EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Advanced Type Systems: Dependent Types, Linear Types, and Effect Systems for Compile-Time Correctness Guarantees

V Sakhtipriyanka

Assistant Professor, Department of Computer Science, Kongunadu Arts and Science College, Coimbatore, India

Article information

Received: 10th February 2026
Received in revised form: 18th March 2026
Accepted: 21st April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20654795

Abstract

A type checker is the most widely deployed program verifier in the world, run by millions of engineers who may never call it that. Most use only its weakest form, which catches little more than adding a number to a string. A richer tradition in programming-language research pushes the type system far further, until it can prove properties that today are checked, if at all, by tests and review. This paper surveys three of the most consequential of these advanced disciplines. Linear and affine types track the use of resources, guaranteeing that a file handle, a lock, or a block of memory is consumed exactly once, which eliminates leaks and use-after-free at compile time and underlies the ownership model that has recently reached mainstream systems languages. Effect systems make a function's side effects, its input and output, its exceptions, its access to state, part of its type, so the compiler can enforce that effects are handled and that pure code stays pure. Dependent types, the most powerful of the three, allow types to depend on values, collapsing the distinction between programs and proofs and making it possible to state and verify full functional correctness within the language itself. We explain the core idea behind each, trace how ideas once confined to proof assistants are entering production languages, and assess the cost, chiefly in type-checking complexity and programmer effort, that has kept the most powerful systems from widespread adoption.

Keywords:- Type Systems, Dependent Types, Linear Types, Affine Types, Effect Systems, Algebraic Effects, Formal Verification, Programming Languages, Curry-Howard Correspondence.

I. INTRODUCTION

Every statically typed language ships a proof checker, though few of its users think of it that way. When the compiler rejects a program because a function expecting an integer was handed a string, it has refuted a small theorem about that program's behaviour before the program ever runs [1]. The discipline of type theory asks an ambitious question: if a type checker can prove this, what else can it prove? The answer, developed over decades of research, is a great deal more than mainstream languages currently exploit, extending all the way to full functional correctness.

The appeal is the timing of the guarantee. A test demonstrates that a program behaved correctly on the inputs the test happened to try; a type proves a property for all inputs, and it does so at compile time, before deployment, with no runtime cost [1]. The catch is expressiveness: the simple type systems that are cheap to check can only express simple properties, and pushing toward richer guarantees means a more powerful, and more demanding, type system. Figure 1 sketches this spectrum, from untyped code through the polymorphism of ordinary functional languages to the advanced disciplines this paper concerns.

We examine three of those disciplines, chosen because each has moved, or is moving, from research into practice. Linear and affine types govern resource usage. Effect systems govern side effects. Dependent types govern logical correctness. Figure 2 summarises what each eliminates. The remainder of the paper treats them in turn, then weighs the adoption cost that determines how far each will spread.

II. LINEAR AND AFFINE TYPES: ACCOUNTING FOR RESOURCES

A. The Idea

Ordinary type systems treat values as freely copyable and discardable, which is exactly wrong for resources. A file handle should be closed once, not zero times and not twice; a lock acquired must be released; a buffer freed must not be touched again. Linear types, derived from Girard's linear logic, enforce that a value is used exactly once, and the affine variant relaxes this to at most once [6], [7]. Under such a system the compiler can guarantee, statically, that a resource is neither forgotten, which would be a leak, nor used after it has been consumed, which would be a use-after-free [6]. The accounting is done entirely at type-checking time and disappears at runtime.

B. From Theory to Mainstream Systems

For years this discipline lived mostly in research languages, but it has now reached production. Rust's ownership model is, in essence, an affine type system in practical clothing: a value has one owner, moving it invalidates the source, and the borrow checker enforces that references do not outlive what they point to, which is precisely how it eliminates the memory-safety errors that dominate systems-software incident reports [9]. Haskell has gained linear types as a first-class extension, allowing libraries to express must-use-once protocols in an otherwise non-linear language without splitting it into two [8]. The significance of these developments is that a once-esoteric corner of type theory turned out to be the key to safe, garbage-collector-free resource management at industrial scale [8], [9].

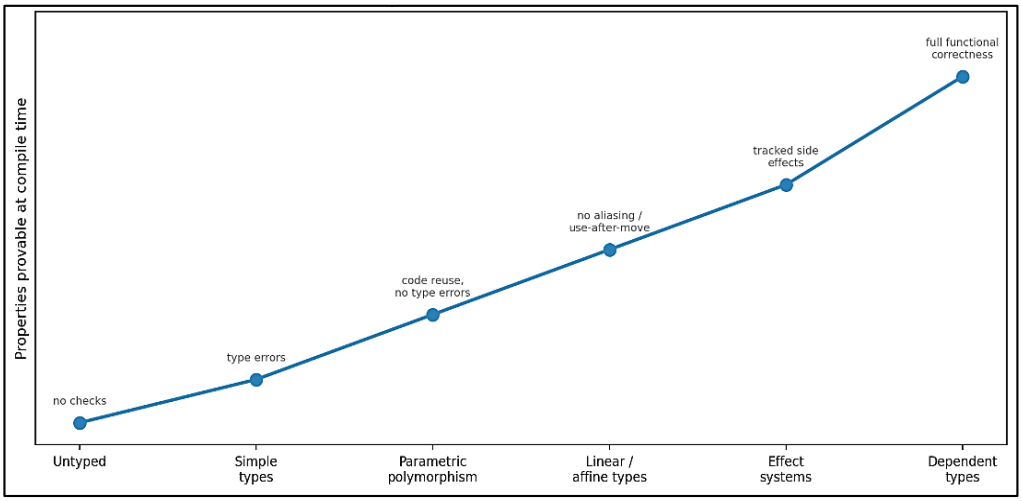

Fig 1: A spectrum of static guarantees. Each step rightward lets the type checker prove a stronger class of property, at the price of a more demanding type system.

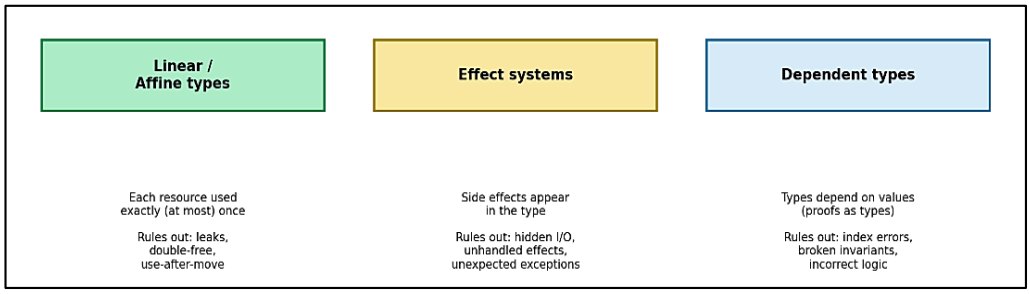

Fig 2: The three advanced disciplines compared by what each eliminates at compile time: linear and affine types account for resources, effect systems track side effects, and dependent types verify full functional correctness.

III. EFFECT SYSTEMS: MAKING SIDE EFFECTS VISIBLE

A. Types That Describe What a Function Does

A function's type ordinarily states what it takes and what it returns, but stays silent about what it does along the way, whether it reads a file, mutates global state, throws an exception, or blocks on the network. An effect system adds that missing information, annotating the type with the set of effects the function may perform [10]. Once effects are visible in types, the compiler can enforce useful disciplines: that a function declared pure performs no I/O, that every effect a program raises is eventually handled, and that effectful and pure code are not silently mixed. This turns a class of latent runtime surprises, an unexpected exception, an unhandled asynchronous operation, into compile-time errors [11].

B. Algebraic Effects and Handlers

The most influential modern formulation is algebraic effects with handlers, which separate the act of raising an effect from the code that interprets it, much as an exception separates a throw from its catch, but generalised to resumable operations [10], [11]. A program performs an abstract operation, and a surrounding handler decides what it means: in production it performs real I/O, in a test it returns canned responses, and the same code serves both because the effect and its interpretation are decoupled. This structure subsumes exceptions, mutable state, generators, and cooperative concurrency under one mechanism, generalising the earlier monadic treatment that first made side effects explicit in pure functional languages [15], and the type system tracks which operations a piece of code may perform [11], [12]. Languages designed around this idea, and extensions retrofitting it onto established ones, are an active and fast-moving area, because the same discipline that makes effects testable also makes concurrency and resource scoping easier to reason about [12].

IV. DEPENDENT TYPES: PROGRAMS AS PROOFS

A. Types That Depend on Values

The most powerful of the three disciplines erases the boundary between types and values. In a dependently typed language a type may mention a value: the type of vectors of length n, where n is an ordinary runtime number, or the type of sorted lists, or the type of a function that provably returns an index within bounds [2], [17]. Because types can now express arbitrary propositions about values, the type checker becomes a proof checker in the full sense, and a program that type-checks is a constructive proof that the stated property holds. This identification of propositions with types and proofs with programs is the Curry-Howard correspondence, and dependent type theory is its most complete realisation [2].

B. From Proof Assistants to Programming Languages

Dependent types first reached maturity in interactive theorem provers such as Coq and Agda, which are used to prove mathematical theorems and to verify software with machine-checked certainty [3], [4]. Two landmark results show what this enables: CompCert, a C compiler whose correctness is formally proved so that it provably preserves program behaviour, and seL4, an operating-system microkernel verified down to its implementation [13], [14]. A parallel effort has tried to make dependent types usable for ordinary programming rather than only proof, producing languages such as Idris that let an engineer state precise specifications and have the compiler enforce them as a normal part of writing code [5]. Recent work on quantitative type theory even unifies the resource tracking of linear types with the precision of dependent types in a single framework, suggesting the three disciplines of this paper are facets of one larger design space [16].

Table 1.Three Advanced Type Disciplines

Discipline	Core guarantee	Representative systems	Adoption stage
Linear / affine types	Resources used once; no leaks or use-after-move	Rust ownership, Linear Haskell	Mainstream (systems)
Effect systems	Side effects tracked and handled in types	Koka, OCaml effects, Eff	Emerging
Dependent types	Full functional correctness; proofs as programs	Coq, Agda, Idris, Lean	Research / verification

V. THE COST OF POWER

If these systems prove so much, the obvious question is why every language does not adopt them, and the answer is cost, paid in two currencies. The first is the difficulty of type checking itself. As types grow more expressive, deciding whether a program type-checks becomes harder, and in the fully dependent case type checking can require the programmer to supply proofs that the compiler cannot infer on its own [1], [5]. The

second currency is human effort. Writing a precise specification and convincing the type checker that the implementation meets it is real work, often comparable to writing the program, and it demands fluency in a style of reasoning that most engineers have never been taught [4], [5].

This cost explains the adoption pattern in Table 1. Linear and affine types reached the mainstream because they buy a large, concrete safety benefit, the elimination of memory errors, for a modest and well-contained increase in annotation burden, as Rust's success demonstrates [9]. Effect systems sit in between, valuable and increasingly practical but still settling into language designs [11], [12]. Full dependent types remain concentrated in verification, where the correctness of a compiler or a kernel justifies an extraordinary investment, rather than in everyday application code [13], [14]. The frontier of practical language design lies in lowering the second cost, finding ways to deliver more of the guarantee with less of the proof burden.

VI. CONVERGENCE

The three disciplines are often presented separately, but the more interesting story is their convergence. Rust took an idea from linear logic and made it ordinary. Mainstream functional languages are absorbing algebraic effects to tame concurrency and side effects. Theorem provers built on dependent types are verifying the very compilers and kernels that everything else runs on, and frameworks such as quantitative type theory show that resource tracking and dependent precision can live in one system [13], [14], [16]. The common thread is a steady migration of guarantees from runtime to compile time, and from external tools, tests, linters, separate verifiers, into the type system that engineers already run on every build. Each discipline is a different answer to the same question of what a compiler can be made to prove [1].

VII. REFINEMENT TYPES: A PRAGMATIC MIDDLE GROUND

Full dependent types are powerful but costly, and a body of work has sought most of the benefit at a fraction of the price through refinement types. A refinement type attaches a logical predicate to an ordinary type, describing not just that a value is an integer but that it is, say, a positive integer or an index strictly less than the length of a given array [19], [20]. The decisive engineering choice is to restrict these predicates to a logic that an automated solver can decide, so that an external satisfiability-modulo-theories solver discharges the proof obligations automatically rather than asking the programmer to write proofs by hand [20]. The result is a system that can verify the absence of out-of-bounds accesses, division by zero, and violated preconditions, properties firmly in the territory of dependent types, while keeping the programmer experience close to ordinary typed programming. Languages and extensions in this family, including refinement-typed dialects of mainstream functional languages and verification-oriented languages, have demonstrated the verification of real libraries with annotation rather than proof, which is why refinement types are among the most promising routes to bringing strong static guarantees into everyday code [19]. The cost reappears at the edges: predicates outside the solver's decidable fragment fall back on manual reasoning, and solver performance becomes part of the build's behaviour.

VIII. SESSION TYPES AND TYPED COMMUNICATION

The disciplines discussed so far govern a single program's values and effects, but distributed and concurrent systems fail most often at the seams, in the protocols by which components communicate. Session types extend the typing discipline to those protocols, describing the permitted sequence of messages on a communication channel as a type: send a request, then receive a response, then either repeat or close [21]. A program that type-checks against a session type is guaranteed to follow the protocol, which rules out whole categories of concurrency bug such as sending a message the other side does not expect or deadlocking on a mismatched exchange. The multiparty generalisation lifts this from two participants to many, deriving each participant's local protocol from a single global description and guaranteeing that, if every participant respects its local type, the ensemble cannot deadlock and communication stays well typed [22]. This is a striking demonstration that types can constrain not only what a program computes but how independent programs interact over time, and it connects the type-systems tradition directly to the correctness of distributed systems, where protocol mismatches are a leading source of failure [21], [22].

IX. TYPE INFERENCE AND THE USABILITY FRONTIER

Whether any of these disciplines reaches mainstream use depends as much on inference as on power, because a guarantee that demands heavy annotation will be resisted no matter how valuable it is. The Hindley-Milner system that underlies the ML and Haskell families occupies a famous sweet spot: it infers the most general type of an expression with no annotations at all, giving strong polymorphic typing for free [18]. The difficulty is that inference degrades as expressiveness grows. Adding subtyping, dependent types, or rich effects can make full inference undecidable, so the more powerful systems necessarily ask the programmer to supply more information [1], [18]. Bidirectional typing is the standard response, splitting the task into checking an expression against an expected type and synthesising a type where none is given, which recovers much of the convenience of inference

in settings where global inference is impossible and also yields more localised, comprehensible error messages [24]. A different compromise, gradual typing, lets static and dynamic typing coexist in one program so that a codebase can be migrated toward stronger guarantees incrementally rather than all at once [23]. Figure 3 places the disciplines of this paper on the two axes that govern adoption, expressive power against automation, and makes the central tension visible: the systems that prove the most tend to automate the least, and the research frontier is the effort to push points toward the upper right, delivering more guarantee for less burden. Table 2 records the correspondence that gives all of this its theoretical unity.

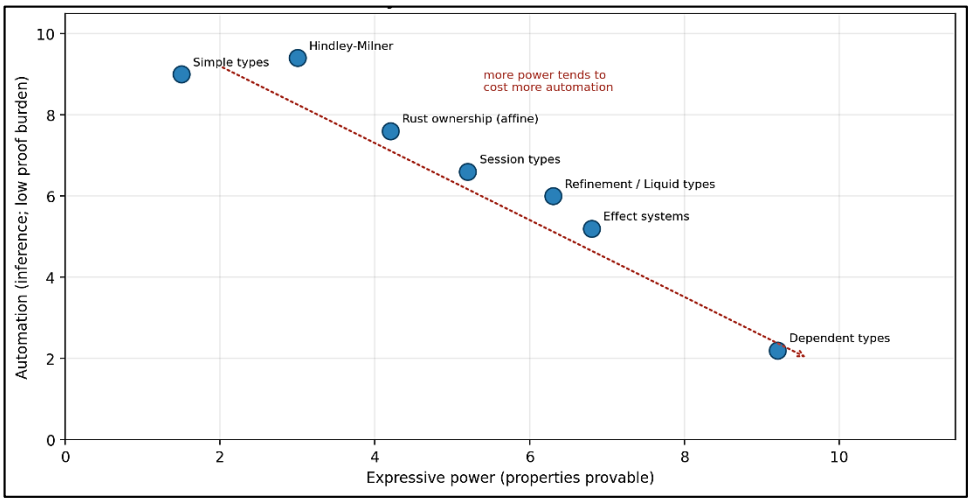

Fig 3: The usability frontier. Each discipline trades expressive power against the automation of type inference and the avoidance of manual proof; the practical research goal is to move systems toward the upper right.

Table 2. The Curry-Howard Correspondence

Logic	Type theory / programming	Consequence
Proposition	Type	A type states a claim about programs
Proof	Program (term)	Writing a program proves the claim
Implication A => B	Function type A -> B	A function transforms evidence
Conjunction / disjunction	Product / sum type	Pairing and choice of evidence
Universal quantifier	Dependent function type	Proofs parameterised by values

X. CONCLUSION

Advanced type systems represent the most practical bridge yet built between everyday programming and formal verification, because they deliver proofs through a mechanism every engineer already uses. Linear and affine types have crossed into the mainstream and now guarantee resource safety in production systems languages [8], [9]. Effect systems are crossing now, promising to make side effects and concurrency explicit and testable [11], [12]. Dependent types remain the demanding frontier, capable of proving full correctness but still costly enough to reserve for software whose failure is intolerable [5], [13], [14]. The trajectory is clear and consistent: properties once relegated to tests, documentation, and hope are moving, one discipline at a time, into the type checker, where the compiler enforces them on every build. The open engineering problem is not whether these guarantees are worth having but how to make them cheap enough that ordinary code can afford them.

REFERENCES

[1] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.
[2] P. Martin-Löf, *Intuitionistic Type Theory*. Naples, Italy: Bibliopolis, 1984.
[3] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin, Germany: Springer, 2004.
[4] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Dept. Comput. Sci. Eng., Chalmers Univ. Technol., Gothenburg, Sweden, 2007.
[5] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *J. Funct. Program.*, vol. 23, no. 5, pp. 552–593, 2013.
[6] P. Wadler, "Linear types can change the world!," in *Programming Concepts and Methods*, M. Broy and C. Jones, Eds. Amsterdam, The Netherlands: North-Holland, 1990, pp. 561–581.

[7] J.-Y. Girard, “Linear logic,” *Theoret. Comput. Sci.*, vol. 50, no. 1, pp. 1–101, 1987.
[8] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, “Linear Haskell: Practical linearity in a higher-order polymorphic language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–29, 2018.
[9] N. D. Matsakis and F. S. Klock II, “The Rust language,” in *Proc. ACM SIGAda Annu. Conf. High Integrity Lang. Technol. (HILT)*, 2014, pp. 103–104.
[10] G. D. Plotkin and M. Pretnar, “Handlers of algebraic effects,” in *Proc. 18th Eur. Symp. Program. (ESOP)*, 2009, pp. 80–94.
[11] A. Bauer and M. Pretnar, “Programming with algebraic effects and handlers,” *J. Log. Algebr. Methods Program.*, vol. 84, no. 1, pp. 108–123, 2015.
[12] D. Hillerström and S. Lindley, “Liberating effects with rows and handlers,” in *Proc. 1st Int. Workshop Type-Driven Develop. (TyDe)*, 2016, pp. 15–27.
[13] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.
[14] G. Klein *et al*., “seL4: Formal verification of an OS kernel,” in *Proc. 22nd ACM Symp. Oper. Syst. Princ. (SOSP)*, 2009, pp. 207–220.
[15] P. Wadler, “Monads for functional programming,” in *Advanced Functional Programming*, Lecture Notes in Computer Science, vol. 925. Berlin, Germany: Springer, 1995, pp. 24–52.
[16] R. Atkey, “Syntax and semantics of quantitative type theory,” in *Proc. 33rd Annu. ACM/IEEE Symp. Logic Comput. Sci. (LICS)*, 2018, pp. 56–65.
[17] H. Xi and F. Pfenning, “Dependent types in practical programming,” in *Proc. 26th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1999, pp. 214–227.
[18] L. Damas and R. Milner, “Principal type-schemes for functional programs,” in *Proc. 9th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1982, pp. 207–212.
[19] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones, “Refinement types for Haskell,” in *Proc. 19th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 2014, pp. 269–282.
[20] P. M. Rondon, M. Kawaguchi, and R. Jhala, “Liquid types,” in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 159–169.
[21] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *Proc. 7th Eur. Symp. Program. (ESOP)*, 1998, pp. 122–138.
[22] K. Honda, N. Yoshida, and M. Carbone, “Multiparty asynchronous session types,” in *Proc. 35th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2008, pp. 273–284.
[23] J. G. Siek and W. Taha, “Gradual typing for functional languages,” in *Proc. 7th Workshop Scheme Funct. Program.*, 2006, pp. 81–92.
[24] B. C. Pierce and D. N. Turner, “Local type inference,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 1–44, 2000.