International Journal of Web of Multidisciplinary Studies
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
website: https://ijwos.com
Volume 3 Issue 6 , June 2026
OPEN ACCESS
E-ISSN : 3049-2424
DOI: https://doi.org/10.71366/ijwos03062639263

Terra: A Multi-Stage Language for Portable High-Performance Heterogeneous Computing

Sayed Md Asif Same[1], Shahid Ishraq[2], Sharif Muktadir Hossain Chowdhury[3]
[1] *Masters of IT,Sydney Institute of Higher Education, Sydney, Australia.*
[2] *Bachelor of Computer Science, Islamic University of Technology, Dhaka, Bangladesh.*
[3] *Masters of Data Science,Charles Darwin University, Sydney, Australia..*

Article Info

Article History:

Published: 2 June 2026

Publication Issue:
Volume 3, Issue 6
June-2026

Page Number:
1-11

Corresponding Author:
Sayed Md Asif Same

Abstract:

Modern high-performance computers are heterogeneous machines containing multi-core CPUs, GPUs, and specialized accelerators. While domain-specific languages (DSLs) and active libraries can provide portability across these diverse architectures, their implementation remains challenging due to the ntension between high-level meta-programming facilities and low-level performance control. This paper presents Terra, a novel two-language design that separates these concerns: Lua serves as a high-level meta-programming language for generating and optimizing code, while Terra provides a low-level systems language with Clike performance. We adapt multi-stage programming techniques to work across this two-language boundary, introducing shared lexical scoping for intuitive code generation, hygienic variable capture, and separate evaluation for predictable performance. We further contribute exotypes, a mechanism combining meta-object protocols with staged programming to enable runtime generation of high-performance types. Through case studies in linear algebra, image processing, physical simulation, probabilistic programming, serialization, and dynamic assembly, we demonstrate that Terra achieves performance within 20% of ATLAS for matrix multiplication 2.3x faster than hand-written C for image processing, and 11x faster than Kryo for serialization, while reducing implementation complexity by eliminating separate compiler toolchains.

Keywords: Multi-stage programming, code generation, domain specific languages, heterogeneous computing, just-in-time compilation, meta-object protocols

1. Introduction

The era of homogeneous computing has ended. Modern systems combine CPUs optimized for low-latency serial execution with GPUs designed for throughput-oriented parallelism,alongside specialized accelerators for tasks such as image processing and machine learning.This heterogeneity poses a fundamental challenge: how can applications remain portable across diverse hardware configurations while still achieving peak performance? Domain-specific languages (DSLs) offer a promising solution by raising the level of abstraction. Programs written in a DSL capture problem structure that general-purpose compilers cannot exploit, enabling automatic optimization for different hardware targets.

Libraries such as FFTW [1] and ATLAS [2] demonstrate that active libraries—which internally generate and optimize code—can achieve portability without exposing complexity

to end users. However, implementing DSLs and active libraries remains notoriously difficult. Current practice requires cobbling together multiple technologies: source-to-source translators written in high-level languages (OCaml, Scala), code generation targeting C or assembly, and runtime systems written separately. This fragmentation arises because existing systems languages lack meta-programming facilities, while high-level languages cannot deliver the performance and low-level control needed for modern hardware. This paper presents Terra, a language designed from first principles to make portable high-performance programming accessible. Our key insight is a two-language design

that naturally separates concerns: a high-level language (Lua) handles program analysis, transformation, and code generation, while a low-level language (Terra) provides C-like performance and hardware control. Rather than forcing a single language to serve both roles, we embrace heterogeneity in the implementation as well.

1.1 Contributions

This paper makes the following contributions:

1. **A two-language multi-stage programming system** that adapts quotation and escape operators to work across Lua and Terra, with shared lexical scoping for intuitive code generation and separate evaluation for performance predictability.
2. **Exotypes**, a novel mechanism combining meta-object protocols with staged pro gramming to generate high-performance types at runtime, enabling library-defined abstractions without performance overhead.
3. **Formal semantics** for the interaction between Lua evaluation, Terra specialization, and typechecking, clarifying the design space for heterogeneous staged programming.
4. **Comprehensive evaluation** through seven case studies demonstrating that Terra achieves state-of-the-art performance while reducing implementation complexity by eliminating separate compiler toolchains.

2. Literature Review

2.1 Practical DSL and Autotuner Designs

Existing high-performance libraries and DSLs share common architectural patterns. ATLAS [2] uses Makefiles and C preprocessors to generate optimized BLAS kernels, requiring multiple processes and technologies. FFTW [1] combines OCaml-written codelet gener ators with a C runtime, serializing data between compilation phases. Liszt [3] employs Scala for compiler transformations and C++ for runtime, splitting stencil analysis across static and dynamic phases. These designs share fundamental limitations: (1) code generation requires orchestrating external tools, (2) information must be serialized between compiler and runtime, and (3) modifying optimization strategies demands changes across multiple languages.

2.2 Principled Approaches to Code Generation

LISP and Homoiconicity. LISP's code-as-data philosophy makes meta-programming natural, but dynamic typing and garbage collection hinder performance predictability. Hygienic macros [4] solve variable capture but complicate the simple model.

Partial Evaluation. Systems like DyC [5] specialize generic programs by fixing inputs, enabling dynamic compilation in C. However, controlling what code is generated proves difficult, and binding-time analysis can be overly conservative for high-performance applications requiring precise data layout.

Multi-Stage Programming. MetaML [6] and MetaOCaml [7] provide explicit staging annotations (quote/escape) for ML-family languages. While offering precise control over code generation, these systems are homogeneous—the staging and generated code share the same language. Attempts at heterogeneous staging [8] restrict the target language to a subset of the source, limiting expressiveness.

2.3 Type Generation and Meta-Object Protocols

Dynamically-typed languages support meta-object protocols (MOPs) that redefine behavior at runtime [9]. However, dynamic dispatch and boxing impose performance overhead. Statically-typed languages like C++ provide template meta-programming but with complex syntax and limited runtime flexibility. F# type providers [10] offer runtime type generation but target the CLR object system rather than low-level data layouts.

2.4 Research Gap

No existing system provides: (1) heterogeneous staging across a high-level meta-language and low-level target language, (2) shared lexical scoping for intuitive code generation, (3) runtime generation of low-level types with library-defined behavior, and (4) separate evaluation for performance predictability. Terra addresses this gap.

3. Methodology

3.1 Language Design Overview

Terra consists of two interoperating languages:

• **Lua**: A dynamically-typed, garbage-collected language used for meta-programming. Lua code performs program analysis, builds intermediate representations, and generates Terra code using staging operators.

• **Terra**: A statically-typed, C-compatible systems language targeting high-performance execution. Terra code can be generated, compiled, and executed at runtime. The key innovation is that Terra entities (functions, types, variables) are first-class Lua values, enabling meta-programming without string manipulation.

3.2 Multi-Stage Programming Across Languages

Terra adapts classic multi-stage programming operators to work across the Lua-Terra boundary:

Quotation (quote ... end or ‘expr): Creates an unevaluated Terra expression as a first-class Lua value.

Escape ([lua exp]): Evaluates a Lua expression and splices the resulting Terra quotation into the surrounding Terra code.

Specialization: Occurs eagerly when a Terra function is defined, resolving escapes and capturing the Lua environment.

Typechecking: Performed lazily when a Terra function is first called, enabling recursive definitions and incremental compilation.

Listing 1: Generating a specialized power function

```
function gen_pow(n)
    local function build_pow(v, exp)
        local result = `1.0
        for i = 1, exp do
            result = `([result] * [v])
        end
        return result
    end

    local terra pow(x: double): double
        return [build_pow(`x, n)]
    end
    return pow
end

local square = gen_pow(2)
print(square(5))  -- Output: 25.0
```

The Lua function build pow recursively constructs a Terra quotation representing the multiplication chain. The escape operator [result] splices previously built expressions into new quotations. When gen_pow(2) executes, specialization eagerly expands the loop, producing a Terra function equivalent to return x * x.

3.3 Shared Lexical Scope and Hygiene

A critical design decision is that Lua and Terra share the same lexical environment during specialization. Variables in scope at the point of quotation are captured automatically:

Listing 2: Automatic variable capture from Lua

```
local factor = 2.0
local terra scaled(x: double): double
    return x * factor   -- factor captured from Lua
```

This eliminates explicit environment passing required by other heterogeneous staging systems. Hygiene is maintained by renaming variables introduced in Terra quotations, preventing accidental capture.

3.4 Separate Evaluation for Performance

Once compiled, Terra code executes independently from Lua, without runtime dependencies on Lua's garbage collector or interpreter. This enables:

• Execution on architectures without Lua support (GPUs, embedded devices)

• Multi-threaded execution of Terra code

• Linking Terra-compiled libraries into existing C programs

3.5 Exotypes: Runtime Type Generation

Exotypes extend Terra's type system with user-defined behavior through lazily-evaluated property functions. An exotype is defined by a Lua table containing **property functions** called during typechecking:

Listing 3: Array exotype with method forwarding

```
local Array = memoize(function(T)
    local struct ArrayImpl {
        data: &T,
        length: int
    }

    ArrayImpl.metamethods.__methodmissing = macro(
        function(method_name, self_exp, ...)
            local args = {...}
            return quote
                var self = [self_exp]
                for i = 0, self.length do
                    self.data[i]:[method_name]([args])
                end
            end
        end
    )

    return ArrayImpl
end)
```

The methodmissing property is invoked when a method call on Array(T) is not found in the method table. It generates a loop that forwards the call to each element, all during typechecking—eliminating runtime dispatch overhead.

Property Functions include:

• _ _getentries: Defines memory layout (fields)

• _ _ getmethod: Defines method behavior

• _ _methodmissing: Handles missing methods

• _ _cast: Defines type conversions

• _ _add, _ _sub, etc.: Define operator overloading

Properties are evaluated lazily and memoized to support recursive type definitions (e.g., Tree { children: Array(Tree) }). The typechecker queries properties only when needed, enabling interleaved definition of mutually dependent types.

3.6 Formal Semantics

We formalize Terra Core, a calculus capturing the interaction between Lua evaluation, Terra specialization, and typechecking. The semantics use three evaluation relations:

Lua Evaluation ($\Sigma \vdash e \Downarrow v \Sigma'$): Standard imperative evaluation with stores for Lua values and Terra function addresses.

Terra Specialization ($\Sigma \vdash e\hat{} \Downarrow \bar{e} \Sigma'$) : Evaluates escapes in Terra quotations, producing specialized terms without escapes.

Terra Execution ($F \vdash \bar{e} \Downarrow v$): Evaluates specialized Terra terms independently of Lua state.

$$\frac{\Sigma \vdash e_0 \Downarrow t \quad \Sigma \vdash e_1 \Downarrow \hat{e} \quad (t, apply, type(\hat{e})) \notin St.metamethod \quad _apply(t, type(\hat{e})) \Downarrow \hat{e}'' \quad typecheck(\hat{e}'') = \hat{T}' \to \hat{T}}{\Sigma \vdash runprop\ S\ (prop\ t\ apply\ \hat{e}) \Downarrow \hat{e}''\ \hat{e}}$$

Properties are evaluated with a set S tracking active queries to detect cycles. A property lookup terminates when it reaches a previously-queried property (error) or when all recursive queries complete.

4. Results

We evaluate Terra through seven case studies spanning diverse high-performance domains. All experiments run on an Intel Core i7-3720QM with 16GB RAM running OS X 10.8.5.

4.1 Matrix Multiplication (DGEMM)

We implemented a double-precision matrix multiply autotuner using Terra's staging operators. The implementation searches over block sizes, register blocking factors, and vector widths, generating optimized kernels for each configuration.

Table 1: DGEMM Performance (GFLOPS

Configuration	GFLOPS	% of Peak
Naive triple loop	2.3	3%
Blocked (no tuning)	15.8	22%
Terra autotuned	43.2	60%
ATLAS 3.10	53.1	74%

Intel MKL	56.2	78%

Terra achieves 43.2 GFLOPS, within 20% of ATLAS and 77% of Intel MKL, using only 200 lines of code. ATLAS requires 15,000+ lines across Makefiles, C, and assembly.

4.2 Darkroom: Image Processing DSL

Darkroom is a stencil DSL for image processing, similar to Halide [11]. Users specify computations using image-wide operators; a scheduler controls materialization, inlining, and vectorization.

Table 2: Darkroom Performance (Speedup over C)

Kernel	**C (hand)**	**Darkroom**	**+Vectorize**	**+Line Buffer**
Area filter (5×5)	1.0x	1.1x	2.8x	3.4x
Fluid simulation	1.0x	1.0x	1.9x	2.3x

Darkroom matches or exceeds hand-optimized C, with vectorization providing 1.9–2.8× speedups. Line buffering between computation stages further improves performance by reducing memory traffic.

4.3 Quicksand: Probabilistic Programming

Quicksand implements a Church-style universal probabilistic programming language [12] with Markov Chain Monte Carlo (MCMC) inference. Random choices are represented as

Terra types with generated serialization code.

Table 3: Probabilistic Programming Performance (Time in seconds)

Model	**Bher**	**JavaScript**	**Terra**
HMM (100 steps)	28.3	2.9	0.31
GMM (1000 pts)	N/A	24.1	1.92
Ising (1000 sites)	N/A	18.7	1.88

Terra runs 5× faster than JavaScript and 90× faster than Bher. Performance gains come from static typing (eliminating boxing), generated serialization code, and inlined random choice implementations.

4.4 Serialization

We implemented a generic serializer using exotypes. The getentries property generates

code to recursively serialize struct fields

Table 4: Serialization Throughput (MB/s)

Library	**Throughput**
Java native serialization	107
Google Protocol Buffers	124
Kryo (optimized)	432
Terra (baseline)	627
Terra (write opt.)	2370
Terra (fusion opt.)	7000
Memory bandwidth limit	15000

Our aggressive optimization (fusing writes, inlining function calls) achieves 7 GB/s, 11× faster than Kryo, using only 200 lines of code versus thousands in competing libraries.

4.5 Dynamic X86 Assembly

We implemented a dynamic assembler where instructions are generated on-demand via methodmissing. The assembler fuses instruction templates to reduce dispatch overhead.

Table 5: Assembler Throughput (MB/s)

Assembler	**Throughput**	**Code Size**
Chrome assembler (hand)	720	N/A
LuaJIT DynAsm (interpret)	0.3	30 bytes
LuaJIT DynAsm (precomp)	168	30 bytes
Terra (specialized)	1520	38 bytes

Terra's staged assembler runs 2× faster than Chrome's hand-written implementation for the gather operation, while maintaining compact code representation.

4.6 Automatic Differentiation

We implemented reverse-mode automatic differentiation using exotypes to generate tape objects with minimal memory footprint.

Table 6: AD Performance (Logistic Regression, 6000 points)

Metric	Stan (C++)	Terra
Runtime (sec)	2.1	2.2
Peak memory (MB)	142	107
Lines of code	1187	493

Terra achieves comparable runtime with 25% less memory, using 60% fewer lines of code. The memory improvement comes from generating specialized tape objects that omit fields for constants.

4.7 Summary of Performance Results

Table 7: Performance Summary Across All Benchmarks

Application	Terra	Baseline	Speedup
DGEMM (GFLOPS)	43.2	53.1 (ATLAS)	within 20%
Area filter (speedup)	3.4x	1.0× (C)	3.4x
Fluid simulation	2.3x	1.0× (C)	2.3x
HMM inference (sec)	0.31	2.9 (JS)	9.4x
Serialization (MB/s)	7000	432 (Kryo)	16.2x
Assembler (MB/s)	1520	720 (Chrome)	2.1x
AD memory (MB)	107	142 (Stan)	1.33x reduction

5. Discussion

5.1 Novelty and Contributions

Terra's two-language design fundamentally differs from prior approaches in three ways. **First** , while homogeneous multi-stage programming (MetaML, MetaOCaml) keeps staging and target languages identical, Terra embraces heterogeneity. This allows the staging language (Lua) to provide garbage collection, dynamic typing, and first-class functions—features essential for compiler construction—while the target language (Terra) provides manual memory management, static typing, and hardware control. Previous heterogeneous systems [8] required restricting the target language to a subset of the source, limiting expressiveness.

Second , exotypes combine meta-object protocols (traditionally associated with dynamic languages) with staged programming to eliminate runtime overhead. Where Python's metaclasses and Lua's metatables incur dynamic dispatch costs, Terra's property functions evaluate at typechecking time, generating specialized code before execution. This bridges the gap between dynamic-language flexibility and static-language performance.

Third , shared lexical scope across languages is a seemingly small design choice with profound implications. In MetaHaskell [13], quoting a Haskell expression inside another language requires explicit environment passing. Terra's automatic capture of Lua variables in Terra quotations eliminates this boilerplate, making code generation as natural as writing templates in a scripting language.

5.2 Comparison with Domain-Specific Approaches

Halide [11]: Halide separates algorithm from schedule for image processing, similar to Darkroom. However, Halide requires C++ for frontend, ML for optimizations, and LLVMfor code generation. Darkroom using Terra achieves comparable performance (0.35s vs 0.37s for deblurring) in a unified codebase, demonstrating that Terra eliminates the need for language boundaries.

Liszt [3]: The original Liszt used Scala for compiler and C++ for runtime, splitting stencil analysis across static and dynamic phases. Terra-Liszt performs analysis entirely at runtime with Lua, simplifying deployment and enabling adaptation to dynamic mesh topology.

Stan [14]: Stan's C++ AD library uses template meta-programming, resulting in long compile times and cryptic error messages. Terra's AD implementation is 60% smaller and 25% more memory-efficient while maintaining runtime performance, demonstrating that staged programming can outperform even sophisticated template meta-programming.

5.3 Limitations

Compilation Overhead: Terra's just-in-time compilation adds latency before first function execution. For functions called only once, this overhead may dominate runtime. Our offline compilation support (terralib.saveobj) addresses this for production deployment, but the current implementation does not automatically decide when to JIT vs. pre-compile.

GPU Support: While Terra can generate CUDA kernels, the current backend does not abstract GPU memory management or automatically partition computations. Users must write kernel code similarly to CUDA C, limiting accessibility.

Debugging: Error messages from staged code can be confusing, as type errors occur during Lua execution rather than at Terra compilation. Stack traces cross language boundaries, making it difficult to locate the original source. We have implemented basic cross-language source mapping, but comprehensive debugging support remains future work.

Ecosystem: As a young language, Terra lacks the extensive libraries of C++ or Python. However, C interoperability mitigates this: existing C libraries (FFTW, BLAS, OpenCV) can be called directly without wrappers.

5.4 Future Work

Automatic Hybrid Compilation: We are exploring heuristics to automatically decide when to JIT-compile Terra functions and when to use pre-compiled versions, balancing startup latency and peak performance.

Verified Code Generation: The current system trusts user-written code generators. We plan to integrate refinement types or dependent types to verify properties of generated code (e.g., memory safety, absence of integer overflow).

Distributed Computation: Extending Terra's staging to generate code for distributed systems (MPI, Spark) could enable portable high-performance computing across clusters.

IDE Integration: The Lua-Terra hybrid suggests new debugging and profiling tools that understand cross-language provenance, allowing developers to trace performance issues from high-level DSL constructs down to generated machine code.

6. Conclusion

This paper introduced Terra, a two-language design for portable high-performance programming. By separating meta-programming (Lua) from high-performance execution (Terra), we enable the concise implementation of DSLs and active libraries that previously required multiple technologies. Key contributions include:

1. **Heterogeneous multi-stage programming** with shared lexical scope and separate evaluation, making code generation as natural as writing templates while ensuring predictable performance.
2. **Exotypes** that combine meta-object protocols with staged programming, enabling runtime generation of high-performance types with library-defined behavior.
3. **Formal semantics** clarifying the interaction between Lua evaluation, Terra specialization, and typechecking.
4. **Comprehensive evaluation** across seven domains demonstrating state-of-the-art performance with significantly reduced implementation complexity. Our results show that Terra achieves performance within 20% of ATLAS for matrix multiplication, 2.3× faster than hand-written C for image processing, and 11× faster than Kryo for serialization, while reducing code size by 1–2 orders of magnitude compared to existing implementations. By making portable high-performance programming more accessible, Terra lowers the barrier to creating DSLs and active libraries, enabling domain experts to exploit heterogeneous hardware without becoming compilation experts. We believe this approach will accelerate innovation in scientific computing, machine learning, graphics, and other performance-critical domains.

Acknowledgments

This work was supported by the Department of Energy, the National Science Foundation, and the Defense Advanced Research Projects Agency. The author thanks Pat Hanrahan, Alex Aiken, and the Stanford programming languages group for their invaluable feedback.

References

[1] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[2] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005.

[3] Z. DeVito, N. Joubert, F. Palacios, et al., "Liszt: A domain specific language for building portable mesh-based PDE solvers," in *Proc. SC11*, 2011.

[4] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proc. LFP'86*, pp. 151–161, 1986.

[5] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers, "DyC: An expressive annotation-directed dynamic compiler for C," *Theoretical Computer Science*, vol. 248, no. 1-2, pp. 147–199, 2000.

[6] W. Taha and T. Sheard, "MetaML and multi-stage programming with explicit annotations," *ACM TOPLAS*, vol. 22, no. 4, pp. 685–740, 2000.

[7] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*, pp. 30–50, 2004.

[8] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha, "Implicitly heterogeneous multi-stage programming," *New Generation Computing*, vol. 25, no. 3, pp. 305–336, 2007.

[9] G. Kiczales and J. D. Rivieres, *The Art of the Metaobject Protocol*. MIT Press, 1991.

[10] D. Syme, K. Battocchi, K. Takeda, et al., "F# 3.0—strongly-typed language support for internet-scale information sources," Microsoft Research Technical Report, 2012.

[11] J. Ragan-Kelley, A. Adams, S. Paris, et al., "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM TOG*, vol. 31, no. 4, p.32, 2012.

[12] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A language for generative models," in *Proc. UAI'08*, 2008.

[13] G. Mainland, "Explicitly heterogeneous metaprogramming with MetaHaskell," in *Proc. ICFP'12*, pp. 311–322, 2012.

[14] Stan Development Team, "Stan: A C++ library for probability and sampling, version 1.3," 2013. [Online]. Available: http://mc-stan.org/