A Type System For Feature Engineering (Extended Abstract)

Weixi Ma
mavc@fastmail.com
Meta
USA

Serena Chan
tkschan@meta.com
Meta
USA

Fei Yu
feiy@meta.com
Meta
USA

1 Introduction

Feature engineering is the preprocess of training and serving machine learning models. Programmers formalize their domain knowledge into data operations that convert raw data to proper model inputs. Features are usually written in languages of the Sql-family, which includes F3 (Facebook Feature Framework, first introduced by Chung and Yang [5]), a compiler developed at Meta.

During the development of F3, we have found that a compiler for feature engineering may benefit from dependent type theory [10] and its implementation techniques. Here, we demonstrate the type system of F3, and its practical applications in daily feature engineering.

2 The F3 Language

An F3 feature describes a data processing flow, using a directed acyclic graph. Nodes of a graph are *operators*. Operators are the built-in functions that are commonly seen in the Sql languages. Figure 1 shows a few operators as examples. An operator may contain some *expressions* to complete the missing expressiveness. F3 programmers usually implement expression-level functions in a different, more performant language and then import these implementations as function symbols in F3 expressions.

Expressions can be formalized by ordinary lambda terms [6]. Since our goal is to illustrate how dependent types improve a Sql compiler, here we focus on operators (the crux of a Sql language) and omit the discussions of expressions (lambda calculus has been better studied).

Figure 2 is an F3 feature. It reads a table from a database. Each row of this table contains one column: an integer `x`. The rest of the operators are self-explanatory except for the `LeftJoin` that creates t_4. The result of a $\texttt{LeftJoin}(t_{left}, t_{right}, k)$ contains all columns in t_{left}; it also contains the columns in t_{right} that pass the condition k.

Another feature may describe the same computation, as in Figure 3. Since the left parent has all its columns preserved, the computation is the same, if we move the `Filter` below the `LeftJoin`.

F3 compiler has a critical mission: to detect such semantic equivalence that underlies different ways of using these Sql operators. Sometimes, the compiler restructures a feature by moving operators around to optimize runtime performance– we need to validate if such a restructure is semantic preserving. Sometimes, the compiler needs to prevent a programmer from deploying a new feature, if an existing feature of the same semantics is detected, to save resources.

p	::=	let_o	features
	|	let_o, p	
o	::=	`Read(table)`	read from database
	|	$\texttt{Add}(t, let_e)$	create a new column
	|	$\texttt{Select}(t, let_e)$	non-preserving `Add`
	|	$\texttt{Filter}(t, e)$	remove some rows
	|	$\texttt{LeftJoin}(t_0, t_1, e)$	left join
	|	$\texttt{RightJoin}(t_0, t_1, e)$	right join
	|	t	tables
e, k	::=	x	columns (variables)
	|	$f(e)$	application
let_o	::=	$t = o$	bind a table
let_e	::=	$x = e$	bind a variable
t, t_0, t_1			table names
x, y, z			variable names
f, g			function symbols

Figure 1. Grammar of F3 features

$t_0 = \texttt{Read}(\{x : int\})$	t_0 is ({x=42}, {x=5})
$t_1 = \texttt{Add}(t_0, y = double(x))$	t_1 is ({x=42, y=84}, {x=5, y=10})
$t_2 = \texttt{Filter}(t_1, less\_than(y, 20))$	t_2 is ({x=5, y=10})
$t_3 = \texttt{Select}(t_0, z = x)$	t_3 is ({z=42}, {z=5})
$t_4 = \texttt{LeftJoin}(t_2, t_3, x \stackrel{?}{=} z)$	t_4 is ({x=5, y=10, z=5})

Figure 2. F3 example 1

$t_0 = \texttt{Read}(\{x : int\})$	t_0 is ({x=42}, {x=5})
$t_1 = \texttt{Add}(t_0, y = double(x))$	t_1 is ({x=42, y=84}, {x=5, y=10})
$t_2 = \texttt{Select}(t_0, z = x)$	t_2 is ({z=42}, {z=5})
$t_3 = \texttt{LeftJoin}(t_1, t_2, x \stackrel{?}{=} z)$	t_3 is ({x=5, y=10, z=5}, {x=42, y=84, z=42})
$t_4 = \texttt{Filter}(t_3, less\_than(y, 20))$	t_4 is ({x=5, y=10, z=5})

Figure 3. F3 example 2

$$
\begin{array}{lcll}
S & ::= & <\tau;\kappa;\rho> & \text{Schemata} \\
\tau & ::= & \epsilon \mid x:T,\tau & \text{DataTypes} \\
\kappa & ::= & \epsilon \mid e,\kappa & \text{NormConds} \\
\rho & ::= & \epsilon \mid x=e,\rho & \text{NormVars} \\
T & ::= & int \mid float \mid bool \mid S \mid ... & \text{types}
\end{array}
$$

$$
\boxed{[\![e]\!]_\rho \Rightarrow e}
$$

$$
\begin{array}{rcl}
[\![x]\!]_\rho & \Rightarrow & \rho(x) \\
[\![f(x)]\!]_\rho & \Rightarrow & [\![f]\!]_\rho([\![x]\!]_\rho)
\end{array}
$$

Figure 4. Grammar of F3 types and NbE rules

3 The F3 Type System

We now introduce our solution to validate semantic equivalence: dependent types.

3.1 Types for tables

As in Figure 4, each table is of type *Schema*, a three-tuple that captures

- *DataTypes* that map column names to their types,
- *NormConds* that collect all normalized conditions (such as a `Filter` node and a join condition), and
- *NormVars* that map columns names to the computations (in their normal forms) that create them.

3.2 NbE for expressions

NbE is a technique to find the normal forms of programs (discussed by Abel et al. [1]). It usually runs a special interpreter that reduces syntactic forms to some internal forms. E.g. `Expression(40 + 2)` and `Expression(double(21))` are both reduced to `Value(42)`. It then runs a reifier to read internal forms back to syntactic forms, e.g., `Expressions(42)`.

Most NbE algorithms resolve $\alpha\beta\eta$-equivalence (described by Barendregt [3]) for general purpose programming languages. For F3's case, we focus only on β, which is about computing the NormVars of columns. Our algorithm is rather straightforward, as in Figure 4. Since operators do not involve any binders (and thus no closures), there is no substantial difference between the syntactic forms and the internal forms. So, we omit the grammar for `Values`.

3.3 Inference for operators

F3 uses a typical bi-directional type checker, as shown by Pierce and Turner [11]. For a program, the type system chooses between type synthesis (generating a type) and type check (validating a user annotation). The choice depends on whether the program is a constructor or an eliminator (described by Dybjer [7]). All F3 operators take forms of function applications (thusly eliminators), which synthesis applies to.

Figure 5 sketches the type inference rules for operators. We write a judgment form for synthesizing a type for an operator as $\vdash o :< \tau;\kappa;\rho >$. (In a more conventional presentation of bi-directional rules, a synthesis judgement may look like $e \Rightarrow t$, while a check judgement is $e \Leftarrow t$. Since this paper does not include any check judgements, we omit this difference.) On operator level, the inference context comes from the parents operators. So, their judgment forms do not contain any explicit inference contexts. On expression level, we write a judgment form for synthesizing a type for an expressions as $\tau \vdash e : T$.

- A `Read` operator refers to a physical table in a database. The table contains a mapping from column names to column data types, τ. The resultant type of Read includes (1) τ, (2) an empty set of conditions, and (3) NormVars that are initialized from τ. $\tau 2\rho$ denotes a function: it creates a mapping from columns names to constant symbols for the physical table. For example, `Read`$(\{x : int\})$: its type is $< x : int;\epsilon;x = t_0.x >$, where $t_0.x$ is a constant symbol, assuming the physical table is named t_0.
- Filter extends κ of its parent node with the normalized condition, and preserves τ and ρ from its parent.
- Add and Select are similar. They synthesize the types of their parents and the new columns. Then, the `Add` operator preserves the DataTypes and NormVars from its parent, while the `Select` operator discards them. In practice, both operators are variable-arity functions.
- LeftJoin combines its two parents. The resultant τ is a union between its parents' τs, assuming no naming conflicts. The join condition, k, is normalized and added to the union of its parents' κs. The most interesting change, however, is in the NormVars.

In $\rho_0 \uplus join(\rho_1)$, $join$ is a function that converts ρ_1 to a new mapping. The new mapping preserves all left-hand sides of ρ_1 and wraps all right-hand sides of ρ_1 in a function application, indicating that a join operator has changed the computation of these columns. For the example in Figure 2, the NormVar of t_2 is $\{x = t_0.x, y = double(t_0.x)\}$, the NormVar of t_3 is $\{z = t_0.x\}$, the resultant NormVar of t_4 is $\{x = t_0.x, y = double(t_0.x), z = join(t_0.x)\}$

4 Applications In Feature Engineering

Here we discuss how the dependent type system enhances F3 compiler.

4.1 Semantic sameness

As shown in Figure 2 and Figure 3, F3 compiler needs to identify meanings among seemingly-different features. With dependent types, the problem of identifying semantic equivalence is reduced to the problem of checking structural equivalence between the results of type synthesis.

$$\boxed{\vdash o :< \tau; \kappa; \rho >}$$

$$\textsc{Read}\ \frac{}{\vdash \texttt{Read}(\tau) :< \tau; \epsilon; \tau 2\rho(\tau) >} \qquad \textsc{Filter}\ \frac{\vdash t :< \tau; \kappa; \rho > \quad \tau \vdash k : bool}{\vdash \texttt{Filter}(t, k) :< \tau; [\![k]\!]_\rho, \kappa; \rho >}$$

$$\textsc{Add}\ \frac{\vdash t :< \tau; \kappa; \rho > \quad \tau \vdash e : T}{\vdash \texttt{Add}(t, x = e) :< x : T, \tau; \kappa; x = [\![e]\!]_\rho, \rho >} \qquad \textsc{Select}\ \frac{\vdash t :< \tau; \kappa; \rho > \quad \tau \vdash e : T}{\vdash \texttt{Select}(t, x = e) :< x : T; \kappa; x = [\![e]\!]_\rho >}$$

$$\textsc{LeftJoin}\ \frac{\vdash t_0 :< \tau_0; \kappa_0; \rho_0 > \quad \vdash t_1 :< \tau_1; \kappa_1; \rho_1 > \quad \tau_0 \uplus \tau_1 \vdash k : bool}{\vdash \texttt{LeftJoin}(t_0, t_1, k) :< \tau_0 \uplus \tau_1; [\![k]\!]_{\rho_0 \uplus \rho_1}, (\kappa_0 \uplus \kappa_1); \rho_0 \uplus join(\rho_1) >}$$

$$\textsc{RightJoin}\ \frac{\vdash t_0 :< \tau_0; \kappa_0; \rho_0 > \quad \vdash t_1 :< \tau_1; \kappa_1; \rho_1 > \quad \tau_0 \uplus \tau_1 \vdash k : bool}{\vdash \texttt{RightJoin}(t_0, t_1, k) :< \tau_0 \uplus \tau_1; [\![k]\!]_{\rho_0 \uplus \rho_1}, (\kappa_0 \uplus \kappa_1); join(\rho_0) \uplus \rho_1 >}$$

Figure 5. Inference rules for operators

In our example, both features have type

$$< \{x : int, y : int, z : int\};$$
$$\{less\_than(double(t_0.x), 20)\};$$
$$\{x = t_0.x, y = double(t_0.x), z = join(t_0.x)\} > .$$

If we replace the `LeftJoins` with `RightJoins` in both examples, then these two features no longer share the same computation, since the `Filter` information of the left parent is now lost. Our type checker is able to detect this–the two features now have different types.

Figure 2 now has

$$< \{x : int, y : int, z : int\};$$
$$\{less\_than(\mathbf{double(t_0.x)}, 20)\};$$
$$\{x = join(t_0.x), y = join(double(t_0.x)), z = t_0.x\} >;$$

Figure 3, instead, has

$$< \{x : int, y : int, z : int\};$$
$$\{less\_than(\mathbf{join(double(t_0.x))}, 20)\};$$
$$\{x = join(t_0.x), y = join(double(t_0.x)), z = t_0.x\} > .$$

NormVars may be impacted by other operators, such as `Union` and `GroupBy`. Their inference rules are similar to `LeftJoin` and `RightJoin`–wrapping existing NormVars within dummy function symbols.

In practice, for the comparison of ρs, we ignore the variables that occur in ρ but not in τ. They are intermediate computations and do not make any difference in the result (assuming no side-effects).

4.2 Privacy enforcement

Not all features can eventually be used in production. Different countries have introduced various legislation on data protection. If a feature contains sensitive columns in its result, then the feature must not be deployed.

The schema NormVars help with such validations. In the future, if one is to detect whether the result of a feature is computed from data that is not policy-compliant, then one may scan the resultant schema of this feature, which contains the normal forms that may trace back to the sensitive columns of the original table.

4.3 Test generation

F3 compiler is accompanied with thousands of unit tests. To validate a compiler transformation, it is time consuming and error-prone to manually create complex features as test inputs. These tests are also difficult to maintain: we have noticed that one change often breaks test cases in other irrelevant compiler rules, due to their hard coded test cases.

Dybjer et al. [8] show that dependent types can be used to generate comprehensive test cases. With such a test generator, we may significantly improve the productivity of compiler developers, by property tests. In a nutshell, generating test cases is just the opposite of normalization. The inference rules in Figure 5 can be used to generate features, if programmers specify their schemata.

5 Related Work

Dependent types have been introduced to Sql-languages to improve authoring experience, such as detecting bugs earlier and requiring fewer type casts. As examples, (1) Chlipala [4]'s Ur is a metaprogramming tool for web applications that builds and runs data queries, (2) Kazerounian et al. [9] have introduced CompRDL for Ruby libraries.

On the other hand, Baltopoulos et al. [2] is motivated similarly as we do: use types as a proxy to help the compiler with semantics understanding. One of their application is to check database integrity.

Acknowledgment

We thank Daniel P. Friedman and the anonymous reviewers of TyDe 2023 for their feedback on the earlier drafts. The first author thanks Yulai Bi, Zhenghao Zhao, and Yining Yang for educating on Sql semantics.

F3 was practically an untyped system, which made it a tremendous engineering effort to materialize these mathematical formulas into a program running in daily production and validating thousands of features. For this, we thank David Chung, Ilina Mitra, Jay Huang, Rocky Liu, Ping Chen, and Yang Qiao for their leadership support and investment in our endeavor. We also thank Junhua Gu and Abhimanyu Sharma for co-leading the execution. We express gratitude to every engineer at Meta who has contributed to rolling out F3's type system.

References

[1] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, April 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.02.025. URL http://www.sciencedirect.com/science/article/pii/S1571066107000977.

[2] Ioannis Baltopoulos, Johannes Borgström, and Andrew Gordon. Maintaining Database Integrity with Refinement Types. pages 484–509, July 2011. ISBN 978-3-642-22654-0. doi: 10.1007/978-3-642-22655-7_23.

[3] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in logic and the foundations of mathematics; v. 103. North-Holland PubCo, Sole distributors for the USA and Canada Elsevier North-Holland, 1981.

[4] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. *ACM SIGPLAN Notices*, 45(6):122–133, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806612. URL https://doi.org/10.1145/1809028.1806612.

[5] David Chung and Qiao Yang. F3: Next-generation Feature Framework at Facebook, December 2020. URL https://shorturl.at/gyMXY.

[6] Alonzo Church. *The Calculi of Lambda-conversion*. Princeton University Press, Humphrey Milford Oxford University Press, 1941.

[7] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4): 440–465, July 1994. ISSN 0934-5043, 1433-299X.

[8] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining Testing and Proving in Dependent Type Theory. In *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 188–203. Springer, Berlin, Heidelberg, September 2003. ISBN 978-3-540-40664-8 978-3-540-45130-3.

[9] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 966–979, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314630. URL https://dl.acm.org/doi/10.1145/3314221.3314630.

[10] Per Martin-Löf. A Theory of Types. 1972.

[11] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL https://dl.acm.org/doi/10.1145/345099.345100.