The Virtues of Semi-Explicit Polymorphism

Frank Emrich
The University of Edinburgh, UK
frank.emrich@ed.ac.uk

Sam Lindley
Heriot-Watt University, UK
s.lindley@hw.ac.uk

Jan Stolarek
The University of Edinburgh, UK
jan.stolarek@ed.ac.uk

Abstract

The usual declarative presentation of ML allows implicit generalisation and instantiation anywhere in a program. We consider a mild variant, Explicit ML, with explicit syntax for generalisation and instantiation. The familiar implicit ML syntax may be recovered by way of syntactic sugar for variables and let-bindings.

FreezeML is a small extension of ML providing first-class polymorphism and sound and complete type inference of principal types, whose typing rules are not declarative.

We show that Explicit ML extends naturally to Explicit FreezeML, a declarative presentation of an explicit variant of FreezeML. The familiar implicit FreezeML syntax may be recovered by way of syntactic sugar. Explicit FreezeML is a conservative extension of both Explicit ML and System F.

1 Introduction

The design of ML is motivated by a desire to write polymorphic programs without laboriously spelling out details of type abstraction and type application. A remarkable feature of ML is that, due to its restricted form of polymorphism, it is unnecessary to write any polymorphism, or indeed any types, at all. The usual declarative presentation of ML [2] exploits this property by not even providing syntax to mark where generalisation and instantiation occur. The usual syntax-directed presentation of ML [1] takes advantage of the fact that it is sufficient to only generalise let-bindings and only (and always) instantiate variables.

As ML programmers we, the authors, prefer the determinism of the syntax-directed presentation and would argue that it is closer to the intuitive model we use in practice when writing and reasoning about ML programs. However, the syntax-directed presentation is non-orthogonal (not declarative) exactly because it fuses generalisation with let-binding and instantiation with variables. Simply by adding explicit syntax for generalisation and instantiation, we obtain a declarative and syntax-directed language, *Explicit ML*, in which the features are orthogonal, and which enjoys the determinism of the usual syntax-directed presentation. Moreover, we may recover the usual implicit version of ML as syntactic sugar.

Explicit ML does not change the expressive power of the language, and on the face of it may seem like a superficial conceptual improvement over implicit ML. However, as we shall see, where it really shines is when we extend ML with first-class polymorphism.

The *prenex polymorphism* of ML only allows top-level quantifiers and only allows quantifiers to be instantiated with monomorphic types. *FreezeML* [4] is a small extension of ML providing first-class polymorphism and sound and complete type inference of principal types. It is part of a large design space of systems bridging the gap between tractable type inference and first-class polymorphism [5–9, 11, 13–17]. FreezeML adds optional type annotations on bound variables and a construct for *freezing* variables, preventing them from being implicitly instantiated. Whilst the previous formulation of FreezeML is not declarative, we introduce *Explicit FreezeML*, a natural extension of Explicit ML, which is both declarative and syntax-directed. We may recover FreezeML as syntactic sugar for Explicit FreezeML.

We distinguish three forms of polymorphism.

implicit	implicit generalisation + instantiation
semi-explicit	explicit generalisation + instantiation
explicit	type abstraction + type application

Prior systems with semi-explicit polymorphism include IFX [11], Poly-ML [5], and QML [13]. They distinguish ML-like type schemes and System F-style explicit polymorphism, whereas (Explicit) FreezeML has only System F types.

The perspective we take in this work is that Explicit ML (or Explicit FreezeML) *is* the programming language, and ML (or FreezeML) is merely syntactic sugar. Figure 1 illustrates the path from syntactic sugar (first column) to programming language (second column) to core language (third column).

The rest of this extended abstract outlines the design of Explicit ML and Explicit FreezeML, desugaring rules, and a succinct equational theory that dictates elaboration to System F. Full details appear in the appendix.

2 Explicit ML

We let S, T range over monomorphic types and E, F range over type schemes. Typing judgements have the form $\Delta; \Gamma \vdash M : E$, stating that term M has type scheme E in type context Δ (a sequence of type variables ranged over by a, b) and term context Γ. (Traditional presentations of ML often elide which type variables Δ are in scope; we prefer to track these explicitly.)

Generalisation. In ML, implicit generalisation is introduced by the following rule.

$$\textsc{I-Gen-Lax}\quad \frac{\Delta, \Delta'; \Gamma \vdash M : S}{\Delta; \Gamma \vdash M : \forall \Delta'.S}$$

It allows terms to be given arbitrarily general types. For instance, the generalised identity function $\lambda x.x$ may be typed

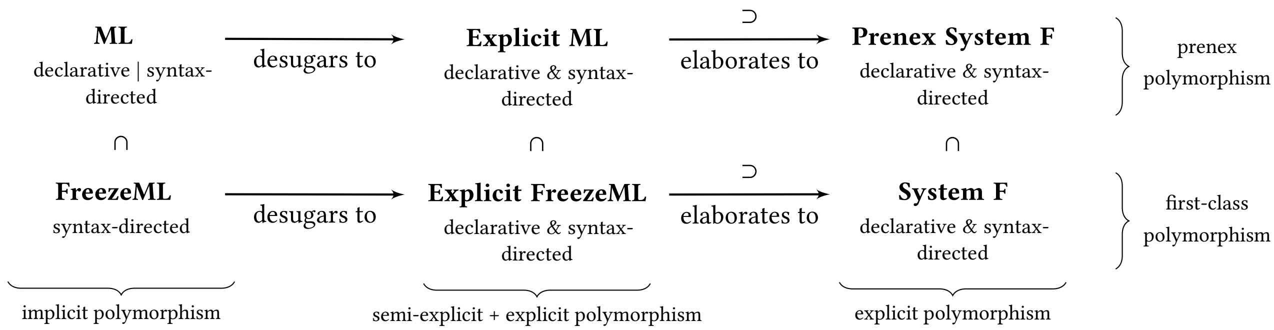

Figure 1. Desugaring and Elaboration of ML and FreezeML

as $\mathsf{Int} \to \mathsf{Int}$, as $\forall a.a \to a$, as $\forall a\, b.(a \to b) \to (a \to b)$, or as infinitely many other types. As it will become necessary later, we adopt a stricter notion of generalisation.

$$\textsc{I-Gen}\quad \frac{\Delta, \Delta'; \Gamma \vdash M : S \qquad \mathsf{principal}(\Delta, \Gamma, M, \Delta', S)}{\Delta; \Gamma \vdash M : \forall \Delta'.S}$$

The principal constraint (Appendix E.2) ensures that generalisation yields the unique most general type. For instance, the generalised identity function $\lambda x.x$ may now only be typed as $\forall a.a \to a$. Explicit ML adopts a variant of I-Gen in which generalisation is explicit in the syntax of terms.

$$\textsc{Gen}\quad \frac{\Delta, \Delta'; \Gamma \vdash M : S \qquad \mathsf{principal}(\Delta, \Gamma, M, \Delta', S)}{\Delta; \Gamma \vdash \Lambda\bullet.M : \forall \Delta'.S}$$

Instantiation. The implicit instantiation rule of ML, substitutes monomorphic types for the body of a term.

$$\textsc{I-Inst}\quad \frac{\Delta; \Gamma \vdash M : \forall \Delta'.S \qquad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M : \sigma(S)}$$

The judgement $\Delta \vdash \sigma : \Delta' \Rightarrow \Delta''$ defines a type instantiation σ mapping type variables in (Δ, Δ') to types with free type variables in (Δ, Δ''), such that $\sigma(a) = a$ for every $a \in \Delta$.

Explicit ML adopts a variation of I-Inst in which generalisation is explicit in the syntax of terms.

$$\textsc{Inst}\quad \frac{\Delta; \Gamma \vdash M : \forall \Delta'.S \qquad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M\bullet : \sigma(S)}$$

Variables and let-binding. We write variables as $\lceil x \rceil$ and let-binding as $\mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N$. We say that such variables are *frozen* as they are not implicitly instantiated. Similarly, we say that such let-bindings are *frozen* as they do not implicitly generalise M.

We now define implicit instantiation of variables and implicit generalisation of let-bindings as syntactic sugar.

$$\begin{array}{rcl} x & \equiv & \lceil x \rceil\bullet \\ \mathbf{let}\ x = M\ \mathbf{in}\ N & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda\bullet.M\ \mathbf{in}\ N \end{array}$$

2.1 Explicit Polymorphism

In addition to the semi-explicit polymorphism we have already seen, we also include fully explicit polymorphism in Explicit ML. This requires a little care. Consider the Prenex System F term $\Lambda a.\lambda x.x$. It is not immediately clear whether this term should have principal type $\forall a.a \to a$ or $\forall a\, b.b \to b$. Exactly the same problem occurs with the term: $\Lambda a.\mathsf{id}$ where $\mathsf{id} : \forall a.a \to a$.

SML [10] resolves the issue by, in both cases selecting $\forall a\, b.b \to b$, supporting explicit type abstraction, but carefully separating type variables that are provided by the programmer from those that are inferred, and not allowing the former to appear in inferred types.

We adopt an approach that avoids any special treatment of type variables but still ensures that the body of a type abstraction has a unique typing. We do so by dividing the syntax of Explicit ML terms into two classes.

$$\begin{array}{lll} \mathsf{ITerm} \ni & & \\ I, J ::= & \lceil x \rceil & \\ & \mid\ \lambda(x : S).I \mid I\, N & \\ & \mid\ \Lambda a.I \mid I\, S & \\ & \mid\ \mathbf{let}\ \lceil x \rceil = I\ \mathbf{in}\ J & \\ & & \\ & \mid\ \Lambda\bullet.M & \end{array} \qquad \begin{array}{lll} \mathsf{MTerm} \ni & & \\ M, N ::= & \lceil x \rceil & \\ & \mid\ \lambda(x : S).M \mid M\, N & \\ & \mid\ \Lambda a.I \mid M\, S & \\ & \mid\ \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N & \\ & \mid\ \lambda x.M & \\ & \mid\ \Lambda\bullet.M & \\ & \mid\ M\bullet & \end{array}$$

The ITerm class consists of Prenex System F extended with (frozen, i.e., non-generalising) let-binding and generalisation. The body of a generalisation need not be an ITerm as generalisation always yields the unique most general type. Similarly, the argument of a function application need not be an ITerm as the type of a function uniquely determines its return type. The MTerm class adds unannotated lambdas and implicit instantiation, these being the only two sources of non-determinism in type inference.

Explicit ML subsumes both Prenex System F and ML: the former directly and the latter via syntactic sugar.

3 Explicit FreezeML

The extension of Explicit ML to Explicit FreezeML is modest. Types may now be fully polymorphic. We let A, B range over System F types. Some care must be taken to manage the separation between monomorphic and polymorphic types. To control where polymorphic instantiation takes place Explicit FreezeML adds a third class of terms.

$$
\begin{array}{rcl}
\mathsf{ITerm} \ni I, J & ::= & \lceil x \rceil \\
& \mid & \lambda(x : A).I \mid I\,Q \\
& \mid & \Lambda a.I \mid I\,A \\
& \mid & \mathbf{let}\ \lceil x \rceil = I\ \mathbf{in}\ J \\
& \mid & \Lambda\bullet.P
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{MTerm} \ni & \mathsf{PTerm} \ni \\
\quad M, N ::= \lceil x \rceil & \quad P, Q ::= \lceil x \rceil \\
\qquad \mid \lambda(x : A).M \mid M\,Q & \qquad \mid \lambda(x : A).P \mid P\,Q \\
\qquad \mid \Lambda a.I \mid M\,A & \qquad \mid \Lambda a.I \mid P\,A \\
\qquad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N & \qquad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q \\
\qquad \mid \lambda x.M & \qquad \mid \lambda x.P \\
\qquad \mid \Lambda\bullet.P & \qquad \mid \Lambda\bullet.P \\
\qquad \mid M\bullet & \qquad \mid P\bullet \\
& \qquad \mid P\star
\end{array}
$$

The PTerm class extends MTerm with a polymorphic instantiation operator $P\star$. The key place where it is important to restrict terms to use monomorphic instantiation is in let-bindings. This restriction prevents "guessing polymorphism", keeping type inference tractable [12, 18]. For the same reason, the typing rule for unannotated lambda abstractions is restricted to monomorphic argument types. The Explicit FreezeML typing judgement has the form $\Delta; \Gamma \vdash P : A$.

We now define the implicit instantiation of variables and implicit generalisation of let-bindings as syntactic sugar.

$$
\begin{array}{rcl}
x & \equiv & \lceil x \rceil\star \\
\mathbf{let}\ x = P\ \mathbf{in}\ Q & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda\bullet.P\ \mathbf{in}\ Q
\end{array}
$$

Moreover, using intermediate syntactic sugar for type-annotated terms and in turn type-annotated generalisation, we define the type-annotated variant of generalising let from FreezeML as syntactic sugar.

$$
\begin{array}{rcl}
(P : A) & \equiv & (\lambda(x : A).\lceil x \rceil)\,P \\
(\Lambda\bullet.P : \forall\Delta.G) & \equiv & \Lambda\Delta.(P : G) \\
\mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q & \equiv & (\lambda(x : A).Q)\,(\Lambda\bullet.P : A)
\end{array}
$$

Here G ranges over *guarded types*, that is, types whose outermost type constructor is not $\forall$. We also define syntactic sugar for non-generalising variants of let in which the let-binding is not syntactically restricted to be an MTerm.

$$
\begin{array}{rcl}
\mathbf{let}'\ x = P\ \mathbf{in}\ Q & \equiv & \mathbf{let}\ \lceil x \rceil = (\Lambda\bullet.P)\bullet\ \mathbf{in}\ Q \\
\mathbf{let}'\ (x : A) = P\ \mathbf{in}\ Q & \equiv & (\lambda(x : A).Q)\,P
\end{array}
$$

In the unannotated case the term $(\Lambda\bullet.P)\bullet$ has the effect of ensuring that all instantiations inside P are monomorphic. We can now implement the value restriction [19] by deciding whether or not to generalise a let-bound term depending on whether it is a syntactic value or not (Appendix G).

Explicit FreezeML subsumes both System F and FreezeML: the former directly and the latter via syntactic sugar.

The type inference algorithm for Explicit FreezeML is a minor adaptation of the one for FreezeML [4], which is itself a routine extension of algorithm W [2].

Equational Reasoning. The equivalence $P \simeq Q$ on terms P and Q is defined only when P and Q have the same type in the same context (i.e., $\Delta; \Gamma \vdash P : A$ and $\Delta; \Gamma \vdash Q : A$). The following rules are the usual β and η-rules of System F.

$$
\begin{array}{llcl}
\beta\text{-rules} & (\lambda(x : A).P)\,Q & \simeq & P[Q/\lceil x \rceil] \\
& (\Lambda a.I)\,A & \simeq & I[A/a] \\
\eta\text{-rules} & \lambda(x : A).P\,\lceil x \rceil & \simeq & P \\
& \Lambda a.I\,a & \simeq & I
\end{array}
$$

The following rules elaborate the additional constructs of Explicit FreezeML into plain System F terms.

$$
\begin{array}{lcl}
\mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q & \simeq & (\lambda(x : A).Q)\,M \\
\lambda x.P & \simeq & \lambda(x : S).P \\
\Lambda\bullet.I & \simeq & \Lambda\Delta.I \\
P\bullet & \simeq & P\,S_1\ \ldots\ S_n \\
P\star & \simeq & P\,A_1\ \ldots\ A_n
\end{array}
$$

Let bindings and unannotated lambdas are expressible using type-annotated lambda abstractions. The last three rules witness the correspondence between generalisation and type abstraction and between instantiation and type application. The third rule applies only once the body of a generalisation has been elaborated. The translation in Appendix E.4 lifts the elaboration rules to a translation on derivations and in so doing proves that we can systematically apply them left-to-right to elaborate to System F.

4 Conclusions and Future Work

FreezeML is a pragmatic extension of ML with first-class polymorphism. In Explicit FreezeML, by making generalisation and instantiation explicit, we have obtained a declarative variant of FreezeML. More ad hoc aspects of FreezeML are accounted for via syntactic sugar on top of Explicit FreezeML.

More sophisticated approaches to first-class polymorphism use heuristics [8, 14, 15] to avoid explicitly marking generalisation and instantiation. We plan to investigate the extent to which we can capture such heuristics via syntactic sugar or lightweight typing extensions on top of Explicit FreezeML. We also plan to extend Explicit FreezeML to support Fω and to adapt Explicit FreezeML to account for features such as typing constraints and bidirectional typing.

Quite apart from first-class polymorphism, we believe that ad hoc conveniences such as implicit generalisation and instantiation are best defined as syntactic sugar. The benefits to designing orthogonal languages with syntax-directed typing rules are both conceptual and practical.

References

[1] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. 1986. A Simple Applicative Language: Mini-ML. In *LISP and Functional Programming*. ACM, 13–27.

[2] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212.

[3] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2019. *FreezeML: Complete and Easy Type Inference for First-Class Polymorphism (extended version)*. Technical Report. arXiv:2004.00396.

[4] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *PLDI*. ACM. https://arxiv.org/abs/2004.00396 To appear.

[5] Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169.

[6] Didier Le Botlan and Didier Rémy. 2003. ML^{F}: raising ML to the power of System F. In *ICFP*. ACM, 27–38.

[7] Daan Leijen. 2007. A type directed translation of MLF to system F. In *ICFP*. ACM, 111–122.

[8] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *ICFP*. ACM, 283–294.

[9] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *POPL*. ACM, 66–77.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

[11] James O'Toole and David K. Gifford. 1989. Type Reconstruction with First-Class Polymorphic Values. In *PLDI*. ACM, 207–217. https://doi.org/10.1145/73141.74836

[12] Frank Pfenning. 1993. On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundam. Inform.* 19, 1/2 (1993), 185–199.

[13] Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-class Polymorphism for ML. In *ML*. ACM, 3–14.

[14] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. In *ICFP*. ACM. Conditionally accepted.

[15] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *PLDI*. ACM, 783–796.

[16] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*. ACM, 251–262.

[17] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *ICFP*. ACM, 295–306.

[18] J. B. Wells. 1994. Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable. In *LICS*. IEEE Computer Society, 176–185.

[19] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and Symbolic Computation* 8, 4 (1995), 343–355.

A Prenex System F

A.1 Syntax of Prenex System F

$$
\begin{array}{ll}
\text{Type Variables} & a, b, c \\
\text{Type Constructors} & D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots \\
\text{Monotypes} & S, T ::= a \mid D\,\overline{S} \\
\text{Type Schemes} & E, F ::= \forall \overline{a}.S \\
\text{Type Contexts} & \Delta ::= \cdot \mid \Delta, a \\
\text{Term Contexts} & \Gamma ::= \cdot \mid \Gamma, x : E \\
\text{Term Variables} & x, y, z \\
\text{Terms} & M, N ::= \lceil x \rceil \mid \lambda(x : S).M \mid M\,N \mid \Lambda a.M \mid M\,S
\end{array}
$$

A.2 Type System of Prenex System F

Well-formed monotypes / type schemes. $\boxed{\Delta \vdash E \text{ ok}}$

$$
\frac{a \in \Delta}{\Delta \vdash a \text{ ok}}
\qquad
\frac{\mathsf{arity}(D) = n \quad \Delta \vdash E_1 \text{ ok} \ \cdots \ \Delta \vdash E_n \text{ ok}}{\Delta \vdash D\,\overline{E} \text{ ok}}
$$

Typing. $\boxed{\Delta; \Gamma \vdash M : E}$

$$
\textsc{Var}\ \frac{x : E \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : E}
\qquad
\textsc{App}\ \frac{\Delta; \Gamma \vdash M : S \to T \quad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M\,N : T}
\qquad
\textsc{TyLam}\ \frac{\Delta, a; \Gamma \vdash M : E}{\Delta; \Gamma \vdash \Lambda a.M : \forall a.E}
$$

$$
\textsc{Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda(x : S).M : S \to T}
\qquad
\textsc{TyApp}\ \frac{\Delta; \Gamma \vdash M : \forall a.E}{\Delta; \Gamma \vdash M\,S : E[S/a]}
$$

A.3 Equational Rules of Prenex System F

As in Section 3 the equivalence $M \simeq N$ on terms M and N is defined only when M and N have the same type in the same context (i.e., $\Delta; \Gamma \vdash M : E$ and $\Delta; \Gamma \vdash N : E$).

$$
\begin{array}{lll}
\beta\text{-rules} & (\lambda(x : S).M)\,N & \simeq\ M[M/\lceil x \rceil] \\
& (\Lambda a.M)\,S & \simeq\ M[S/a] \\
\\
\eta\text{-rules} & \lambda(x : S).M\,\lceil x \rceil & \simeq\ M \\
& \Lambda a.M\,a & \simeq\ M
\end{array}
$$

B Explicit ML

B.1 Syntax of Explicit ML

Types.

$$
\begin{array}{ll}
\text{Type Variables} & a, b, c \\
\text{Type Constructors} & D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots \\
\text{Monotypes} & S, T ::= a \mid D\,\overline{S} \\
\text{Type Schemes} & E, F ::= \forall \overline{a}.S \\
\text{Type Instantiation} & \sigma ::= \emptyset \mid \sigma[a \mapsto S] \\
\text{Type Contexts} & \Delta ::= \cdot \mid \Delta, a \\
\text{Term Contexts} & \Gamma ::= \cdot \mid \Gamma, x : E
\end{array}
$$

Terms.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{ITerm} \ni I, J & ::= & \lceil x \rceil & \mathsf{MTerm} \ni M, N & ::= & \lceil x \rceil \\
& \mid & \lambda(x:S).I \mid I\,N & & \mid & \lambda(x:S).M \mid M\,N \\
& \mid & \Lambda a.I \mid I\,S & & \mid & \Lambda a.I \mid M\,S \\
& \mid & \mathbf{let}\ \lceil x \rceil = I\ \mathbf{in}\ J & & \mid & \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N \\
& & & & \mid & \lambda x.M \\
& \mid & \Lambda\bullet.M & & \mid & \Lambda\bullet.M \\
& & & & \mid & M\bullet
\end{array}
$$

B.2 Type System of Explicit ML

Well-formed monotypes / type schemes. $\boxed{\Delta \vdash E\ \mathit{ok}}$

$$
\frac{a \in \Delta}{\Delta \vdash a\ \mathsf{ok}}
\qquad
\frac{\mathsf{arity}(D) = n \qquad \Delta \vdash E_1\ \mathsf{ok} \qquad \cdots \qquad \Delta \vdash E_n\ \mathsf{ok}}{\Delta \vdash D\ \overline{E}\ \mathsf{ok}}
\qquad
\frac{\Delta, a \vdash E\ \mathsf{ok}}{\Delta \vdash \forall a.E\ \mathsf{ok}}
$$

Instantiation. $\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow \Delta''}$

$$
\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Delta'}
\qquad
\frac{\Delta \vdash \sigma : \Delta' \Rightarrow \Delta'' \qquad \Delta, \Delta'' \vdash S\ \mathsf{ok}}{\Delta \vdash \sigma[a \mapsto S] : (\Delta', a) \Rightarrow \Delta''}
$$

Principality. $\boxed{\mathsf{principal}(\Delta, \Gamma, M, \Delta', E)}$

$$
\begin{array}{l}
\mathsf{principal}(\Delta, \Gamma, M, \Delta', E') = \\
\quad \Delta' = \mathsf{ftv}(E') - \Delta \ \text{ and } \ \Delta, \Delta'; \Gamma \vdash M : E' \ \text{ and} \\
\quad (\text{for all } \Delta'', E'' \mid \text{if } \Delta'' = \mathsf{ftv}(E'') - \Delta \text{ and} \\
\qquad\qquad \Delta, \Delta''; \Gamma \vdash M : E'' \\
\qquad\qquad \text{then there exists } \sigma \text{ such that} \\
\qquad\qquad \Delta \vdash \sigma : \Delta' \Rightarrow \Delta'' \text{ and } \sigma(E') = E'')
\end{array}
$$

Typing. $\boxed{\Delta; \Gamma \vdash M : E}$

$$
\textsc{Var}\ \frac{x : E \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : E}
\qquad
\textsc{Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda(x:S).M : S \to T}
\qquad
\textsc{App}\ \frac{\Delta; \Gamma \vdash M : S \to T \quad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M\,N : T}
$$

$$
\textsc{TyLam}\ \frac{\Delta, a; \Gamma \vdash I : E}{\Delta; \Gamma \vdash \Lambda a.I : \forall a.E}
\qquad
\textsc{TyApp}\ \frac{\Delta; \Gamma \vdash M : \forall a.E}{\Delta; \Gamma \vdash M\,S : E[S/a]}
$$

$$
\textsc{Let}\ \frac{\Delta; \Gamma \vdash M : E \qquad \Delta; \Gamma, x : E \vdash N : F}{\Delta; \Gamma \vdash \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N : F}
\qquad
\textsc{U-Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \to T}
$$

$$
\textsc{Gen}\ \frac{\Delta, \Delta'; \Gamma \vdash M : E \qquad \mathsf{principal}(\Delta, \Gamma, M, \Delta', E)}{\Delta; \Gamma \vdash \Lambda\bullet.M : \forall\Delta'.E}
\qquad
\textsc{MonoInst}\ \frac{\Delta; \Gamma \vdash M : \forall\Delta'.S \quad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M\bullet : \sigma(S)}
$$

B.3 Equational Rules of Explicit ML

As in Section 3 the equivalence $M \simeq N$ on terms M and N is defined only when M and N have the same type in the same context (i.e., $\Delta; \Gamma \vdash M : E$ and $\Delta; \Gamma \vdash N : E$).

$$\begin{array}{lll}
\beta\text{-rules} & (\lambda(x:S).M)\,N & \simeq M[N/\lceil x\rceil] \\
 & (\Lambda a.I)\,S & \simeq I[S/a] \\[1em]
\eta\text{-rules} & \lambda(x:S).M\,\lceil x\rceil & \simeq M \\
 & \Lambda a.I\,a & \simeq I \\[1em]
\text{elaboration rules} & \textbf{let}\ \lceil x\rceil = M\ \textbf{in}\ N & \simeq (\lambda(x:S).N)\,M \\
 & \lambda x.M & \simeq \lambda(x:S).M \\
 & \Lambda\bullet.I & \simeq \Lambda\Delta.I \\
 & M\bullet & \simeq M\,S_1\,\ldots\,S_n
\end{array}$$

B.4 Translation from Explicit ML to Prenex System F

$$\left[\!\!\left[\frac{x:E\in\Gamma}{\Delta;\Gamma\vdash\lceil x\rceil:E}\right]\!\!\right]=x \qquad \left[\!\!\left[\frac{\Delta;\Gamma,x:A\vdash M:T}{\Delta;\Gamma\vdash\lambda(x:S).M:S\to T}\right]\!\!\right]=\lambda(x:S).[\![M]\!] \qquad \left[\!\!\left[\frac{\Delta;\Gamma\vdash M:S\to T \qquad \Delta;\Gamma\vdash N:S}{\Delta;\Gamma\vdash M\,N:T}\right]\!\!\right]=[\![M]\!]\,[\![N]\!]$$

$$\left[\!\!\left[\frac{\Delta,a;\Gamma\vdash I:E}{\Delta;\Gamma\vdash\Lambda a.I:\forall a.E}\right]\!\!\right]=\Lambda a.[\![I]\!] \qquad \left[\!\!\left[\frac{\Delta;\Gamma\vdash M:\forall a.E}{\Delta;\Gamma\vdash M\,S:E[S/a]}\right]\!\!\right]=[\![M]\!]\,S$$

$$\left[\!\!\left[\frac{\Delta;\Gamma\vdash M:E \qquad \Delta;\Gamma,x:E\vdash N:F}{\Delta;\Gamma\vdash\textbf{let}\ \lceil x\rceil=M\ \textbf{in}\ N:F}\right]\!\!\right]=(\lambda(x:E).\,[\![N]\!])\,[\![M]\!] \qquad \left[\!\!\left[\frac{\Delta;\Gamma,x:S\vdash M:T}{\Delta;\Gamma\vdash\lambda x.M:S\to T}\right]\!\!\right]=\lambda(x:S).[\![M]\!]$$

$$\left[\!\!\left[\frac{\Delta,\Delta';\Gamma\vdash M:E \qquad \mathsf{principal}(\Delta,\Gamma,M,\Delta',E)}{\Delta;\Gamma\vdash\Lambda\bullet.M:\forall\Delta'.E}\right]\!\!\right]=\Lambda\Delta'.[\![M]\!] \qquad \left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma\vdash M:\forall\Delta'.S\\ \Delta\vdash\sigma:\Delta'\Rightarrow\cdot\end{array}}{\Delta;\Gamma\vdash M\bullet:\sigma(S)}\right]\!\!\right]=[\![M]\!]\,\sigma(\Delta')$$

C ML

C.1 Syntax of ML

Types.

$$\begin{array}{ll}
\text{Type Variables} & a,b,c \\
\text{Type Constructors} & D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots \\
\text{Monotypes} & S,T ::= a \mid D\,\overline{S} \\
\text{Type Schemes} & E,F ::= \forall\overline{a}.S \\
\text{Type Instantiation} & \sigma ::= \emptyset \mid \sigma[a\mapsto S] \\
\text{Type Contexts} & \Delta ::= \cdot \mid \Delta,a \\
\text{Term Contexts} & \Gamma ::= \cdot \mid \Gamma,x:E
\end{array}$$

Terms.

$$\begin{array}{rl}
M,N ::= & x \\
\mid & \lambda x.M \mid M\,N \\
\mid & \textbf{let}\ x=M\ \textbf{in}\ N
\end{array}$$

C.2 Type System of ML

Well-formed monotypes / type schemes. $\boxed{\Delta\vdash E\ ok}$

$$\frac{a\in\Delta}{\Delta\vdash a\ \mathsf{ok}} \qquad \frac{\mathsf{arity}(D)=n \qquad \Delta\vdash S_1\ \mathsf{ok} \quad \cdots \quad \Delta\vdash S_n\ \mathsf{ok}}{\Delta\vdash D\,\overline{S}\ \mathsf{ok}} \qquad \frac{\Delta,a\vdash E\ \mathsf{ok}}{\Delta\vdash\forall a.E\ \mathsf{ok}}$$

Instantiation. $\boxed{\Delta\vdash\sigma:\Delta'\Rightarrow\Delta''}$

$$\frac{}{\Delta\vdash\emptyset:\cdot\Rightarrow\Delta'} \qquad \frac{\Delta\vdash\sigma:\Delta'\Rightarrow\Delta'' \qquad \Delta,\Delta''\vdash S\ \mathsf{ok}}{\Delta\vdash\sigma[a\mapsto S]:(\Delta',a)\Rightarrow\Delta''}$$

Syntax-directed Typing Judgement. $\boxed{\Delta; \Gamma \vdash M : S}$

$$\textsc{VarInst}\ \frac{x : \forall \Delta'.S \in \Gamma \qquad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash x : \sigma(S)} \qquad \textsc{U-Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \to T}$$

$$\textsc{App}\ \frac{\Delta; \Gamma \vdash M : S \to T \qquad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M\, N : T} \qquad \textsc{LetGen}\ \frac{\Delta' = \mathsf{ftv}(S) - \Delta \qquad \Delta, \Delta'; \Gamma \vdash M : S \qquad E = \forall \Delta'.S \qquad \Delta; \Gamma, x : E \vdash N : T}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : T}$$

Declarative Typing Judgement. $\boxed{\Delta; \Gamma \vdash M : E}$

$$\textsc{Var}\ \frac{x : E \in \Gamma}{\Delta; \Gamma \vdash x : E} \qquad \textsc{U-Lam}\ \frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \to T} \qquad \textsc{App}\ \frac{\Delta; \Gamma \vdash M : S \to T \qquad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M\, N : T}$$

$$\textsc{Let}\ \frac{\Delta; \Gamma \vdash M : E \qquad \Delta; \Gamma, x : E \vdash N : F}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : F} \qquad \textsc{I-Gen-Lax}\ \frac{\Delta, \Delta'; \Gamma \vdash M : S}{\Delta; \Gamma \vdash M : \forall \Delta'.S} \qquad \textsc{I-Inst}\ \frac{\Delta; \Gamma \vdash M : \forall \Delta'.S \qquad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M : \sigma(E)}$$

C.3 Desugaring from ML to Explicit ML

$$\begin{array}{rcl} x & \equiv & \lceil x \rceil \bullet \\ \mathbf{let}\ x = M\ \mathbf{in}\ N & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda \bullet.M\ \mathbf{in}\ N \end{array}$$

D System F

D.1 Syntax of System F

Type Variables	a, b, c
Type Constructors	$D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots$
Types	$A, B ::= a \mid D\,\overline{A} \mid \forall a.A$
Type Contexts	$\Delta ::= \cdot \mid \Delta, a$
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Term Variables	x, y, z
Terms	$M, N ::= \lceil x \rceil \mid \lambda(x : A).M \mid M\, N \mid \Lambda a.M \mid M\, A$

D.2 Type System of System F

Well-formed types. $\boxed{\Delta \vdash A\ \mathsf{ok}}$

$$\frac{a \in \Delta}{\Delta \vdash a\ \mathsf{ok}} \qquad \frac{\mathsf{arity}(D) = n \qquad \Delta \vdash A_1\ \mathsf{ok} \cdots \Delta \vdash A_n\ \mathsf{ok}}{\Delta \vdash D\,\overline{A}\ \mathsf{ok}} \qquad \frac{\Delta, a \vdash A\ \mathsf{ok}}{\Delta \vdash \forall a.A\ \mathsf{ok}}$$

Typing. $\boxed{\Delta; \Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A} \qquad \textsc{App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\, N : B} \qquad \textsc{TyLam}\ \frac{\Delta, a; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \Lambda a.M : \forall a.A}$$

$$\textsc{Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda(x : A).M : A \to B} \qquad \textsc{TyApp}\ \frac{\Delta; \Gamma \vdash M : \forall a.B}{\Delta; \Gamma \vdash M\, A : E[A/a]}$$

D.3 Equational Rules of System F

As in Section 3 the equivalence $M \simeq N$ on terms M and N is defined only when M and N have the same type in the same context (i.e., $\Delta; \Gamma \vdash M : A$ and $\Delta; \Gamma \vdash N : A$).

$$
\begin{array}{lrcl}
\beta\text{-rules} & (\lambda(x:A).M)\,N & \simeq & M[N/\lceil x \rceil] \\
 & (\Lambda a.M)\,A & \simeq & M[A/a] \\[1em]
\eta\text{-rules} & \lambda(x:A).M\,\lceil x \rceil & \simeq & M \\
 & \Lambda a.M\,a & \simeq & M
\end{array}
$$

E Explicit FreezeML

E.1 Syntax of Explicit FreezeML

Types.

$$
\begin{array}{ll}
\text{Type Variables} & a, b, c \\
\text{Type Constructors} & D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots \\
\text{Types} & A, B ::= a \mid D\,\overline{A} \mid \forall a.A \\
\text{Monotypes} & S, T ::= a \mid D\,\overline{S} \\
\text{Guarded Types} & G ::= a \mid D\,\overline{A} \\
\text{Monomorphic Instantiation} & \sigma ::= \emptyset \mid \sigma[a \mapsto S] \\
\text{Polymorphic Instantiation} & \delta ::= \emptyset \mid \delta[a \mapsto A] \\
\text{Type Contexts} & \Delta ::= \cdot \mid \Delta, a \\
\text{Term Contexts} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}
$$

Terms.

$$
\begin{array}{lll}
\mathsf{ITerm} \ni I, J ::= \lceil x \rceil & \mathsf{MTerm} \ni M, N ::= \lceil x \rceil & \mathsf{PTerm} \ni P, Q ::= \lceil x \rceil \\
\quad \mid \lambda(x:A).I \mid I\,Q & \quad \mid \lambda(x:A).M \mid M\,Q & \quad \mid \lambda(x:A).P \mid P\,Q \\
\quad \mid \Lambda a.I \mid I\,A & \quad \mid \Lambda a.I \mid M\,A & \quad \mid \Lambda a.I \mid P\,A \\
\quad \mid \mathbf{let}\ \lceil x \rceil = I\ \mathbf{in}\ J & \quad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N & \quad \mid \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q \\
 & \quad \mid \lambda x.M & \quad \mid \lambda x.P \\
\quad \mid \Lambda \bullet.P & \quad \mid \Lambda \bullet.P & \quad \mid \Lambda \bullet.P \\
 & \quad \mid M \bullet & \quad \mid P \bullet \\
 & & \quad \mid P \star
\end{array}
$$

E.2 Type System of Explicit FreezeML

Well-formed types. $\boxed{\Delta \vdash A\ \mathsf{ok}}$

$$
\frac{a \in \Delta}{\Delta \vdash a\ \mathsf{ok}}
\qquad
\frac{\mathsf{arity}(D) = n \qquad \Delta \vdash A_1\ \mathsf{ok} \qquad \cdots \qquad \Delta \vdash A_n\ \mathsf{ok}}{\Delta \vdash D\,\overline{A}\ \mathsf{ok}}
\qquad
\frac{\Delta, a \vdash A\ \mathsf{ok}}{\Delta \vdash \forall a.A\ \mathsf{ok}}
$$

Monomorphic instantiation. $\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \Delta''}$

$$
\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_\bullet \Delta'}
\qquad
\frac{\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \Delta'' \qquad \Delta, \Delta'' \vdash S\ \mathsf{ok}}{\Delta \vdash \sigma[a \mapsto S] : (\Delta', a) \Rightarrow_\bullet \Delta''}
$$

Polymorphic instantiation. $\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta''}$

$$
\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_\star \Delta'}
\qquad
\frac{\Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \qquad \Delta, \Delta'' \vdash A\ \mathsf{ok}}{\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_\star \Delta''}
$$

Principality judgement. $\boxed{\mathrm{principal}(\Delta, \Gamma, P, \Delta', A')}$

$$\begin{array}{l}\mathrm{principal}(\Delta, \Gamma, P, \Delta', A') = \\ \qquad \Delta' = \mathrm{ftv}(A') - \Delta \;\text{ and }\; \Delta, \Delta'; \Gamma \vdash P : A' \;\text{ and} \\ \qquad (\text{for all } \Delta'', A'' \mid \text{if } \Delta'' = \mathrm{ftv}(A'') - \Delta \text{ and} \\ \qquad\qquad\qquad\qquad\quad \Delta, \Delta''; \Gamma \vdash P : A'' \\ \qquad\qquad\qquad\qquad\quad \text{then there exists } \delta \text{ such that} \\ \qquad\qquad\qquad\qquad\quad \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta(A') = A'')\end{array}$$

Typing judgement. $\boxed{\Delta; \Gamma \vdash P : A}$

$$\textsc{Var}\;\frac{x : E \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : E} \qquad \textsc{Lam}\;\frac{\Delta; \Gamma, x : S \vdash P : T}{\Delta; \Gamma \vdash \lambda(x : S).P : S \to T} \qquad \textsc{App}\;\frac{\Delta; \Gamma \vdash P : A \to B \quad \Delta; \Gamma \vdash Q : A}{\Delta; \Gamma \vdash P\,Q : B}$$

$$\textsc{TyLam}\;\frac{\Delta, a; \Gamma \vdash I : E}{\Delta; \Gamma \vdash \Lambda a.I : \forall a.E} \qquad \textsc{TyApp}\;\frac{\Delta; \Gamma \vdash M : \forall a.E}{\Delta; \Gamma \vdash M\,S : E[S/a]}$$

$$\textsc{Let}\;\frac{\Delta; \Gamma \vdash M : E \qquad \Delta; \Gamma, x : E \vdash N : F}{\Delta; \Gamma \vdash \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N : F} \qquad \textsc{U-Lam}\;\frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \to T}$$

$$\textsc{Gen}\;\frac{\Delta, \Delta'; \Gamma \vdash M : E \qquad \mathrm{principal}(\Delta, \Gamma, M, \Delta', E)}{\Delta; \Gamma \vdash \Lambda\bullet.M : \forall\Delta'.E} \qquad \textsc{MonoInst}\;\frac{\Delta; \Gamma \vdash P : \forall\Delta'.S \quad \Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \cdot}{\Delta; \Gamma \vdash P\bullet : \sigma(S)} \qquad \textsc{PolyInst}\;\frac{\Delta; \Gamma \vdash P : \forall\Delta'.A \quad \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot}{\Delta; \Gamma \vdash P\star : \delta(A)}$$

E.3 Equational Rules of Explicit FreezeML

As in Section 3 the equivalence $P \simeq Q$ on terms P and Q is defined only when P and Q have the same type in the same context (i.e., $\Delta; \Gamma \vdash P : A$ and $\Delta; \Gamma \vdash Q : A$).

$$\begin{array}{lrcl}\beta\text{-rules} & (\lambda(x : A).P)\,Q & \simeq & P[Q/\lceil x \rceil] \\ & (\Lambda a.I)\,A & \simeq & I[A/a] \\ \\ \eta\text{-rules} & \lambda(x : A).P\,\lceil x \rceil & \simeq & P \\ & \Lambda a.I\,a & \simeq & I \\ \\ \text{elaboration rules} & \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q & \simeq & (\lambda(x : A).Q)\,M \\ & \lambda x.P & \simeq & \lambda(x : S).P \\ & \Lambda\bullet.I & \simeq & \Lambda\Delta.I \\ & P\bullet & \simeq & P\,S_1\ \ldots\ S_n \\ & P\star & \simeq & P\,A_1\ \ldots\ A_n\end{array}$$

E.4 Translation from Explicit FreezeML to System F

$$\left\llbracket \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A} \right\rrbracket = x \qquad \left\llbracket \frac{\Delta; \Gamma, x : A \vdash P : B}{\Delta; \Gamma \vdash \lambda(x : A).P : A \to B} \right\rrbracket = \lambda(x : A).\llbracket P \rrbracket \qquad \left\llbracket \frac{\Delta; \Gamma \vdash P : A \to B \qquad \Delta; \Gamma \vdash Q : A}{\Delta; \Gamma \vdash P\, Q : B} \right\rrbracket = \llbracket P \rrbracket\, \llbracket Q \rrbracket$$

$$\left\llbracket \frac{\Delta, a; \Gamma \vdash I : A}{\Delta; \Gamma \vdash \Lambda a.I : \forall a.A} \right\rrbracket = \Lambda a.\llbracket I \rrbracket \qquad \left\llbracket \frac{\Delta; \Gamma \vdash P : \forall a.B}{\Delta; \Gamma \vdash P\, A : B[A/a]} \right\rrbracket = \llbracket P \rrbracket\, A$$

$$\left\llbracket \frac{\Delta; \Gamma \vdash M : A \qquad \Delta; \Gamma, x : A \vdash Q : B}{\Delta; \Gamma \vdash \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ Q : B} \right\rrbracket = (\lambda(x : A).\, \llbracket Q \rrbracket)\, \llbracket M \rrbracket \qquad \left\llbracket \frac{\Delta; \Gamma, x : S \vdash P : B}{\Delta; \Gamma \vdash \lambda x.P : S \to B} \right\rrbracket = \lambda(x : S).\llbracket P \rrbracket$$

$$\left\llbracket \frac{\Delta, \Delta'; \Gamma \vdash P : A \qquad \mathsf{principal}(\Delta, \Gamma, P, \Delta', A)}{\Delta; \Gamma \vdash \Lambda \bullet .P : \forall \Delta'.A} \right\rrbracket = \Lambda \Delta'.\llbracket P \rrbracket \qquad \left\llbracket \frac{\begin{array}{c}\Delta; \Gamma \vdash P : \forall \Delta'.G \\ \Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \cdot\end{array}}{\Delta; \Gamma \vdash P\bullet : \sigma(G)} \right\rrbracket = \llbracket P \rrbracket\, \sigma(\Delta')$$

$$\left\llbracket \frac{\begin{array}{c}\Delta; \Gamma \vdash P : \forall \Delta'.G \\ \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot\end{array}}{\Delta; \Gamma \vdash P\star : \delta(G)} \right\rrbracket = \llbracket P \rrbracket\, \delta(\Delta') =$$

F FreezeML

F.1 Syntax of FreezeML

Types.

Type Variables	a, b, c
Type Constructors	$D ::= \mathsf{Int} \mid \mathsf{List} \mid \to \mid \times \mid \ldots$
Types	$A, B ::= a \mid D\,\overline{A} \mid \forall a.A$
Monotypes	$S, T ::= a \mid D\,\overline{S}$
Guarded Types	$G ::= a \mid D\,\overline{A}$
Polymorphic Instantiation	$\delta ::= \emptyset \mid \delta[a \mapsto A]$
Term Variables	x, y, z
Type Contexts	$\Delta ::= \cdot \mid \Delta, a$
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

Terms.

$$\begin{array}{lrl}\text{Terms} & P, Q ::= & x \mid \lceil x \rceil \mid \lambda x.P \\ & \mid & \lambda(x : A).P \mid P\, Q \\ & \mid & \mathbf{let}\ x = P\ \mathbf{in}\ Q \\ & \mid & \mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q\end{array}$$

F.2 Type System of FreezeML

Well-formed types. $\boxed{\Delta \vdash A\ \mathsf{ok}}$

$$\frac{a \in \Delta}{\Delta \vdash a\ \mathsf{ok}} \qquad \frac{\mathsf{arity}(D) = n \qquad \Delta \vdash A_1\ \mathsf{ok} \qquad \cdots \qquad \Delta \vdash A_n\ \mathsf{ok}}{\Delta \vdash D\,\overline{A}\ \mathsf{ok}} \qquad \frac{\Delta, a \vdash A\ \mathsf{ok}}{\Delta \vdash \forall a.A\ \mathsf{ok}}$$

Polymorphic instantiation. $\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''}$

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_{\star} \Delta'} \qquad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta'' \qquad \Delta, \Delta'' \vdash A\ \mathsf{ok}}{\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_{\star} \Delta''}$$

Principality judgement. $\boxed{\mathrm{principal}(\Delta, \Gamma, P, \Delta', A')}$

$$\begin{array}{l}\mathrm{principal}(\Delta, \Gamma, P, \Delta', A') = \\ \quad \Delta' = \mathrm{ftv}(A') - \Delta \ \text{ and } \ \Delta, \Delta'; \Gamma \vdash P : A' \ \text{ and} \\ \quad (\text{for all } \Delta'', A'' \mid \text{if } \Delta'' = \mathrm{ftv}(A'') - \Delta \text{ and} \\ \quad \quad \quad \quad \quad \quad \quad \Delta, \Delta''; \Gamma \vdash P : A'' \\ \quad \quad \quad \quad \quad \quad \quad \text{then there exists } \delta \text{ such that} \\ \quad \quad \quad \quad \quad \quad \quad \ \Delta \vdash \delta : \Delta' \Rightarrow_\star \Delta'' \text{ and } \delta(A') = A'')\end{array}$$

Typing judgement. $\boxed{\Delta; \Gamma \vdash P : A}$

In contrast to Emrich et al. [4], we first present a simplified variant of FreezeML that does not incorporate the value restriction. In Appendix G we describe how to adapt the following to support the value restriction.

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : A} \qquad \textsc{VarInst}\ \frac{\begin{array}{c}x : \forall \Delta'.G \in \Gamma \\ \Delta \vdash \delta : \Delta' \Rightarrow_\star \cdot\end{array}}{\Delta; \Gamma \vdash x : \delta(G)}$$

$$\textsc{U-Lam}\ \frac{\Delta; \Gamma, x : S \vdash P : B}{\Delta; \Gamma \vdash \lambda x.P : S \to B} \qquad \textsc{Lam}\ \frac{\Delta; \Gamma, x : A \vdash P : B}{\Delta; \Gamma \vdash \lambda(x : A).P : A \to B}$$

$$\textsc{App}\ \frac{\Delta; \Gamma \vdash P : A \to B \qquad \Delta; \Gamma \vdash Q : A}{\Delta; \Gamma \vdash P\, Q : B}$$

$$\textsc{LetGen}\ \frac{\begin{array}{c}\Delta' = \mathrm{ftv}(A') - \Delta \qquad A = \forall \Delta'.A' \qquad \Delta, \Delta''; \Gamma \vdash P : A' \qquad \Delta; \Gamma, x : A \vdash Q : B \\ \mathrm{principal}(\Delta, \Gamma, P, \Delta', A')\end{array}}{\Delta; \Gamma \vdash \mathbf{let}\ x = P\ \mathbf{in}\ Q : B}$$

$$\textsc{A-LetGen}\ \frac{A = \forall \Delta'.G \qquad \Delta, \Delta'; \Gamma \vdash P : G \qquad \Delta; \Gamma, x : A \vdash Q : B}{\Delta; \Gamma \vdash \mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q : B}$$

F.3 Desugaring from FreezeML to Explicit FreezeML

$$\begin{array}{rcl}x & \equiv & \lceil x \rceil \star \\ \mathbf{let}\ x = P\ \mathbf{in}\ Q & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda \bullet.P\ \mathbf{in}\ Q \\ (P : A) & \equiv & (\lambda(x : A).\lceil x \rceil)\, P \\ (\Lambda \bullet.P : \forall \Delta.G) & \equiv & \Lambda \Delta.(P : G) \\ \mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q & \equiv & (\lambda(x : A).Q)\ (\Lambda \bullet.P : A)\end{array}$$

G Incorporating the Value Restriction

None of the calculi presented in this work obey the value restriction [19], which is used in ML-like languages to retain type soundness in the presence of side effects (e.g., mutable references). We revisit versions of ML and FreezeML that do obey the value restriction (the latter following Emrich et al. [3]), and show how the desugaring to the corresponding explicit calculus has to be updated to incorporate the value restriction.

For the remaining systems displayed in Figure 1 (Prenex System F, System F, Explicit ML, Explicit FreezeML), incorporating the value restriction it suffices to restrict the body of the type abstraction and generalisation operators to be syntactic values.

G.1 ML

Syntax. We define the grammar of syntactic values as follows.

$$\mathrm{Val} \ni V, W ::= x \mid \lambda x.M \mid \mathbf{let}\ x = V\ \mathbf{in}\ W$$

Typing. We define the following helper function.

$$\mathsf{gen}(\Delta, A, M) = \begin{cases} \mathsf{ftv}(A) - \Delta & \text{if } M \in \mathsf{Val} \\ \cdot & \text{otherwise} \end{cases}$$

We then replace the ML typing rule LetGen of the syntax-directed variant of ML (Appendix C.2) by the following rule.

$$\textsc{LetGen}\quad \frac{\begin{array}{c}\Delta' = \mathsf{gen}(\Delta, S, M) \quad \Delta, \Delta'; \Gamma \vdash M : S \\ E = \forall \Delta'.S \quad \Delta; \Gamma, x : E \vdash N : T\end{array}}{\Delta; \Gamma \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : T}$$

To adapt the declarative presentation, it suffices to limit the rule I-Gen-Lax to syntactic values.

Desugaring to Explicit ML. We replace the desugaring rule for **let** with the following:

$$\begin{array}{lcll} \mathbf{let}\ x = V\ \mathbf{in}\ N & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda\bullet.V\ \mathbf{in}\ N & \\ \mathbf{let}\ x = M\ \mathbf{in}\ N & \equiv & \mathbf{let}\ \lceil x \rceil = M\ \mathbf{in}\ N & \text{if } M \notin \mathsf{Val} \end{array}$$

G.2 FreezeML

Syntax. The grammar is augmented as follows:

$$\begin{array}{lrl} \text{Monomorphic Instantiation} & \sigma ::= & \emptyset \mid \sigma[a \mapsto S] \\ \text{Values} & \mathsf{Val} \ni V, W ::= & x \mid \lceil x \rceil \mid \lambda x.P \mid \lambda(x : A).P \mid \mathbf{let}\ x = V\ \mathbf{in}\ W \mid \mathbf{let}\ (x : A) = V\ \mathbf{in}\ W \\ \text{Guarded Values} & \mathsf{GVal} \ni U ::= & x \mid \lambda x.P \mid \lambda(x{:}A).P \mid \mathbf{let}\ x = V\ \mathbf{in}\ U \mid \mathbf{let}\ (x : A) = V\ \mathbf{in}\ U \end{array}$$

Typing. We define the following helper judgements and functions.

$$\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \Delta''}$$

$$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow_\bullet \Delta'} \qquad \frac{\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \Delta'' \quad \Delta, \Delta'' \vdash S\ \mathsf{ok}}{\Delta \vdash \sigma[a \mapsto S] : (\Delta', a) \Rightarrow_\bullet \Delta''}$$

$$\boxed{(\Delta, \Delta', P, A') \Updownarrow A}$$

$$\frac{P \in \mathsf{GVal}}{(\Delta, \Delta', P, A') \Updownarrow \forall \Delta'.A'} \qquad \frac{\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \cdot \quad P \notin \mathsf{GVal}}{(\Delta, \Delta', P, A') \Updownarrow \sigma(A')}$$

$$\mathsf{gen}(\Delta, A, P) = \begin{cases} (\Delta', \Delta') & \text{if } P \in \mathsf{GVal} \\ (\cdot, \quad \Delta') & \text{otherwise} \end{cases} \qquad \mathsf{split}(\forall \Delta.G, P) = \begin{cases} (\Delta, G) & \text{if } P \in \mathsf{GVal} \\ (\cdot, \forall \Delta.G) & \text{otherwise} \end{cases}$$
$$\text{where } \Delta' = \mathsf{ftv}(A) - \Delta$$

(The judgement $\Delta \vdash \sigma : \Delta' \Rightarrow_\bullet \Delta''$ is the monomorphic instantiation judgement of Explicit FreezeML.)
We replace the FreezeML typing rules LetGen and A-LetGen with the following rules.

$$\textsc{LetGen'}\quad \frac{\begin{array}{c}(\Delta', \Delta'') = \mathsf{gen}(\Delta, A', P) \quad (\Delta, \Delta'', P, A') \Updownarrow A \quad \Delta, \Delta''; \Gamma \vdash P : A' \quad \Delta; \Gamma, x : A \vdash Q : B \\ \mathsf{principal}(\Delta, \Gamma, P, \Delta'', A')\end{array}}{\Delta; \Gamma \vdash \mathbf{let}\ x = P\ \mathbf{in}\ Q : B}$$

$$\textsc{A-LetGen'}\quad \frac{(\Delta', A') = \mathsf{split}(A, P) \quad \Delta, \Delta'; \Gamma \vdash P : A' \quad \Delta; \Gamma, x : A \vdash Q : B}{\Delta; \Gamma \vdash \mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q : B}$$

Desugaring to Explicit FreezeML. We replace the desugaring rule for **let** with the following two rules according to whether the bound term is a guarded value or not.

$$
\begin{array}{lcll}
\mathbf{let}\ x = U\ \mathbf{in}\ Q & \equiv & \mathbf{let}\ \lceil x \rceil = \Lambda\bullet.U\ \mathbf{in}\ Q & \\
\mathbf{let}\ x = P\ \mathbf{in}\ Q & \equiv & \mathbf{let}\ \lceil x \rceil = (\Lambda\bullet.\lambda().P)\bullet\ ()\ \mathbf{in}\ Q & \text{if } P \notin \mathsf{GVal}
\end{array}
$$

Here, () is the usual data constructor of the unit type and thunking enables us to treat P as a value, as per the value restriction.

We replace the desugaring rule for type-annotated **let** with the following two rules.

$$
\begin{array}{lcll}
\mathbf{let}\ (x : A) = U\ \mathbf{in}\ Q & \equiv & (\lambda(x : A).Q)\ (\Lambda\bullet.U : A) & \\
\mathbf{let}\ (x : A) = P\ \mathbf{in}\ Q & \equiv & (\lambda(x : A).Q)\ P & \text{if } P \notin \mathsf{GVal}
\end{array}
$$

We rely on the syntactic sugar for type-annotated terms and type-annotated generalisation from Section 3; the latter being restricted appropriately to accommodate the value restriction.

$$
\begin{array}{rcl}
(P : A) & \equiv & (\lambda(x : A).\lceil x \rceil)\ P \\
(\Lambda\bullet.U : \forall\Delta.G) & \equiv & \Lambda\Delta.(U : G)
\end{array}
$$