Freezing Bidirectional Typing (Extended Abstract)*

Wenhao Tang
wenhao.tang@ed.ac.uk
The University of Edinburgh, UK

Shengyi Jiang
syjiang@cs.hku.hk
The University of Hong Kong, China

Bruno C. d. S. Oliveira
bruno@cs.hku.hk
The University of Hong Kong, China

Sam Lindley
sam.lindley@ed.ac.uk
The University of Edinburgh, UK

Abstract

We propose Frost, a novel approach to bidirectional type inference for first-class polymorphism. Conventional bidirectional typing usually considers unidirectional information flow between functions and arguments, typically from the former to the latter. Frost allows local type information to flow back and forth between functions and arguments via *pattern inference*. A *freezing* operator is used to control the direction of information flow when there is no best choice. The flexible information flow of Frost enables expressive and predictable inference of polymorphic instantiation and implicit type abstraction. We conjecture that Frost admits a simple, sound and complete type inference algorithm and generalizes to type inference for modal types.

Keywords: first-class polymorphism, bidirectional typing

1 Introduction

ML-style type inference [5, 17] has enjoyed great success in programming languages for inferring rank-1 polymorphic types without any user annotations including type abstraction and instantiation. Nonetheless, polymorphic types are treated as second-class, since quantifiers must be top-level and instantiated with monomorphic types. Many approaches to type inference for *first-class polymorphism*, which allows quantifiers to appear anywhere and be instantiated with polymorphic types, have been proposed in the past thirty years [1, 10, 11, 14–16, 18, 20, 21]. These approaches make different trade-offs between user-provided annotations, predictability, and complexity of the type inference algorithms. However, first-class polymorphism still remains one of the most challenging problems of type inference.

Bidirectional typing [6, 7, 13, 19, 26] for *higher-rank polymorphism* (which allows quantifiers to appear anywhere but restricts instantiation to monomorphic types) has enjoyed great success. Key to this success is the property that when restricting to monomorphic instantiation, we can easily guess monotypes and solve them via unification. There are two modes in bidirectional typing: a *checking* mode which checks a term against an input type, and a *synthesis* mode which synthesises an output type from a term. The local type information provided by the checking mode helps with inferring *implicit type abstraction* and *polymorphic arguments*.

We start from the observation that local type information also provides valuable guidance on *polymorphic instantiation* for first-class polymorphism. However, traditional bidirectional type systems place rather severe restrictions on the flow of local type information. We extend bidirectional typing with expressive information flow and a freezing operator to control flow directions inspired by FreezeML [10]. We propose Frost[1], a novel type system for first-class polymorphism based on bidirectional typing. Frost follows the principle shared by many existing systems that polymorphism should never be guessed. Frost only instantiates a type with a polymorphic type when the local type information requires such a type; otherwise it guesses a monomorphic type.

Local type information in Frost mainly serves three purposes: implicit type abstraction, polymorphic arguments, and polymorphic instantiation. However, theses purposes typically require different directions of information flow between functions and arguments. Consider the application $M\ N$. When type information flows from M to N, we can type check N and infer potential implicit type abstraction. When type information flows from N to M, we can instantiate M with the information provided by N. To support both directions simultaneously, we first infer a *pattern* for the argument N. A pattern is a type skeleton with holes and only provides the type information independent of the surrounding context. That is, no matter where a term is, its type must be consistent with its pattern. The pattern of N always flows to M before the type information provided by M flows back to N. To control which information flows from arguments to functions, we introduce a term-level freezing operator. Freezing a term makes its type independent of the surrounding context and thus can be collected in patterns and used for synthesising the type of other terms.

In summary, Frost has the following main advantages.

- Predictability. Frost has a clear declarative presentation.
- Simplicity. Frost does not require new type syntax and admits a sound and complete algorithmic type system with simple constraint solving (work in progress).
- Expressiveness. Compared to QuickLook [20], which also uses local type information for first-class polymorphism, the pattern inference and freezing of Frost enable more fine-grained control over local information flow, allowing more programs to be synthesised and checked.

The remainder of this extended abstract gives an overview of Frost with examples and illustrative typing rules. For space reasons, we do not discuss the specification of Frost in this

*This is an extended abstract for the ML family workshop 2025.

[1]Frost forms when vapour *flows* to cold surfaces and *freezes* into ice crystals.

extended abstract. Frost is still work in progress. We are working on formalising the type inference algorithm and proving its soundness and completeness. We also plan to extend Frost to type inference for modal effect types, inferring implicit modality introduction and elimination [23].

2 Frost

2.1 Flowing from Functions to Arguments

Most bidirectional type systems [4, 7, 8, 12, 27] infer the type of the function first, instantiate the type, and use its argument types to type check each argument. The local type information flows from functions to arguments as illustrated in the following informal rule.

FunToArg

$$\frac{\Gamma \vdash M \Rightarrow A \qquad A \preceq A_1 \to B \qquad \Gamma \vdash N \Leftarrow A_1}{\Gamma \vdash M\ N \Rightarrow B}$$

We write $\Rightarrow$ for synthesis mode and $\Leftarrow$ for checking mode. We write $A \preceq B$ if A can be instantiated to B. As a result, the argument N is in checking mode, allowing us to infer an implicit type abstraction for it. When N is a lambda $\lambda x.N'$, the checking mode also gives us the type of the parameter x. For instance, consider the function application $\texttt{poly}\ (\lambda x.x)$ where $\texttt{poly} : (\forall a.a \to a) \to \texttt{Int}$. Using the argument type $\forall a.a \to a$ of $\texttt{poly}$ to type check $\lambda x.x$, we can infer that there is an implicit type abstraction Λa wrapping the argument $\lambda x.x$, and the parameter x should have type a.

Frost supports the information flow from functions to arguments. However, only having this direction of information flow is not enough as we have no guidance when we need to instantiate the function type. Previous systems adopting this rule solely mitigate the problem by adopting unification [4, 7, 12, 27] on top of the bidirectional typing to allow undirectional information flow. However, since arbitrary implicit polymorphic instantiation is undecidable [3, 24], they only consider predicative higher-rank polymorphism where instantiation is restricted to monomorphic types, In contrast, Frost enables implicit polymorphic instantiation by allowing selective reversal of the direction of information flow.

2.2 Flowing from Arguments to Functions

Local type inference [19] supports inferring argument types first and using these types to guide the typing of a function as illustrated in the following informal rule.

ArgToFun

$$\frac{\Gamma \vdash N \Rightarrow A_1 \qquad \Gamma \vdash M \Rightarrow A \qquad A \preceq A_1 \to B \text{ guided by argument type } A_1}{\Gamma \vdash M\ N \Rightarrow B}$$

When the function has a polymorphic type, the type information from arguments provides useful guidance on how we should instantiate the polymorphic function type, especially enabling polymorphic instantiation. For instance, consider the application $\texttt{head ids}$ where $\texttt{head} : \forall a.\texttt{List}\ a \to a$ and $\texttt{ids} : \texttt{List}\ (\forall a.a \to a)$. We know that the first argument type $\texttt{List}\ a$ of $\texttt{head}$ should match the type of $\texttt{ids}$. Thus, a should be instantiated to the polymorphic type $\forall a.a \to a$.

The information flow from arguments to functions is also useful for assigning types to parameters of unannotated lambda abstractions. For example, consider inferring the type of the application $(\lambda x.x)\ \texttt{ids}$. It is clear that the parameter x should have exactly the type of the argument $\texttt{ids}$. However, the ArgToFun rule is not powerful enough to cover this case. Xie and Oliveira [26] propose a special *application* mode $\Gamma \mid \Psi \vdash M \Rightarrow A$ which extends synthesis mode with a context Ψ collecting the types of arguments that M is applied to. When an application $M\ N$ is encountered, the type of the argument N is first inferred and added to Ψ when typing M as illustrated by the following rule.

LetArgGoFirst [26]

$$\frac{\Gamma \mid \Psi \vdash N \Rightarrow A \qquad \Gamma \mid A, \Psi \vdash M \Rightarrow A \to B}{\Gamma \mid \Psi \vdash M\ N \Rightarrow B}$$

Their system supports examples like $(\lambda x.x)\ \texttt{ids}$ but does not support polymorphic instantiation.

Frost supports information flow from arguments to functions and uses it for both polymorphic instantiation and assigning types to parameters of unannotated lambda abstractions. In fact, similarly to application mode, the typing judgement of Frost has the form $\Gamma \mid \rho \vdash M \Rightarrow A$. The main difference is that ρ does not contain types for arguments but their patterns, the key feature that enables Frost to support both directions of information flow simultaneously.

2.3 Flowing Back and Forth via Patterns

Naively having both directions of information flow in Sections 2.1 and 2.2 would require backtracking, which is unrealistic for practical type inference. Instead of backtracking or relying on ad-hoc heuristics, Frost supports both directions of information flow by first collecting pattern information from arguments, then inferring the function type guided by the pattern information, and finally using the argument types of the inferred function type to type check each argument. The typing rule for application in Frost is as follows.

T-App

$$\frac{\Gamma \mid P, \rho \vdash M \Rightarrow A \to B \qquad \Gamma \vdash N \Rightarrow\!\!\!\Rightarrow P \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \mid \rho \vdash M\ N \Rightarrow B}$$

The pattern inference judgement $\Gamma \vdash N \Rightarrow\!\!\!\Rightarrow P$ infers a pattern P for a term N under context Γ. We add this pattern P to the pattern row ρ when inferring the type of M, similar to the LetArgGoFirst rule. Finally we type check the argument N, similar to the FunToArg rule. A pattern is a type with holes (for which we use the ghost symbol 👻) representing unknown information. We say that two patterns are *consistent* if they are equivalent modulo unknown information [22]. If N has pattern P, every type N must be consistent with the pattern P. Formally, if $\Gamma \vdash N \Rightarrow\!\!\!\Rightarrow P$, then for any type A such that $\Gamma \vdash N \Leftarrow A$, we have that A is

consistent with P. In other words, the pattern P gives the type information provided by N independently of where N is and which information the context provides.

For instance, the term $\lambda x.x$ has pattern $\forall\text{👻}.\text{👻} \to \text{👻}$, which covers any function type with an arbitrary number of top-level quantifiers.[2] As another example, the term `ids` has pattern $\forall\text{👻}.\texttt{List}\ (\forall a.a \to a)$. The $\forall\text{👻}.$ reflects potential implicit type abstraction of `ids`. Note that as with many other systems for first-class polymorphism, we only care about shallow instantiation [2] where the polymorphic type $\forall a.a \to a$ inside `List` cannot be instantiated.

Back to the example `poly` $(\lambda x.x)$ in Section 2.1. The pattern $\forall\text{👻}.\text{👻} \to \text{👻}$ of $\lambda x.x$ does not contribute to the typing of `poly` but is harmless since it is consistent with $\forall a.a \to a$.

Back to the example `head ids` in Section 2.2, what really flows from `ids` to `head` is not the type of `ids` but its pattern $\forall\text{👻}.\texttt{List}\ (\forall a.a \to a)$. We still know that we should instantiate `head` with $\forall a.a \to a$ because the argument type `List` a must be consistent with the pattern $\forall\text{👻}.\texttt{List}\ (\forall a.a \to a)$.

The above examples essentially only use one direction of information flow. We give another example which uses both directions of information flow. Consider a higher-rank polymorphic function $\texttt{poly}' : \forall a.(\forall b.b \to b \times a) \to a$. Frost infers the type `List` $(\forall a.a \to a)$ for the application $\texttt{poly}'\ (\lambda x.(x, \texttt{ids}))$. First, the pattern information $\forall\text{👻}.\text{👻} \to \text{👻} \times (\forall\text{👻}.\texttt{List}\ (\forall a.a \to a))$ of the argument $\lambda x.(x, \texttt{ids})$ flows to $\texttt{poly}'$. This information tells the function to instantiate a to `List` $(\forall a.a \to a)$. After instantiation, we get the argument type $\forall b.b \to b \times \texttt{List}\ (\forall a.a \to a)$, which we use to type check the argument $\lambda x.(x, \texttt{ids})$ and infers that there should be an implicit type abstraction Λb.

2.4 Freezing Flow

Sometimes the pattern is not as informative as we want. For instance, consider the application `single id` where $\texttt{single} : \forall a.a \to \texttt{List}\ a$. For the argument `id`, the most precise pattern information we could collect is $\forall\text{👻}.\text{👻} \to \text{👻}$ because of implicit instantiation and type abstraction. This pattern information does not tell us anything useful for polymorphic instantiation of `single`. In this case Frost would just guess a monotype consistent with $\forall\text{👻}.\text{👻} \to \text{👻}$ to instantiate a and finally infer `List` $(\tau \to \tau)$ with some monotype τ for `single id`. This is unsatisfying as `List` $(\forall a.a \to a)$ is also a perfectly reasonable type for `single id` which even more precisely reflects the point that each element in the list is itself a polymorphic function. Inferring this type requires more information from `id` to flow to the function `single` and guides its instantiation.

We introduce a term-level *freezing* operator to control the directions of information flow. When a term is frozen, its type is what we infer from it and cannot depend on the surrounding context. As a result, we can directly use the inferred type as its pattern without considering potential implicit type abstraction and instantiation. In other words, previously for a term there are inward and outward information flows, while freezing a term fixes the direction to outward only. Back to the example of `single id`, we could freeze `id` by writing $\lceil\texttt{id}\rceil$. Frost infers the type `List` $(\forall a.a \to a)$ for `single` $\lceil\texttt{id}\rceil$. As another example, given $\texttt{fint} : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$, the application $\texttt{fint}\ \lceil\texttt{id}\rceil$ is ill-typed because freezing `id` disables instantiation.

Freezing naturally propagates through program structures such as let-bindings, bodies of lambda abstractions, and data constructors. For instance, Frost still infers the type `List` $(\forall a.a \to a)$ for `single` $(\textbf{let}\ x = 42\ \textbf{in}\ \lceil\texttt{id}\rceil)$.

2.5 More Examples

Since Frost collects patterns of arguments in the typing judgement, information from multiple arguments can be used in together to guide polymorphic instantiations. For instance, given $\texttt{choose} : \forall a.a \to a \to a$, Frost infers the type $(\forall a.a \to a) \times (\forall a.a \to a)$ for the following term.

$$\texttt{choose}\ (\lambda x.x, \lceil\texttt{id}\rceil)\ (\lceil\texttt{id}\rceil, \lambda x.x)$$

For the same reason of collecting argument patterns, Frost is not sensitive to η-expansion. For instance, η-expanding `choose` in the above term does not prevent Frost from inferring the type $(\forall a.a \to a) \times (\forall a.a \to a)$.

Frost allows instantiation to happen after each step of application. For instance, Frost infers type `Int` for `head ids` 42. After the first application `head ids` we get the type $\forall a.a \to a$, which is instantiated with `Int` before the second application.

Frost also allows information to flow from one argument to another argument by using the function as a bridge. Consider the application `choose` $\lceil\texttt{id}\rceil\ (\lambda x.x)$. Frost infers that there is an implicit type abstraction for $\lambda x.x$. The information flows from the first argument to `choose` and then to the second argument. Frost is not sensitive to the order of arguments and still works for `choose` $(\lambda x.x)\ \lceil\texttt{id}\rceil$.

Frost adopts the ML value restriction [25]: implicit type abstraction is only allowed for values. For instance, `poly (id id)` is ill-typed because `id id` is not a value. We can make it well-typed by freezing: `poly (id` $\lceil\texttt{id}\rceil$`)`. Now $\texttt{id}\ \lceil\texttt{id}\rceil$ directly gives the type $\forall a.a \to a$ without implicit type abstraction.

A more thorough demonstration of Frost can be found in Appendix A including both positive and negative ones, comparing against QuickLook [20], GI [21], HMF [15], and FreezeML [10]. Most examples are from Serrano et al. [20, 21], with some new examples demonstrating Frost's advantage on handling bidirectional information flow.

[2] Unlike dynamic types in gradual typing, our ghosts not only appear as types but also appear as $\forall\text{👻}.$ which matches any number of quantifiers.

References

[1] Didier Le Botlan and Didier Rémy. 2003. ML$^{\text{F}}$: raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 27–38. doi:10.1145/944705.944709

[2] Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking stability by being lazy and shallow: lazy and shallow instantiation is user friendly. In *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*, Jurriaan Hage (Ed.). ACM, 85–97. doi:10.1145/3471874.3472985

[3] Jacek Chrząszcz. 1998. Polymorphic subtyping without distributivity. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–355.

[4] Chen Cui, Shengyi Jiang, and Bruno C. d. S. Oliveira. 2023. Greedy Implicit Bounded Quantification. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 2083–2111. doi:10.1145/3622871

[5] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. doi:10.1145/582153.582176

[6] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. doi:10.1145/3450952

[7] Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. doi:10.1145/2500365.2500582

[8] Jana Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 281–292. doi:10.1145/964001.964025

[9] Frank Emrich. 2024. *Complete and easy type Inference for first-class polymorphism.* Ph. D. Dissertation. The University of Edinburgh, UK. doi:10.7488/era/4152

[10] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: complete and easy type inference for first-class polymorphism. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 423–437. doi:10.1145/3385412.3386003

[11] Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169. doi:10.1006/INCO.1999.2830

[12] Shengyi Jiang, Chen Cui, and Bruno C. d. S. Oliveira. 2025. Bidirectional Higher-Rank Polymorphism with Intersection and Union Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 2118–2148. doi:10.1145/3704907

[13] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Funct. Program.* 17, 1 (2007), 1–82. doi:10.1017/S0956796806006034

[14] András Kovács. 2020. Elaboration with first-class implicit function types. *Proc. ACM Program. Lang.* 4, ICFP (2020), 101:1–101:29. doi:10.1145/3408983

[15] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 283–294. doi:10.1145/1411204.1411245

[16] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 66–77. doi:10.1145/1480881.1480891

[17] Robin Milner, Mads Tofte, and Robert Harper. 1990. *Definition of standard ML.* MIT Press.

[18] Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL (2024), 1418–1450. doi:10.1145/3632890

[19] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. doi:10.1145/345099.345100

[20] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP (2020), 89:1–89:29. doi:10.1145/3408971

[21] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 783–796. doi:10.1145/3192366.3192389

[22] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. https://api.semanticscholar.org/CorpusID:1398902

[23] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2025. Modal Effect Types. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1130–1157. doi:10.1145/3720476

[24] Jerzy Tiuryn and Pawel Urzyczyn. 1996. The subtyping problem for second-order types is undecidable. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science.*

[25] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.

[26] Ningning Xie Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 272–299. doi:10.1007/978-3-319-89884-1_10

[27] Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. Elementary Type Inference. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:28. doi:10.4230/LIPICS.ECOOP.2022.2

A Comparison of Type Systems for First-Class Polymorphism

Figure 1 compares Frost with QuickLook (QL) [20], GI [21], HMF [15], and FreezeML [10] via examples taken from Serrano et al. [20, 21]. Figure 2 continues the comparison with new examples demonstrating the expressiveness of flexible information flow and freezing of Frost. Type signatures of functions appeared in these tables are given in Figure 3.

All systems included in the comparison table only require simple constraint solving for type inference, mostly just unification of polymorphic types. There are other systems like MLF [1], HML [16], SuperF [18] which introduce new type syntax and sophisticated constraint solving. The performance of these systems on examples in Figure 1 can be found in the literature [9, 18, 20].

		Frost	QL	GI	HMF	FreezeML
A	**Polymorphic instantiation**					
A1	$\lambda x\, y.y$	●	●	●	●	●
A2	`choose id`	●	●	●	●	●
A3	`choose nil id`	●	●	●	●	●
A4	$\lambda(x : \forall a.a \to a).x\, x$	●	●	●	●	●
A5	`id auto`	●	●	●	●	●
A6	`id auto'`	●	○	●	●	●
A7	`choose id auto`	●	●	●	●	●
A8	`choose id auto'`	○	○	○	○	○
A9	f (`choose id`) `ids` where $f : \forall a.(a \to a) \to$ `List` $a \to a$	●	●	○	○	◐
A10	`poly id` | `poly` $(\lambda x.x)$ | `id poly` $(\lambda x.x)$	●	●	●	●	◐
A11 [20]	k $(\lambda f.(f\ 42, f\ \texttt{true})$ $xs)$ where $k : \forall a.a \to$ `List` $a \to$ `Int`, xs : `List` $((\forall a.a \to a) \to (\texttt{Int}, \texttt{Bool}))$	●	●	○	○	○
A12	`app poly id` | `revapp id poly`	●	●	●	●	◐
A13	`app runST argST` | `revapp argST runST`	●	●	●	●	◐
B	**Functions on polymorphic lists**					
B1	`tail ids` | `head ids` | `single id`	●	●	●	●	●
B2	`cons id ids`	●	●	●	●	◐
B3	`cons` $(\lambda x.x)$ `ids`	●	●	●	●	◐
B4	`append (single inc) (single id)`	●	●	●	●	●
B5 [20]	`append (single id) ids`	●	●	○	○	◐
B6	`map poly (single id)`	●	●	○	○	◐
B7	`map head (single ids)`	●	●	●	●	●
B8	`head ids true`	●	●	●	●	◐
C	**Inference of polymorphic lambda binders and generalisation points**					
C1a	$\lambda f.(f\ 42, f\ \texttt{true})$	○	○	○	○	○
C1b	$\lambda(f : \forall a.a \to a).(f\ 42, f\ \texttt{true})$	●	●	●	●	●
C1c [20]	g $(\lambda f.(f\ 42, f\ \texttt{true}))$ where $g : ((\forall a.a \to a) \to \texttt{Int} \times \texttt{Bool}) \to 1$	●	●	○	○	○
C2	r $(\lambda x\, y.y)$ where $r : (\forall a.a \to \forall b.b \to b) \to$ `Int`	●	●	○	○	◐
E	***η*-expansion**					
E1a	$k\ h\ lst$	○	○	○	○	○
E1b	k $(\lambda x.h\ x)$ lst where lst : `List` $(\forall a.\texttt{Int} \to a \to a)$, $k : \forall a.a \to$ `List` $a \to a$, h : `Int` $\to \forall a.a \to a$	●	●	●	●	◐
E2a [20]	$\lambda x.$`poly` x	○	○	○	○	○
E2b [20]	$(\lambda x.$`poly` $x) : (\forall a.a \to a) \to \texttt{Int} \times \texttt{Bool}$	●	●	○	●	○
E3a	`app poly id`	●	●	●	●	◐
E3b [20]	`app` $(\lambda x.$`poly` $x)$ `id` Frost: `app` $(\lambda x.$`poly` $x)$ `[id]`	◐	○	○	○	○
E4a [20]	`map poly ids`	●	●	●	●	●
E4b [20]	`map` $(\lambda x.$`poly` $x)$ `ids`	●	●	○	○	○
E5a [20]	`compose poly head`	●	●	●	●	●
E5b [20]	$\lambda xs.$`poly (head` xs`)`	○	○	○	○	○

Figure 1. Comparison of type systems with FCP. Cover all examples in GI [21] and QL [20] (examples from QL are marked). ● means well-typed; ○ means ill-typed; ◐ means well-typed after adding type-free annotations (e.g., the freezing operator). We provide the well-typed terms with freezing operators in Frost in the case of ◐.

		Frost	QL	GI	HMF	FreezeML
E	**η-expansion (new)**					
E6	$(\lambda f.\texttt{app}\ f)\ \texttt{poly}\ \texttt{id}$	●	○	○	○	○
E7	$(\lambda f.\texttt{app}\ \texttt{poly}\ f)\ \texttt{id}$ Frost: $(\lambda f.\texttt{app}\ \texttt{poly}\ f)\ \lceil\texttt{id}\rceil$	◐	○	○	○	○
$\mathbb{B}$	**β-expansion (new)**					
$\mathbb{B}$1	$(\lambda x.\texttt{app})\ 42\ \texttt{poly}\ \texttt{id}$	●	○	●	●	◐
$\mathbb{B}$2	$(\textbf{let}\ x = 42\ \textbf{in}\ \texttt{app})\ \texttt{poly}\ \texttt{id}$	●	○	●	●	◐
$\mathbb{B}$3	$\texttt{app}\ ((\lambda x.\texttt{poly})\ 42)\ \texttt{id}$	●	○	●	●	◐
F	**Freezing and bidirectional information flow (new)**					
F1	$f\ (\lambda x.\texttt{ids})$ where $f : \forall a.(\texttt{Int} \to a) \to a$	●	○	●	●	●
F2	$f\ (\lambda x.(x, \texttt{ids}))$ where $f : \forall a.(\forall b.b \to b \times a) \to a$	●	○	●	●	◐
F3	$f\ (\lambda x.\texttt{ids})$ where $f : \forall a.((\forall b.b \to b) \to a) \to a$	●	○	○	○	○
F4	$(\lambda f.(f\ 42, f\ \text{true}))\ \texttt{id}$ Frost: $(\lambda f.(f\ 42, f\ \text{true}))\ \lceil\texttt{id}\rceil$	◐	○	○	○	○
F5	$\texttt{pair}\ (\lambda x.x)\ 42 : ((\forall a.a \to a) \to \texttt{Int}) \times \texttt{Int}$	●	●	○	○	○
F6	$\texttt{choose}\ \texttt{churchNil}\ \texttt{churchIds}$	●	○	●	●	●
F7	$\texttt{churchHead}\ (\texttt{choose}\ \texttt{churchNil}\ \texttt{churchIds})$ Frost: $\texttt{churchHead}\ (\texttt{choose}\ \texttt{churchNil}\ \lceil\texttt{churchIds}\rceil)$	◐	○	●†	●†	○
F8a	$\texttt{polys}\ (\texttt{single}\ \texttt{id})$	●	●	○	○	◐
F8b	$\textbf{let}\ xs = \texttt{single}\ \texttt{id}\ \textbf{in}\ \texttt{polys}\ xs$ Frost: $\textbf{let}\ xs = \texttt{single}\ \lceil\texttt{id}\rceil\ \textbf{in}\ \texttt{polys}\ xs$	◐	○	○	○	◐

Figure 2. Comparison of type systems with FCP (continued). New examples. † means well-typed only without value restriction.

$\texttt{head} : \forall a.\texttt{List}\ a \to a$
$\texttt{tail} : \forall a.\texttt{List}\ a \to \texttt{List}\ a$
$\texttt{append} : \forall a.\texttt{List}\ a \to \texttt{List}\ a \to \texttt{List}\ a$
$\texttt{nil} : \forall a.\texttt{List}\ a$
$\texttt{cons} : \forall a.a \to \texttt{List}\ a \to \texttt{List}\ a$
$\texttt{map} : \forall a\,b.(a \to b) \to \texttt{List}\ a \to \texttt{List}\ b$
$\texttt{single} : \forall a.a \to \texttt{List}\ a$
$\texttt{length} : \forall a.\texttt{List}\ a \to \texttt{Int}$
$\texttt{compose} : \forall a\,b\,c.(b \to c) \to (a \to b) \to a \to c$
$\texttt{runST} : \forall a.(\forall s.\texttt{ST}\ s\ a) \to a$
$\texttt{app} : \forall a\,b.(a \to b) \to a \to b$
$\texttt{pair} : \forall a\,b.a \to b \to a \times b$
$\texttt{argST} : \forall s.\texttt{ST}\ s\ \texttt{Int}$
$\texttt{revapp} : \forall a\,b.a \to (a \to b) \to b$
$\texttt{ChurchList}\ a \doteq \forall b.b \to (a \to b \to b) \to b$
$\texttt{id} : \forall a.a \to a$
$\texttt{auto} : (\forall a.a \to a) \to (\forall a.a \to a)$
$\texttt{churchNil} : \forall a.\texttt{ChurchList}\ a$
$\texttt{ids} : [\forall a.a \to a]$
$\texttt{auto}' : \forall b.(\forall a.a \to a) \to (b \to b)$
$\texttt{churchIds} : \texttt{ChurchList}\ (\forall a.a \to a)$
$\texttt{inc} : \texttt{Int} \to \texttt{Int}$
$\texttt{poly} : (\forall a.a \to a) \to \texttt{Int} \times \texttt{Bool}$
$\texttt{churchHead} : \forall a.\texttt{ChurchList}\ a \to a$
$\texttt{choose} : \forall a.a \to a \to a$
$\texttt{polys} : \texttt{List}\ (\forall a.a \to a) \to \texttt{Int} \times \texttt{Bool}$

Figure 3. Type signatures for functions used in the comparison.