

# Coming to Terms with Your Choices

An Existential Take on Dependent Types

GEORG SCHMID, EPFL, Switzerland

OLIVIER BLANVILLAIN, EPFL, Switzerland

JAD HAMZA, EPFL, Switzerland

VIKTOR KUNČAK, EPFL, Switzerland

Type-level programming is an increasingly popular way to obtain additional type safety. Unfortunately, it remains a second-class citizen in the majority of industrially-used programming languages. We propose a new dependently-typed system with subtyping and singleton types whose goal is to enable type-level programming in an accessible style. At the heart of our system lies a non-deterministic choice operator. We argue that embracing non-determinism is crucial for bringing dependent types to a broader audience of programmers, since real-world programs will inevitably interact with imprecisely-typed, or even impure code. Furthermore, we show that singleton types combined with the choice operator can serve as a replacement for many type functions of interest in practice. We establish the soundness of our approach using the Coq proof assistant. Our soundness approach models non-determinism using additional function arguments to represent choices. We represent type-level computation using singleton types and existential types that quantify over choice arguments. To demonstrate the practicality of our type system, we present an implementation as a modification of the Scala compiler. We provide a case study in which we develop a strongly-typed wrapper for Spark datasets.

## 1 INTRODUCTION

Dependent types have been met with considerable interest from the research community in recent years. Their primary application so far has been in proof assistants such as Agda [Norell 2007], Coq [Bertot and Castran 2010] and Idris [Brady 2013], where they provide a sound and expressive foundation for theorem proving. However, dependent types are still largely absent from general-purpose programming languages, despite a long history of lightweight approaches [Xi and Pfenning 1998]. In the context of Haskell, much research has gone into extending the language to support computations on types, for instance in the form of functional dependencies [Jones 2000], type families [Kiselyov et al. 2010] and promoted datatypes [Yorgey et al. 2012]. These techniques have seen adoption by Haskell programmers, showing that there is a real demand for such mechanisms. Furthermore, recent research has explored how dependent types could be added to the language for the same purpose [Eisenberg 2016; Weirich et al. 2017]. In a largely orthogonal direction, inference for dependent refinement types is reaching significant maturity [Vazou et al. 2015, 2018, 2017].

Dependently-typed languages often rely on a unified syntax to describe both terms and types. The simplicity of this approach is unfortunately at odds with the design of most programming languages, where types and terms are expressed using separate syntactic categories. Singleton types provide a simple solution to this problem by allowing every term to be represented as a type. The singleton type of a term therefore gives us the most precise specification for that term.

In this paper, we report on our attempt at combining an industrial mixed-paradigm language, Scala, with dependent types. We offer both a formalization of our type system and a discussion of

---

Authors' addresses: Georg Schmid, LARA, EPFL, Switzerland, georg.schmid@epfl.ch; Olivier Blanvillain, LAMP, EPFL, Switzerland, olivier.blanvillain@epfl.ch; Jad Hamza, LARA, EPFL, Switzerland, jad.hamza@epfl.ch; Viktor Kunčák, LARA, EPFL, Switzerland, viktor.kuncak@epfl.ch.

---

2020. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Programming Languages*.

the challenges faced in a practical implementation. It is our hope that the present paper will serve as a guide for other language implementors interested in pushing the limit of their type systems by adding dependent types.

Unlike proof assistants, we do not aim to use types as a general-purpose logic, which would favor designs ensuring totality of functions through termination checks. Instead, our focus is on improving type safety by increasing the expressive power of the type system.

We present  $\lambda_{<:\{\}}^{\text{nd}}$ , a dependently-typed calculus with subtyping and singleton types. The main novelty of our calculus is a new approach to expressing type-level computation that, at first, seems diametrically opposed to the purity other systems favor. A new term is added for non-deterministic choice from a base type, similar to Floyd’s choice operator [Floyd 1967]. Designing a sound system in the presence of non-determinism is challenging. Our solution provides systematic translation of non-determinism using additional parameters that are existentially quantified at a syntactically well-defined point. Consequently, a term in  $\lambda_{<:\{\}}^{\text{nd}}$  may reduce to different values. Our system generalizes the traditional notion of singleton type: when the lifted term  $t$  contains a non-deterministic choice, the resulting type  $\{t\}$  denotes the set of values that  $t$  could possibly reduce to. As a result, our type system is capable of type computations by manipulating types which are based on terms, but can nonetheless contain more than a single value. In combination with subtyping, this allows us to seamlessly integrate with impure, or imprecisely-typed programs.

Our contributions are as follows:

- We present our calculus  $\lambda_{<:\{\}}^{\text{nd}}$ , which illustrates the novel elements of our extension to Scala (Section 3). The type system of  $\lambda_{<:\{\}}^{\text{nd}}$  combines dependent types, subtyping and a generalization of singleton types to non-deterministic terms. We demonstrate how the interplay of these features allows us to leverage term-level programs for type-level computation.
- We provide a soundness proof of  $\lambda_{<:\{\}}^{\text{nd}}$  by reusing reducibility semantics of System FR [Hamza et al. 2019]. Using its semantics we prove the soundness of our rules (Section 4). These proofs are mechanized using the Coq proof assistant [Bertot and Castran 2010]. The formalization is available in the additional materials.
- We show a concrete use-case of our system by implementing it as an extension of Scala (Section 5), and using it to develop a strongly-typed wrapper for Apache Spark [Zaharia et al. 2016] (Section 6). Thanks to dependent types, we can statically ensure the type safety of database operations such as join and filter. We compare our implementation with an equivalent implicit-based one and show remarkable compilation time savings.

## 2 MOTIVATING EXAMPLES

We begin by motivating why dependent types are desirable in general purpose programming, and how one might use them to improve type safety. In our first example, we design an API that keeps track of database tables’ schemas in the type. We demonstrate how dependently-typed list operations can be used to compute schemas resulting from join operations at the type level. Our second example shows how to build a safer version of the zip operation on lists that only accepts equally-sized arguments. The examples in this section are written in our dependently-typed extension of Scala described in Section 5.

### 2.1 Safe Join

As a first step, we show how our system supports type-level programming in the style of term-level programs. Consider the following definition of the list datatype, which is standard Scala up the **dependent** keyword:

```
sealed trait Lst { . . . }
```

```

dependent case class Cons(head: Any, tail: Lst) extends Lst
dependent case class Nil() extends Lst

```

We can define list concatenation in the usual functional style of Scala<sup>1</sup>, that is, using pattern matching and recursion:

```

sealed trait Lst:
  dependent def concat(that: Lst) <: Lst =
    this match
      case Cons(x, xs) => Cons(x, concat(xs, that))
      case Nil() => that

```

By annotating a method as **dependent**, the user instructs our system that the result type of `concat` should be as precise as its implementation. Effectively, this means that the body of `concat` is lifted to the type level, and will be partially evaluated at every call site to compute a precise result type which *depends* on the given inputs. For recursive **dependent** methods such as `concat`, we infer types that include calls to `concat` itself. The `<:` annotation lets us provide an upper bound on `concat`'s result type, which will be used while type checking the method's definition. Finally, by qualifying the definition of `Cons` and `Nil` as **dependent** we also allow their constructors and extractors to be lifted to the type level. Using these definitions, we can now request the precise type whenever we manipulate lists by annotating the new **val** binding with **dependent**:

```

dependent val l1 = Cons("A", Nil())
dependent val l2 = Cons("B", Nil())
dependent val l3 = l1.concat(l2)
l3.size: { 2 }
l3: { Cons("A", Cons("B", Nil())) }

```

Enclosing a pure term in braces (`{ ... }`) denotes the singleton type of that term. In the last two lines of this example we are therefore asking our system to prove that `l3` has size 2 and is equivalent to `Cons("A", Cons("B", Nil()))`.

In Scala we often deal with impure or imprecisely-typed code, however. To integrate with such terms, we provide the `choose[T]` construct. Operationally, we interpret `choose[T]` as a non-deterministic choice from `T`, which can be modeled faithfully on the type level as an existentially quantified inhabitant of `T` in a singleton type. Thus, we equate `{ choose[T] }` to `T`, and when typing an impure term such as `Cons(readString(), Nil())` we can assign the precise type `{ Cons(choose[String], Nil()) }`. Returning to the previous example, this means that even in the presence of impurity, we can perform useful type-level computation and checking:

```

dependent val l2 = Cons(readString(), Nil())
dependent val l3 = l1.concat(l2)
l3: { Cons("A", Cons(choose[String], Nil())) }

```

In a style similar to `concat`, we can define `remove` on `Lst`:

```

sealed trait Lst:
  dependent def remove(e: String) <: Lst =
    this match
      case Cons(head, tail) =>
        if (e == head) tail
        else Cons(head, tail.remove(e))

```

<sup>1</sup>Our examples use the indentation-based syntax introduced in Scala 3.0.

```
case _ => throw new Error("element not found")
```

Removing "B" yields the expected result, while trying to remove "C" from 13 leads to a *compilation error*, since the given program will provably fail at runtime.

```
13.remove("B"): { Cons("A", Nil()) }
13.remove("C") // Error: element not found
```

The lists we defined so far can be used to implement a type-safe interface for database tables.

```
dependent case class Table(schema: Lst, data: spark.DataFrame):
  dependent def join(right: Table, col: String) <: Table =
    val s1 = this.schema.remove(col)
    val s2 = right.schema.remove(col)
    val newSchema = Cons(col, s1.concat(s2))
    val newData = this.data.join(right.data, col)
    new Table(newSchema, newData)
```

In this example, we wrap a weakly-typed implementation of Spark's `DataFrame` in the **dependent** class `Table`. The first argument of this class represents the schema of the table as a precisely-typed list. The second argument is the underlying `DataFrame`. In the implementation of `join`, we execute the join operation on the underlying tables (`newData`) and compute the resulting schema corresponding to that join (`newSchema`). By annotating the `join` method as **dependent**, the resulting schema is reflected in the type:

```
dependent val schema1 = Cons("age", Cons("name", Nil()))
dependent val schema2 = Cons("name", Cons("unit", Nil()))
dependent val table1 = Table(schema1, . . . )
dependent val table2 = Table(schema2, . . . )
dependent val joined = table1.join(table2, "name")
joined: { Table(Cons("name", Cons("age", Cons("unit", Nil()))), choose[DataFrame]) }
```

Reflecting table schemas in types increases type safety over the existing weakly-typed interface. For instance, it becomes possible to raise compile-time errors when a user tries to use non-existent columns. This is an improvement over the underlying Spark implementation that would instead fail at runtime.

## 2.2 Safe Zip

Our first example demonstrated how dependent methods allow inference of precise types. Conversely, we can also use singleton types to constrain method parameters further. In this example, our goal is to write a safer wrapper for functions like `zip` that should only be applicable to lists of the same length. To accomplish this, we can constrain the second parameter of `zip` as follows:

```
def safeZip(xs: Lst, ys: { sizedLike(xs) }) = unsafeZip(xs, ys)
```

Here we would like `{ sizedLike(xs) }` to be inhabited by all lists of equal length as `xs`, regardless of their elements' values. How can this be achieved, given that `sizedLike(xs)` is a term? By exploiting the non-deterministic interpretation of `choose[T]`, we can provide a succinct definition for `sizedLike`:

```
dependent def sizedLike(xs: Lst) <: Lst =
  xs match
    case Nil() => Nil()
    case Cons(x, ys) => Cons(choose[Any], sizedLike(ys))
```

Consider, for instance, the meaning of  $\{ \text{ sizedLike}(xs) \}$  for  $xs = \text{Cons}(1, \text{Cons}(2, \text{Nil}()))$ . After reduction, we obtain  $\{ \text{Cons}(\text{choose}[\text{Any}], \text{Cons}(\text{choose}[\text{Any}], \text{Nil}())) \}$ , which is a type that represents all lists of size 2. Thus `safeZip` requires that every caller prove that  $xs$  and  $ys$  are of the same length, which ensures that the underlying implementation in `unsafeZip` will never fail or truncate elements from one of the lists.

Note that, unlike `concat` and `remove` that can be used both on the term and the type level, `sizedLike` is here intended to be used as a type function, but not at runtime.

### 2.3 Discussion: From Choices to Existentials

Note that  $\{ \text{ sizedLike}(xs) \}$  cannot be readily expressed using existential types and singletons alone. The given list  $xs$  might be of an arbitrary size, so the number of existentials needed for all the occurrences of `choose[Any]` is abstract at this point. More specifically,  $\{ \text{ sizedLike}(xs) \}$  can be seen as a union of an unknown number of existential types:

$$\{ \text{nil} \} \cup \exists x_1 : \text{Top}. \{ \text{cons } x_1 \text{ nil} \} \cup \exists x_1 : \text{Top}. \exists x_2 : \text{Top}. \{ \text{cons } x_1 (\text{cons } x_2 \text{ nil}) \} \cup \dots$$

An important contribution of our type system is that it allows users to express such existential quantifications conditional on the program unfolding. Our calculus (described in Section 3) achieves this by encoding all non-deterministic choices using a single existential per-type annotation. In particular, we represent  $\{ \text{ sizedLike}(xs) \}$  by

$$\exists z : \text{Trail}. \{ \text{ sizedLike}' z xs \}$$

where  $(\text{ sizedLike}' z xs)$  is defined by

$$xs \text{ match nil}; x, y \Rightarrow \text{cons } (\text{unpack } z.1) (\text{ sizedLike}' z.2.3 y)$$

Conceptually,  $z : \text{Trail}$  corresponds to a map of input values passed to a deterministic version of the program, i.e., `sizedLike'`. Programs resulting from our encoding are pure and deterministic, so we can perform equational reasoning and apply well-understood techniques for designing sound type systems. At the same time, our encoding is adequate with respect to non-determinism (which, in turn, can approximate other language features). In our example,  $(\text{unpack } z.1)$  extracts the value at index `.1` from the input  $z$ . Note that using the argument of the recursive call,  $z.2.3$ , we ensure that each invocation of `choose[T]` in the original program is translated with a different index (Subsection 3.2). This is necessary for `sizedLike'` to faithfully model the original (non-deterministic) `sizedLike`, in the sense that each invocation of `choose[T]` can be mapped to a different value. For instance, when  $xs$  is a concrete list of two elements, we end up with a type encoded as

$$\exists z : \text{Trail}. \{ \text{cons } (\text{unpack } z.1) (\text{cons } (\text{unpack } z.2.3.1) \text{ nil}) \}$$

which, given our interpretation of the `Trail` type, selections like  $z.1$ , and the `unpack` operation, is equivalent to all the lists of two elements.

During type checking, we explicitly eliminate the references to `unpack` and replace them by fresh existentials:

$$\exists x_1 : \text{Top}. \exists x_2 : \text{Top}. \{ \text{cons } x_1 (\text{cons } x_2 \text{ nil}) \}$$

That is, we “untangle” individual existentials that had previously been tied up together (Subsection 3.4). As part of our overall soundness proof (Section 4) we show that untangling produces equivalent types, which allows us to match different occurrences of types containing non-deterministic choices when they denote the same sets of values.

Our type system rules are designed to support type checking with such existential types and subtyping. We find that it achieves an appealing combination of expressive power and simplicity: the developers can denote types using functions that generate sets of values, instead of manipulating syntactic representations of types. Even if our current set of type-checking rules does not cover as

many type equivalences as we may wish to have, our soundness approach based on reducibility semantics and System FR [Hamza et al. 2019] allows us to modularly introduce and prove additional rules in the future.

<b>Terms and Types of <math>\lambda_{&lt;\{\}}^{\text{nd}}</math>:</b>	
$p, t$	$:= x \mid \lambda x:T. t \mid t t \mid \text{nil} \mid \text{cons } t t \mid t \text{ match } t; x, y \Rightarrow t \mid \text{fix}_n(x:T \Rightarrow t, t) \mid \text{choose}[B]$
$S, T, U, V$	$:= B \mid \{t\}_T \mid \Pi x:T. T$
$B$	$:= \text{Top} \mid \text{List}$
<b>Values:</b>	
$v, v^{\text{Top}}$	$:= x \mid \lambda x:T. t \mid v^{\text{List}}$
$v^{\text{List}}$	$:= \text{nil} \mid \text{cons } v v^{\text{List}}$

Fig. 1. The terms and types for  $\lambda_{<\{\}}^{\text{nd}}$ .

<b>Evaluation contexts:</b>	
$\mathcal{E}$	$:= [] \mid \mathcal{E} t \mid v \mathcal{E} \mid \text{cons } \mathcal{E} t \mid \text{cons } v \mathcal{E} \mid \mathcal{E} \text{ match } t; x, y \Rightarrow t \mid$
<b>Term evaluation:</b>	
$\frac{t \rightarrow_{\beta} t'}{\mathcal{E}[t] \rightarrow_{\beta} \mathcal{E}[t']}$	(BCTX)
$\frac{}{(\lambda x:A. t) v \rightarrow_{\beta} t[x \mapsto v]}$	(BAPP)
$\frac{}{(\text{nil match } t_1; x, y \Rightarrow t_2) \rightarrow_{\beta} t_1}$ (BMATCHNIL)	
$\frac{}{((\text{cons } v_1 v_2^{\text{List}}) \text{ match } t_1; x, y \Rightarrow t_2) \rightarrow_{\beta} t_2[x \mapsto v_1][y \mapsto v_2^{\text{List}}]}$ (BMATCHCONS)	
$\frac{n = n' + 1}{\text{fix}_n(x:A \Rightarrow t_1, t_2) \rightarrow_{\beta} t_1[x \mapsto \text{fix}_{n'}(x:A \Rightarrow t_1, t_2)]}$ (BFIXREC)	
$\frac{n = 0}{\text{fix}_n(x:A \Rightarrow t_1, t_2) \rightarrow_{\beta} t_2}$ (BFIXDEFAULT)	
$\frac{fV(v) = \emptyset}{\text{choose}[\text{Top}] \rightarrow_{\beta} v}$	(BCHOOSETOP)
$\frac{fV(v^{\text{List}}) = \emptyset}{\text{choose}[\text{List}] \rightarrow_{\beta} v^{\text{List}}}$	(BCHOOSELIST)

Fig. 2. The term evaluation rules and evaluation contexts.

### 3 OUR SYSTEM

We present a calculus and a type system that capture the core mechanisms required for type-level computation in a dependently-typed language with subtyping. While an implementation on top of Scala must operate in the presence of a much more general subtyping relation, our formalism does not cover all the features of Scala's type system. In the following section, we introduce a functional language with primitives for operating on Lisp-like lists, which gives similar power as the closed type hierarchies that our Scala-implementation can reason about. An extension to other algebraic data types should be straightforward. Our calculus also supports non-deterministic choice from base types and Top. This choice operator allows us, on the one hand, to model imprecisely-typed functions and, on the other hand, to emulate type-level computation.

#### 3.1 Syntax and Semantics

The terms and types of our calculus,  $\lambda_{\leq 1}^{\text{nd}}$ , are defined in Figure 1. We consider terms and types equivalent up to alpha-renaming. As usual, variables are named  $x, y$  or  $z$ . We denote the set of free variables of a term  $t$  by  $\text{fv}(t)$ . Our language contains first-class functions and constructors for lists, along with pattern matching, and a fixpoint combinator. Programs in our language always terminate because our fixpoint combinator is bounded to a maximum recursion depth and returns a default value otherwise. In  $\text{fix}_n(x : T \Rightarrow t_1, t_2)$ ,  $n$  corresponds to the maximum recursion depth,  $t_1$  is the body of fix and  $t_2$  is the default value. We expect that our approach extends to more general solutions, for example, requiring proofs of termination as in most dependently-typed languages [Bertot and Castran 2010; Norell 2007], or controlling reduction on the type level using iso-types [Yang et al. 2016].

The small-step operational semantics given in Figure 2 is mostly standard, save for two aspects. First, term evaluation does not get stuck on variables (we include them among the values  $v$ ) and behaves non-deterministically on the term  $\text{choose}[B]$ , which evaluates to an arbitrary value of type  $B$  (i.e., base types or Top). Unlike many other dependently-typed systems, this allows us to express more than just purely-functional programs, as  $\text{choose}[B]$  conservatively models a term lacking referential transparency. Second,  $\text{choose}[B]$  allows us to model the situation in which parts of our program may be pure, but are typed in a less precise manner.

Besides the dependent products usually found in dependently-typed languages, we also include singleton types [Hayashi 1991], denoted  $\{t\}_U$ , which are inhabited only by terms observationally equivalent to  $t$ . The *underlying* type  $U$  provides an upper bound for the singleton type and is used to guide type inference. For instance,  $\text{nil}$  can be typed precisely as  $\{\text{nil}\}_{\text{List}}$ . The identity function on Top can be typed as:

$$\{\lambda x : \text{Top}. x\}_{\Pi x : \text{Top}. \{x\}_{\text{Top}}}$$

For better legibility we often write singletons without their underlying types:  $\{\lambda x : \text{Top}. x\}$ .

When used in a type annotation,  $\text{choose}[B]$  existentially quantifies over an arbitrary value in  $B$ . As a result, the base type List is equivalent to  $\{\text{choose}[\text{List}]\}$  which in turn allows us to express the type of non-empty lists as follows:

$$\{\text{cons}(\text{choose}[\text{Top}]) (\text{choose}[\text{List}])\}$$

During type checking, our system rewrites  $\text{choose}[B]$  to explicit existential quantifications, that are not available in the surface syntax. Internally, we end up with the following type for the above example:

$$\exists x_1 : \text{Top}. \exists x_2 : \text{List}. \{\text{cons } x_1 \ x_2\}$$

<b>Terms and Types of <math>\lambda_{&lt;(),\{}}^{\text{det}}</math>:</b>	
$p, t$	$:= x \mid \lambda x:T. t \mid t t \mid \text{nil} \mid \text{cons } t t \mid t \text{ match } t; x, y \Rightarrow t \mid$ $\text{fix}_n(x:T \Rightarrow t, t) \mid t.1 \mid t.2 \mid t.3$
$S, T, U, V$	$:= B \mid \{t\}_T \mid \Pi x:T. T \mid$ $\text{Cons } T_1 T_2 \mid t \text{ Match } T; x, y \Rightarrow T \mid \exists x:T. T \mid \text{Trail}$
<b>Values:</b>	$v, v^{\text{Top}} := x \mid \lambda x:T. t \mid v^{\text{List}} \mid v.1 \mid v.2 \mid v.3$

Fig. 3. The terms and types in  $\lambda_{<(),\{}}^{\text{det}}$ . Constructs not present in  $\lambda_{<(),\{}}^{\text{nd}}$  are marked in gray.

Semantically, this type corresponds to the infinite union over all elements  $x_1: \text{Top}, x_2: \text{List of } \{\text{cons } x_1 x_2\}$ . As a first step towards representing the impure  $\text{choose}[B]$  construct, we translate programs in  $\lambda_{<(),\{}}^{\text{nd}}$  to a deterministic language, as described below.

### 3.2 Lowering to a Deterministic Language

In this section, we detail how we eliminate the non-deterministic  $\text{choose}[B]$  construct. The essence of our translation is to collect all the choices that a non-deterministic execution might need and turn them into an input argument of a deterministic version of the program. Our translation is therefore analogous to a translation from a non-deterministic Turing machine to a deterministic machine that acts as the corresponding verifier [Sipser 2013, Theorem 7.20 on p. 294].

The encoding is performed before type checking, and as a consequence  $\text{choose}[B]$  is absent from subsequent typing rules. Depending on the context where  $\text{choose}[B]$  occurs, it takes on different meanings. In the context of terms,  $\text{choose}[B]$  refers to a specific value in  $B$ , picked non-deterministically during program execution. When invoked from inside a singleton type, such as in  $\{\text{choose}[B]\}$ , our translation will give it the meaning of all values in  $B$ . This result arises due to existential quantification over choices, which the translation introduces independently for each type annotation in the program.

We define a lowering from  $\lambda_{<(),\{}}^{\text{nd}}$ , the surface language, to  $\lambda_{<(),\{}}^{\text{det}}$ , which we then use in subsequent type checking. In Figure 3 we give the terms and types of  $\lambda_{<(),\{}}^{\text{det}}$  with the differences to  $\lambda_{<(),\{}}^{\text{nd}}$  highlighted in gray. First, note the absence of  $\text{choose}[B]$ , which is eliminated by the lowering. We include types for list constructors ( $\text{Cons } T_1 T_2$ ) and matches. The type for matches,  $t \text{ Match } T_2; x, y \Rightarrow T_3$ , represents either  $T_2$  or the substituted form of  $T_3$ , depending on the value of  $t$ . These types are used later, during type inference, and guide subtyping. The other additions, i.e., existential types, the base type  $\text{Trail}$ , and selections on trails,  $t.n$ , are discussed below in the lowering step.

**3.2.1 Encoding  $\text{choose}[T]$ .** Lowering produces a deterministic program that, thanks to an extra parameter, captures all of the potential behaviors of the original (non-deterministic) program. We express the lowered program as a function of *trails*. Intuitively, a trail  $\tau$  contains all the information necessary to recover the non-deterministic choices made in a concrete execution of the original program.



$$\begin{aligned}
\langle\langle T \rangle\rangle &: \text{Type} \rightarrow \text{Type} \\
\langle\langle B \rangle\rangle &:= B \\
\langle\langle \{t\}_T \rangle\rangle &:= \exists z: \text{Trail}. \{\langle\langle t \rangle\rangle^z\}_{\langle\langle T \rangle\rangle} \quad \text{where } z \text{ is fresh} \\
\langle\langle \Pi x: S. T \rangle\rangle &:= \Pi z: \text{Trail}. \Pi x: \langle\langle S \rangle\rangle. \langle\langle T \rangle\rangle \quad \text{where } z \text{ is fresh} \\
\langle\langle \text{Cons } T_1 T_2 \rangle\rangle &:= \text{Cons } \langle\langle T_1 \rangle\rangle \langle\langle T_2 \rangle\rangle \\
\langle\langle t \text{ Match } T_2; x, y \Rightarrow T_3 \rangle\rangle &:= \langle\langle t \rangle\rangle^z \text{ Match } \langle\langle T_2 \rangle\rangle; x, y \Rightarrow \langle\langle T_3 \rangle\rangle \quad \text{where } z \text{ is fresh} \\
\langle\langle t \rangle\rangle^p &: \text{Term} \rightarrow \text{Term} \rightarrow \text{Term} \\
\langle\langle \text{choose}[B] \rangle\rangle^p &:= \text{unpack}_B p \\
\langle\langle \lambda x: T. t \rangle\rangle^p &:= \lambda z: \text{Trail}. \lambda x: \langle\langle T \rangle\rangle. \langle\langle t \rangle\rangle^z \quad \text{where } z \text{ is fresh} \\
\langle\langle t_1 t_2 \rangle\rangle^p &:= t'_1 p.3 t'_2 \quad \text{where } t'_1 = \langle\langle t_1 \rangle\rangle^{p.1} \text{ and } t'_2 = \langle\langle t_2 \rangle\rangle^{p.2} \\
\langle\langle x \rangle\rangle^p &:= x \\
\langle\langle \text{nil} \rangle\rangle^p &:= \text{nil} \\
\langle\langle \text{cons } t_1 t_2 \rangle\rangle^p &:= \text{cons } \langle\langle t_1 \rangle\rangle^{p.1} \langle\langle t_2 \rangle\rangle^{p.2} \\
\langle\langle t_1 \text{ match } t_2; x, y \Rightarrow t_3 \rangle\rangle^p &:= \langle\langle t_1 \rangle\rangle^{p.1} \text{ match } \langle\langle t_2 \rangle\rangle^{p.2}; x, y \Rightarrow \langle\langle t_3 \rangle\rangle^{p.3} \\
\langle\langle \text{fix}_n(x: T \Rightarrow t_1, t_2) \rangle\rangle^p &:= \text{fix}_n(x: \langle\langle T \rangle\rangle \Rightarrow \langle\langle t_1 \rangle\rangle^{p.1}, \langle\langle t_2 \rangle\rangle^{p.2})
\end{aligned}$$

Fig. 4. The rules for lowering programs in  $\lambda_{<:, \{ \}}^{\text{nd}}$  to  $\lambda_{<:, \{ \}}^{\text{det}}$ , yielding a deterministic program without the non-deterministic  $\text{choose}[B]$  construct.

Given a program  $t$  in  $\lambda_{<:, \{ \}}^{\text{nd}}$ , the lowering yields  $t_f = \lambda z_\alpha: \text{Trail}. \langle\langle t \rangle\rangle^{z_\alpha}$  in  $\lambda_{<:, \{ \}}^{\text{det}}$ , which encodes the behavior of  $t$  as a pure function. That is, for any given (potentially non-deterministic) reduction resulting in  $v$ , there exists a trail  $\tau$  such that  $(t_f \tau) \rightarrow_\beta^* \langle\langle v \rangle\rangle^2$ .

In Figure 4 we describe the transformation of terms,  $\langle\langle t \rangle\rangle^p$ , and the transformation of types,  $\langle\langle T \rangle\rangle$ . At its core,  $\langle\langle t \rangle\rangle^p$  replaces each invocation of  $\text{choose}[B]$  by an application of a function  $\text{unpack}_B$  to a trail. Given the original program, one of its non-deterministic executions can be characterized by a mapping from every invocation of  $\text{choose}[B]$  to the resulting value in  $B$ . With respect to our evaluation relation  $\rightarrow_\beta$ , such a mapping can be obtained by recording the sequence of non-deterministic choices in  $\text{BCHOOSETOP}$  and  $\text{BCHOOSELIST}$ . The initial trail  $z_\alpha$  used to evaluate the lowered program corresponds to a complete mapping for some non-deterministic execution. Throughout the lowered program we build up *selections* on the initial trail using  $t.n$ , which correspond to subtrails. Calls to  $\text{unpack}_B$  then use the given subtrail to return a value. In our translation we take care never to apply  $\text{unpack}_B$  to the same trail twice: Doing so would incorrectly constrain the outcome of the corresponding invocations to be coupled together.

In the translation of abstractions, we create a fresh trail parameter  $z$ , which is then used to translate the function's body. This is essential, as it ensures that in each function invocation we allow for different non-deterministic choices. Note that it does not seem feasible to enumerate all the possible invocations of  $\text{choose}[B]$  statically: For one, the outcome of a  $\text{choose}[B]$  might

<sup>2</sup>We omit the trail argument in  $\langle\langle v \rangle\rangle$ , as it is irrelevant when translating values, which can only contain  $\text{choose}[B]$  underneath lambdas.

influence the control flow of the original program, and, in general, the length of the execution may be unbounded. To translate an application, we select on the current trail and pass it as the additional argument. Extending the selection is crucial to ensure that recursive calls can be distinguished in their non-deterministic choices. Consequently, we also adapt types that occur in the annotations of abstractions and fixpoints using  $\langle\langle T \rangle\rangle$ . In particular,  $\Pi$ -types are rewritten to account for the newly-introduced Trail parameter.

Note that in  $\langle\langle T \rangle\rangle$  we do not propagate and extend an existing trail as we do with  $p$  in  $\langle\langle t \rangle\rangle^p$ . When translating a singleton type  $\{t\}_U$  we instead wrap the resulting type in a fresh existential type  $\exists z : \text{Trail.}$ , which is used in the translation of  $t$ . This is what gives  $\text{choose}[B]$  its dual meaning at the type level: Rather than referring to one particular choice, it encompasses all of them.

Our lowering is related to monadic encodings in the style of Wadler [Wadler 1990]. Our encoding is simpler than a typical State monad because we only care about the distinctness of trails, rather than encoding the evaluation order and threading the resulting state from one subterm to another.

**3.2.2 Trails, More Carefully.** We will now give a more concrete definition of what properties a trail, and the operations that act upon it, must satisfy. We organize the sequence of values of a trail  $\tau$  as a ternary tree. Leaves of this tree contain a value, and a tag that encodes the type of the value. Consider  $t..p$  as notation for  $(\dots(t.n_1)\dots).n_k$ , i.e., applying a series of selections  $p = .n_1 \dots .n_k$  where  $n_1, \dots, n_k \in \{1, 2, 3\}$  to  $t$ . Given a trail  $\tau$  and selections  $p$ ,  $\tau..p$  represents the subtree of  $\tau$  when selecting the  $n_i$ -th child at the  $i$ -th level of  $\tau$ . For trees  $\tau, \tau'$  and a selection  $p$ ,  $(\text{update } \tau p \tau')$  replaces the subtree selected by  $p$  in  $\tau$  by  $\tau'$ , so that  $(\text{update } \tau p \tau')..p = \tau'$ . The  $\text{unpack}_B : (\Pi x : \text{Trail. } B)$  function returns the value at the root of the given tree, if the type-tag of the value there encodes  $B$ , and nil otherwise.

### 3.3 The Type System

We introduce our type system for  $\lambda_{<.,\downarrow}^{\text{det}}$  which consists of several inter-dependent relations:

- type inference and checking ( $\Uparrow$  and  $\Downarrow$  in Figure 5),
- subtyping ( $<$ : in Figure 6), and
- type normalization ( $\rightarrow_N$  in Figure 7).

To improve legibility of the rules, we omit well-formedness conditions, and presume that types are well-formed in the given context. For singleton types, in particular, we maintain the assumption that for any  $\{t\}_U$  we encounter,  $t$  inhabits  $U$ . Similarly, for every list match type  $(t \text{ Match } T_2; x, y \Rightarrow T_3)$  we assume that  $t$  inhabits List.

*Type inference and underlying types.* Figure 5 presents rules that infer the most precise type for a given term  $t$ . In particular, type inference will yield a singleton type  $\{t\}_U$ , if  $t$  is well-typed. For each construct we attach an upper bound as the singleton's underlying type  $U$ . In TABS, for instance, we “tag” the singleton type inferred for a function with the corresponding  $\Pi$ -type, and in TCONS we attach a special type  $\text{Cons } T_1 T_2$  only present during type checking. The underlying type is used to guide checks in TAPP and various subtyping rules.

In TAPP, in particular, we expose and match against the underlying function type of  $t_1$  using the auxiliary  $[\cdot]$  function. Our goal here is to check that applying  $t_1$  to  $t_2$  is safe, as usual, while also maintaining a precise version of the underlying type. Assuming we inferred  $V = \{t_1\}_{\Pi x:S. T}$  for  $t_1$ ,  $[V]$  will yield a  $\Pi$ -type equivalent to  $t_1$  for all  $x$  in  $S$ , i.e.,  $\Pi x : S. \{t_1 x\}_T$ . We then substitute the argument term in the result type, yielding  $\{t_1 t_2\}_{T[x \mapsto t_2]}$ . In TAPP, TCONS and TFIX we also refer to a type-checking relation  $(t \Downarrow T)$  which is defined as a shorthand (see TCHECK) for inferring the type of  $t$  and checking against the expected type  $T$  using the subtyping relation.

$\frac{\Gamma(x) = T}{\Gamma \vdash x \uparrow \{x\}_T} \quad (\text{TVar})$	$\frac{\Gamma, x : S \vdash t \uparrow T}{\Gamma \vdash \lambda x : S. t \uparrow \{\lambda x : S. t\}_{\Pi x S. T}} \quad (\text{TAbs})$
$\frac{\Gamma \vdash t_1 \uparrow V \quad [V] = \Pi x : S. T \quad \Gamma \vdash t_2 \Downarrow S}{\Gamma \vdash t_1 t_2 \uparrow T[x \mapsto t_2]} \quad (\text{TApp})$	
$\frac{\Gamma, x : T \vdash t_1 \Downarrow T \quad \Gamma \vdash t_2 \Downarrow T}{\Gamma \vdash \text{fix}_n(x : T \Rightarrow t_1, t_2) \uparrow \{\text{fix}_n(x : T \Rightarrow t_1, t_2)\}_T} \quad (\text{TFix})$	
$\frac{}{\Gamma \vdash \text{nil} \uparrow \{\text{nil}\}_{\text{List}}} \quad (\text{TNIL})$	
$\frac{\Gamma \vdash t_1 \uparrow T_1 \quad \Gamma \vdash t_2 \uparrow T_2 \quad \Gamma \vdash T_2 <: \text{List}}{\Gamma \vdash \text{cons } t_1 t_2 \uparrow \{\text{cons } t_1 t_2\}_{\text{Cons } T_1 T_2}} \quad (\text{TCons})$	
$\frac{\Gamma \vdash t_1 \Downarrow \text{List} \quad \Gamma \vdash t_2 \uparrow T_2 \quad \Gamma, x : \text{Top}, y : \text{List} \vdash t_3 \uparrow T_3}{\Gamma \vdash t_1 \text{ match } t_2; x, y \rightarrow t_3 \uparrow \{t_1 \text{ match } t_2; x, y \rightarrow t_3\}_{t_1 \text{ Match } T_2; x, y \Rightarrow T_3}} \quad (\text{TMATCH})$	
$\frac{\Gamma \vdash t \Downarrow \text{Trail}}{\Gamma \vdash t.k \uparrow \{t.k\}_{\text{Trail}}} \quad (\text{TDot})$	$\frac{\Gamma \vdash t \uparrow T' \quad \Gamma \vdash T' <: T}{\Gamma \vdash t \Downarrow T} \quad (\text{TCHECK})$
$[\{t\}_U] := \begin{cases} \Pi x : S. \{t x\}_T & \text{if } U = \Pi x : S. T \text{ for some types } S, T \\ [U] & \text{otherwise} \end{cases}$	
$[T] := T \quad \text{if there exist no } t \text{ and } U \text{ such that } T = \{t\}_U$	

Fig. 5. The inference and checking rules.

*Subtyping and type normalization.* The subtyping relation is given in Figure 6. Rules for reflexivity (SUBREFL),  $\Pi$ -types (SUBPI), Top and the List base type (SUBTOP, SUBCONS1) are standard. The Cons type introduced during inference can be subtyped covariantly (SUBCONS2); the type  $(x \text{ Match } T_2; x, y \Rightarrow T_3)$  assigned to matches behaves like a union of  $T_2$  and  $T_3$ , while allowing  $T_3$  to retain variables bound in the pattern (SUBMATCH). Using SUBSING we can approximate a singleton type  $\{t\}_{T_1}$  occurring on the left-hand side by its upper bound  $T_1$  and continue subtyping from there (SUBSING).

Our system allows for computation on types to take place during subtyping. Subtyping rule SUBNORM bundles two kinds of normalizing behavior: We first reduce both sides  $T_1$  and  $T_2$  using type normalization. We then attempt to replace any newly-exposed occurrences of  $\text{unpack}_B$  by fresh existentials of type  $B$  via the untangle function  $\mathcal{U}(\cdot)$ .

The rules for type normalization are detailed in Figure 7. We merely distribute over  $\Pi$ -types, existentials, and Cons-types (NPI, NEXISTS1, NEXISTS2, NCONS). Since  $S$  is assumed to be inhabited in existential types  $\exists x : S. T$ , we eliminate such quantifications whenever the result type  $T$  does not contain  $x$  free (NEXISTS1).

Singleton types  $\{t\}_U$  may be normalized using NSING, in which we first reduce  $t$  using beta-delta reduction (Figure 8). Beta-delta reduction is defined as a context-aware extension of beta reduction

$\frac{}{\Gamma \vdash T <: T} \quad (\text{SUBREFL})$	$\frac{}{\Gamma \vdash T <: \text{Top}} \quad (\text{SUBTOP})$
$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{t\}_{T_1} <: T_2} \quad (\text{SUBSING})$	$\frac{\Gamma, x : S \vdash T <: U}{\Gamma \vdash \exists x : S. T <: U} \quad (\text{SUBEXISTSLEFT})$
$\frac{\{t\}_U = \text{solve}_x(T_1, S, T_2) \quad \Gamma \vdash \{t\}_U <: S \quad \Gamma \vdash T_1 <: T_2[x \mapsto t]}{\Gamma \vdash T_1 <: \exists x : S. T_2} \quad (\text{SUBEXISTSRIGHT})$	
$\frac{}{\Gamma \vdash \text{Cons } S \ T <: \text{List}} \quad (\text{SUBCONS1})$	$\frac{\Gamma \vdash S_1 <: S_2 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \text{Cons } S_1 \ T_1 <: \text{Cons } S_2 \ T_2} \quad (\text{SUBCONS2})$
$\frac{\Gamma \vdash S_2 <: T \quad \Gamma, x : \text{Top}, y : \text{List} \vdash S_3 <: T}{\Gamma \vdash t_1 \text{ Match } S_2; x, y \Rightarrow S_3 <: T} \quad (\text{SUBMATCH})$	
$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \Pi x : S_1. T_1 <: \Pi x : S_2. T_2} \quad (\text{SUBPI})$	
$\frac{\Gamma \vdash T_1 \rightarrow_N T'_1 \quad \Gamma \vdash T_2 \rightarrow_N T'_2 \quad \Gamma \vdash \mathcal{U}(T'_1) <: \mathcal{U}(T'_2)}{\Gamma \vdash T_1 <: T_2} \quad (\text{SUBNORM})$	

Fig. 6. The subtyping rules.

(seen previously in Figure 2) with a new rule BDDDELTA, which allows the elimination of variables whose precise definition is known from the context (a similar evaluation relation is found in [Courant 2003]). Delta reduction steps may lead the underlying type  $U$  to go out-of-sync with the newly computed term  $t'$ . For instance, given the context  $\Gamma = x : \{\text{nil}\}_{\text{List}}$  and the type  $\{x\}_{\text{Top}}$ , if we were to normalize using the beta-delta reduction  $x \rightarrow_{\beta\delta}^* \text{nil}$  alone, we would arrive at  $\{\text{nil}\}_{\text{Top}}$ . We can improve upon this — and in fact might rely on it in later subtyping queries — by redoing type inference on  $t'$ , yielding a singleton type with a better bound (in our example,  $\{\text{nil}\}_{\text{List}}$ ).

The rules for match allow reduction of  $(t \text{ Match } T_2; x, y \Rightarrow T_3)$  depending on the beta-delta reduction of  $t$ . That is, we normalize to  $T_2$  when  $t = \text{nil}$  (NMATCH1), and to  $T_3$  when  $t$  is a cons (NMATCH2). In the latter case, we add precisely-typed bindings that allow for  $x$  and  $y$  to be  $\delta$ -reduced during the normalization of  $T_3$ . If  $t$  does not fit either case, we instead normalize to a type that incorporates the reduced  $t$ .

*Subtyping existential types.* Existentials only enter the program when lowering type annotations in  $\lambda_{<:\{\}}^{\text{nd}}$  to  $\lambda_{<:\{\}}^{\text{det}}$ , and in SUBNORM via  $\mathcal{U}(\cdot)$ . When encountered on the left-hand side, existential types are eliminated by adding  $x : S$  to the context (SUBEXISTSLEFT). When an existential occurs on the right-hand side, we try to guess a valid instantiation  $t$  for  $x$  (SUBEXISTSRIGHT). The subroutine that guesses  $t$  is modelled abstractly by  $\text{solve}_x(T_1, S, T_2)$ , which is expected to return a singleton  $\{t\}_U$ . We make no assumptions on the implementation of  $\text{solve}_x$ , but verify that the outcome is a valid solution by checking that it conforms to  $S$  and makes the instantiated right-hand side a super-type of the left-hand side. In Subsection 3.5 we discuss one possible concrete implementation of  $\text{solve}_x$ .

$\frac{T \in \{\text{Top}, \text{List}\}}{\Gamma \vdash T \rightarrow_N T} \text{ (NBASE)}$	$\frac{\Gamma \vdash t \rightarrow_{\beta\delta}^* t' \quad \Gamma \vdash t' \uparrow \{t''\}_V}{\Gamma \vdash \{t\}_U \rightarrow_N \{t''\}_V} \text{ (NSING)}$
$\frac{\Gamma \vdash S \rightarrow_N S' \quad \Gamma, x : S' \vdash T \rightarrow_N T'}{\Gamma \vdash \Pi x : S. T \rightarrow_N \Pi x : S'. T'} \text{ (NPI)}$	
$\frac{\Gamma \vdash S \rightarrow_N S' \quad \Gamma, x : S' \vdash T \rightarrow_N T' \quad x \notin \text{fv}(T)}{\Gamma \vdash \exists x : S. T \rightarrow_N T'} \text{ (NEXISTS1)}$	
$\frac{\Gamma \vdash S \rightarrow_N S' \quad \Gamma, x : S' \vdash T \rightarrow_N T' \quad x \in \text{fv}(T)}{\Gamma \vdash \exists x : S. T \rightarrow_N \exists x : S'. T'} \text{ (NEXISTS2)}$	
$\frac{\Gamma \vdash T_1 \rightarrow_N T'_1 \quad \Gamma \vdash T_2 \rightarrow_N T'_2}{\Gamma \vdash \text{Cons } T_1 T_2 \rightarrow_N \text{Cons } T'_1 T'_2} \text{ (NCONS)}$	
$\frac{\Gamma \vdash t \rightarrow_{\beta\delta}^* \text{nil} \quad \Gamma \vdash T_2 \rightarrow_N T'_2}{\Gamma \vdash t \text{ Match } T_2; x, y \Rightarrow T_3 \rightarrow_N T'_2} \text{ (NMATCH1)}$	
$\frac{\Gamma \vdash t \rightarrow_{\beta\delta}^* \text{cons } t_1 t_2 \quad \Gamma, x : \{t_1\}_{\text{Top}}, y : \{t_2\}_{\text{List}} \vdash T_3 \rightarrow_N T'_3}{\Gamma \vdash t \text{ Match } T_2; x, y \Rightarrow T_3 \rightarrow_N T'_3} \text{ (NMATCH2)}$	
$\frac{\Gamma \vdash t \rightarrow_{\beta\delta}^* t' \quad \text{if neither of the above rules apply}}{\Gamma \vdash t \text{ Match } T_2; x, y \Rightarrow T_3 \rightarrow_N t' \text{ Match } T_2; x, y \Rightarrow T_3} \text{ (NMATCH3)}$	

Fig. 7. The type normalization rules.

$\frac{\Gamma \vdash t \rightarrow_{\beta\delta} t'}{\Gamma \vdash \mathcal{E}[t] \rightarrow_{\beta\delta} \mathcal{E}[t']} \text{ (BDCCTX)}$	$\frac{\Gamma(x) = \{t\}_U}{\Gamma \vdash x \rightarrow_{\beta\delta} t} \text{ (BDDDELTA)}$	$\frac{t \rightarrow_{\beta} t'}{\Gamma \vdash t \rightarrow_{\beta\delta} t'} \text{ (BDBETA)}$
--	--	--

Fig. 8. The rules of beta-delta reduction.

### 3.4 Untangling Trails

In Subsection 3.2 we explained how to translate the non-deterministic choose[B] construct into an application of  $\text{unpack}_B$  to a trail. Therefore, during type checking, we often face subtyping queries involving applications of  $\text{unpack}_B$  on the right-hand side. For instance, when checking the program

$$(\lambda x : \{\text{cons choose[Top] choose[List]}\}. x) (\text{cons nil nil})$$

we will encounter the following subtyping query:

$$\{\text{cons nil nil}\} <: \exists z : \text{Trail}. \{\text{cons } (\text{unpack}_{\text{Top}} z.1) (\text{unpack}_{\text{List}} z.2)\}$$

Though the right-hand side is an existential type, this query cannot be solved by  $\text{SUBEXISTSRIGHT}$  directly, unless the solve subroutine possesses some deep knowledge about  $\text{unpack}_B$ . That is, a

$$\begin{array}{l}
\mathcal{U} : \text{Type} \rightarrow \text{Type} \qquad \qquad \qquad \text{“Untangle all Trail existentials”} \\
\mathcal{U}(\exists x : \text{Trail}. T) := \mathcal{W}_x(\text{ps}, \mathcal{U}(T)) \quad \text{where ps} = \text{trailsOf}_x(\mathcal{U}(T)) \\
\mathcal{U}(\exists x : S. T) := \exists x : \mathcal{U}(S). \mathcal{U}(T) \quad \text{if } S \neq \text{Trail} \\
\mathcal{U}(\Pi x : S. T) := \Pi x : \mathcal{U}(S). \mathcal{U}(T) \\
\mathcal{U}(\{t\}_U) := \{t\}_{\mathcal{U}(U)} \qquad \qquad \mathcal{U}(\text{Cons } T_1 \ T_2) := \text{Cons } \mathcal{U}(T_1) \ \mathcal{U}(T_2) \\
\mathcal{U}(B) := B \qquad \qquad \mathcal{U}(t \text{ Match } T_2; x, y \Rightarrow T_3) := t \text{ Match } \mathcal{U}(T_2); x, y \Rightarrow \mathcal{U}(T_3) \\
\\
\mathcal{W}_x : \text{Id} \rightarrow 2^{\text{Term}} \rightarrow \text{Type} \rightarrow \text{Type} \quad \text{“Untangle one Trail existential”} \\
\mathcal{W}_x(\{ \}, T) := T \\
\mathcal{W}_x(\{p\} \uplus \text{ps}, T) := \begin{cases} \mathcal{W}_x(\text{ps}, \exists y : B. T') & \text{if } p \notin \text{trailsOf}_x(T') \\ \mathcal{W}_x(\text{ps}, \exists y : \text{Trail}. T'') & \text{otherwise} \end{cases} \\
\text{where } T' \text{ is } T \text{ with all occurrences of } (\text{unpack}_B x..p) \text{ replaced by } y, \\
T'' \text{ is } T \text{ with all occurrences of } x..p \text{ replaced by } y, \text{ and } y \text{ is fresh.} \\
\\
\text{trailsOf}_x(T) \subseteq \text{Term} \qquad \qquad \qquad \text{“Collect all trails rooted in } x \text{”} \\
\text{trailsOf}_x(T) \text{ is a set of maximal selections } p \text{ where } x..p \text{ appears in } T
\end{array}$$

Fig. 9. The untangle function  $\mathcal{U}$  and additional auxiliary functions.

priori it is not evident that there exists a trail  $z$  such that both  $(\text{unpack}_{\text{Top}} z.1)$  and  $(\text{unpack}_{\text{List}} z.2)$  reduce to nil.

Using the properties on trails and  $\text{unpack}_B$  introduced in Subsection 3.2.1 we can prove that the type on the right-hand side is, in fact, equivalent to an explicitly quantified version, i.e.,

$$\exists z : \text{Trail}. \{ \text{cons } (\text{unpack}_{\text{Top}} z.1) \ (\text{unpack}_{\text{List}} z.2) \} = \exists x_1 : \text{Top}. \exists x_2 : \text{List}. \{ \text{cons } x_1 \ x_2 \}$$

To see why the inclusion from left to right holds, consider any trail  $\tau$  with values  $v_1^{\text{Top}}$  and  $v_2^{\text{List}}$  stored at indices .1 and .2. We can thus instantiate  $x_1$  and  $x_2$  to  $v_1^{\text{Top}}$  and  $v_2^{\text{List}}$ , to obtain the same term on both sides. From right to left we can construct a tree containing the values of  $x_1$  and  $x_2$  at indices .1 and .2. This same reasoning can be applied to all functions of Trail that we encounter after our lowering to  $\lambda_{<:\{ \}}^{\text{det}}$ . Furthermore, it generalizes to an arbitrary number of selections on  $z$ , as long as the selections are not prefixes of one another, which is ensured by our lowering step.

To exploit this property, we define the untangle function  $\mathcal{U}(\cdot)$ , which transforms the left-hand side of the equality above to the right-hand side. We use  $\mathcal{U}(\cdot)$  during normalization in SUBNORM. In our example, this leads to a simpler subtyping query:

$$\{ \text{cons nil nil} \} <: \exists x_1 : \text{Top}. \exists x_2 : \text{List}. \{ \text{cons } x_1 \ x_2 \}$$

At this point we can apply SUBEXISTSRIGHT twice, which could find valid assignments for both  $x_1$  and  $x_2$  (i.e., nil) using straightforward unification.

The definition of  $\mathcal{U}(T)$  is given in Figure 9. Given the conditions on trails mentioned above, we prove untangling always yields equivalent types (see Section 4).

### 3.5 From Rules to Algorithms

The type system we presented above comes close to being algorithmic. All of the rules for type inference and most of the rules for subtyping and type normalization are already syntax-directed. To derive the normalization of an existential type one has to choose between `NEXISTS1` and `NEXISTS2`, but only one of them will ever succeed due to the condition on  $x$  being free in  $T'$ . It is therefore straightforward to formulate these cases as a single, effective rule. In the remainder, we note two more substantial adjustments that are needed for an effective formulation of our type system.

The first adjustment is to only apply `SUBNORM` (type normalization) at the very beginning of subtyping queries (for example, in `TCHECK`), and before any subderivation that adds a binder to the context (for example, in the second premise of `SUBPI`). Applying `SUBNORM` must remain optional, however, since forcing normalization can lead to subderivations that grow *ad infinitum*, for instance by normalizing under matches and re-entering type inference in `NSING`. Beyond making `SUBNORM` optional, in practice it is useful to allow for a fast path in subtyping. Given a subtyping query  $T_1 <: T_2$ , one can first try to prove a stronger subtyping relation, where the left-hand side  $T_1$  is approximated by  $[T_1]$ . We found that this greatly reduces the need for complex subtyping derivations, e.g., when checking against the List base type in `TCONS` and `TMATCH`.

The second adjustment lies in `SUBEXISTSRIGHT`, where we require a procedure for `solvex`. In principle, `solvex(T1, S, T2)` could do arbitrarily deep reasoning about the involved types, but our experience shows that a unification-like procedure is sufficient for the use cases presented in this paper. We experimented with a particularly simple variant: `solvex` performs a separate subtyping query which constrains  $x$  in a greedy manner using a modified version of `SUBREFL`. In this modified rule, the computation of (syntactic) type equality looks for any appearances of  $x$ . If  $x$  appears on either side of the comparison, the corresponding term on the other side is picked as a greedy solution of `solvex`. This naive syntactic approach can result in an instantiation that is not well-formed in the original context  $\Gamma$ , in which case we simply fail. One could, of course, try to incrementally improve this approach by trying to rewrite  $t$  to an equivalent term well-formed in  $\Gamma$ . It would be interesting to explore a more general constraint-solving approach.

## 4 SOUNDNESS

In this section we discuss the formal soundness proof for  $\lambda_{<:, \{ \}}^{\text{det}}$ . As a starting point, we use System FR [Hamza et al. 2019], a calculus that was recently presented as a foundation for the Stainless program verifier [LARA 2019]. Our formalization in Coq is available as additional material in the submission. We give an embedding  $\langle \cdot \rangle$  of  $\lambda_{<:, \{ \}}^{\text{det}}$  terms and types into System FR in Figure 10.

*Embedding terms.* Functions and applications are represented trivially using System FR's lambda abstractions and applications, which behave identically to ours. Lists are encoded in the typical way as a sum of unit for nil and a pair of a head and a tail for cons.

Our embedding of `fix` ensures trivial termination, following the bounded recursion behavior of  $\lambda_{<:, \{ \}}^{\text{nd}}$ . While we believe that the addition of general recursion to  $\lambda_{<:, \{ \}}^{\text{nd}}$  would not present a problem (as discussed before in Subsection 3.1), System FR is normalizing, and thus prevents us from lifting this restriction in our current formalization.

*Embedding types.*  $\Pi$ -types along with existential types are represented trivially. The type of lists is expressed in the usual way through a recursive type. A singleton type  $\{t\}_T$  is encoded using the type  $\{ \{ v : T \mid v \equiv t \} \}$  of System FR, which represents all values in  $T$  that are observationally equivalent to  $t$ . Observational equivalence is supported as a type in the current open-source formalization of System FR, even though this type was not supported in the paper [Hamza et al. 2019]. The  $(\text{Cons } T_1 \ T_2)$  type of  $\lambda_{<:, \{ \}}^{\text{det}}$  is translated by existentially quantifying over any combination of values

**Translation of terms to System FR:**

$$\begin{aligned}
\langle x \rangle &:= x \\
\langle \lambda x:T. t \rangle &:= \lambda x. \langle t \rangle \\
\langle t_1 t_2 \rangle &:= \langle t_1 \rangle \langle t_2 \rangle \\
\langle \text{nil} \rangle &:= \text{left } (\text{Unit} + (\text{Top}, \langle \text{List} \rangle)) (()) \\
\langle \text{cons } t_1 t_2 \rangle &:= \text{right } (\text{Unit} + (\text{Top}, \langle \text{List} \rangle)) (\langle t_1 \rangle, \langle t_2 \rangle) \\
\langle t_1 \text{ match } t_2; x, y \Rightarrow t_3 \rangle &:= \text{either\_match} (\langle t_1 \rangle, z \Rightarrow \langle t_2 \rangle, z \Rightarrow \langle t_3 \rangle [x \mapsto \pi_1 z] [y \mapsto \pi_2 z]) \\
\langle \text{fix}_n(x:X \Rightarrow t_1, t_2) \rangle &:= \text{fix}(x \Rightarrow \lambda y:\text{Nat. match}(y, \langle t_2 \rangle, y' \Rightarrow \langle t_1 \rangle [x \mapsto x y']) ) n
\end{aligned}$$

**Translation of types to System FR:**

$$\begin{aligned}
\langle \text{Top} \rangle &:= \text{Top} \\
\langle \text{List} \rangle &:= \forall n. \text{Rec}(n)(X \Rightarrow \text{Unit} + (\text{Top}, X)) \\
\langle \{t\}_T \rangle &:= \{ \{ v : \langle T \rangle \mid v \equiv \langle t \rangle \} \} \\
\langle \Pi x:S. T \rangle &:= \Pi x:\langle S \rangle. \langle T \rangle \\
\langle \text{Cons } T_1 T_2 \rangle &:= \exists x_1:\langle T_1 \rangle. \exists x_2:\langle T_2 \rangle. \{ \langle \text{cons } x_1 x_2 \rangle \}_{\text{List}} \\
\langle t \text{ Match } T_2; x, y \Rightarrow T_3 \rangle &:= \{ \{ v : \langle T_2 \rangle \mid t \equiv \text{left } () \} \} \cup \\
&\quad \exists y_1:\text{Top}. \exists y_2:\langle \text{List} \rangle. \\
&\quad \{ \{ v : \langle T_3 \rangle [x_1 \mapsto y_1] [x_2 \mapsto y_2] \mid t \equiv \text{right } (y_1, y_2) \} \} \\
\langle \exists x:S. T \rangle &:= \exists x:\langle S \rangle. \langle T \rangle
\end{aligned}$$

Fig. 10. The embedding of  $\lambda_{<:\{\}}^{\text{det}}$  terms and types into System FR.

in  $T_1$  and  $T_2$ . For the type of matches,  $(t \text{ Match } T_2; x, y \Rightarrow T_3)$ , we take the union of each of  $T_2$ 's and  $T_3$ 's interpretation, conditional on whether the scrutinee  $t$  reduces to nil or a cons.

Given that System FR assigns a reducibility semantics to its types, our embedding also affords us with denotations for all the types of  $\lambda_{<:\{\}}^{\text{det}}$ . That is, given the set of reducible values  $[[T]]_v$  of type  $T$  in System FR, the meaning of a type  $T'$  in  $\lambda_{<:\{\}}^{\text{det}}$  is given by  $[[\langle T' \rangle]]_v$ . For instance,  $[[\langle \text{List} \rangle]]_v = \{ \text{cons } v_1 (\dots (\text{cons } v_n \text{ nil}) \dots) \mid n \geq 0, \forall i. v_i \in [[\langle \text{Top} \rangle]]_v \}$ . Existential types  $[[\langle \exists x:S. T \rangle]]_v$  are the union of all  $[[\langle T[x \mapsto s] \rangle]]_v$  for all  $s \in [[\langle S \rangle]]_v$ .

*Formalized soundness statement.* Using the above embedding, we have proved that all of the rules for type inference, subtyping and type normalization presented in Subsection 3.3 are admissible with respect to the reducibility semantics of types. We built our mechanization on top of System FR's existing Coq formalization. The respective lemmas are proven under the additional assumptions given to us via the well-formedness rules mentioned in Subsection 3.3. Namely, the following are assumed to hold:

- In rules for type inference, subtyping, and type normalization, we require well-formedness in the current context and inhabitedness for singleton and list match types.
- During delta-beta reduction, we require terms to be normalizing in the current context.
- Trails and their operations are kept abstract and specified using axioms in file `Trail.v`.

The entirety of our definitions and proofs consists of ~7k lines of Coq in addition to the previous development of System FR soundness, which consisted of ~20k lines.



We can thus state soundness for  $\lambda_{<:\{\}}^{\text{det}}$  programs in terms of the reducibility judgment  $\Gamma \vDash t : T$  of System FR. The latter holds when, for all substitutions  $\gamma$ , such that for all  $(x, S) \in \Gamma$  we have  $\gamma(x) \in \llbracket \gamma(S) \rrbracket_v$ ,  $\gamma(t) \in \llbracket \gamma(T) \rrbracket_v$ . Let  $\langle \Gamma \rangle$  be the context with all  $\lambda_{<:\{\}}^{\text{det}}$  types embedded into System FR types.

**THEOREM (SOUNDNESS).** *Given a context  $\Gamma$  and a  $\lambda_{<:\{\}}^{\text{det}}$  term  $t$ , if type inference yields a type  $T$ , then  $t$  is reducible at that type. That is, if  $\Gamma \vdash t \uparrow T$  holds, then  $\langle \Gamma \rangle \vDash \langle t \rangle : \langle T \rangle$ .*

Note that the traditional notion of type safety for  $t$  follows, i.e., well-typedness of  $t$  implies the existence of value  $v$  such that  $t \rightarrow_{\beta}^* v$ , since  $\langle t \rangle$  is normalizing exactly when  $t$  is. Similarly, using the correspondence of  $\lambda_{<:\{\}}^{\text{det}}$  programs to non-deterministic  $\lambda_{<:\{\}}^{\text{nd}}$  programs after lowering (see Subsection 3.2), we get type safety for  $\lambda_{<:\{\}}^{\text{nd}}$ .

## 5 IMPLEMENTATION

In this section we give an overview of how we extended Scala with dependent types. This development was an experiment to explore the feasibility of adding dependent types in Scala. We implemented our prototype as an extension of Dotty, the reference compiler for future versions of the Scala language. Our presentation focuses on several facets of the implementation that are not reflected in the formalism presented in Section 3.

On a syntactic level, our Scala extension consists of three additions:

- the singleton types syntax  $\{ t \}$ ,
- the **dependent** modifier for methods, values and classes,
- the `choose[T]` construct.

The newly-introduced singleton type syntax enables a subset of Scala expressions to be used in types. This subset approximately corresponds to the core functional subset of Scala, plus the `choose[T]` construct, as illustrated in  $\lambda_{<:\{\}}^{\text{nd}}$ . Within this subset, the main differences between our formalism and implementation lie in the handling of pattern matching.

### 5.1 Pattern Matching

Pattern matching in Scala supports a wide range of matching techniques [Emir et al. 2007]. For example, *extractor patterns* rely on user-defined methods to extract values from objects. As a result, these custom extractors can contain arbitrary side effects. Our implementation limits the kind of patterns available in types to the two simplest forms: decomposition of case classes and the type-tests/type-casts patterns.

During type normalization, our system evaluates pattern matching expressions according to Scala's runtime semantics, that is, patterns are checked top-to-bottom, and type-tests are evaluated using runtime type information available after type erasure.

For example, consider the following pattern matching expression:

```
s match { case _: T1 => v1 case _: T2 => v2 }
```

When used in a type, this expression reduces to  $v_1$  if the scrutinee's type is a subtype of  $T_1$ . In order to reduce to  $v_2$ , type normalization must make sure  $T_1$  and the scrutinee's type are disjoint, namely that the dynamic type of  $s$  cannot possibly be smaller than  $T_1$ . Disjointness proofs are built using static knowledge about the class hierarchy and make use of the guarantees implied by the **sealed** and **final** qualifiers, which are Scala's way of declaring closed-type hierarchies.

## 5.2 Two Modes of Type Inference

In order to retain backwards-compatibility, our system supports two modes of type inference: the precise inference mode which infers singleton types, and the default inference mode that corresponds to Scala's current type-inference algorithm. Concretely, users opt into our new inference mode using the **dependent** qualifier on methods, values, and classes.

When inferring the result type of a **dependent** method, our system lifts the method's body into a type. This lifting will be precise for the subset of expressions that is representable in types, and approximative for the rest. When we encounter an unsupported construct, we compute its type using the default mode, yielding a type  $T$  which we then integrate in the lifted body as `choose[T]`.

For example, given the following definition:

```
dependent def getName(personalized: Boolean) =
  if (personalized) readString() else "Joe"
```

our system infers the following result type:

```
{ if (personalized) choose[String] else "Joe" }
```

Scala requires recursive methods to have an explicit result type, and this restriction also applies to **dependent** methods. However, in the case of a **dependent** method, an explicit result type is only used as an upper bound for the actual precise result type and will only be used to type-check the method's body. At other call sites, the (precise) inferred result type is used. Bounds of dependent methods are written using a special syntax ( $<: T$ ), which emphasizes the difference from normal result types ( $: T$ ).

## 5.3 Approximating Side Effects

*State.* Scala's type system permits uncontrolled side effects in programs. Given the absence of an effect system, result types of methods do not convey any information about the potential use of side effects in the method body. The situation is analogous for **dependent** methods. Thanks to `choose[T]` we can still formulate precise result types when terms depend on the result of side-effectful operations. Since we uniformly approximate all side effects, we avoid the situation where a type refers to a value that may be modified during the program execution. For instance, if  $z$  is a mutable integer variable, we will never introduce  $z$  in a singleton type, but we can still assign a better type than `Lst` to an expression like `Cons(z, Nil())`, that is, `{ Cons(choose[Int], Nil()) }`.

*Exceptions.* Similarly to how we model other side effects, exceptions are approximated in types. Our type-inference algorithm uses a new error type, `Error(e)`, which we infer when raising an exception with `throw e`. Exception handlers are typed imprecisely using the default mode of type-inference. Exceptions thrown in statement positions are not reflected in singleton types, since the type of `{e1; e2}` is simply `{ e2 }`. However, exceptions thrown in tail positions (such as in `remove` from Section 2) can lead to types normalizing to `Error(e)`. In these cases, our type system can prove that the program execution will encounter exceptional behavior, and reports a compilation error. This approach is conservative in that it might reject programs that recover from exceptions. Also note that this is a sanity check, rather than a guarantee of no exceptions occurring at runtime. That is, depending on which rules are used during subtyping, it is possible to succeed without entering type normalization, resulting in such errors going undetected. Despite these shortcomings, our treatment of exceptions results in a practical way to raise compile-time errors. It would be interesting to explore the addition of an effect system to our Scala extension and formalization.

## 5.4 Virtual Dispatch

Our extension does not model virtual dispatch explicitly in singleton types. Instead, the result type of a method call  $t.m(\dots)$  is always the result type of  $m$  in  $t$ 's static type. Consequently, **dependent** methods effectively become **final**, given that only a provably-equivalent implementation could be used to override it.

Special care must be taken when an imprecisely-typed method is overridden with a **dependent** one. In this situation, the result type of a method invocation can lose precision depending on type of the receiver. Calls to the `equals` methods are a common example of this: `equals` is defined at the top of Scala's type hierarchy as referential equality and can be overridden arbitrarily. Given a class `Foo` with a **dependent** overrides of `equals`, calls to `Foo.equals(Any)` and `Any.equals(Foo)` are not equivalent; the former precisely reflects the equality defined in `Foo` whereas the latter merely returns a `Boolean`.

## 5.5 Termination

We distinguish two important aspects of termination.

The first question is whether type-checked programs are guaranteed to terminate. For simplicity, our work side-steps this question, requiring bounds for recursion. A more general solution would be to compute or infer such bounds using measure functions, as done in System FR [Hamza et al. 2019]. Another approach would be to extend our translation of non-determinism to permit non-termination. We consider this aspect orthogonal to the objectives of this paper. Our work targets general-purpose programming language whose type safety is defined with regards to its runtime semantics and that may include non-terminating interactive computations.

The second question is termination of our type checker. Non-termination of type checking implies that the type checker can give three possible answers, "type correct", "type incorrect" or "do not know" (or timeout). Treating "do not know" as "type incorrect" makes the non-termination unproblematic from a soundness perspective. A similar argument is made for other dependently-typed languages with unbounded recursion, such as Dependent Haskell [Eisenberg 2016] or Cayenne [Augustsson 1998]. In practice, our system deals with infinite loops using a fuel mechanism. Every evaluation step consumes a unit of fuel, and an error is reported when the compiler runs out of fuel. The default fuel limit can be increased via a compiler flag to enable arbitrarily long compilation times.

# 6 USE CASE

In this section, we extend the motivating example presented in Section 2 by building a type-safe interface for Spark datasets. We use dependent types to implement a simple domain-specific type checker for the SQL-like expressions used in Spark. We then compare the compilation time of our dependently-typed interface against an equivalent encoding based on implicits.

## 6.1 A Type-Safe Database Interface

The type-safe interface presented in this section illustrates the expressive power of our system and is implemented purely as a library. For brevity, our presentation only covers a small part of Spark's dataset interface, but the approach can be scaled to cover that interface in its entirety. The type safety of database queries is a canonical example and has been studied in many different settings [Chlipala 2010; Kazerounian et al. 2019; Leijen and Meijer 1999; Meijer et al. 2006].

The example built in Section 2 uses lists of column names to represent schemas. A straightforward improvement is to also track the type of columns as part of the schema. Instead of using column

names directly, we introduce the following `Column` class with a phantom type parameter `T` for the column type, and a field name for the column name:

```
dependent case class Column[T](name: String) { . . . }
```

Table schemas become lists of `Column`-s and thereby gain precision. The definition of `join` given in Section 2 can be adapted to this new schema encoding to prevent joining two tables that have columns with matching names but different types.

A large proportion of the weakly-typed Spark interface is dedicated to building expressions on table columns. Such expressions can currently be built from strings, in a subset of SQL, or using a Scala DSL which is essentially untyped.

The lack of type safety for column expressions can be particularly dangerous when mixing columns of different types. The pitfall is caused by Spark's inconsistency: depending on types of columns and operations involved, programs will either crash at runtime, or, more dangerously, data will be silently converted from one type to another.

By keeping track of column types it becomes possible to enforce the well-typedness of column expressions. As an example, consider the following Spark program:

```
table.filter(table.col("a") + table.col("b") === table.col("c"))
```

We would like our interface to enforce the following safety properties:

- Columns *a*, *b* and *c* are part of the schema of *table*.
- Addition is well-defined on columns *a* and *b*.
- The result of adding columns *a* and *b* can be compared with column *c*.
- The overall column expression yields a `Boolean`, which conforms to `filter`'s argument type.

Automatic conversions during equality checks can be prevented by restricting column equality to expressions of the same type `T`:

```
dependent case class Column[T](k: String):
  def ===(that: Column[T]): Column[Boolean] = Column(s"(${this.k} === ${that.k})")
```

Addition in Spark is defined between numeric types and characters. The result type of an addition depends on the operand types. For numeric types, Spark will pick the larger of the operand types according to the following ordering: `Double > Long > Int > Byte`. The situation is quite surprising with characters as any addition involving a `Char` will result in a `Double`.

Dependent types can be used to precisely model these conversions. We define a type function to compute the result type of additions:

```
def addRes(a: Any, b: Any) =
  (a, b) match
  case (_, Char, _, Char | Byte | Int | Long | Double) => choose[Double]
  case (_, Byte, _, Byte | Int | Long | Double)       => b
  case (_, Int, _, Int | Long | Double)               => b
  case (_, Long, _, Long | Double)                   => b
  case (_, Double, _, Double)                        => choose[Double]
  case (_, Byte | Int | Long | Double, _)            => addRes(b, a)
  case _ => throw new Error("incompatible types in addition")
type AddRes[A, B] = { addRes(choose[A], choose[B]) }
```

Also note the use of recursion in the second-to-last case, to avoid duplicating symmetric cases. The `AddRes` type can be used to define a `Column` addition that accurately models Spark's runtime:

```
dependent case class Column[T] private (k: String):
```

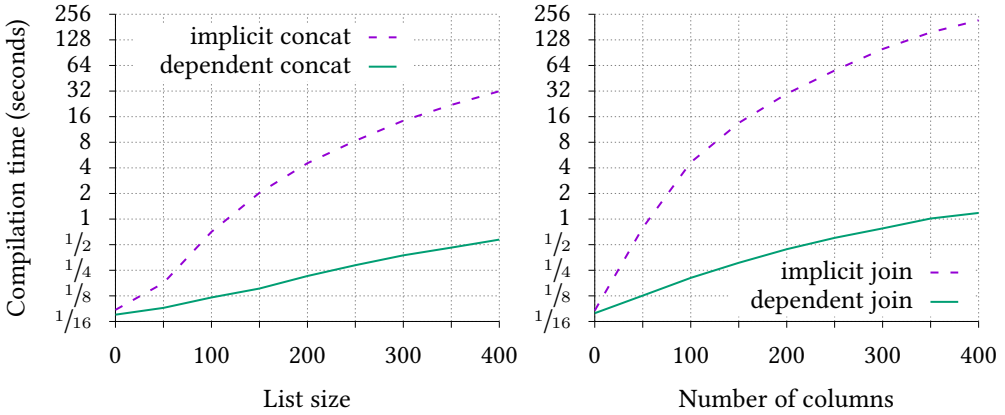


Fig. 11. Comparing the compilation times of two implementations of list concatenation and join, log. scale.

```
dependent def +[U](that: Column[U]) <: Column[_] =
  Column[AddRes[T, U]](s"${this.k} + ${that.k}")
```

Allowing programmers to construct `Column`s from string literals would defeat the purpose of a type-safe interface. Instead, programmers should extract columns from a `Table`'s schema. For that purpose, we implement the `col` method on `Table` and annotate the `Column` constructor as private.

```
dependent case class Table(schema: Lst, data: spark.DataFrame):
  dependent def col(name: String) <: Column[_] =
    dependent def find(key: String, list: Lst) <: Any =
      list match
        case Cons(head: Column[_], tail) =>
          if (head.k == key) head else find(key, tail)
        case _ => throw new Error("column not found in schema")
      find(name, schema)
  dependent def filter(predicate: Column[Boolean]) <: Table =
    new Table(this.schema, this.data.filter(predicate.k))
```

The `col` method is implemented using a nested dependent method to find the column corresponding to the given name. Thanks to the dependent annotation, the type-checker is able to statically evaluate calls to `col`. Assuming the table's schema contains a column `a` of type `Int` and columns `b` and `c` of type `Long`, the compiler will be able to infer types as follows:

```
val pred = table.col("a") + table.col("b") === table.col("c")
// Infers: { Column[Int]("a") } { Column[Long]("b") } { Column[Long]("c") }
```

Given our definitions of column addition and equality, the overall `pred` expression is typed as `Column[Boolean]`. All the safety properties stated above are therefore enforced by the dependently-typed interface presented in this section.

## 6.2 Comparison to an Existing Technique

Programmers have managed to find clever encodings that circumvent the lack of first-class support for type-level programming in many languages. These encodings can be very cumbersome, as they often entail poor error reporting and a negative impact on compilation times [McBride 2002],

[Kiselyov et al. 2004]. In Scala, implicits are the primary mechanism by which programmers implement type-level programming [Odersky et al. 2018].

Frameless [Frameless Contributors 2020] is a Scala library that implements a type-safe interface for Spark by making heavy use of implicits. Most type-level computations in this library are performed on the heterogeneous lists provided by Shapeless [Sabin 2020].

We compared the dependently-typed Spark interface presented in this section against the implicit-based implementation of Frameless. To do so, we isolated the implicit-based implementation of the `join` operation on table schemas, and compared its compilation time against the dependently-typed version presented in this section. To evaluate the scalability of both approaches we generated test cases with varying schema sizes and compiled each test case in isolation. A similar comparison is done for list concatenation, which constitutes a building block of `join`.

Figure 11 shows that, in both benchmarks, the dependently-typed implementation compiles faster than the version with implicits, and compilation time scales better with the size of the input. In the `join` benchmark, we see that the implicit-based implementation exceeds 30 seconds of compilation time around the 200 columns mark, and continues to grow quadratically. This can be explained by the nature of implicit resolution, which might backtrack during its search. The compilation time of the dependently-typed implementation grows linearly and stays below one second until the 350 columns mark. We were able to observe similar trends in the concatenation benchmark. These measurements were obtained by averaging 120 compilations on a warm compiler, and have been performed on an i7-7700K Processor running Oracle JVM 1.8.0 on Linux.

## 7 RELATED WORK

As of today, Haskell is perhaps closest to becoming dependently-typed among the general-purpose programming languages used in industry. Haskell's type families [Kiselyov et al. 2010] provide a direct way to express type-level computations. Other language extensions such as functional dependencies [Jones 2000] and promoted datatypes [Yorgey et al. 2012] are also moving Haskell towards dependent types. Nevertheless, programming in Haskell remains significantly different from using full-spectrum dependently-typed languages. A significant difference is that Haskell imposes a strict separation between terms and types. As a result, writing dependently-typed programs in Haskell often involves code duplication between types and terms. These redundancies can be somewhat avoided using the singletons package [Eisenberg and Weirich 2012], which uses meta-programming to automatically generate types from datatypes and function definitions.

In the context of Haskell, Eisenberg's work on Dependent Haskell [Eisenberg 2016] is closest to ours, in that it adds first-class support for dependent types to an established language, in a backwards-compatible way. Dependent Haskell supports general recursion without termination checks, which makes it less suitable for theorem proving. While we share similar goals, our work is differentiated by the contrasting paradigms of Scala and Haskell. Like many object-oriented languages, Scala is primarily built around subtyping and does not restrict the use of side effects. Furthermore, Eisenberg's system provides control over the relevance of values and type parameters. In contrast, our system does not support any erasure annotations and simply follows Scala's canonical erasure strategy: types are systematically erased to JVM types, and terms are left untouched. Weirich established a fully mechanized type safety proof for the core of Dependent Haskell using the Coq proof assistant [Weirich et al. 2017].

Cayenne is a Haskell-like language with dependent types introduced in 1998 by Augustsson [Augustsson 1998]. Like Dependent Haskell, it resembles our system in its treatment of termination, and differs by being a purely functional programming language. Cayenne's treatment of erasure is similar to Scala's: types are systematically erased. Augustsson proves that Cayenne's erasure is semantics-preserving, but does not provide any other metatheoretical results.



Adding dependent types to object-oriented languages is a remarkably under-explored area of research. A notable exception is the recent work of Kazerounian et al. on adding dependent types to Ruby [Kazerounian et al. 2019]. Their goals are very much aligned with ours: using type-level programming to increase program safety. Given the extremely dynamic nature of Ruby, it is unsurprising that their solution greatly differs from ours. In their work, type checking happens entirely at runtime and has to be performed at every function invocation to account for possible changes in function definitions. Safety is obtained by inserting dynamic checks, similarly to gradual typing.

The work of Campos and Vasconcelos on DOL (Dependent Object-oriented Language) [Campos and Vasconcelos 2018] shares similar goals but is limited to inequality constraints on integer parameters (in the style of [Xi and Pfenning 1998]).

Dependently-typed lambda calculi with subtyping were described at least as far back as 1988 by Cardelli [Cardelli 1988]. His type system is much more expressive than ours and allows bounded quantification over both types and terms using the notion of a Type type and power types. Unlike our system, which is designed with the concrete evaluation of types in mind, Cardelli does not provide semantics for his system and leaves the equivalence relation among types unspecified.

In [Aspinall 1994] Aspinall introduces  $\lambda_{\leq}$ , a dependently-typed system with subtyping and singleton types that resembles ours in its type language. His equivalence relation on types is more powerful and is not syntax-directed, unlike our type evaluation relation. Furthermore, singleton types in his work are indexed by the type through which equality is “viewed”, thereby enabling a form of polymorphism beyond ours. Aspinall’s system also has primitive types and allows for atomic subtyping among them, but no congruence rules, hence partially-widened forms like  $\{\text{cons choose}[\text{Top}] \text{ nil}\}$  cannot be represented.

System  $\lambda P_{\leq}$  [Aspinall and Compagnoni 1996] combines subtyping and dependent types in the Edinburgh Logical Framework. In this work, Aspinall et al. propose a type-checking algorithm for  $\lambda P_{\leq}$  which they show to be complete and terminating. Their system uses a kinding relation to ensure well-formedness of type applications. A kind system is not required in  $\lambda_{\leq}^{\text{nd}_{\{<, \{\}$  as we emulate type applications inside singleton types.

In [Stone and Harper 2000], Stone and Harper describe a dependently-typed calculus with singleton kinds and subkinding. Their type-and-kind system is similar to Aspinall’s  $\lambda_{\leq}$  term-and-type system, but operates one level up the hierarchy.

More recently, Courant [Courant 2003] developed a variant of Aspinall’s  $\lambda_{\leq}$  with a type-inference algorithm that he proves to be sound and complete. The main takeaway from Courant’s work is the inclusion of a coercion rule in delta reduction. These coercions are used to “tag” variables with their declared type, which prevents these types from being lost during substitution. Our formalism resembled Courant’s system, it shares the SUBSING subtyping rule (SUB/SINGL in Courant’s work), and  $\beta\delta$ -reduction.

Pure Type Systems [Barendregt 1991] provide a unified presentation of systems of dependently-typed  $\lambda$ -calculus by using a single syntactic category for both terms and types.

In [Zwanenburg 1999], Zwanenburg defines an extension of pure type systems that include both subtyping and bounded quantification. A central design decision of his system is that subtyping rules do not depend on typing rules. The absence of circularity simplifies both the theory and the metatheory, at the cost of having to define subtyping on pseudoterms rather than only well-typed terms. Another limitation of Zwanenburg’s theory is that it cannot be extended with a Top-type.

Pure Subtype Systems [Hutchins 2010] is another framework with unified syntax; it differs from traditional approaches in that it uses a single relation, subtyping, that subsumes typing, subtyping, and type evaluation as found in our system. Their system allows for partially-widened types similar

to ours and also enables the computation with different levels of precision. For instance, it can conclude that  $\text{Int} + 5$  can be approximated as  $\text{Int}$ . The paper presents a partial investigation of the metatheory, but the proof of soundness remains incomplete. Nevertheless, Hutchins reports that he has not been able to construct a counter-example, even with the addition of fixpoints.

In [Yang and Oliveira 2017], Yang and Oliveira propose a dependently-typed generalization of System  $F_{\leq}$  with unified syntax and a single relation that subsumes typing and subtyping. In their system, type computations are driven by cast operators: each reduction or expansion step requires an annotation to explicitly instruct the type checker to take a step. Explicit casts make it possible to allow general recursion without compromising decidability of type checking. It would be interesting to study variants of  $\lambda_{\leq, \dagger}^{\text{nd}}$  based on explicit casts instead of our finitized fix.

Dependent-object types [Amin and Rompf 2017] model the core of Scala’s type system and include type members and path-dependent types, which are not represented in our formalism. Even though they introduce a form of dependency, path-dependent types were not designed for type-level computation, rendering their original goals largely orthogonal to ours.

## REFERENCES

- Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*. ACM.
- David Aspinall. 1994. Subtyping with singleton types. In *International Workshop on Computer Science Logic*. Springer.
- David Aspinall and Adriana Compagnoni. 1996. Subtyping dependent types. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE.
- Lennart Augustsson. 1998. Cayenne — a Language with Dependent Types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP’98)*. ACM.
- Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of functional programming* 1, 2 (1991).
- Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq’Art The Calculus of Inductive Constructions* (1st ed.). Springer.
- Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013).
- Joana Campos and Vasco T Vasconcelos. 2018. Dependent Types for Class-based Mutable Objects. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Luca Cardelli. 1988. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’88)*. ACM.
- Adam Chlipala. 2010. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. In *PLDI’10: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*.
- Judicaël Courant. 2003. Strong normalization with singleton types. *Electronic Notes in Theoretical Computer Science* 70, 1 (2003).
- Richard A Eisenberg. 2016. *Dependent types in Haskell: Theory and practice*. Ph.D. Dissertation. University of Pennsylvania.
- Richard A Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons, In Haskell Symposium 2012. *ACM SIGPLAN* 47, 12.
- Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP’07)*. Springer-Verlag, Berlin, Heidelberg.
- Robert W. Floyd. 1967. Nondeterministic Algorithms. *J. ACM* 14, 4 (Oct. 1967).
- Frameless Contributors. 2016–2020. Frameless. <https://github.com/typelevel/frameless>.
- Jad Hamza, Nicolas Voirol, and Viktor Kunčák. 2019. System FR: Formalized Foundations for the Stainless Verifier. *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019).
- Susumu Hayashi. 1991. Singleton, union and intersection types for program extraction. In *International Symposium on Theoretical Aspects of Computer Software (TACS’91)*. Springer.
- DeLesley S. Hutchins. 2010. Pure Subtype Systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’10)*. ACM.
- Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP ’00)*. Springer-Verlag, London, UK, UK.
- Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S Foster, and David Van Horn. 2019. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.



- Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. 2010. Fun with type functions. In *Reflections on the Work of CAR Hoare*. Springer.
- Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM.
- EPFL IC LARA. 2019. Stainless: Formal Verification for Scala. <https://stainless.epfl.ch/>.
- Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages*. ACM.
- Conor McBride. 2002. Faking it: Simulating dependent types in Haskell. *Journal of functional programming* 12, 4-5 (2002).
- Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM.
- Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.
- Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simply: Foundations and Applications of Implicit Function Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)*. ACM.
- Miles Sabin. 2011–2020. Shapeless. <https://github.com/milessabin/shapeless>.
- Michael Sipser. 2013. *Introduction to the Theory of Computation (3rd ed.)*. Cengage Learning. ISBN-13: 978-1-133-18779-0.
- Christopher A Stone and Robert Harper. 2000. Deciding type equivalence in a language with singleton kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
- Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM.
- Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018).
- Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2017).
- Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*.
- Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A Eisenberg. 2017. A specification for dependent types in Haskell. In *Proceedings of the ACM on Programming Languages (ICFP'17)*, Vol. 1. ACM.
- Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal.
- Yanpeng Yang, Xuan Bi, and Bruno CDS Oliveira. 2016. Unified syntax with iso-types. In *Asian Symposium on Programming Languages and Systems*. Springer.
- Yanpeng Yang and Bruno CDS Oliveira. 2017. Unifying typing and subtyping. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).
- Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12)*. ACM.
- Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016).
- Jan Zwanenburg. 1999. Pure type systems with subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer.