] of each variable in the TWIR. For each variable, a `MemoryAcquire` call instruction is placed at the head of each interval, and `MemoryRelease` is placed at the tail. Both `MemoryAcquire` and `MemoryRelease` are written polymorphically and are noop for unmanaged objects and `ReferenceIncrement` and `ReferenceDecrement` for reference counted objects.

Expression Boxing and Unboxing – The compiler follows the existing interpreter's semantics to F1 seamlessly integrate with it. To the Wolfram interpreter, all functions have the signature of `TypeSpecification[{"Expression"}→"Expression"]`. Therefore, the compiler wraps each compiled function with an auxiliary function. The auxiliary function takes the input expression, unpacks and checks (using the compiled function type signature) if it matches the expected number of arguments and types. The auxiliary function then calls the user function and packs the output into an expression. This auxiliary function is the one called by the Wolfram interpreter.

Soft Numerical Failure – The compiler needs to invoke the interpreter as a fallback when F2 runtime numerical errors are thrown. All machine numerical operations are checked for errors by the compiler runtime. Similar to how abortable evaluation is handled, cleanup code is inserted into the TWIR function to handle numeric exceptions. Numerical exceptions are propagated to the top-level auxiliary function which calls the interpreter to rerun the function. For example, consider the `cfib` function in Section 4.1. When the compiled code detects an integer overflow (e.g. `cfib[200]`), it print a warning message and switch to the interpreter which evaluates the function with arbitrary precision integer:

```
In[2]:= cfib[200]

        CompiledCodeFunction: A compiled code runtime error occurred; reverting to
                uncompiled evaluation: IntegerOverflow ≫

Out[2]= 280571172992510140037611932413038677189525
```

Symbolic Computation – Users can perform F8 symbolic computation within compiled code. Symbolic code still utilize the Wolfram Engine, but uses threaded interpretation [27] to bypass the Wolfram interpreter. Symbolic values have the `"Expression"` type. `cf=FunctionCompile[Function[{Typed[arg1,"Expression"],Typed[arg2,"Expression"]}, arg1+arg2]]`, for example, can be compiled. Evaluating `cf[1,2]` returns `3`, evaluating `cf[x,y]` returns `x+y`, and evaluating `cf[x,Cos[y]+Sin[z]]` returns `x+Cos[y]+Sin[z]`.

Escape to Interpreter – When hosted within the Wolfram Engine, compiled code may escape and call the interpreter using `KernelFunction`. A pass lowers `KernelFunction` into a call to the interpreter. This, seamless transition between using compiled and interpreted code, allows users to F9 gradually migrate their code into the compiler.

4.6 Code Generation

The final step is to generate executable code. Code generation only operates on the fully typed TWIR code, and a compile error is issued if any variable type is missing. F4 Multiple backends are supported by the compiler and an API for users to plugin their own backend. Currently, the compiler uses the LLVM backend by default and prototype backends exist to target C++, the existing Wolfram Virtual Machine, WebAssembly, and NVIDIA PTX.

Library Export – All code is JITed by default, but ahead of time compilation to a dynamic library can be performed using the F10 `FunctionCompileExportLibrary` function. The exported library code can be loaded into a Wolfram Engine session using `LibraryFunctionLoad` — enabling ahead of time compilation. Code can also be exported to a standalone static library and used within external applications. Currently, when using code in standalone mode, certain functionalities such as interpreter integration and abortable code are disabled, since they depend on the Wolfram Engine.

4.7 Extending the Compiler

Care was taken to ensure the extensibility of the design and implementation. Users can extend the compiler by adding new macro rules, type system definitions, or transformation passes. Macros and type systems are defined within an environment which is passed in at `FunctionCompile` time. Passes can be enabled during the `FunctionCompile` call. Macro rules, type system definitions, and passes can be predicated on the `FunctionCompile` options, or AST/IR analysis. For example, we can write a macro predicated on the `TargetSystem` option to `FunctionCompile` to transform a `Map` function into one that runs on CUDA if the `TargetSystem` is CUDA:

```
RegisterMacro[macroEnv, Map,
  Conditioned[#TargetSystem≡"CUDA"&]@
  Map[f_, lst_] → CUDA`Map[f, lst]
]
```

4.8 Applicability to Other Languages

The proposed compiler design, analysis, and transformation are relevant to many scripting languages. We found the pipeline MExpr → WIR → TWIR → code generation and the ability to perform analysis and transforms at any stage to be very productive. Some analysis are simpler to perform at a high level, whereas other are more general at the low level IR [51].

The compiler's integration with the interpreter is one of its key features. There has been significant research on compiling scripting language compilers [14, 34, 49]. These compilers, however, have limited use in industry, since they inevitably encounter code that cannot be compiled. Our pragmatic solution to enable the use of the interpreter for "uncompilable" blocks of code increases compiler adoption. Futhermore, it allows the language to evolve without being limited by what can be compiled. Another applicable feature is the compiler's abortability feature. Clones of the Wolfram Notebook interface exist for other scripting languages [46, 59, 63, 67], and abort behavior is desired but, to our knowledge, no other languages support abortable code.

5 Implementation

Compiler Implementation – The compiler is implemented entirely within the Wolfram Language and is on the order of 100,000 lines of code. Passes average around 200 lines of code, and there are over 100 AST or IR passes. The compiler is backed by over 4,000 test cases which are run daily on the supported architectures. These test cases range from simple unit tests to more complicated application tests. Virtually no modifications were needed to the Wolfram Engine during the course of the compiler development. Currently, the compiler supports approximately 2,000 functions which cover 31 functionality areas of the Wolfram Language. Active work on expanding the list of support of functions is in progress, and a mechanism exists to prioritize function support by mining documentation pages and programs.

Compiler Availability – The compiler is in production [98] and is a long-term development project. The compiler is distributed as a package that is updated independently of the Wolfram Engine. Updates are rolled periodically and automatically pushed to users (typically on a monthly basis) from the Wolfram package repository. The new compiler is readily available to developers [100] and is supported on Linux (32-, and 64-bit x86, and 64-bit ARM), Windows (32-, and 64-bit x86), and macOS (64-bit x86). The compiler can also be used within the Wolfram Cloud.

Compiler Extensibility – One of the goals of the compiler is to lower the cost of adding or modifying compiler features. Extending the compiler leverages its API and requires no `C` programming or extensive knowledge of compiler internals. Using the API, several developers have prototyped auto-compilation of functions used by numerical solvers (such as `NDSolve`, `FindRoot`, etc.), performed AST and IR manipulation for automatic differentiation [16], targeted CUDA for acceleration, and deployed compiled functions to WebAssembly to speedup `Dynamic` plots in the Wolfram Cloud Notebooks.

6 Evaluation

The performance of the new Wolfram compiler is measured using an internal benchmark suite. The benchmark suite is run daily and measures all aspects of the compiler: compilation time, time to run specific passes, performance of simple code, and performance of large programs. To highlight the performance aspects of the compiled functions, we select 7 simple benchmarks from the benchmark suite. The benchmarks cover diverse functionality areas. We compare against the pre-existing bytecode compiler as well as hand-optimized `C` implementations. The `C` implementations do not support abortability behavior. All benchmarks are run within Wolfram Engine v12 and compiler version v1.0.1.0 on a MacBook Pro 2.9 GHz Intel Core i7 running macOS Mojave v10.14.6.

Since passes can be turned on or off, we can examine the causes of added overhead when performance is not what is expected. We look at abortability, since it has the biggest impact. Abort checking can be toggled as a compiler option or selectively on expressions by wrapping them with the `Native`AbortInhibit` decorator. In general, as shown in Figure 2, the new compiler's performance is comparable to the hand-written highly tuned `C` implementation:

FNV1a – Computes the FNV1a hash [25] of a string of length 10^6. Since strings are not supported within the bytecode compiler, a workaround is used to represent them as an integer vector of their character codes. Since 64-bit integers are the lowest integer datatype supported by the bytecode compiler, the bytecode compiled function operates on `int64` rather than `uint8`. The new compiler has builtin support for strings and operates on the UTF8 bytes within the string.

Mandelbrot – Computes the Mandelbrot set on the $[-1, 1]\times[-1, 0.5]$ region with 0.1 resolution. The benchmark's loop body is computationally heavy and, therefore, the extra the abort checking overhead at the function header is insignificant to the overall runtime.

Dot – Performs the `Dot` product of two 1000×1000 matrices. The `C` implementations use the MKL [42] library to perform the `cblas_dgemm` operation. Both the new compiler and bytecode compiler leverage the Wolfram Engine's runtime to perform the matrix multiplication. The Wolfram Engine's runtime in turn calls the MKL library. Since all implementations use the MKL library and MKL is non-abortable (being closed source), no performance difference is observed.

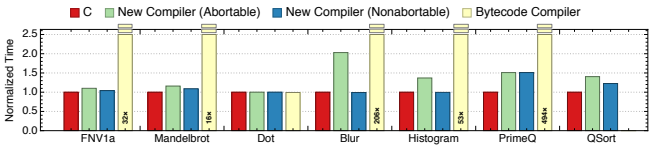

Figure 2. Seven benchmarks are selected from the benchmark suite. Results are normalized to highly optimized hand-written `C` implementation. For the bytecode compiler, slowdown is capped at 2.5 and the actual slowdown shown within the bar.

Blur – Performs a 3×3 Gaussian blur on a 1000×1000 single-channel image. An abort check is placed at the head of both loops in the benchmark. Since the body of the inner function loop is a trivial stencil computation, abortability adds considerable overhead.

Histogram – Performs 256-bin histogram on a uniformly distributed list of 10^6 integers. Histogram is limited by memory load operations and abort checking inhibits vectorized loads.

PrimeQ – Checks the primality of intergers in the range $\left[0, 10^6\right]$ using the Rabin-Miller primality test [60, 80]. To avoid testing primality for small integers, a 2^{14} seed table is generated using the Wolfram interpreter and embedded into the compiled code as a constant array. This table is also pasted into the hand-written `C` code. Due to non-optimal handling of constant arrays, we observe a 1.5× performance degradation. This issue is fixed in the upcoming version of the compiler.

QSort – Performs text-book quicksort on a pre-sorted 2^{15} list. The code is polymorphic and written in a functional style, where user define and pass the comparator function as an argument to the program. Function passing cannot be represented in the bytecode compiler, and therefore this program cannot be represented using the bytecode compiler. A 1.2× slowdown over hand-crafted C code is incurred, since the mutability semantics do not allow sorting to happen in place and a copy of the input list is made.

The new compiler consistently outperforms the bytecode compiler. This is due to two general shortcoming of the bytecode compiler. The first reason is that the bytecode compiler cannot inline functions, and for these benchmarks, which have very tight loops that operate on machine numbers, overhead due to function dispatch is large. For example, disabling function inline within the new compiler results in a 10× slowdown for Mandelbrot over the `C` implementation. The second is due to unboxing of arrays and array indexing semantics. The bytecode compiler operates on boxed array, and therefore any operation on arrays incurs unboxing overhead. Futhermore, since Wolfram Language's supports negative indexing, all array accesses must be predicated at runtime. I.e. `arry[idx]` is executed as `arry[If[idx≥0,idx,Length[arry]-idx]]`. The new compiler address the first shortcoming by inlining primitive functions, and the second by adding optimizations to reduces the frequency of array unboxing and removal of redundant array indexing checks.

7 Related Work

Since we referenced related works throughout this paper, this section only highlights work dealing with overall compiler design of high level scripting languages. Compiling high level scripting languages is an active research topic. There are two schools of thought on compiling these languages. The first is to develop a new language that has the feel of a scripting language, but is in fact statically compiled [10, 11]. The other is to accelerate preexisting programming languages by either reducing the interpreter overhead [78], performing trace compilation [14, 39], or looking at a subset of the language [4, 49, 54]. This paper took the latter approach and proposed a design that addresses the objectives and constraints imposed by the Wolfram language. While certain aspects of the Wolfram Language have been studied [28, 29, 70, 71], to our knowledge, we are the first to describe a compiler for the Wolfram Language.

8 Conclusion

With programmer's time being an ever-more scarce resource, compiler developers need to support languages that increase programmer's productivity. The Wolfram Language has a history of providing solutions to computation problems in areas such as algebra, data science, calculus, statistics, and optimization. Through its wide domain coverage, melding of symbolic and numerical computation, and interactive Notebook interface, the Wolfram language provides a rich platform that increases programmer's productivity. This paper describes the design and implementation of the Wolfram Language compiler. It shows that a careful compiler design maintains the productivity and the expected behavior of Wolfram Language while producing compiled code that rivals hand-written `C` code. The compiler is production grade and is a prominent feature of the Wolfram Engine v12 release.

A Artifact Appendix

The artifact shows how to run the compiler and the benchmarks presented in the paper. We will also describe how to get the intermediate textual representations of the compilation process. The artifact does require the use of the Wolfram Engine which is free to download from https://www.wolfram.com/engine/ and we provide a docker image to automate the process. The code is not restricted to run within the Wolfram Engine and can be run within any Mathematica or any Wolfram Notebook environment. The only requirement is that Mathematica must be version 12 or greater. The artifact

also requires internet access to download the Wolfram Engine along with other support functions which have been deployed to the Wolfram repository.

A.1 Artifact Check-List (Meta-Information)

- **Run-time environment:** Wolfram Engine
- **Hardware:** X86 or amd64 Windows, Linux, or MacOS
- **Metrics:** Wall-clock time
- **Experiment Workflow:** See bellow.
- **Publicly available:** Yes. Can be downloaded from www.wolfram.com and the compiler is soon to enabled within the cloud at cloud.wolfram.com.
- **Licenses:** Wolfram licence available at https://www.wolfram.com/legal/agreements/wolfram-mathematica/
- **Documentation :** The Compiler is documented at https://wolfr.am/FLVQZ6xI. The documentation contains more examples than what is presented in the paper
- **Program:** FNV1A, Mandelbrot, Dot, Blur, Histogram, PrimeQ, QuickSort
- **Workflow frameworks used:** Docker.
- **Archived:** https://doi.org/10.5281/zenodo.3558339 and https://wolfr.am/I2CE1Lw1.

A.2 Description

The code can be executed within any Wolfram Engine and all the code presented in the paper is valid Wolfram programs. The Wolfram Engine can be installed on the local system. For convenience, we provide a docker file which users can build and use. We also provide the benchmark programs used for the paper experiments.

A.2.1 Hardware and Software dependencies. The compiler runs natively on Windows (32- and 64-bit x86), Linux (32- and 64-bit x86 and ARM), and MacOS (64-bit x86).

A.3 Installation

The Wolfram Engine along with the compiler is free to download from https://www.wolfram.com/engine/. This artifact provides a ready-made docker image of the engine as well. A docker image is provided as an artifact, since the image is simple, it is embedded in this document:

```
FROM ubuntu

RUN apt update && apt install -y curl avahi-daemon wget
     sshpass sudo locales locales-all ssh vim expect
     libfontconfig1 libgl1-mesa-glx libasound2

RUN echo "en_US.UTF-8 UTF-8" > /etc/locale.gen && locale-gen

RUN wget https://account.wolfram.com/download/public/wolfram
     -engine/desktop/LINUX && sudo bash LINUX -- -auto -
     verbose && rm LINUX

CMD ["/usr/bin/wolframscript"]
```

Copy the above into a file called `Dockerfile` and then build the docker image using

```
> docker build -t [[user_name]]/wolframengine:1.0 .
```

where `[[user_name]]` is your dockerhub user name.

A.4 Activating and running the Wolfram Engine

The following steps are required to activate the Wolfram Engine.

A.4.1 Activating the Wolfram Engine. To activate the Wolfram Engine you need to start the docker image you just created and visit wolfram.com/developer-license to get a license for the Free Wolfram Engine for Developers. You will be asked to sign up for a Wolfram ID. Launch the docker image and sign in with your our own Wolfram ID and type your Wolfram ID password. This will activate your Wolfram Engine. After the Wolfram Engine activates, evaluate the `$PasswordFile` command. The output is the location of the password file. Next, evaluate `FilePrint[$PasswordFile]` to print the content of the password file. Finally, quit the Wolfram Engine session using the `Quit` command. An example session is shown bellow:

```
> docker run -it [[user_name]]/wolframengine:1.0
The Wolfram Engine requires one-time activation on this
     computer.

Visit https://wolfram.com/developer-license to get your free
      license.

Wolfram ID: yourwolframid@example.com
Password:
Wolfram Engine activated. See https://www.wolfram.com/
     wolframscript/ for more information.
Wolfram Language 12.0.0 Engine for Linux x86 (64-bit)
Copyright 1988-2019 Wolfram Research, Inc.

In[1]:= $PasswordFile

Out[1]= /root/.WolframEngine/Licensing/mathpass

In[2]:= FilePrint[$PasswordFile]
1e1d781ed0a3 6520-03713-97466 4304-2718-2K5ATR
     5095-179-696:2,0,8,8:80001:20190627

In[3]:= Quit
```

A.4.2 Licence Persistence. Because Docker images starts a "clean" environment each time, the password information is lost across sessions. To persist the password information between sessions, one needs to store the password file on the host machine. On the host machine, first create a `Licensing` directory which will hold the password file:

```
> mkdir Licensing
```

Then, create a file called `mathpass` within the `Licensing` directory and copy the password (the output from `In[2]` above) into it.

A.4.3 Running the Wolfram Engine. To launch the Wolfram Engine with the password file from the host machine, use the `-v` option to mount the directory and make the `./Licensing` directory available to the docker image under `/root/.WolframEngine/Licensing`. If everything works

correctly, you should now be able to launch the docker image without being prompted with activation requests each time.

```
> docker run -it -v ./Licensing:/root/.WolframEngine/
    Licensing [[user_name]]/wolframengine:1.0
Wolfram Language 12.0.0 Engine for Linux x86 (64-bit)
Copyright 1988-2019 Wolfram Research, Inc.

In[1]:= $MachineName
Out[1]= 861d2b5cd33f

In[2]:= $Version
Out[2]= 12.0.0 for Linux x86 (64-bit) (May 19, 2019)
```

A.5 Experiment Workflow

We will first describe how to perform basic compilation using the compiler, we will then describe how to get intermediate representation of these basic programs, and then show how to run the benchmarks presented in the paper on a supported system. We will conclude by briefly describing how to extend the compiler and where to get additional examples. In this section, we will assume the user is using an interactive session in the Wolfram Engine.

A.6 Intermediate Representations

First, it is useful to demonstrate how the compiler pipeline works at a basic level. We will use

```
In[1]:= addOne = Function[Typed[arg, "MachineInteger"], arg
    + 1];
```

as a running example to show the high-level API of the compiler. The evaluator can copy the following code (following the `In[]:=` statement) and evaluate it within their session to get the results presented

As described in the paper, the compiler uses a pipeline to translate the input AST into first a desugared AST, then a TWIR and WIR, then the backends can generate LLVM, assembly, or a library. Not all of these transformations are available in the `System` context, so the following code imports the `Compile` context:

```
In[2]:= Needs["Compile`"]
```

A.6.1 Generating AST. The `CompileToAST` returns a data structure which holds the AST of the input expression. For printing purposes, we convert the data structure to a string using the `"toString"` method. No macros are apply to the `addOne` and therefore the code is unchanged.

```
In[3]:= CompileToAST[addOne]["toString"]
Out[3]= "Function[{Typed[arg, \"MachineInteger\"]}, arg + 1]
    "
```

A.6.2 Generating WIR. To generate the internal WIR representation one uses the `CompileToIR` function where all passes are disabled. We, again, only show the textual description of the data structure returned.

```
In[4]:= CompileToIR[addOne,"OptimizationLevel"->None]["
    toString"]
Out[4]= Main::Information={"inlineInformation"->{ "
    inlineValue"->Automatic, "isTrivial"->False}, "
    ArgumentAlias"->False, "Profile"->False, "AbortHandling
    "->True}
Main
start(1):
2 | %1:TypeSpecifier[MachineInteger] = LoadArgument arg
3 | %3 = Plus: %1, 1
4 | Jump end(2)
end(2):
6 | Return %3
```

A.6.3 Generating TWIR. To generate the internal TWIR representation one uses the `CompileToIR` function. Note how all types are now annotated and the functions are resolved to their implementations. Here `checked_binary_plus_Integer64_Integer64` is a function defined within the compiler runtime library.

```
In[5]:= CompileToIR[addOne]["toString"]
Out[5]= Main::Information={"inlineInformation"->{"
    inlineValue"->Automatic, "isTrivial"->True}, "
    ArgumentAlias"->False, "Profile"->False, "AbortHandling
    "->True}
Main : (Integer64)->Integer64
start(1):
2 | %1:I64 = LoadArgument arg
3 | %7:I64 = Call Native`PrimitiveFunction[
    checked_binary_plus_Integer64_Integer64]:(I64,I64)->I64
     [%1, 1:I64]
4 | Return %7
```

A.6.4 Generating LLVM. The LLVM IR can be generated using the documented `FunctionCompileExportString` function.

```
In[6]:= FunctionCompileExportString[addOne,"LLVM"]
Out[6]= ; ModuleID = 'WolframCompiledFunction$6'
source_filename = "WolframCompiledFunction$6"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-darwin"

; Function Attrs: inlinehint
define dllexport i64 @Main(i64 %var1) local_unnamed_addr #0
    {
start_1:
  %var7 = tail call i64
      @checked_binary_plus_Integer64_Integer64(i64 %var1,
      i64 1), !linkage !5, !name !6
  ret i64 %var7
}

declare i64 @checked_binary_plus_Integer64_Integer64(i64,
    i64) local_unnamed_addr

attributes #0 = { inlinehint }

!llvm.ident = !{!0}
!wolfram.version = !{!1}
!wolfram.build_time = !{!2}
!wolfram.options = !{!3}
!wolfram.expr = !{!4}

!0 = !{!"Wolfram Compiler 1."}
!1 = !{!"12.0.0 for Mac OS X x86 (64-bit) (April 7, 2019)"}
!2 = !{!"Tue 12 Nov 2019 12:38:09"}
```

```
!3 = !{!"<|PassLogger -> Automatic, AddRuntime ->
     SharedLibrary, LazyJIT -> True, LLVMOptimization ->
     ClangOptimization[1], LoopHints -> Automatic,
     MachineArchitecture -> Automatic, CreateWrapper ->
     False, ExpressionInterface -> Automatic,
     TargetArchitecture -> Automatic, TargetSystemID ->
     Automatic, TargetTriple -> Automatic, DataLayout ->
     Automatic, Debug -> False, LLVMDebug -> False,
     ExceptionsModel -> Automatic, AbortHandling -> False,
     PassOptions -> {}|>"}
!4 = !{!"Program[Function[{Typed[arg, \22MachineInteger
     \22]}, arg + 1]]"}
!5 = !{!"Runtime"}
!6 = !{!"Native`PrimitiveFunction[
     checked_binary_plus_Integer64_Integer64]"}
```

A.6.5 Generating Assembly.

The system's assembly instructions can be generated using the documented `FunctionCompileExportString` function.

```
In[7]:= FunctionCompileExportString[addOne,"Assembler"]
Out[7]=  .section __TEXT,__text,regular,pure_instructions
  .globl _Main
  .p2align 4, 0x90
_Main:
  .cfi_startproc
  movl $1, %esi
  jmp _checked_binary_plus_Integer64_Integer64
  .cfi_endproc


.subsections_via_symbols
```

A.6.6 Generating a Library.

The compiler supports ahead of time compilation by generating a library which can be subsequently used within the Wolfram Engine. One can export a function as a library using the documented `FunctionCompileExportLibrary` function.

```
In[8]:= lib = FunctionCompileExportLibrary["addOne.dylib",
     addOne]
```

A.7 Evaluating the Benchmarks

With the basics demonstrated, we will show how to run the benchmarks presented in the paper within the Wolfram Engine. These benchmarks can be evaluated against hand-written C code to validate the results in the paper.

First, download the programs from the

```
In[9]:= CGOBenchmark = ResourceFunction["https://wolfr.am/
     I2CE1Lw1"]
```

The downloaded `CGOBenchmark` is a hash table where the keys are the name of the benchmarks and the values are their implementation. To evaluate `dot`, for example, a user writes

```
In[10]:= n = 100;
a = RandomReal[1, {n,n}];
b = RandomReal[1, {n,n}];
CGOBenchmark["dot"][a,b]
```

A user can view the definition of the benchmarks either by looking at the posted files on the artifact repository or by doing

```
In[11]:= CGOBenchmark["dot"]
```

Futher examples and information are viewable online at https://wolfr.am/I2CE1Lw1 and an example notebook can be generated pragmatically using

```
In[12]:= ResourceObject[CGOBenchmark]["ExampleNotebook"]
```

The implementation of each benchmark are also accessible by accessing the `"implementation"` field within the benchmark resource. For example, to get the implementation of the `mandelbrot` benchmark you do:

```
In[13]:= First[CGOBenchmark["implementation"]["mandelbrot"
     ]]["Input"]
Out[13]= Function[{Typed[pixel0, "ComplexReal64"]},
 Module[{iters = 1, maxIters = 1000, pixel = pixel0},
  While[iters < maxIters && Abs[pixel] < 2, pixel = pixel^2
       + pixel0;
   iters++]; iters]]
```

More information about the repository is available at https://wolfr.am/I3kiVNaV.

A.8 Experiment Customization

The code provided can be edited to explore other aspects of the compiler. While the compiler's source code is not open source, but it is readable and is located in `"Location" /. PacletInformation[ First[PacletFind["Compile"]]]`. As described in the paper, the code base is written purely using the Wolfram Language and a user can modify the code to suit their needs.

A.9 Additional Examples

More examples of the usage of the compiler are available at the Wolfram documentation site https://reference.wolfram.com/language/guide/CodeCompilation.html. Futhremore, stand-alone examples are available as part of the feature release page on the Wolfram website https://www.wolfram.com/language/12/code-compilation.

B Artifact Repository

The main artifact of the paper is the Wolfram compiler which is available within the Wolfram Engine. The specific version used in the paper is `1.0.1.0` and can be installed using

```
In[14]:= PacletInstall[{"Compile", "1.0.1.0"}]
```

We also provide the source code for all the benchmarks presented in the paper through the Wolfram artifact repository. Users download the code using

```
In[15]:= ResourceFunction["https://wolfr.am/I2CE1Lw1"]
```

The artifact published above uses the Wolfram repository which is a permanent repository and forms the backbone of Wolfram knowledge bases such as Wolfram Alpha. The code is also published to https://doi.org/10.5281/zenodo.3558339 which contains the C and bytecode compiler implementation of the benchmarks as well.

References

[1] Martha LL Abell and James P Braselton. 2017. *Mathematica by Example*. Elsevier.

[2] Michael D. Adams. 2015. Towards the Essence of Hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*, Vol. 50. ACM, ACM Press, 457–469.

[3] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a symposium on Compiler optimization* -, Vol. 5. ACM, ACM Press, 1–19.

[4] George Almási and David Padua. 2002. MaJIC. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation - PLDI '02*, Vol. 37. ACM, ACM Press, 294–303.

[5] Sandra Alves and Mário Florido. 2002. Type Inference using Constraint Handling Rules. *Electron. Notes Theor. Comput. Sci.* 64 (Sept. 2002), 56–72.

[6] Marc Auslander and Martin Hopkins. 1982. An overview of the PL.8 compiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction - SIGPLAN '82*, Vol. 17. ACM, ACM Press, 22–31.

[7] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Comput. Sci. Eng.* 13, 2 (March 2011), 31–39.

[8] David A. Belsley. 1989. Mathematica: A critical appraisal. *Computer Science in Economics and Management* 2, 2 (1989), 171–178.

[9] Jean-Philippe Bernardy and Nicolas Pouillard. 2014. Names for free: Polymorphic views of names and binders. *ACM SIGPLAN Notices* 48, 12 (2014), 13–24.

[10] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and performance reconciled by design. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 1–23.

[11] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (Jan. 2017), 65–98.

[12] Benoit Boissinot, Florian Brandner, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello. 2011. A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs. In *Asian Symposium on Programming Languages and Systems*. Springer, 137–154.

[13] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. 2012. SSI Properties Revisited. *TECS* 11S, 1 (June 2012), 1–23.

[14] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems - ICOOOLPS '09*. ACM, ACM Press, 18–25.

[15] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and efficient construction of static single assignment form. In *International Conference on Compiler Construction*. Springer, 102–122.

[16] H Martin Bücker, George Corliss, Paul Hovland, Uwe Naumann, and Boyana Norris. 2006. *Automatic Differentiation: Applications, Theory, and Implementations*. Vol. 50. Springer Berlin Heidelberg.

[17] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type inference for static compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, Vol. 51. ACM, ACM Press, 410–429.

[18] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *SIGPLAN Not.* 34, 10 (Oct. 1999), 1–19.

[19] William Clinger. 1991. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91*, Vol. 91. ACM Press, 155–162.

[20] John Cocke. 1970. Global common subexpression elimination. *SIGPLAN Not.* 5, 7 (July 1970), 20–24.

[21] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.

[23] R Kent Dybvig. 1992. *Writing hygienic macros in scheme with syntax-case*. Technical Report. Technical Report 356, Indiana Computer Science Department (June 1992).

[24] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic abstraction in scheme. *Lisp and Symbolic Computation* 5, 4 (Dec. 1993), 295–326.

[25] Donald Eastlake, Tony Hansen, Glenn Fowler, Kiem-Phong Vo, and Landon Noll. 2018. The fnv non-cryptographic hash algorithm. (2018).

[26] Dirk Eddelbuettel and Romain François. 2011. Rcpp: Seamless-randc++integration. *J. Stat. Soft.* 40, 8 (2011), 1–18.

[27] M Anton Ertl and David Gregg. 2001. The behavior of efficient virtual machine interpreters on modern architectures. In *European Conference on Parallel Processing*. Springer, 403–413.

[28] R. J. Fateman. 1990. Advances and trends in the design and construction of algebraic manipulation systems. In *Proceedings of the international symposium on Symbolic and algebraic computation - ISSAC '90*. ACM, ACM Press, 60–67.

[29] Richard J. Fateman. 1992. A review of mathematica. *Journal of Symbolic Computation* 13, 5 (May 1992), 545–579.

[30] Matthew Flatt. 2016. Binding as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, Vol. 51. ACM, ACM Press, 705–717.

[31] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together. *J. Funct. Prog.* 22, 2 (March 2012), 181–216.

[32] Thom Frühwirth. 1998. Theory and practice of constraint handling rules. *The Journal of Logic Programming* 37, 1-3 (Oct. 1998), 95–138.

[33] Thom Frühwirth. 2006. Constraint handling rules. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '06*. Springer, ACM Press, 90–107.

[34] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: Enabling static optimization of scientific Python programs. *Comput. Sci. Disc.* 8, 1 (March 2015), 014001.

[35] Louis-Julien Guillemette and Stefan Monnier. 2007. A type-preserving closure conversion in haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell '07*. ACM, ACM Press, 83–92.

[36] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138.

[37] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. 2003. Helium, for learning Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '03*. ACM, ACM Press, 62–71.

[38] Bastiaan J Heeren. 2005. *Top quality type error messages*. Utrecht University.

[39] Andrei Homescu and Alex Şuhan. 2011. HappyJIT. In *Proceedings of the 7th symposium on Dynamic languages - DLS '11*, Vol. 47. ACM, ACM Press, 25–36.

[40] Paul Hudak and Joseph H. Fasel. 1992. A gentle introduction to Haskell. *SIGPLAN Not.* 27, 5 (May 1992), 1–52.

[41] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell. In *Proceedings of the third ACM SIGPLAN*

conference on History of programming languages - HOPL III. ACM, ACM Press, 12–1.

[42] Intel. [n.d.]. The Intel math kernel library. https://software.intel.com/en-us/mkl. Accessed: 2019-08-30.

[43] Mohamed Ismail and G. Edward Suh. 2018. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, 36–47.

[44] Mark P Jones. 1992. A theory of qualified types. In *European symposium on programming*. Springer, 287–306.

[45] Mark P. Jones. 1994. *Qualified Types*. Vol. 9. Cambridge University Press.

[46] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan-Kelley, and Carol Willing. 2018. Binder 2.0 - Reproducible, interactive, sharable environments for science at scale. In *Proceedings of the 17th Python in Science Conference*. SciPy, 87–90.

[47] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. *Partial dead code elimination*. Vol. 29. ACM.

[48] Lawrence S. Kroll. 1989. Mathematica—A System for Doing Mathematics by Computer. Wolfram Research. *The American Mathematical Monthly* 96, 9 (Nov. 1989), 855–861.

[49] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*. ACM, ACM Press, 7.

[50] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE Computer Society, IEEE, 75.

[51] Chris Lattner and Jacques Pienaar. 2019. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law. (2019).

[52] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS. In *Proceedings of the 9th symposium on Dynamic languages - DLS '13*, Vol. 49. ACM, ACM Press, 1–16.

[53] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Typed Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications - Dyla'14*. ACM, ACM Press, 1–10.

[54] B. Marsolf, K. Gallivan, and E. Gallopoulos. 1995. The interactive restructuring of MATLAB programs using the FALCON environment. In *Proceedings Innovative Architecture for Future Generation High-Performance Processors and Systems*. Springer, IEEE Comput. Soc, 269–288.

[55] Microsoft. [n.d.]. Visual Studio Code. https://code.visualstudio.com. Accessed: 2019-08-30.

[56] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (Dec. 1978), 348–375.

[57] Nomair A. Naeem and Ondrej Lhoták. 2009. Efficient alias set analysis using SSA form. In *Proceedings of the 2009 international symposium on Memory management - ISMM '09*. ACM, ACM Press, 79–88.

[58] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory Pract. Obj. Syst.* 5, 1 (Jan. 1999), 35–55.

[59] Fernando Perez and Brian E. Granger. 2007. IPython: A System for Interactive Scientific Computing. *Comput. Sci. Eng.* 9, 3 (2007), 21–29.

[60] RGE Pinch. 1993. Some primality testing algorithms. *Notices Amer. Math. Soc* 40 (1993), 1203–1210.

[61] Nicolas Pouillard and François Pottier. 2012. A unified treatment of syntax with binders. *J. Funct. Prog.* 22, 4-5 (Aug. 2012), 614–704.

[62] G. Ramalingam. 2002. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.* 24, 5 (Sept. 2002), 455–490.

[63] Bernadette M. Randles, Irene V. Pasquetto, Milena S. Golshan, and Christine L. Borgman. 2017. Using the Jupyter Notebook as a Tool for Open Science: An Empirical Study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, IEEE, 1–2.

[64] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2005. EDUCATIONAL PEARL: A Nanopass framework for compiler education. *J. Funct. Prog.* 15, 05 (June 2005), 653.

[65] Patrick Scheibe. [n.d.]. Wolfram Language Plugin 2019 for IntelliJ platform based IDE's. https://wlplugin.halirutan.de. Accessed: 2019-08-30.

[66] Dag Sverre Seljebotn. 2009. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science Conference*, Vol. 37.

[67] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (Nov. 2014), 151–152.

[68] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual machine showdown. *TACO* 4, 4 (Jan. 2008), 1–36.

[69] Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[70] Mark Sofroniou and Giulia Spaletta. 2005. Precise numerical computation. *The Journal of Logic and Algebraic Programming* 64, 1 (July 2005), 113–134.

[71] Mark Sofroniou and Giulia Spaletta. 2008. Extrapolation methods in mathematica. *JNAIAM J. Numer. Anal. Indust. Appl. Math* 3 (2008), 105–121.

[72] Gabriel Southern and Jose Renau. 2016. Overhead of deoptimization checks in the V8 javascript engine. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, 1–10.

[73] Paul A. Steckler and Mitchell Wand. 1997. Lightweight closure conversion. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 48–86.

[74] Peter J. Stuckey and Martin Sulzmann. 2002. A theory of overloading. *SIGPLAN Not.* 37, 9 (Sept. 2002), 167–178.

[75] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*, Vol. 43. ACM, ACM Press, 395–406.

[76] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. 2014. The typed racket guide.

[77] William M. Waite and Gerhard Goos. 1984. *Compiler Construction*. Springer New York.

[78] Haichuan Wang, David Padua, and Peng Wu. 2015. Vectorization of apply to reduce interpretation overhead of R. *SIGPLAN Not.* 50, 10 (Oct. 2015), 400–415.

[79] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210.

[80] Eric W Weisstein. 2002. Rabin-Miller Strong Pseudoprime Test. (2002).

[81] Paul R Wellin, Richard J Gaylord, and Samuel N Kamin. 2005. *An introduction to programming with Mathematica*. Cambridge University Press.

[82] Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In *Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10*. ACM, ACM Press, 170–179.

[83] Stephen Wolfram. 2017. *An Elementary Introduction to the Wolfram Langauge*. Wolfram Media, Incorporated.

[84] Wolfram Research. [n.d.]. Compiling Wolfram Language Expressions. https://wolfr.am/GgNqTiOn. Accessed: 2019-08-30.

[85] Wolfram Research. [n.d.]. Controlling Infinite Evaluation. https://reference.wolfram.com/language/tutorial/ControllingInfiniteEvaluation.html. Accessed: 2019-08-30.

[86] Wolfram Research. [n.d.]. Mathematica 12. https://www.wolfram.com/mathematica/new-in-12/. Accessed: 2019-08-30.

[87] Wolfram Research. [n.d.]. Mathematica 2. https://reference.wolfram.com/legacy/v2/. Accessed: 2019-08-30.

[88] Wolfram Research. [n.d.]. Mathematica Quick Revision History. https://www.wolfram.com/mathematica/quick-revision-history.html. Accessed: 2019-08-30.
[89] Wolfram Research. [n.d.]. Mathematica Reference. https://reference.wolfram.com/language/. Accessed: 2019-08-30.
[90] Wolfram Research. [n.d.]. The Mathematica Story: A Scrapbook. http://www.wolfram.com/mathematica/scrapbook/. Accessed: 2019-08-30.
[91] Wolfram Research. [n.d.]. Patterns and Transformation Rules. https://reference.wolfram.com/language/tutorial/PatternsAndTransformationRules.html. Accessed: 2019-08-30.
[92] Wolfram Research. [n.d.]. Principles of Evaluation. https://reference.wolfram.com/language/tutorial/PrinciplesOfEvaluation.html. Accessed: 2019-08-30.
[93] Wolfram Research. [n.d.]. The Wolfram High-Performance Computing. https://www.wolfram.com/solutions/industry/hpc/. Accessed: 2019-08-30.
[94] Wolfram Research. [n.d.]. The Wolfram Microcontroller Kit. https://www.wolfram.com/language/12/microcontroller-kit. Accessed: 2019-08-30.
[95] Wolfram Research. [n.d.]. The Wolfram Microcontroller Kit. https://www.wolfram.com/language/12/microcontroller-kit. Accessed: 2019-08-30.
[96] Wolfram Research. [n.d.]. Version 12 Launches Today! (And It's a Big Jump for Wolfram Language and Mathematica). "https://wolfr.am/Gnl8JY7p". Accessed: 2019-08-30.
[97] Wolfram Research. [n.d.]. We've Come a Long Way in 30 Years (But You Haven't Seen Anything Yet!). https://wolfr.am/Gnl6Pekr. Accessed: 2019-08-30.
[98] Wolfram Research. [n.d.]. What's New in Mathematica 12: Code Compilation. https://www.wolfram.com/language/12/code-compilation. Accessed: 2019-08-30.
[99] Wolfram Research. [n.d.]. Wolfram Cloud. https://www.wolframcloud.com. Accessed: 2019-08-30.
[100] Wolfram Research. [n.d.]. Wolfram Engine. https://www.wolfram.com/engine/. Accessed: 2019-08-30.
[101] Wolfram Research. [n.d.]. Wolfram Language Principles and Concepts. https://www.wolfram.com/language/principles/. Accessed: 2019-08-30.
[102] Wolfram Research. [n.d.]. Wolfram Mathematica. https://www.wolfram.com/mathematica/. Accessed: 2019-08-30.
[103] Wolfram Research. [n.d.]. Wolfram Notebook. http://www.wolfram.com/notebooks/. Accessed: 2019-08-30.
[104] Wolfram Research. [n.d.]. Wolfram Workbench. https://www.wolfram.com/workbench/. Accessed: 2019-08-30.