Imperative Syntax for Dependent Types

BHAKTI SHAH, University of St Andrews, United Kingdom
EDWIN BRADY, University of St Andrews, United Kingdom

The average programming language researcher holds the opinion that syntax is the least important part of a programming language. The average programmer disagrees. The syntax of a functional, dependently typed programming language differs significantly from that of a typical imperative language. As a part of our goal to develop a dependent type theory and programming language tailored to imperative programmers, we wish to explore the impact of syntax on comprehensibility and usability of a dependently typed language, specifically for experienced imperative programmers. To this end, we develop a prototypical imperative syntax that can be transformed into an existing dependently typed language, Idris. We develop several transformation algorithms to take imperative constructs such as loops and statements from this syntax to terms that typecheck in Idris, allowing programs to be executed within Idris's ecosystem. We also develop an algorithm to automatically derive decidable equality instances for custom types, enabling dependent elimination with no additional effort from the programmer. As future work, we aim to conduct a formal user study to evaluate the effectiveness of this syntax for our purposes.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

ACM Reference Format:
Bhakti Shah and Edwin Brady. 2025. Imperative Syntax for Dependent Types. *Proc. ACM Program. Lang.* 1, HATRA, Article 1 (July 2025), 9 pages.

1 INTRODUCTION

In the space of programming languages, principled user studies are quite uncommon [Stefik and Hanenberg 2017]. In a study by Ko et al., an analysis of 1701 papers published at top software engineering conferences over a period of 10 years revealed that while 1392 (81%) of these papers involved some form of software engineering "tool", the number of controlled studies with human participants was just 44. Often the humans using the tool are the authors of the papers themselves, and the results of such studies are obviously subject to significant biases.

While several factors contribute to the usability of a programming language, we would like to highlight *syntax*. Stefik and Siebert conducted a formal study of the features of programming language syntax. While their work itself was focused on novice programmers, their study participants consisted of both programmers and non-programmers, and useful conclusions can be drawn that generalize to experienced programmers. In particular, one of the findings we would like to highlight is the (perhaps seemingly obvious) fact that as programmers gain experience, they rate familiar syntax higher. While we were unable to find formal studies in this space, it is clear that the functional style of programming usually adapted by most dependently typed languages differs significantly from the imperative style of programming used in most mainstream languages. It is colloquially well accepted that the learning curve for dependently typed programming languages is quite high. It seems reasonable to postulate that unfamiliar syntax plays a role in making dependently typed languages harder to learn, especially for experienced programmers. The question is, how much?

Authors' addresses: Bhakti Shah, School of Computer Science, University of St Andrews, United Kingdom, bs266@st-andrews.ac.uk; Edwin Brady, School of Computer Science, University of St Andrews, Street1 Address1, United Kingdom, ecb10@st-andrews.ac.uk.

2025. ACM 2475-1421/2025/7-ART1
https://doi.org/

We hypothesise that *imperative syntax for a dependent type theory can enhance the usability of dependent types for an experienced programmer.* In other words, simply altering surface syntax without changing the core type theory of an existing dependently typed language can lead to a more usable language for experienced imperative programmers not familiar with dependently typed programming.

We develop a syntax resembling a medley of syntactic features seen in "mainstream" imperative programming features, namely C++, Java, and Python. This syntax is "compiled" to code in Idris using simple syntactic transformations for features that exist in Idris, and algorithmic transformations for features that do not. In particular, we develop a methodology for transformation of boolean statements into their propositional equivalents to enable dependent elimination via case analysis. We also develop an algorithm for automated derivation of decidable equality for user-defined types. Our eventual goal is to conduct a formal study surveying experienced imperative programmers on the usability of this syntax, using methodology detailed by Coblenz et al. Having a fully-functional and expressive typechecker allows for interactivity within such a study, as dependently typed programming is, after all, an interactive discipline.

2 SYNTAX

Even before we get to features of dependently typed programming, we must tackle non-dependent, functional features that sit at the foundations of dependent types. Plenty of "mainstream" imperative languages have a type system. A lot of them have type systems that are quite complex and expressive, but they follow the imperative programming paradigm rather than the functional one. This means that familiar concepts such as algebraic data types and pattern matching may not look exactly as they do in functional programming languages, but exist in spirit in a style adapted to the imperative style of the language they exist in. For example, there is an existing repository of emulations of algebraic data types in a variety of languages [Wikipedia contributors 2024]. With the aim of designing a syntax loosely familiar for "most" imperative programmers, we looked at three popular languages: Java, Python, and C++. All these languages have constructs to emulate some subset of algebraic data types. We discuss these features below, with an example definition of a `List` type in each language in Table 1.

While Python supports generics, for brevity, we define a non-polymorphic list, using *dataclasses* and *tagged unions*. Similarly, in C++, we exclude templates and define a non-polymorphic list using *structs* and *variants*. Since Java is an object-oriented language, it has always had support for "ADT-like" structures using class hierarchies and objects. However, practically using these design patterns is tedious, error-prone, and verbose [Smith 2025]. Newer features added to Java allow for a much cleaner implementation of ADTs, using *sealed interfaces* [Goetz 2019] and *records*.

While both Python and C++ declare the type constructors and the type itself in separate constructs, Java declares the constructors definitionally within the type (or interface), and we believe this is a better representation for data-type declarations. As such, we adapt a syntax loosely representing Java, specifically for data types.

We now give the syntactic definitions of the features in our language. A program consists of a list of imports, followed by type and function declarations.

```
i // Imports
...
i
td | rd | f // declarations
...
td | rd | f
```

Table 1. A definition of a `List` type in different languages.

Python	`@dataclass`
	`class Nil:`
	`  pass`
	`@dataclass`
	`class Cons:`
	`  head: int`
	`  tail: List`
	`List = Nil | Cons`

C++	`struct Nil final {};`
	`struct Cons final {`
	`  int head;`
	`  std::unique_ptr<std::variant<Nil, Cons>> tail;`
	`};`
	`using List = std::variant<Nil, Cons>;`

Java	`sealed interface List<T> {`
	`  record Nil<T>() implements List<T> {}`
	`  record Cons<T>(T head, List<T> tail)`
	`  implements List<T> {}`
	`}`

Imports are libraries in Idris that must be imported to use built-in types and functions.

```
import Library.Module // as in Idris
```

Types, constructors, and functions may take implicit and explicit named arguments, enclosed in angle brackets (inspired by generics in Java and C++) and parentheses. We use the traditional curly brace syntax as seen in Java and C++ for scoping purposes.

```
td := type tname<t v,...,t v>(t v,...,t v) {
    constructor c<t v,...,t v>(t v,...,t v)
        of tname<e,...,e>(e,...,e);
    ...
    constructor c<t v,...,t v>(t v,...,t v)
        of tname<e,...,e>(e,...,e);
    }
rd := record rname<t v,...,t v>(t v,...,t v) {
    t v;
    ...
    t v;
    }
f := func fname<t v,...,t v>(t v,...,t v) of t { s }
```

Function bodies are *statements*. A statement is one of the following: declaration and assignment of a variable, assignment to a (pre-defined) variable, `return`, skip (`;;`), list of statements, `if` or `eif` (essentially eliminator constructs, explained in detail in Section 3) blocks, `switch-case`, `while` (or `ewhile` for elimination, explained in Section 3) loop. With the exception of `eif` and `ewhile`, these are all constructs commonly found in imperative programming languages, so their purpose should be self-explanatory.

```
s := let t v = e; | v = e; | return e; | ;;
```

```
  | s
    ...
    s
  | if (e) { s } else { s } | eif (e) { s } else { s }
  | switch (e,...,e){
      case (e,...,e) { s }
      ...
      case (e,...,e) { s }
      default { s } // optionally
    }
  | while (e) { s } | ewhile (e) { s }
```

Terms are distinct from statements: one can think of terms as constructs that are close to identical in functional and imperative languages. A term can be one of the following: type, variable, function application, function, natural number, boolean, unit, fully applied constructor, sum of two terms, wildcard, boolean equality or inequality of two terms, boolean negation of a term, boolean union and disjunction, and if-else, as a term. We make a distinction between if-else statements and terms semantically, but they have the same syntax.

```
e := t | v | e(e,...,e) | f | n | b | () | c(e,...,e) | e + e | _
    | e == e | e < e | ! e | e && e | e || e | if (e) { e } else { e }
b := True | False
n := 0,1,2...
```

Any types defined in Idris can be used in the surface language by simply using surface syntax: so a type such as `Vect n t` in Idris is simply represented as `Vect(n, t)`, using the traditional imperative function call syntax. Technically this means we do not need to have special syntactic constructs for any predefined types at all, but for convenience we add a few types, including `Ty`, representing `Type` in Idris.

```
t := Nat | Bool | Unit | Func(t v,...,t v => t) | Ty | tname(e,...,e)
```

We now give a few examples of transformations from the surface language to Idris in Table 2. We only show the more trivial transformations here, saving the algorithmic transforms for section 3. Type declarations in the surface language differ purely syntactically from Idris. We incorporate some very specific transforms for common patterns to enable easier typechecking in Idris: for example, terms of the form `1 + e` are transformed into `S e`. The surface language makes use of switch-case syntax for pattern matching, which is translated into the pattern matching case statements in Idris. Variable re-assignment is simply emulated using existing variable shadowing. If statements followed by additional tail statements are transformed into if statements with each branch containing the tail, and the same applies for switch-case statements followed by additional tail statements.

Table 2. Example programs in the surface language and their (mostly trivial) translations to Idris.

Surface Language	Idris

`type Vect(Nat n, Ty t) {`
` constructor Nil() of Vect(0, t);`
` constructor Cons(t head,`
`  Vect(n, t) tail)`
`  of Vect(1+n, t);`
`}`
	`data Vect : (n : Nat) -> (t : Type)`
	` -> Type where`
	`Nil : Vect 0 t`
	`Cons : (head : t) -> (tail : Vect n t)`
	` -> Vect (S n) t`

```
func replicate<Ty t>(t x, Nat n)
 of Vect(t, n) {
  switch(n) {
     case (0) {return Nil;}
     case (S(n)) {
      return Cons (x, replicate(x,n));
  }}}
```

```
replicate : {t : Type} -> t -> Nat
 -> Vect t n
replicate x n = case (n) of
   0 => Nil
   S n => Cons x (replicate x n)
```

```
func varManip (Nat x, Nat y) of Nat {
  let Nat z = x + y;
  if (z < 10) {
      z = z + 10;
  } else {
      z = z + 1;
  }
  z = z + x;
  return z;
}
```

```
varManip : Nat -> Nat -> Nat
varManip x y =
 let z : Nat = (x + y) in
  if (z < 10) then
   let z : Nat = (z + 10) in
    let z : Nat = (z + x) in
     z
  else
   let z : Nat = (z + 1) in
    let z : Nat = (z + x) in
     z
```

None of these transformations are particularly complex. The real transformations come when we bring loops into play. The next sections explore the non-trivial transforms applied to go from the surface language to Idris.

3 TRANSFORMATION OF WHILE LOOPS

While the surface language has while loops, Idris does not. It is not sufficient to simply transform the loop: one must account for the surrounding context, as the loop is not guaranteed to be the toplevel construct in a function. For simplicity, we assume the function body is a (non-empty) list of statements; it is easy to convert single statement bodies into a singleton list.

We define a recursive algorithm to convert a list of statements into a term in Idris, along with required helper declarations. The idea is to add these helper declarations to a `where` clause in the toplevel function containing the statement list. We recursively add any new variables defined to a list of header variables, obeying block scoping rules for variable definition. For a while loop in the form below,

```
func f(t1 v1,...,tn vn) of rt {
  head
  while(condition) {
    body
  }
  tail
}
```

It is transformed into a function with a recursive helper, as seen below.

```
func f(t1 v1,...,tn vn) of t {
  head
  frec(v_1,...,tn vn, vh1,...,vhn)
}
where
func frec(t1 v1,...tn vn, th1 vh1,...,thn vhn) of t {
    if(condition) {
```

```
      body
      f_rec(v_1,...,t_n v_n, v_h1,...,v_hn)
    } else {
      tail
    }
}
```

We are a bit loose with our definition of the function header, as the while loop may be within a nested statement; we assume the header text remains untouched. The recursive function generated is passed the original function parameters, along with any variables defined in the header. In the case of conditional branches in the header, a distinct recursive helper is generated for each branch, as the function signature changes based on what variables are defined in the header. We transform the while loop into a simple if statement. The positive branch of the if contains a recursive call to the helper appended to the original loop body. The negative branch contains the tail statements from the original function. This simple transformation works for any nested combination of statements, including nested and sequential loops, and loops in conditional branches.

This works well, but in a dependently typed setting, we need *more*. Conditional branches may need to be interpreted as refinements on the type of a value, to enable dependent elimination to satisfy typechecker constraints. Such a branch on a boolean does not yield dependent elimination, and to get transformed programs to typecheck as easily as possible, we must enter the land of propositions. We explore this in the next section.

4 ENABLING DEPENDENT PATTERN ELIMINATION

Since our goal is for our surface syntax to resemble a familiar imperative language, we must try to handle dependent type shenanigans under the hood wherever possible. We offer a limited solution for types with decidable equality: boolean equality checks are transformed into propositional ones, and if statements are transformed into case statements. This enables the body of the branch to utilise the proof of equality in its statements, and because of Idris' elaborate elaboration, often auto-implicitly supplies proofs to the terms that need them.

Not all conditional statements require such a transform: sometimes, booleans are just booleans. We use different keywords for loops and conditional statements to indicate when such a transform is to be applied: `ewhile` and `eif`. The surface language supports boolean equality and inequality, however transformations are only semantically valid in Idris if `DecEq` instantiations of the type exist. We also include a translation from boolean to propositional inequalities of natural numbers for simplicity, and because the required instantiations exist in the Idris standard library.

5 DERIVING DECIDABLE EQUALITY FOR CUSTOM TYPES

Since the use of custom user-defined types is more than likely, we also develop an algorithm for automated derivation of decidable equality instances for any types defined. We give an example for the standard `Vect` type, as defined in Table 2 — the definition in the surface language automatically generates this instance along with the translated type declaration in Idris. This allows users to write arbitrary types and get dependent elimination, as seen above, for free.

```
{n : Nat} -> {t : Type} -> (DecEq t) => DecEq (Vect n t) where
  decEq Nil Nil = Yes Refl
  decEq (Cons h1 t1) (Cons h2 t2) with (decEq h1 h2)
    decEq (Cons h1 t1) (Cons h1 t2) | Yes Refl with (decEq t1 t2)
      decEq (Cons h1 t1) (Cons h1 t1) | Yes Refl | Yes Refl = Yes Refl
      decEq (Cons h1 t1) (Cons h1 t2) | Yes Refl | No prf = No (\h =>
        prf (case h of Refl => Refl))
```

```
    decEq (Cons h1 t1) (Cons h2 t2) | No prf = No (\h =>
      prf (case h of Refl => Refl))
```

We determine the necessary constraints based on the type parameters. Constraints for type families are constructed accordingly: an example can be seen below. We add all type parameters as implicit arguments to the instance, to assist the typechecker.

```
data DPair : (a : Type) -> (p : (x : a) -> Type) -> Type where
MkDPair : (x : a) -> (y : (p x)) -> DPair a p

{a : Type} -> {p : (x : a) -> Type} -> (DecEq a, (x : a) -> DecEq (p x))
 => DecEq (DPair a p) where
...
```

The argument cases for the `decEq` function are generated by taking the cartesian product of constructors, and then throwing out pairs of constructors that have definitionally unequal types. For example, in the Vector example, there are no cases for `decEq Nil Cons` and `decEq Cons Nil` because they do not have the same type. Note that this means constructors with definitionally unequal but semantically unifiable types will not have cases generated: this is something we wish to fix in future work that is able to leverage the type system of Idris during translation. We pattern match on the decidable equality of constructor arguments recursively, giving the appropriate proof for each branch. Constructor arguments that appear in the type are skipped — in the case of the `Singleton` type, the argument is not pattern matched on: equality of types implies equality of values.

6 EXAMPLE: LINEAR SEARCH THROUGH A VECTOR

We now demonstrate an example of a program utilizing the all features and transforms described above. Consider a linear search for a value `x` through a `Vect` of length `n`, as in Listing 7. While it is possible to use our custom defined `Vect` if we define an index function (which is straightforward), we simply use the Idris defined type `Vect` with a defined index function for brevity. The function either returns `Nothing` if the value is not found, or some index guaranteed to be within the bounds of the vector (using the `Fin` type) if it succeeds. This is transformed into the program in Listing 8. Because of the use of the `eif` and `ewhile` keywords, both boolean conditionals are transformed into propositional statements, and we match on the constructors of the `Dec` type. Note that we must still specify a conversion from `Nat` to `Fin` in the surface language, but the proof that the conversion function requires must not be specified: Idris will fill it in from the context because of the explicit pattern match.

```
import Decidable.Equality
import Data.Vect

func search (Nat n, Vect(n, Nat) ls, Nat x) of Maybe(Fin(n)) {
  let Nat i = 0;
  let Maybe(Fin(n)) ret = Nothing;
  ewhile(i < n) {
      eif (index(natToFinLT(i), ls) == x) {
          ret = Just(natToFinLT(i));
      }
      else {;;}
      i = 1 + i;
  }
  return ret;
```

```
}
```

Listing 7. Linear search through a Vect in the surface language

```
import Decidable.Equality
import Data.Vect

search : (n : Nat) -> (ls : Vect n Nat) -> (x : Nat) -> Maybe (Fin n)
search n ls x =
  let i : Nat = 0 in
    let ret : Maybe (Fin n) = (Nothing) in
      (search_rec0 n ls x i ret)
where
  search_rec0 : (n : Nat) -> (ls : Vect n Nat) -> (x : Nat) -> (i : Nat)
  -> (ret : Maybe (Fin n)) -> Maybe (Fin n)
  search_rec0 n ls x i ret =
    (case (isLT i n) of
      (No noprf) => ret
      Yes yesprf => (case (decEq (index (natToFinLT i) ls) x) of
        No noprf => let i : Nat = (S i) in
          (search_rec0 n ls x i ret)
        Yes yesprf => let ret : Maybe (Fin n) = Just (natToFinLT i) in
          let i : Nat = S i in
            search_rec0 n ls x i ret))
```

Listing 8. Linear search through a Vect, translated to Idris

7 CONCLUSION

While naïve transformations of imperative features using primarily syntactic constructs have some promising results, it is clear that for true, non-trivial usability, the development of a specialized type system is required. For example, the simulation of mutability without actual mutability is quite pointless if one is truly trying to mutate variables in-place. Since this is something that fundamentally clashes with the underlying type-system, it is simply not achievable through pure syntax.

Another consideration is that these are all programs that "just work": most programmers will, at some point, write code that does not compile. In those cases, error messages may be even more incomprehensible than those one would encounter while writing Idris code by hand, simply due to the "unnatural" code generated by algorithmic transformation. Comprehensibility of error messages is known to be an important factor in the usability of programming languages, and one of the biggest factors when dealing with dependently typed languages [Juhosova et al. 2025]. It is also not contested that dependently typed programming languages are associated with some particularly nasty error messages, particularly during type-checking and unification. One common problem is associated with *definitional equality* of types. Indeed, "error messages that result from failures of definitional equality are not always very easy to understand, because they may be phrased in terms of the internals of functions" [Christiansen 2023].

These observations make it clear that just introducing syntax is not remotely enough — significant semantic developments need to be made at the type-theoretic level to build a truly usable language with these features. However, for our purpose of studying syntax, we believe the limited space of reading and executing correct programs is enough, and our developed syntax fulfills its purpose.

We designed an imperative-style syntax for an existing dependently typed language, Idris. The purpose of this experiment is not to create a usable language, rather to create a potential candidate for a usable syntax that can be used for a language more type-theoretically suited to imperative programming. Semantics are important, but syntax is too — and this direction of work aims to explore the syntactic considerations associated with designing a dependently typed language with imperative features. We aim to investigate our hypotheses about the usability of such a syntax with this minimal prototype, intended as a contained environment for a comprehensibility user study.

REFERENCES

David Thrane Christiansen. 2023. *Functional Programming in Lean.*

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2020. *PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design.* (2020). https://arxiv.org/abs/1912.04719 arXiv: 1912.04719 [cs.HC].

Brian Goetz. 2019. *Data Classes and Sealed Types for Java.* (2019). https://openjdk.org/projects/amber/design-notes/records-and-sealed-classes.

Sara Juhosova, Andy Zaidman, and Jesper Cockx. Apr. 2025. " Pinpointing the Learning Obstacles of an Interactive Theorem Prover." In: *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC).* IEEE Computer Society, Los Alamitos, CA, USA, (Apr. 2025), 159–170. doi:10.1109/ICPC66645.2025.00024.

Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. Feb. 2015. "A practical guide to controlled experiments of software engineering tools with human participants." *Empirical Softw. Engg.*, 20, 1, (Feb. 2015), 110–141. doi:10.1007/s10664-013-9279-3.

Magnus Smith. 2025. (2025). https://blog.scottlogic.com/2025/01/20/algebraic-data-types-with-java.html.

Andreas Stefik and Stefan Hanenberg. 2017. "Methodological Irregularities in Programming-Language Research." *Computer*, 50, 8, 60–63. doi:10.1109/MC.2017.3001257.

Andreas Stefik and Susanna Siebert. Nov. 2013. "An Empirical Investigation into Programming Language Syntax." *ACM Trans. Comput. Educ.*, 13, 4, (Nov. 2013). doi:10.1145/2534973.

Wikipedia contributors. 2024. *Comparison of programming languages (algebraic data type) — Wikipedia, The Free Encyclopedia.* (2024). https://en.wikipedia.org/w/index.php?title=Comparison_of_programming_languages_(algebraic_data_type)&oldid=1266445555.