

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.

$$\frac{\text{I-GEN-LAX}}{\Delta, \Delta'; \Gamma \vdash M : S} \quad \frac{\Delta, \Delta'; \Gamma \vdash M : S}{\Delta; \Gamma \vdash M : \forall \Delta'. S}$$

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

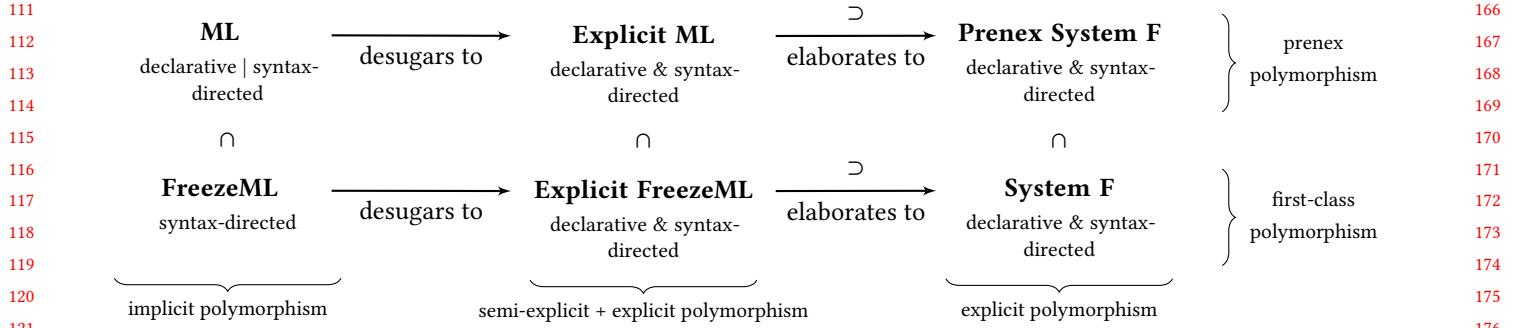


Figure 1. Desugaring and Elaboration of ML and FreezeML

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

$$\frac{\text{I-GEN} \quad \Delta, \Delta'; \Gamma \vdash M : S \quad \text{principal}(\Delta, \Gamma, M, \Delta', S)}{\Delta; \Gamma \vdash M : \forall \Delta'. S}$$

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 $\forall a.a \rightarrow a$. Explicit ML adopts a variant of I-GEN in which generalisation is explicit in the syntax of terms.

$$\frac{\text{GEN} \quad \Delta, \Delta'; \Gamma \vdash M : S \quad \text{principal}(\Delta, \Gamma, M, \Delta', S)}{\Delta; \Gamma \vdash \Lambda \bullet. M : \forall \Delta'. S}$$

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

$$\frac{\text{I-INST} \quad \Delta; \Gamma \vdash M : \forall \Delta'. S \quad \Delta \vdash \sigma : \Delta' \Rightarrow .}{\Delta; \Gamma \vdash M : \sigma(S)}$$

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.

$$\frac{\text{INST} \quad \Delta; \Gamma \vdash M : \forall \Delta'. S \quad \Delta \vdash \sigma : \Delta' \Rightarrow .}{\Delta; \Gamma \vdash M \bullet : \sigma(S)}$$

Variables and let-binding. We write variables as $[x]$ and let-binding as $\text{let } [x] = M \text{ in } N$. 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.

$$\begin{aligned} x &\equiv [x] \bullet \\ \text{let } x = M \text{ in } N &\equiv \text{let } [x] = \Lambda \bullet. M \text{ in } N \end{aligned}$$

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 $\forall a.a \rightarrow a$ or $\forall a.b.b \rightarrow b$. Exactly the same problem occurs with the term: $\Lambda a.id$ where $id : \forall a.a \rightarrow a$.

SML [10] resolves the issue by, in both cases selecting $\forall a.b.b \rightarrow b$, 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.

$$\begin{array}{ll} \text{ITerm } \ni & \text{MTerm } \ni \\ I, J ::= [x] & M, N ::= [x] \\ | \lambda(x : S). I \mid I N & | \lambda(x : S). M \mid M N \\ | \Lambda a. I \mid I S & | \Lambda a. I \mid M S \\ | \text{let } [x] = I \text{ in } J & | \text{let } [x] = M \text{ in } N \\ | \Lambda \bullet. M & | \lambda x. M \\ & | \Lambda \bullet. M \\ & | M \bullet \end{array}$$

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.

$$\begin{aligned} \text{ITerm } \ni I, J &:= [x] \\ &\quad | \lambda(x : A).I \mid IQ \\ &\quad | \Lambda a.I \mid IA \\ &\quad | \text{let } [x] = I \text{ in } J \\ &\quad | \Lambda \bullet .P \\ \\ \text{MTerm } \ni M, N &:= [x] \\ &\quad | \lambda(x : A).M \mid MQ \\ &\quad | \Lambda a.I \mid MA \\ &\quad | \text{let } [x] = M \text{ in } N \\ &\quad | \lambda x.M \\ &\quad | \Lambda \bullet .P \\ &\quad | M \bullet \\ \\ \text{PTerm } \ni P, Q &:= [x] \\ &\quad | \lambda(x : A).P \mid PQ \\ &\quad | \Lambda a.I \mid PA \\ &\quad | \text{let } [x] = M \text{ in } Q \\ &\quad | \lambda x.P \\ &\quad | \Lambda \bullet .P \\ &\quad | P \bullet \\ &\quad | P \star \end{aligned}$$

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.

$$\begin{aligned} x &\equiv [x]\star \\ \text{let } x = P \text{ in } Q &\equiv \text{let } [x] = \Lambda \bullet .P \text{ in } Q \end{aligned}$$

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.

$$\begin{aligned} (P : A) &\equiv (\lambda(x : A).[x])P \\ (\Lambda \bullet .P : \forall \Delta.G) &\equiv \Lambda \Delta.(P : G) \\ \text{let } (x : A) = P \text{ in } Q &\equiv (\lambda(x : A).Q)(\Lambda \bullet .P : A) \end{aligned}$$

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.

$$\begin{aligned} \text{let}' x = P \text{ in } Q &\equiv \text{let } [x] = (\Lambda \bullet .P)\bullet \text{ in } Q \\ \text{let}' (x : A) = P \text{ in } Q &\equiv (\lambda(x : A).Q)P \end{aligned}$$

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.

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

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

$$\begin{array}{ll} \text{let } [x] = M \text{ in } Q & \simeq (\lambda(x : A).Q)M \\ \lambda x.P & \simeq \lambda(x : S).P \\ \Lambda \bullet .I & \simeq \Lambda \Delta.I \\ P \bullet & \simeq PS_1 \dots S_n \\ P \star & \simeq PA_1 \dots A_n \end{array}$$

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 $F\omega$ 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.

331 References

- 332 [1] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and
333 Gilles Kahn. 1986. A Simple Applicative Language: Mini-ML. In *LISP*
334 and *Functional Programming*. ACM, 13–27.
- 335 [2] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for
336 Functional Programs. In *POPL*. ACM Press, 207–212.
- 337 [3] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan
338 Coates. 2019. *FreezeML: Complete and Easy Type Inference for First-Class*
339 *Polymorphism (extended version)*. Technical Report. arXiv:2004.00396.
- 340 [4] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan
341 Coates. 2020. FreezeML: Complete and Easy Type Inference for First-
342 Class Polymorphism. In *PLDI*. ACM. <https://arxiv.org/abs/2004.00396>.
343 To appear.
- 344 [5] Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit First-Class
345 Polymorphism for ML. *Inf. Comput.* 155, 1-2 (1999), 134–169.
- 346 [6] Didier Le Botlan and Didier Rémy. 2003. ML^F: raising ML to the power
347 of System F. In *ICFP*. ACM, 27–38.
- 348 [7] Daan Leijen. 2007. A type directed translation of MLF to system F. In
349 *ICFP*. ACM, 111–122.
- 350 [8] Daan Leijen. 2008. HMF: simple type inference for first-class polymor-
351 phism. In *ICFP*. ACM, 283–294.
- 352 [9] Daan Leijen. 2009. Flexible types: robust type inference for first-class
353 polymorphism. In *POPL*. ACM, 66–77.
- 354 [10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997.
355 *The Definition of Standard ML (Revised)*. MIT Press.
- 356
- 357
- 358
- 359
- 360
- 361
- 362
- 363
- 364
- 365
- 366
- 367
- 368
- 369
- 370
- 371
- 372
- 373
- 374
- 375
- 376
- 377
- 378
- 379
- 380
- 381
- 382
- 383
- 384
- 385
- [11] James O’Toole and David K. Gifford. 1989. Type Reconstruction with
386 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
388 Type Reconstruction. *Fundam. Inform.* 19, 1/2 (1993), 185–199.
- [13] Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-
389 class Polymorphism for ML. In *ML*. ACM, 3–14.
- [14] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios
390 Vytiniotis. 2020. A quick look at impredicativity. In *ICFP*. ACM. Con-
391 ditionally accepted.
- [15] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon
392 Peyton Jones. 2018. Guarded impredicative polymorphism. In *PLDI*.
393 ACM, 783–796.
- [16] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones.
394 2006. Boxy types: inference for higher-rank types and impredicativity.
395 In *ICFP*. ACM, 251–262.
- [17] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones.
396 2008. FPH: first-class polymorphism for Haskell. In *ICFP*. ACM, 295–
397 306.
- [18] J. B. Wells. 1994. Typability and Type-Checking in the Second-Order
398 lambda-Calculus are Equivalent and Undecidable. In *LICS*. IEEE Com-
399 puter Society, 176–185.
- [19] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *Lisp and*
400 *Symbolic Computation* 8, 4 (1995), 343–355.
- 401
- 402
- 403
- 404
- 405
- 406
- 407
- 408
- 409
- 410
- 411
- 412
- 413
- 414
- 415
- 416
- 417
- 418
- 419
- 420
- 421
- 422
- 423
- 424
- 425
- 426
- 427
- 428
- 429
- 430
- 431
- 432
- 433
- 434
- 435
- 436
- 437
- 438
- 439
- 440

441	A Prenex System F	496	
442	A.1 Syntax of Prenex System F	497	
443		498	
444		499	
445	Type Variables	a, b, c	500
446	Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$	501
447	Monotypes	$S, T ::= a \mid D \bar{S}$	502
448	Type Schemes	$E, F ::= \forall \bar{a}. S$	503
449	Type Contexts	$\Delta ::= \cdot \mid \Delta, a$	504
450	Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : E$	505
451	Term Variables	x, y, z	506
452	Terms	$M, N ::= [x] \mid \lambda(x : S). M \mid M N \mid \Lambda a. M \mid M S$	507
453			508
454	A.2 Type System of Prenex System F	509	
455	<i>Well-formed monotypes / type schemes.</i>	$\boxed{\Delta \vdash E \text{ ok}}$	510
456			511
457			512
458			513
459	$a \in \Delta$	$\Delta \vdash a \text{ ok}$	514
460			515
461		$\frac{\text{arity}(D) = n}{\Delta \vdash E_1 \text{ ok} \cdots \Delta \vdash E_n \text{ ok}} \Delta \vdash D \bar{E} \text{ ok}$	516
462	<i>Typing.</i>	$\boxed{\Delta; \Gamma \vdash M : E}$	517
463			518
464			519
465			520
466	VAR	$\frac{\text{APP}}{\Delta; \Gamma \vdash x : E}$	521
467		$\Delta; \Gamma \vdash M : S \rightarrow T$	
468		$\Delta; \Gamma \vdash N : S$	
469		$\frac{\text{TyLAM}}{\Delta; \Gamma \vdash M N : T}$	
470		$\Delta; \Gamma \vdash \lambda(x : S). M : \forall a. E$	
471	A.3 Equational Rules of Prenex System F	526	
472	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 $\Delta; \Gamma \vdash N : E$).	527	
473		528	
474		529	
475		530	
476	β -rules	$(\lambda(x : S). M) N \simeq M[M/[x]]$	531
477		$(\Lambda a. M) S \simeq M[S/a]$	532
478			533
479	η -rules	$\lambda(x : S). M [x] \simeq M$	534
480		$\Lambda a. M a \simeq M$	535
481			536
482			537
483	B Explicit ML	538	
484	B.1 Syntax of Explicit ML	539	
485	<i>Types.</i>	$\boxed{\Delta \vdash E \text{ ok}}$	540
486			541
487			542
488	Type Variables	a, b, c	543
489	Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$	544
490	Monotypes	$S, T ::= a \mid D \bar{S}$	545
491	Type Schemes	$E, F ::= \forall \bar{a}. S$	546
492	Type Instantiation	$\sigma ::= \emptyset \mid \sigma[a \mapsto S]$	547
493	Type Contexts	$\Delta ::= \cdot \mid \Delta, a$	548
494	Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : E$	549
495			550

551	Terms.	606
552		607
553	$\text{ITerm} \ni I, J ::= [x]$	608
554	$\lambda(x : S).I \mid I N$	609
555	$\Lambda a.I \mid I S$	610
556	$\text{let } [x] = I \text{ in } J$	611
557	$\Lambda\bullet.M$	612
558		613
559		614
560		615
561	B.2 Type System of Explicit ML	616
562	<i>Well-formed monotypes / type schemes.</i> $\boxed{\Delta \vdash E \text{ ok}}$	617
563		618
564	$\frac{a \in \Delta}{\Delta \vdash a \text{ ok}}$	619
565	$\frac{\text{arity}(D) = n \quad \Delta \vdash E_1 \text{ ok} \quad \dots \quad \Delta \vdash E_n \text{ ok}}{\Delta \vdash D \bar{E} \text{ ok}}$	620
566		621
567		622
568	<i>Instantiation.</i> $\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow \Delta''}$	623
569		624
570		625
571		626
572	$\frac{}{\Delta \vdash \emptyset : \cdot \Rightarrow \Delta'}$	627
573		628
574	<i>Principality.</i> $\boxed{\text{principal}(\Delta, \Gamma, M, \Delta', E)}$	629
575		630
576		631
577		632
578	$\text{principal}(\Delta, \Gamma, M, \Delta', E') =$	633
579	$\Delta' = \text{ftv}(E') - \Delta$ and $\Delta, \Delta'; \Gamma \vdash M : E'$ and	634
580	(for all Δ'', E'' if $\Delta'' = \text{ftv}(E'') - \Delta$ and	635
581	$\Delta, \Delta''; \Gamma \vdash M : E''$	636
582	then there exists σ such that	637
583	$\Delta \vdash \sigma : \Delta' \Rightarrow \Delta''$ and $\sigma(E') = E''$	638
584		639
585	<i>Typing.</i> $\boxed{\Delta; \Gamma \vdash M : E}$	640
586		641
587		642
588	VAR	643
589	$x : E \in \Gamma$	
590	$\Delta; \Gamma \vdash [x] : E$	
591		644
592	LAM	645
593	$\Delta; \Gamma, x : S \vdash M : T$	
594	$\Delta; \Gamma \vdash \lambda(x : S).M : S \rightarrow T$	
595		646
596	APP	647
597	$\Delta; \Gamma \vdash M : S \rightarrow T$	
598	$\Delta; \Gamma \vdash N : S$	
599	$\Delta; \Gamma \vdash MN : T$	
600		648
601	TYLAM	649
602	$\Delta, a; \Gamma \vdash I : E$	
603	$\Delta; \Gamma \vdash \Lambda a.I : \forall a.E$	
604		650
605	TyAPP	651
606	$\Delta; \Gamma \vdash M : \forall a.E$	
607	$\Delta; \Gamma \vdash MS : E[S/a]$	
608		652
609	LET	653
610	$\Delta; \Gamma \vdash M : E \quad \Delta; \Gamma, x : E \vdash N : F$	
611	$\Delta; \Gamma \vdash \text{let } [x] = M \text{ in } N : F$	
612	U-LAM	654
613	$\Delta; \Gamma, x : S \vdash M : T$	
614	$\Delta; \Gamma \vdash \lambda x.M : S \rightarrow T$	
615		655
616	MONOINST	656
617	$\Delta; \Gamma \vdash M : \forall \Delta'.S$	
618	$\Delta \vdash \sigma : \Delta' \Rightarrow \cdot$	
619	$\Delta; \Gamma \vdash M\bullet : \sigma(S)$	
620		657
621	B.3 Equational Rules of Explicit ML	
622	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 $\Delta; \Gamma \vdash N : E$).	658
623		659
624		660

661					716
662	β -rules	$(\lambda(x : S).M) N \simeq M[N/\lceil x \rceil]$			717
663		$(\Lambda a.I) S \simeq I[S/a]$			718
664					719
665	η -rules	$\lambda(x : S).M \lceil x \rceil \simeq M$			720
666		$\Lambda a.I a \simeq I$			721
667					722
668	elaboration rules	$\text{let } \lceil x \rceil = M \text{ in } N \simeq (\lambda(x : S).N) M$			723
669		$\lambda x.M \simeq \lambda(x : S).M$			724
670		$\Lambda \bullet .I \simeq \Lambda \Delta .I$			725
671		$M \bullet \simeq M S_1 \dots S_n$			726

B.4 Translation from Explicit ML to Prenex System F

672					727
673					728
674	$\left[\frac{x : E \in \Gamma}{\Delta; \Gamma \vdash \lceil x \rceil : E} \right] = x$	$\left[\frac{\Delta; \Gamma, x : A \vdash M : T}{\Delta; \Gamma \vdash \lambda(x : S).M : S \rightarrow T} \right] = \lambda(x : S).\llbracket M \rrbracket$	$\left[\frac{\Delta; \Gamma \vdash M : S \rightarrow T \quad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash M N : T} \right] = \llbracket M \rrbracket \llbracket N \rrbracket$		729
675					730
676					731
677		$\left[\frac{\Delta, a; \Gamma \vdash I : E}{\Delta; \Gamma \vdash \Lambda a.I : \forall a.E} \right] = \Lambda a.\llbracket I \rrbracket$	$\left[\frac{\Delta; \Gamma \vdash M : \forall a.E}{\Delta; \Gamma \vdash M S : E[S/a]} \right] = \llbracket M \rrbracket S$		732
678					733
679					734
680		$\left[\frac{\Delta; \Gamma \vdash M : E \quad \Delta; \Gamma, x : E \vdash N : F}{\Delta; \Gamma \vdash \text{let } \lceil x \rceil = M \text{ in } N : F} \right] = (\lambda(x : E).\llbracket N \rrbracket)\llbracket M \rrbracket$	$\left[\frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x.M : S \rightarrow T} \right] = \lambda(x : S).\llbracket M \rrbracket$		735
681					736
682					737
683					738
684		$\left[\frac{\Delta, \Delta'; \Gamma \vdash M : E \quad \text{principal}(\Delta, \Gamma, M, \Delta', E)}{\Delta; \Gamma \vdash \Lambda \bullet .M : \forall \Delta'.E} \right] = \Lambda \Delta' .\llbracket M \rrbracket$	$\left[\frac{\Delta; \Gamma \vdash M : \forall \Delta'.S \quad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M \bullet : \sigma(S)} \right] = \llbracket M \rrbracket \sigma(\Delta')$		739
685					740
686					741
687					742

C ML

C.1 Syntax of ML

Types.

692	Type Variables	a, b, c			747
693	Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$			748
694	Monotypes	$S, T ::= a \mid D \bar{S}$			749
695	Type Schemes	$E, F ::= \forall a.S$			750
696	Type Instantiation	$\sigma ::= \emptyset \mid \sigma[a \mapsto S]$			751
697	Type Contexts	$\Delta ::= \cdot \mid \Delta, a$			752
698	Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : E$			753
699					754

Terms.

700	$M, N ::= x$				755
701		$ \lambda x.M \mid MN$			756
702		$ \text{let } x = M \text{ in } N$			757
703					758

C.2 Type System of ML

704	Well-formed monotypes / type schemes.	$\boxed{\Delta \vdash E \text{ ok}}$			759		
705					760		
706	$a \in \Delta$	$\frac{\text{arity}(D) = n}{\Delta \vdash a \text{ ok}}$	$\Delta \vdash S_1 \text{ ok}$	\dots	$\Delta \vdash S_n \text{ ok}$	$\frac{\Delta, a \vdash E \text{ ok}}{\Delta \vdash \forall a.E \text{ ok}}$	761
707							762
708							763
709							764

Instantiation.

710	$\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow \Delta''}$	$\frac{\Delta \vdash \sigma : \Delta' \Rightarrow \Delta'' \quad \Delta, \Delta'' \vdash S \text{ ok}}{\Delta \vdash \sigma[a \mapsto S] : (\Delta', a) \Rightarrow \Delta''}$			765
711					766
712					767
713					768
714	$\Delta \vdash \emptyset : \cdot \Rightarrow \Delta'$				769
715					770

771	<i>Syntax-directed Typing Judgement.</i>	$\boxed{\Delta; \Gamma \vdash M : S}$	826
772			827
773			828
774	VARINST	$\frac{x : \forall \Delta'. S \in \Gamma \quad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash x : \sigma(S)}$	829
775			830
776			831
777	APP	$\frac{\Delta; \Gamma \vdash M : S \rightarrow T \quad \Delta; \Gamma \vdash N : S}{\Delta; \Gamma \vdash MN : T}$	832
778			833
779	LETGEN	$\frac{\Delta' = \text{ftv}(S) - \Delta \quad \Delta, \Delta'; \Gamma \vdash M : S \quad E = \forall \Delta'. S \quad \Delta; \Gamma, x : E \vdash N : T}{\Delta; \Gamma \vdash \text{let } x = M \text{ in } N : T}$	834
780			835
781			836
782	<i>Declarative Typing Judgement.</i>	$\boxed{\Delta; \Gamma \vdash M : E}$	837
783			838
784			839
785			840
786	VAR	$\frac{x : E \in \Gamma}{\Delta; \Gamma \vdash x : E}$	841
787	U-LAM	$\frac{\Delta; \Gamma, x : S \vdash M : T}{\Delta; \Gamma \vdash \lambda x. M : S \rightarrow T}$	842
788			843
789			844
790	LET	$\frac{\Delta; \Gamma \vdash M : E \quad \Delta; \Gamma, x : E \vdash N : F}{\Delta; \Gamma \vdash \text{let } x = M \text{ in } N : F}$	845
791	I-GEN-LAX	$\frac{\Delta, \Delta'; \Gamma \vdash M : S}{\Delta; \Gamma \vdash M : \forall \Delta'. S}$	846
792			847
793	I-INST	$\frac{\Delta; \Gamma \vdash M : \forall \Delta'. S \quad \Delta \vdash \sigma : \Delta' \Rightarrow \cdot}{\Delta; \Gamma \vdash M : \sigma(E)}$	848
794			849
795	C.3 Desugaring from ML to Explicit ML		850
796			851
797	$x \equiv [x] \bullet$		852
798	$\text{let } x = M \text{ in } N \equiv \text{let } [x] = \Lambda \bullet. M \text{ in } N$		853
799			854
800	D System F		855
801	D.1 Syntax of System F		856
802			857
803	Type Variables	a, b, c	858
804	Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$	859
805	Types	$A, B ::= a \mid D \bar{A} \mid \forall a. A$	860
806	Type Contexts	$\Delta ::= \cdot \mid \Delta, a$	861
807	Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$	862
808	Term Variables	x, y, z	863
809	Terms	$M, N ::= [x] \mid \lambda(x : A). M \mid MN \mid \Lambda a. M \mid MA$	864
810			865
811	D.2 Type System of System F		866
812	Well-formed types.	$\boxed{\Delta \vdash A \text{ ok}}$	867
813			868
814	$a \in \Delta$	$\text{arity}(D) = n$	869
815	$\frac{}{\Delta \vdash a \text{ ok}}$	$\frac{\Delta \vdash A_1 \text{ ok} \cdots \Delta \vdash A_n \text{ ok}}{\Delta \vdash D \bar{A} \text{ ok}}$	870
816			871
817		$\frac{\Delta, a \vdash A \text{ ok}}{\Delta \vdash \forall a. A \text{ ok}}$	872
818	<i>Typing.</i>	$\boxed{\Delta; \Gamma \vdash M : A}$	873
819			874
820			875
821	APP		876
822	VAR	$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash [x] : A}$	877
823	TYLAM	$\frac{\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A}{\Delta, a; \Gamma \vdash M : A}$	878
824	LAM	$\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda(x : A). M : A \rightarrow B}$	879
825	TYAPP	$\frac{\Delta; \Gamma \vdash M : \forall a. B}{\Delta; \Gamma \vdash MA : E[A/a]}$	880

881	D.3 Equational Rules of System F	936
882	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 $\Delta; \Gamma \vdash N : A$).	937
883		938
884		939

885	β -rules	$(\lambda(x : A).M)N \simeq M[N/\lceil x \rceil]$	940
886		$(\Lambda a.M)A \simeq M[A/a]$	941
887			942
888	η -rules	$\lambda(x : A).M \lceil x \rceil \simeq M$	943
889		$\Lambda a.M a \simeq M$	944
890			945
891			946

892 E Explicit FreezeML 947

893 E.1 Syntax of Explicit FreezeML 948

894 *Types.* 949

896	Type Variables	a, b, c	951
897	Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$	952
898	Types	$A, B ::= a \mid D\bar{A} \mid \forall a.A$	953
899	Monotypes	$S, T ::= a \mid D\bar{S}$	954
900	Guarded Types	$G ::= a \mid D\bar{A}$	955
901	Monomorphic Instantiation	$\sigma ::= \emptyset \mid \sigma[a \mapsto S]$	956
902	Polymorphic Instantiation	$\delta ::= \emptyset \mid \delta[a \mapsto A]$	957
903	Type Contexts	$\Delta ::= \cdot \mid \Delta, a$	958
904	Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$	959
905			960

906 *Terms.* 961

907	$I\text{-Term } \exists I, J ::= \lceil x \rceil$	$M\text{-Term } \exists M, N ::= \lceil x \rceil$	$P\text{-Term } \exists P, Q ::= \lceil x \rceil$	962
908	$ \lambda(x : A).I \mid I Q$	$ \lambda(x : A).M \mid M Q$	$ \lambda(x : A).P \mid P Q$	963
909	$ \Lambda a.I \mid I A$	$ \Lambda a.I \mid M A$	$ \Lambda a.I \mid P A$	964
910	$ \text{let } \lceil x \rceil = I \text{ in } J$	$ \text{let } \lceil x \rceil = M \text{ in } N$	$ \text{let } \lceil x \rceil = M \text{ in } Q$	965
911		$ \lambda x.M$	$ \lambda x.P$	966
912		$ \Lambda\bullet.P$	$ \Lambda\bullet.P$	967
913		$ M\bullet$	$ P\bullet$	968
914			$ P\star$	969
915				970

916 E.2 Type System of Explicit FreezeML 971

917 *Well-formed types.* $\boxed{\Delta \vdash A \text{ ok}}$ 972

919	$a \in \Delta$	$\text{arity}(D) = n$	$\Delta \vdash A_1 \text{ ok}$	\cdots	$\Delta \vdash A_n \text{ ok}$	$\Delta, a \vdash A \text{ ok}$	974
920						$\Delta \vdash D\bar{A} \text{ ok}$	975
921	$\Delta \vdash a \text{ ok}$						976
922							977

923 *Monomorphic instantiation.* $\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \Delta''}$ 978

925	$\Delta \vdash \emptyset : \cdot \Rightarrow_{\bullet} \Delta'$	$\Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \Delta''$	$\Delta, \Delta'' \vdash S \text{ ok}$	980
926				981
927		$\Delta \vdash \sigma[a \mapsto S] : (\Delta', a) \Rightarrow_{\bullet} \Delta''$		982
928				983

929 *Polymorphic instantiation.* $\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''}$ 984

931	$\Delta \vdash \emptyset : \cdot \Rightarrow_{\star} \Delta'$	$\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''$	$\Delta, \Delta'' \vdash A \text{ ok}$	986
932				987
933		$\Delta \vdash \delta[a \mapsto A] : (\Delta', a) \Rightarrow_{\star} \Delta''$		988
934				989
935				990

991	<i>Principality judgement.</i>	$\boxed{\text{principal}(\Delta, \Gamma, P, \Delta', A')}$	1046	
992			1047	
993			1048	
994			1049	
995			1050	
996		$\text{principal}(\Delta, \Gamma, P, \Delta', A') =$	1051	
997		$\Delta' = \text{ftv}(A') - \Delta$ and $\Delta, \Delta'; \Gamma \vdash P : A'$ and	1052	
998		(for all Δ'', A'' if $\Delta'' = \text{ftv}(A'') - \Delta$ and	1053	
999		$\Delta, \Delta''; \Gamma \vdash P : A''$	1054	
1000		then there exists δ such that	1055	
1001		$\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''$ and $\delta(A') = A''$)	1056	
1002			1057	
1003	<i>Typing judgement.</i>	$\boxed{\Delta; \Gamma \vdash P : A}$	1058	
1004			1059	
1005			1060	
1006			1061	
1007			1062	
1008	VAR	LAM	APP	1063
1009	$x : E \in \Gamma$	$\Delta; \Gamma, x : S \vdash P : T$	$\Delta; \Gamma \vdash P : A \rightarrow B$	1064
1010			$\Delta; \Gamma \vdash Q : A$	1065
1011	$\Delta; \Gamma \vdash [x] : E$	$\Delta; \Gamma \vdash \lambda(x : S).P : S \rightarrow T$	$\Delta; \Gamma \vdash PQ : B$	1066
1012				1067
1013		LET		1068
1014	$\Delta; \Gamma \vdash M : E$	$\Delta; \Gamma, x : E \vdash N : F$	U-LAM	1069
1015		$\Delta; \Gamma \vdash \text{let } [x] = M \text{ in } N : F$	$\Delta; \Gamma, x : S \vdash M : T$	1070
1016				1071
1017				1072
1018	GEN	$\Delta, \Delta'; \Gamma \vdash M : E$	MONOINST	1073
1019		$\text{principal}(\Delta, \Gamma, M, \Delta', E)$	$\Delta; \Gamma \vdash P : \forall \Delta'.S$	1074
1020			$\Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \cdot$	1075
1021		$\Delta; \Gamma \vdash \Lambda \bullet. M : \forall \Delta'.E$	$\Delta; \Gamma \vdash P \bullet : \sigma(S)$	1076
1022	E.3 Equational Rules of Explicit FreezeML			1077
1023	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 $\Delta; \Gamma \vdash Q : A$).			1078
1024				1079
1025				1080
1026				1081
1027				1082
1028				1083
1029	β -rules	$(\lambda(x : A).P)Q \simeq P[Q/[x]]$		1084
1030		$(\Lambda a.I)A \simeq I[A/a]$		1085
1031				1086
1032	η -rules	$\lambda(x : A).P[x] \simeq P$		1087
1033		$\Lambda a.I a \simeq I$		1088
1034				1089
1035	elaboration rules	$\text{let } [x] = M \text{ in } Q \simeq (\lambda(x : A).Q)M$		1090
1036		$\lambda x.P \simeq \lambda(x : S).P$		1091
1037		$\Lambda \bullet.I \simeq \Lambda \Delta.I$		1092
1038		$P \bullet \simeq PS_1 \dots S_n$		1093
1039		$P \star \simeq PA_1 \dots A_n$		1094
1040				1095
1041				1096
1042				1097
1043				1098
1044				1099
1045				1100

1101 **E.4 Translation from Explicit FreezeML to System F**

$$\begin{array}{c}
 \frac{}{\llbracket x : A \in \Gamma \rrbracket = x} \quad \frac{}{\llbracket \Delta; \Gamma, x : A \vdash P : B \rrbracket = \lambda(x : A). \llbracket P \rrbracket} \quad \frac{\Delta; \Gamma \vdash P : A \rightarrow B \quad \Delta; \Gamma \vdash Q : A}{\Delta; \Gamma \vdash P Q : B} = \llbracket P \rrbracket \llbracket Q \rrbracket \\
 \\
 \frac{}{\llbracket \Delta, a; \Gamma \vdash I : A \rrbracket = \Lambda a. \llbracket I \rrbracket} \quad \frac{}{\llbracket \Delta; \Gamma \vdash \Lambda a. I : \forall a. A \rrbracket = \llbracket P \rrbracket A} \\
 \\
 \frac{\Delta; \Gamma \vdash M : A \quad \Delta; \Gamma, x : A \vdash Q : B}{\Delta; \Gamma \vdash \text{let } \lceil x \rceil = M \text{ in } Q : B} = (\lambda(x : A). \llbracket Q \rrbracket) \llbracket M \rrbracket \quad \frac{}{\llbracket \Delta; \Gamma, x : S \vdash P : B \rrbracket = \lambda(x : S). \llbracket P \rrbracket} \\
 \\
 \frac{\Delta, \Delta'; \Gamma \vdash P : A \quad \text{principal}(\Delta, \Gamma, P, \Delta', A)}{\Delta; \Gamma \vdash \Lambda \bullet. P : \forall \Delta'. A} = \Lambda \Delta'. \llbracket P \rrbracket \quad \frac{\Delta; \Gamma \vdash P : \forall \Delta'. G \quad \Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \cdot}{\Delta; \Gamma \vdash P \star : \sigma(G)} = \llbracket P \rrbracket \sigma(\Delta') \\
 \\
 \frac{\Delta; \Gamma \vdash P : \forall \Delta'. G \quad \Delta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot}{\Delta; \Gamma \vdash P \star : \delta(G)} = \llbracket P \rrbracket \delta(\Delta') =
 \end{array}$$

1113 **F FreezeML**

1114 **F.1 Syntax of FreezeML**

1115 *Types.*

Type Variables	a, b, c	1182
Type Constructors	$D ::= \text{Int} \mid \text{List} \mid \rightarrow \mid \times \mid \dots$	1183
Types	$A, B ::= a \mid D \bar{A} \mid \forall a. A$	1184
Monotypes	$S, T ::= a \mid D \bar{S}$	1185
Guarded Types	$G ::= a \mid D \bar{A}$	1186
Polymorphic Instantiation	$\delta ::= \emptyset \mid \delta[a \mapsto A]$	1187
Term Variables	x, y, z	1188
Type Contexts	$\Delta ::= \cdot \mid \Delta, a$	1189
Term Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$	1190

1137 *Terms.*

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

1143 **F.2 Type System of FreezeML**

1144 *Well-formed types.* $\boxed{\Delta \vdash A \text{ ok}}$

$$\frac{a \in \Delta}{\Delta \vdash a \text{ ok}} \quad \frac{\text{arity}(D) = n \quad \Delta \vdash A_1 \text{ ok} \quad \dots \quad \Delta \vdash A_n \text{ ok}}{\Delta \vdash D \bar{A} \text{ ok}} \quad \frac{\Delta, a \vdash A \text{ ok}}{\Delta \vdash \forall a. A \text{ ok}}$$

1150 *Polymorphic instantiation.* $\boxed{\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \Delta''}$

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

1211 **Principality judgement.** $\boxed{\text{principal}(\Delta, \Gamma, P, \Delta', A')}$ 1266

1212

1213

1214

1215

1216

1217

1218

1219

1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320

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

1220 **Typing judgement.** $\boxed{\Delta; \Gamma \vdash P : A}$ 1275

1221

1222 In contrast to Emrich et al. [4], we first present a simplified variant of FreezeML that does not incorporate the value 1277
1278 restriction. In Appendix G we describe how to adapt the following to support the value restriction.

$$\frac{\text{VAR}}{x : A \in \Gamma} \quad \frac{\text{VARINST}}{x : \forall \Delta'. G \in \Gamma} \quad \frac{\Delta \vdash \delta : \Delta' \Rightarrow_{\star} \cdot}{\Delta; \Gamma \vdash x : \delta(G)}$$

$$\frac{\text{U-LAM}}{\Delta; \Gamma, x : S \vdash P : B} \quad \frac{\text{LAM}}{\Delta; \Gamma, x : A \vdash P : B} \quad \frac{\Delta; \Gamma \vdash \lambda x. P : S \rightarrow B}{\Delta; \Gamma \vdash \lambda(x : A). P : A \rightarrow B}$$

$$\frac{\text{APP}}{\Delta; \Gamma \vdash P : A \rightarrow B \quad \Delta; \Gamma \vdash Q : A} \quad \frac{}{\Delta; \Gamma \vdash P Q : B}$$

$$\frac{\text{LETGEN}}{\Delta' = \text{ftv}(A') - \Delta \quad A = \forall \Delta'. A' \quad \Delta, \Delta''; \Gamma \vdash P : A' \quad \Delta; \Gamma, x : A \vdash Q : B} \quad \text{principal}(\Delta, \Gamma, P, \Delta', A')}$$

$$\frac{}{\Delta; \Gamma \vdash \text{let } x = P \text{ in } Q : B}$$

$$\frac{\text{A-LETGEN}}{A = \forall \Delta'. G \quad \Delta, \Delta'; \Gamma \vdash P : G \quad \Delta; \Gamma, x : A \vdash Q : B} \quad \frac{}{\Delta; \Gamma \vdash \text{let } (x : A) = P \text{ in } Q : B}$$

F.3 Desugaring from FreezeML to Explicit FreezeML

$$\begin{aligned} x &\equiv [x]_{\star} \\ \text{let } x = P \text{ in } Q &\equiv \text{let } [x] = \Lambda \bullet. P \text{ in } Q \\ (P : A) &\equiv (\lambda(x : A). [x]) P \\ (\Lambda \bullet. P : \forall \Delta. G) &\equiv \Lambda \Delta. (P : G) \\ \text{let } (x : A) = P \text{ in } Q &\equiv (\lambda(x : A). Q) (\Lambda \bullet. P : A) \end{aligned}$$

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 1309
1310 soundness in the presence of side effects (e.g., mutable references). We revisit versions of ML and FreezeML that do 1311
1312 obey the value restriction (the latter following Emrich et al. [3]), and show how the desugaring to the corresponding explicit 1313
1314 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 1315
1316 the value restriction it suffices to restrict the body of the type abstraction and generalisation operators to be syntactic 1317
1318 values.

G.1 ML

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

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

1321 **Typing.** We define the following helper function. 1376

1322

$$1323 \text{gen}(\Delta, A, M) = \begin{cases} \text{ftv}(A) - \Delta & \text{if } M \in \text{Val} \\ 1324 . & \text{otherwise} \end{cases}$$

1325 We then replace the ML typing rule LETGEN of the syntax-directed variant of ML (Appendix C.2) by the following rule. 1380

1326

$$\frac{\text{LETGEN}}{\Delta' = \text{gen}(\Delta, S, M) \quad \Delta, \Delta'; \Gamma \vdash M : S \\ E = \forall \Delta'. S \quad \Delta; \Gamma, x : E \vdash N : T} \quad \Delta; \Gamma \vdash \text{let } x = M \text{ in } N : T$$

1331 To adapt the declarative presentation, it suffices to limit the rule I-GEN-LAX to syntactic values. 1387

1333 **Desugaring to Explicit ML.** We replace the desugaring rule for let with the following: 1388

1334

$$\frac{\text{let } x = V \text{ in } N \equiv \text{let } [x] = \Lambda \bullet. V \text{ in } N}{\text{let } x = M \text{ in } N \equiv \text{let } [x] = M \text{ in } N} \quad \text{if } M \notin \text{Val}$$

1335 G.2 FreezeML

1340 **Syntax.** The grammar is augmented as follows: 1395

1341

1342 Monomorphic Instantiation	$\sigma ::= \emptyset \mid \sigma[a \mapsto S]$	1397
1343 Values	$\text{Val} \ni V, W ::= x \mid [x] \mid \lambda x. P \mid \lambda(x : A). P \mid \text{let } x = V \text{ in } W \mid \text{let } (x : A) = V \text{ in } W$	1398
1344 Guarded Values	$\text{GVal} \ni U ::= x \mid \lambda x. P \mid \lambda(x : A). P \mid \text{let } x = V \text{ in } U \mid \text{let } (x : A) = V \text{ in } U$	1399

1346 **Typing.** We define the following helper judgements and functions. 1401

1347 $\boxed{\Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \Delta''}$ 1402

1348

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

1349 $\boxed{(\Delta, \Delta', P, A') \Downarrow A}$ 1403

1350

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

1355

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

1364 (The judgement $\Delta \vdash \sigma : \Delta' \Rightarrow_{\bullet} \Delta''$ is the monomorphic instantiation judgement of Explicit FreezeML.) 1419

1365 We replace the FreezeML typing rules LETGEN and A-LETGEN with the following rules. 1420

1366

$$\frac{\text{LETGEN}' \\ (\Delta', \Delta'') = \text{gen}(\Delta, A', P) \quad (\Delta, \Delta'', P, A') \Downarrow A \quad \Delta, \Delta''; \Gamma \vdash P : A' \quad \Delta; \Gamma, x : A \vdash Q : B \\ \text{principal}(\Delta, \Gamma, P, \Delta'', A')} {\Delta; \Gamma \vdash \text{let } x = P \text{ in } Q : B}$$

1371

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

1431 *Desugaring to Explicit FreezeML.* We replace the desugaring rule for `let` with the following two rules according to whether
 1432 the bound term is a guarded value or not.
 1433 $\text{let } x = U \text{ in } Q \equiv \text{let } [x] = \Lambda \bullet. U \text{ in } Q$
 1434 $\text{let } x = P \text{ in } Q \equiv \text{let } [x] = (\Lambda \bullet. \lambda(). P) \bullet () \text{ in } Q \quad \text{if } P \notin \text{GVal}$

1435 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.
 1436 We replace the desugaring rule for type-annotated `let` with the following two rules.
 1437

$$\begin{aligned} 1438 \quad \text{let } (x : A) = U \text{ in } Q &\equiv (\lambda(x : A). Q) (\Lambda \bullet. U : A) \\ 1439 \quad \text{let } (x : A) = P \text{ in } Q &\equiv (\lambda(x : A). Q) P \quad \text{if } P \notin \text{GVal} \end{aligned}$$

1440 We rely on the syntactic sugar for type-annotated terms and type-annotated generalisation from Section 3; the latter being
 1441 restricted appropriately to accommodate the value restriction.
 1442

$$\begin{aligned} 1443 \quad (P : A) &\equiv (\lambda(x : A). [x]) P \\ 1444 \quad (\Lambda \bullet. U : \forall \Delta. G) &\equiv \Lambda \Delta. (U : G) \end{aligned}$$

1445
 1446
 1447
 1448
 1449
 1450
 1451
 1452
 1453
 1454
 1455
 1456
 1457
 1458
 1459
 1460
 1461
 1462
 1463
 1464
 1465
 1466
 1467
 1468
 1469
 1470
 1471
 1472
 1473
 1474
 1475
 1476
 1477
 1478
 1479
 1480
 1481
 1482
 1483
 1484
 1485

1486
 1487
 1488
 1489
 1490
 1491
 1492
 1493
 1494
 1495
 1496
 1497
 1498
 1499
 1500
 1501
 1502
 1503
 1504
 1505
 1506
 1507
 1508
 1509
 1510
 1511
 1512
 1513
 1514
 1515
 1516
 1517
 1518
 1519
 1520
 1521
 1522
 1523
 1524
 1525
 1526
 1527
 1528
 1529
 1530
 1531
 1532
 1533
 1534
 1535
 1536
 1537
 1538
 1539
 1540