A type directed translation of MLF to System F

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

The MLF type system by Le Botlan and Rémy (2003) is a natural extension of Hindley-Milner type inference that supports full first-class polymorphism, where types can be of higher-rank and impredicatively instantiated. Even though MLF is theoretically very attractive, it has not seen widespread adoption. We believe that this partly because it is unclear how the rich language of MLF types relate to standard System F types. In this article we give the first type directed translation of MLF terms to System F terms. Based on insight gained from this translation, we also define "Rigid MLF" (MLF$^=$), a restriction of MLF where all bound values have a System F type. The expressiveness of MLF$^=$ is the same as that of boxy types, but MLF$^=$ needs fewer annotations and we give a detailed comparison between them.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Languages, Design, Theory

Keywords First-class polymorphism, System F, MLF

1. Introduction

The MLF type system by Le Botlan and Rémy (2003) is a natural extension of Hindley-Milner type inference that supports full first-class polymorphism, where any value can have a polymorphic type. MLF has a strong theoretical foundation and requires very few type annotations. Even though MLF is very attractive for these reasons, it has not seen widespread adoption. We believe that this is partly because the type language of MLF is richer than that of System F. Besides that this is harder on the programmer, it is also harder to write a compiler for a language based on MLF: many compilers use an intermediate language based on System F, and the relation between MLF terms and System F terms far from clear.

In an attempt to remedy this situation, Leijen and Löh (2005) described a type inference algorithm that returns System F terms for well-typed MLF terms. Unfortunately, such algorithm does not give particular *insight* into the relation between MLF type rules and the translated System F terms. Furthermore, we believe that one of the reasons that a type directed translation was not given in the previous article is that it is surprisingly subtle to do so – indeed, this is the main technical contribution of this paper. Specifically:

- We are the first to give a type directed translation from MLF to System F terms, and we prove that the translation is sound. The translation of MLF types that go 'beyond System F' (namely flexibly bound polymorphic types) is done using evidence translation.
- We show that a translation is possible where no evidence is passed for polymorphic types with rigid bounds, which correspond naturally to inlined System F types. This greatly simplifies the translated System F terms, but we need to define the translation on slightly modified type rules. We show that the modified type rules are sound and complete with respect to the original MLF type rules.
- It is widely believed that a System F translation of impredicative types may need to traverse structures at runtime to apply coercion terms. We show conclusively that this is not the case for MLF and that there exists a concise translation that builds coerced terms directly without runtime traversals.
- The type directed System F translation leads naturally to the definition of a restriction of MLF, called "Rigid MLF" (MLF$^=$), where all bound values have a standard System F type. We show that MLF$^=$ has the same expressiveness as boxy type inference (Vytiniotis et al. 2006), but needs fewer annotations. Since MLF$^=$ hides MLF types, it can work well as a simplified version of MLF where the programmer only works with regular System F types.

In the next section, we start with an introduction to MLF and explain the MLF type rules. Section 3 first gives an overview of the difficulties of translating MLF to System F, and introduces a modified set of type rules on which we define a type directed System F translation. Section 4 defines MLF$^=$, a restriction of MLF where all values have System F types. Finally, in the related work section (Section 5) we give a detailed comparison between Hindley-Milner, MLF, MLF$^=$, and boxy types.

2. An introduction to MLF

Functional languages with type inference are almost always based on the Hindley-Milner type system (Hindley 1969; Milner 1978) – and for good reasons. Type inference based on Hindley-Milner can automatically infer most general, or *principal*, types for expressions without further type annotations. Also, the type system is *sound*, and well-typed programs cannot "go wrong".

To achieve automatic type inference, the Hindley-Milner type system restricts polymorphism and polymorphic values are not first-class citizens. In particular, function arguments can only be monomorphic. Formally, this means that universal quantifiers can only appear at the outermost level (i.e. higher-ranked types are disallowed), and quantified variables can only be instantiated with monomorphic types (i.e. impredicative types are disallowed).

The MLF type system by LeBotlan and Remy (Le Botlan 2004; Le Botlan and Rémy 2003) is a natural extension of Hindley-Milner

ICFP'07, October 1–3, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-815-2/07/0010…$5.00

that lifts these restrictions and supports first-class polymorphism with higher-ranked and impredicative types. In contrast to Hindley-Milner, some type annotations are required in MLF to avoid guessing polymorphic types. Take for example the following program:

$$\mathit{poly}\ f = (f\ 1, f\ \mathit{True})$$

This program is rejected by both MLF and Hindley-Milner. The program would be accepted in MLF if we annotate the argument f with a polymorphic type, like $\forall\alpha.\,\alpha \to \alpha$ or $\forall\alpha.\,\alpha \to \mathit{Int}$. The type of a polymorphic argument like f cannot be inferred automatically since there exist many types for f none of which are an instance of the other – there is no principal type.

The reader might be worried that many type annotations are needed, but such is not the case. MLF has a remarkable property that the *only* required type annotations are on *arguments that are used polymorphically*. For example,

$$\mathit{poly}\ (f :: \forall\alpha.\,\alpha \to \alpha) = (f\ 1, f\ \mathit{True})$$

is a well-typed MLF program. Note that the annotation rule implies that all programs accepted by Hindley-Milner are also accepted by MLF. Moreover, the annotation rule only requires annotations on polymorphic arguments that are also *used* polymorphically and not merely 'passed through'. For example:

$$\begin{array}{l} \mathit{polyL} :: [\forall\alpha.\,\alpha \to \alpha] \to (\mathit{Int}, \mathit{Bool}) \quad \text{-- inferred} \\ \mathit{polyL}\ \mathit{xs} = \mathit{poly}\ (\mathit{head}\ \mathit{xs}) \end{array}$$

is accepted without type annotations even though xs has an (inferred) impredicative polymorphic type. We consider this an important property for abstraction and modularity, since we can (re)use standard functions like *head*, with type $\forall\alpha.\,[\alpha] \to \alpha$, for lists that contain polymorphic values. As another example, consider the application function:

$$\begin{array}{l} \mathit{apply} :: \forall\alpha\beta.\,(\alpha \to \beta) \to \alpha \to \beta \quad \text{-- inferred} \\ \mathit{apply}\ f\ x = f\ x \end{array}$$

Of course, the direct application $\mathit{poly}\ \mathit{id}$ is well typed in MLF, but also the abstracted application $\mathit{apply}\ \mathit{poly}\ \mathit{id}$. No type annotations are needed, except for the argument annotation of *poly*. Note that to type check $\mathit{apply}\ \ \mathit{poly}\ \ \mathit{id}$, the α quantifier in the type of *apply* is impredicatively instantiated to the polymorphic type of id, namely $\forall\alpha.\,\alpha \to \alpha$. More generally, if an application $e_1\ e_2$ is well typed in MLF, than the expression $\mathit{apply}\ \ e_1\ \ e_2$ is also well typed. In general, this applies to functors (*apply*) applying polymorphic functions (*poly*) over structures holding polymorphic values (id). For example, we can use the standard *map* function to apply the *poly* function to the elements of a list of identity functions, as in $\mathit{map}\ \mathit{poly}\ [\mathit{id}, \mathit{id}]$, without any further type annotations.

2.1 Bounded types

Essential to type inference is the ability to assign principal types to expressions. In the presence of impredicative polymorphism however, we need more than standard System F types to have principal types. Consider the following program

$$\mathbf{let}\ \mathit{ids} = [\mathit{id}]\ \mathbf{in}\ (\mathit{polyL}\ \mathit{ids}, \mathit{ids} +\!\!+ [\mathit{inc}])$$

where inc has type $\mathit{Int} \to \mathit{Int}$ and the append function $(+\!\!+)$ has type $\forall\alpha.\,[\alpha] \to [\alpha] \to [\alpha]$. In a setting with first-class polymorphism we can assign two types to ids, namely a list of polymorphic identity functions, $[\forall\alpha.\,\alpha \to \alpha]$, or a polymorphic list of monomorphic identity functions, $\forall\alpha.\,[\alpha \to \alpha]$. Indeed, in the body of the let expression, we use ids with both types: the polyL function requires the first type, while appending to an $[\mathit{Int} \to \mathit{Int}]$ list requires the second type. Unfortunately, neither of these types is an instance of the other – so what type should we give to ids to make the program well typed?

Monomorphic types	
$\tau ::= \alpha$	type variable
$\quad\mid\ c\ \tau_1\ \ldots\ \tau_n$	constructor application
Polymorphic types	
$\sigma ::= q.\sigma$	quantified type
$\quad\mid\ \tau$	mono type
$\quad\mid\ \bot$	most polymorphic type
Quantifier	
$q\ ::= \forall(\alpha \diamond \sigma)$	a bounded quantifier
$\diamond\ ::= \geqslant \mid =$	a bound is flexible or rigid
Prefix	
$Q\ ::= q_1, \ldots, q_n$	a prefix is a list of quantifiers
Syntactic sugar	
$\forall\alpha\ \ = \forall(\alpha \geqslant \bot)$	
$Q.\tau = q_1.\ \ldots\ .q_n.\tau$	quantify using a prefix

Figure 1. MLF types.

MLF solves this dilemma by going beyond standard System F types and assigns a type that can be instantiated to both of the previous types. The MLF type for ids is $\forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,[\beta]$, which is read as "a list of β, for all types β that are an instance of (or equal to) $\forall\alpha.\,\alpha \to \alpha$". As we will see later, both of our previous types are instances of this type:

$$\begin{array}{l} \forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,[\beta] \ \sqsubseteq\ [\forall\alpha.\,\alpha \to \alpha] \\ \forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,[\beta] \ \sqsubseteq\ \forall\alpha.\,[\alpha \to \alpha] \end{array}$$

where $\sigma_1 \sqsubseteq \sigma_2$ states that σ_2 is an instance of σ_1. We call $\geqslant$ a *flexible* bound. For regularity, MLF also introduces *rigid* bounds, written as $=$. The type $\forall(\beta = \forall\alpha.\,\alpha \to \alpha).\,[\beta]$ is read as "a list of β, where β is as polymorphic as $\forall\alpha.\,\alpha \to \alpha$". Indeed, the type $[\forall\alpha.\,\alpha \to \alpha]$ is just a syntactic shorthand for the previous type and we generally inline rigid bounds for notational convenience.

2.2 MLF types formally

Figure 1 defines the grammar of MLF types. Monomorphic types τ are either a type variable α, or a type constructor application. We assume that the function constructor $\to$ is part of the type constructors and we do not need to treat it specially. Polymorphic types σ are either a monomorphic type, the most polymorphic type $\bot$, or a polymorphic type quantified with a bound $\forall(\alpha \diamond \sigma_1).\,\sigma_2$, where the bound $\diamond$ is either flexible ($\geqslant$) or rigid ($=$).

A quantifier with a flexible bound can be instantiated to any instance of its bound. In particular, a quantifier $\forall(\alpha \geqslant \bot)$ can be instantiated to any type since $\bot$ is the most polymorphic type. For example, the full MLF type for the identity function is:

$$\mathit{id} :: \forall(\alpha \geqslant \bot).\,\alpha \to \alpha$$

We call $\forall(\alpha \geqslant \bot)$ an *unconstrained bound* and usually shorten it to $\forall\alpha$. The $\bot$ type is defined as equivalent to $\forall(\alpha \geqslant \bot).\,\alpha$. Note that in Hindley-Milner all quantifiers are always unconstrained.

A list of quantifiers is called a *prefix* and denoted as Q. We always assume that the quantified variables are distinct and form the domain of Q, written as $\mathsf{dom}(Q)$. If all quantifiers are unconstrained, we call Q an *unconstrained prefix*.

2.3 Type rules of MLF

The type rules for MLF are given in Figure 2. Given the expressiveness of MLF, they are surprisingly simple and very similar to the type rules of Hindley-Milner. A derivation $(Q)\ \Gamma \vdash e : \sigma$ means that under a prefix Q and type environment Γ the expression e has

Var
$$\frac{x:\sigma\in\Gamma}{(Q)\,\Gamma\vdash x:\sigma}$$

App
$$\frac{(Q)\,\Gamma\vdash e_1:\tau_2\to\tau\quad(Q)\,\Gamma\vdash e_2:\tau_2}{(Q)\,\Gamma\vdash e_1\ e_2:\tau}$$

Fun
$$\frac{(Q)\,\Gamma,x:\tau_1\vdash e:\tau_2}{(Q)\,\Gamma\vdash\lambda x.e:\tau_1\to\tau_2}$$

Let
$$\frac{(Q)\,\Gamma\vdash e_1:\sigma_1\quad(Q)\,\Gamma,x:\sigma_1\vdash e_2:\sigma_2}{(Q)\,\Gamma\vdash\mathbf{let}\ x=e_1\ \mathbf{in}\ e_2:\sigma_2}$$

Gen
$$\frac{(Q,\alpha\diamond\sigma_1)\,\Gamma\vdash e:\sigma_2\quad\alpha\notin\mathsf{ftv}(\Gamma)}{(Q)\,\Gamma\vdash e:\forall(\alpha\diamond\sigma_1).\,\sigma_2}$$

Inst
$$\frac{(Q)\,\Gamma\vdash e:\sigma_1\quad(Q)\ \sigma_1\sqsubseteq\sigma_2}{(Q)\,\Gamma\vdash e:\sigma_2}$$

Ann
$$\frac{(Q)\,\Gamma\vdash e:\sigma_1\quad(Q)\,\sigma\mathrel{\sqsubseteq\!\!\!\!\!\!-}\sigma_1}{(Q)\,\Gamma\vdash(e::\sigma):\sigma}$$

Figure 2. Type rules

type σ. The type environment Γ contains the types of all the free *term* variables in e. The prefix Q contains the bounds of all the free *type* variables in Γ, σ, and e[1]. In Hindley-Milner, the prefix Q is usually left implicit since all the bounds of the free type variables are always unconstrained.

The application rule App applies a function to a monomorphic argument. Remarkably, this is still expressive enought to apply functions to polymorphic arguments since the prefix Q can contain a polymorphic bound of a (monomorphic) type variable. As an example, we derive the type for the application *poly id*, where σ_{id} stands for the type $\forall\alpha.\,\alpha\to\alpha$, τ for $(Int, Bool)$, and $\sigma_{id}\to\tau$ as a shorthand for $\forall(\beta=\sigma_{id}).\,\beta\to\tau$.

$$\dfrac{\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{\begin{array}{c}\dfrac{poly:\sigma_{id}\to\tau\in\Gamma}{(\forall\alpha=\sigma_{id})\,\Gamma\vdash poly:\sigma_{id}\to\tau}\\(\forall\alpha=\sigma_{id})\ \ \sigma_{id}\to\tau\sqsubseteq\alpha\to\tau\end{array}}{(\forall\alpha=\sigma_{id})\,\Gamma\vdash poly:\alpha\to\tau}\quad\dfrac{\begin{array}{c}\dfrac{id:\sigma_{id}\in\Gamma}{(\forall\alpha=\sigma_{id})\,\Gamma\vdash id:\sigma_{id}}\\(\forall\alpha=\sigma_{id})\ \ \sigma_{id}\sqsubseteq\alpha\end{array}}{(\forall\alpha=\sigma_{id})\,\Gamma\vdash id:\alpha}}{(\forall\alpha=\sigma_{id})\,\Gamma\vdash poly\ id:\tau}\end{array}}{()\,\Gamma\vdash poly\ id:\forall(\alpha=\sigma_{id}).\,\tau}}{}\\()\ \ \forall(\alpha=\sigma_{id}).\,\tau\sqsubseteq\tau\end{array}}{()\,\Gamma\vdash poly\ id:(Int,Bool)}$$

In the above derivation it is essential that we can instantiate σ_{id} to α under the prefix $(\forall\alpha=\sigma_{id})$. The instantiation rule Inst is used twice to instantiate both the argument type of *poly* and *id* to α after which can we use the application rule App. Finally, the generalization rule Gen is used to remove the assumption from the prefix, and instantiation is applied to remove the (now) dead quantifier, deriving $(Int, Bool)$.

The annotation rule Ann defines the type rule for type annotations, using the abstraction relation $\mathrel{\sqsubseteq\!\!\!\!\!\!-}$. We define this relation (together with instantiation) in the next section. Note that MLF has just a single rule for type annotations, and there is no need for special type propagation rules. In particular, a lambda bound annotation $\lambda(x::\sigma).e$ is syntactic sugar for

$$\lambda x.\ \mathbf{let}\ x=(x::\sigma)\ \mathbf{in}\ e$$

As such, the essence of MLF is contained in the instance relation ($\sqsubseteq$) in rule Inst, and the abstraction relation ($\mathrel{\sqsubseteq\!\!\!\!\!\!-}$) in the annotation rule Ann. In the next section, we formally define these relations and explain the annotation rule in detail.

R-Trans
$$\frac{(Q)\ \ \sigma_1\oplus\sigma_2\quad(Q)\ \ \sigma_2\oplus\sigma_3}{(Q)\ \ \sigma_1\oplus\sigma_3}$$

R-Prefix
$$\frac{(Q,\alpha\diamond\sigma)\ \ \sigma_1\oplus\sigma_2\quad\alpha\notin\mathsf{dom}(Q)}{(Q)\ \ \forall(\alpha\diamond\sigma).\,\sigma_1\oplus\forall(\alpha\diamond\sigma).\,\sigma_2}$$

Figure 3. General rules, where $\oplus$ stands for $\equiv$ (equivalence), $\mathrel{\sqsubseteq\!\!\!\!\!\!-}$ (abstraction), or $\sqsubseteq$ (instantiation).

Eq-Refl
$$(Q)\ \ \sigma\equiv\sigma$$

Eq-Var
$$(Q)\ \ \forall(\alpha\diamond\sigma).\,\alpha\equiv\sigma$$

Eq-Free
$$\frac{\alpha\notin\mathsf{ftv}(\sigma_2)}{(Q)\ \ \forall(\alpha\diamond\sigma_1).\,\sigma_2\equiv\sigma_2}$$

Eq-Mono
$$\frac{\forall(\alpha\diamond\sigma_0)\in Q\quad(Q)\ \sigma_0\equiv\tau_0}{(Q)\ \ \tau\equiv\tau[\alpha\mapsto\tau_0]}$$

Eq-Comm
$$\frac{\alpha_1\neq\alpha_2\quad\alpha_1\notin\mathsf{ftv}(\sigma_2)\quad\alpha_2\notin\mathsf{ftv}(\sigma_1)}{(Q)\ \ \forall(\alpha_1\diamond_1\sigma_1)(\alpha_2\diamond_2\sigma_2).\,\sigma\equiv\forall(\alpha_2\diamond_2\sigma_2)(\alpha_1\diamond_1\sigma_1).\,\sigma}$$

Eq-Context
$$\frac{(Q)\ \ \sigma_1\equiv\sigma_2}{(Q)\ \ \forall(\alpha\diamond\sigma_1).\,\sigma\equiv\forall(\alpha\diamond\sigma_2).\,\sigma}$$

Figure 4. Equivalence

A-Equiv
$$\frac{(Q)\ \ \sigma_1\equiv\sigma_2}{(Q)\ \ \sigma_1\mathrel{\sqsubseteq\!\!\!\!\!\!-}\sigma_2}$$

A-Hyp
$$\frac{\forall(\alpha=\sigma)\in Q}{(Q)\ \ \sigma\mathrel{\sqsubseteq\!\!\!\!\!\!-}\alpha}$$

A-Context
$$\frac{(Q)\ \ \sigma_1\mathrel{\sqsubseteq\!\!\!\!\!\!-}\sigma_2}{(Q)\ \ \forall(\alpha=\sigma_1).\,\sigma\mathrel{\sqsubseteq\!\!\!\!\!\!-}\forall(\alpha=\sigma_2).\,\sigma}$$

Figure 5. Abstraction

I-Bottom
$$(Q)\ \ \bot\sqsubseteq\sigma$$

I-Abstract
$$\frac{(Q)\ \ \sigma_1\mathrel{\sqsubseteq\!\!\!\!\!\!-}\sigma_2}{(Q)\ \ \sigma_1\sqsubseteq\sigma_2}$$

I-Hyp
$$\frac{\forall(\alpha\geqslant\sigma)\in Q}{(Q)\ \ \sigma\sqsubseteq\alpha}$$

I-Rigid
$$(Q)\ \ \forall(\alpha\geqslant\sigma_1).\,\sigma_2\sqsubseteq\forall(\alpha=\sigma_1).\,\sigma_2$$

I-Context
$$\frac{(Q)\ \ \sigma_1\sqsubseteq\sigma_2}{(Q)\ \ \forall(\alpha\geqslant\sigma_1).\,\sigma\sqsubseteq\forall(\alpha\geqslant\sigma_2).\,\sigma}$$

Figure 6. Instance

2.4 Instantiation, abstraction, and equivalence

Figure 6 formally defines the *instance* relation ($\sqsubseteq$). Through rule I-Abstract, the instance relation includes the *abstraction* relation ($\mathrel{\sqsubseteq\!\!\!\!\!\!-}$) defined in Figure 5. Similarly, through rule A-Equiv, the

[1] due to annotations.

$$
\begin{array}{lll}
\mathsf{nf}(\tau) & = \tau & \\
\mathsf{nf}(\bot) & = \bot & \\
\mathsf{nf}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \mathsf{nf}(\sigma_2) & \text{iff } \alpha \notin \mathsf{ftv}(\sigma_2) \\
\mathsf{nf}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \mathsf{nf}(\sigma_1) & \text{iff } \mathsf{nf}(\sigma_2) = \alpha \\
\mathsf{nf}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \mathsf{nf}(\sigma_2[\alpha \mapsto \tau]) & \text{iff } \mathsf{nf}(\sigma_1) = \tau \\
\mathsf{nf}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \forall(\alpha\diamond\mathsf{nf}(\sigma_1)).\,\mathsf{nf}(\sigma_2) &
\end{array}
$$

Figure 7. Normal form

abstraction relation includes the *equivalence* relation ($\equiv$) defined in Figure 4 and we have the following inclusion: ($\sqsubseteq$) $\subseteq$ ($\mathrel{\sqsubset\!\!\!\!-}$) $\subseteq$ ($\equiv$).

All three relations also include the general rules given in Figure 3. That is, all relations are transitive through R-Trans, and all rules can be applied under a common prefix through rule R-Prefix. Note that the condition $\alpha \notin \mathsf{dom}(Q)$ in R-Prefix can always be satisfied through alpha renaming.

2.4.1 Equivalence

The equivalence relation in Figure 4 defines an equivalence on types that abstracts from syntactical artifacts, like the order of the quantifiers. For example, rule Eq-Free states that unbound quantifiers are not meaningful, while rule Eq-Comm allows rearrangement of independent quantifiers. Rule Eq-Mono is the only rule that reads information from the prefix, and inlines monomorphic bounds. For example, assuming $Q' = (\forall(\alpha\diamond Int)$, we can derive:

$$
\frac{\dfrac{\forall(\alpha\diamond Int) \in (QQ') \quad (QQ')\ \ Int \equiv Int}{(Q, \forall(\alpha\diamond Int))\ \ \alpha \to \alpha \equiv Int \to Int} \qquad \alpha \notin \mathsf{dom}(Q)}{(Q)\ \ \forall(\alpha\diamond Int).\,\alpha \to \alpha \equiv \forall(\alpha\diamond Int).\,Int \to Int}
$$

Using transitivity R-Trans and rule Eq-Free, we can now conclude that $\forall(\alpha\diamond Int).\,\alpha \to \alpha$ is equivalent to $Int \to Int$. For monomorphic types it does not matter whether the bound is flexible or rigid.

The context rule Eq-Context states we can substitute equivalent types under any bound (rigid or flexible). For example, we can derive $\forall(\beta = \forall(\alpha \geqslant \sigma_{id}).\,\alpha).\,[\beta] \equiv \forall(\beta = \sigma_{id}).\,[\beta]$ by the context rule and Eq-Var.

2.4.2 Normal form

To abstract away from syntactical artifacts, Figure 7 defines a *normal form* function $\mathsf{nf}(\cdot)$ that maps equivalent types to the same normal form up to rearrangement of their quantifiers (Le Botlan 2004). The normal form of a type is always equivalent to that type, i.e. $\mathsf{nf}(\sigma) \equiv \sigma$.

2.4.3 Abstraction

The abstraction relation in Figure 5 extends the equivalence relation. In particular, it can read information from the prefix by the abstraction hypothesis rule A-Hyp. If a quantifier α is rigidly bound to a type σ in the prefix, we can *abstract* a type σ by α. Note that abstraction is not symmetric, and the opposite direction is not allowed. This is essential for type inference since we can infer that a given polymorphic type can be abstracted but we cannot infer the other direction.

The abstraction relation nicely interacts with the type rules for lambda expression (Fun) and type annotations (Ann) in Figure 2. The Fun rule requires the lambda bound argument x to have a *monomorphic* type, just like Hindley-Milner – but how are we able to type a function like *poly* that requires a polymorphic argument?

The key to typing such function is to give the argument x a monomorphic type α that is rigidly bound in the prefix to the polymorphic type $\forall(\alpha = \sigma)$. The polymorphic type of α is later *revealed* by the type annotation through A-Hyp. Take for example the desugared version of *poly*:

$$
poly = \lambda f_0.\ \mathbf{let}\ f = (f_0 :: \sigma_{id})\ \mathbf{in}\ (f\ 1, f\ \ True)
$$

We assume that e stands for the body of the lambda expression. The typing derivation for *poly* uses generalization to introduce the polymorphic bound, and assigns a monomorphic type α to the type of the lambda bound argument f_0:

$$
\frac{\dfrac{\dfrac{\vdots}{(Q, \forall(\alpha = \sigma_{id}))\ \Gamma, f_0 : \alpha \vdash e : (Int, Bool)}}{(Q, \forall(\alpha = \sigma_{id}))\ \Gamma \vdash \lambda f_0.e : \alpha \to (Int, Bool)} \quad \alpha \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash \lambda f_0.e : \forall(\alpha = \sigma_{id})\ (\alpha \to (Int, Bool))}
$$

Given this environment, we are now able to type the body of the lambda expression. In particular, we can use f with the polymorphic type σ_{id} in the body of the $\mathbf{let}$ since we can derive that the expression $(f_0 :: \sigma_{id})$ has type σ_{id}:

$$
\frac{\dfrac{f_0 : \alpha \in (\Gamma, f_0 : \alpha)}{(...)\ \Gamma, f_0 : \alpha \vdash f_0 : \alpha} \qquad \dfrac{\forall(\alpha = \sigma_{id}) \in (Q, \forall(\alpha = \sigma_{id}))}{(Q, \forall(\alpha = \sigma_{id}))\ \ \sigma_{id} \mathrel{\sqsubset\!\!\!\!-} \alpha}}{(Q, \forall(\alpha = \sigma_{id}))\ \Gamma, f_0 : \alpha \vdash (f_0 :: \sigma_{id}) : \sigma_{id}}
$$

The rule A-Hyp applies here directly, and the type annotation rule reveals the polymorphic type of α. Note that the annotation is required since we cannot instantiate rigidly bound type variables, i.e. we cannot derive $(\forall(\alpha = \sigma_{id}))\ \ \alpha \sqsubseteq \sigma_{id}$.

2.4.4 Instantiation

The instantiation relation in Figure 6 extends abstraction and equivalence through rule I-Abstract. The rule I-Bottom states that the most polymorphic type $\bot$ can be instantiated to any other type. While abstraction can use rigid bounds from the prefix, instantiation can use flexible bounds from the prefix through the hypothesis rule I-Hyp. If the prefix contains a quantifier $\forall(\alpha \geqslant \sigma)$ we know that α is an instance of σ and therefore we can always safely instantiate a type σ to α.

The context rule I-Context works *only* under flexible bounds. The context rule works nicely with I-Bottom to do standard instantiation of unconstrained bounds. For example:

$$
\begin{array}{l}
\forall\alpha.\,\alpha \to \alpha \\
\ = \{\ \text{sugar}\ \ \} \\
\forall(\alpha \geqslant \bot).\,\alpha \to \alpha \\
\ \sqsubseteq \{\ \text{I-Context, I-Bottom}\ \ \} \\
\forall(\alpha \geqslant Int).\,\alpha \to \alpha \\
\ \sqsubseteq \{\ \text{I-Abstract, A-Equiv, Eq-Prefix, Eq-Mono, Eq-Free}\ \ \} \\
Int \to Int
\end{array}
$$

3. From MLF to System F

In this section, we will develop a type directed translation from MLF terms to System F terms. Such translation is very useful in practice as compilers that use MLF type inference, can use System F terms as their intermediate language. More importantly, such translation gives us a better understanding of the expressiveness and operational behaviour of MLF programs. In particular,

- It is widely believed that a System F translation of impredicative types may need to traverse structures at runtime to apply coercion terms, where further instantiations can even lead to multiple traversals. The translation we develop in this section shows that this is not the case for MLF. Instead we show that we can always build the coerced terms directly in a single pass without any traversals.

- The System F translation leads us naturally to a restriction of MLF where all bound values have a System F type and where the System F translation is particularly efficient.
- Extending MLF with other type system extensions usually requires a good understanding of the corresponding evidence translation. For example, to use MLF as a basis for languages that support qualified types, like Haskell, it is important to have an evidence translation of predicates which fits seamlessly with the presented System F translation.

3.1 Translating flexible bounds

The main difficulty in giving a System F translation of MLF terms occurs when the MLF terms have a type that goes beyond System F, i.e. polymorphic types that are flexibly bound. Consider:

$$\mathbf{let}\ \mathit{ids} = [\mathit{id}]\ \mathbf{in}\ (\mathit{polyL}\ \mathit{ids}, \mathit{ids} +\!\!+ [\mathit{inc}])$$

where ids has type $\forall(\alpha \geqslant \sigma_{id}).\, [\alpha]$. Unfortunately, there is no direct System F equivalent of this type, and in a naïve System F translation, ids can have two different System F types, namely $\forall\alpha \cdot [\alpha \to \alpha]$ or $[\forall\alpha \cdot \alpha \to \alpha]$. The first type is required by the append, while the second type is required by the polyL application. Effectively, MLF delays the instantiation and later ids is instantiated in two fundamentally different ways. In contrast, in System F this choice has to be made up front.

A potential way of translating this program to System F is to assign the type $[\forall\alpha \cdot \alpha \to \alpha]$ to ids, and explicitly coerce the type by mapping a coercion function over all element types that instantiates them $\mathit{Int} \to \mathit{Int}$ functions. In practice though this solution is unacceptable since the type inferencer might suddenly introduce coercion terms that need to traverse arbitrary structures at runtime! This is often seen as one of the fatal properties of impredicative type systems (Peyton Jones et al. 2007) and boxy type inference (Vytiniotis et al. 2006) was specifically designed to avoid this behaviour by rejecting the above program.

Fortunately, there exists a very elegant solution for MLF, first described by Leijen and Löh (2005): for every non-trivial flexible bound, we pass in evidence on how to instantiate the term – effectively delaying the instantiation to the call site. A flexible bound $\forall(\alpha \geqslant \sigma)$ states that α can be any instance of σ. Each such bound is translated into an explicit witness function of type $\sigma \to \alpha$ that transforms any type σ into its instantiation α. For example, the System F translation of the $\mathit{ids} :: \forall(\alpha \geqslant \sigma_{id}).\, [\alpha]$ value is:

$$\begin{array}{l} \mathit{ids} :: \forall\alpha \cdot (\sigma_{id} \to \alpha) \to [\alpha] \\ \mathit{ids} = \Lambda\alpha \cdot \lambda(\mathsf{v} :: \sigma_{id} \to \alpha) \cdot \mathit{single}\ \alpha\ (\mathsf{v}\ \mathit{id}) \end{array}$$

where we assume a function $\mathit{single} :: \forall\alpha \cdot \alpha \to [\alpha]$ that creates a singleton list. Note in particular how the witness v takes the identity function to type α. In the body of the $\mathbf{let}$ expression, ids can now instantiated to two different types. For polyL, we simply pass an identity function that leaves the elements polymorphic:

$$\mathit{polyL}\ \sigma_{id}\ (\mathit{ids}\ \sigma_{id}\ (\lambda(\mathsf{x} :: \sigma_{id}) \cdot \mathsf{x}))$$

For the append though, the evidence instantiates all elements to $\mathit{Int} \to \mathit{Int}$ functions:

$$\mathit{ids}\ (\mathit{Int} \to \mathit{Int})\ (\lambda(\mathsf{x} :: \sigma_{id}) \cdot \mathsf{x}\ \mathit{Int})$$

For the above example, the evidence just changes the type parameters. In the presence of qualified types evidence terms can change runtime terms too. For example, the term $[\mathit{inc}]$ could have the type $\forall(\alpha \geqslant \sigma).\, [\alpha]$ where σ equals $\forall\alpha.\, \mathit{Num}\ \alpha \Rightarrow \alpha \to \alpha$. The System F translation passes an explicit runtime dictionary as evidence for $\mathit{Num}\ \alpha$ to such function. In that case the witness for ids transforms a term of type σ_{id} to σ by adding the dictionary argument:

$$\mathit{ids}\ \sigma\ (\lambda(\mathsf{x} :: \sigma_{id}) \cdot (\Lambda\alpha \cdot \lambda(\mathit{num} :: \mathit{Num}\ \alpha) \cdot \mathsf{x}\ \alpha))$$

$$\sigma^* = \mathsf{ft}(\mathsf{nf}(\sigma))$$

$$\begin{array}{ll} \mathsf{ft}(\tau) & = \tau \\ \mathsf{ft}(\bot) & = \forall\alpha \cdot \alpha \\ \mathsf{ft}(\forall(\alpha = \sigma_1).\, \sigma_2) & = \mathsf{ft}(\sigma_2)[\alpha \mapsto \mathsf{ft}(\sigma_1)] \\ \mathsf{ft}(\forall(\alpha \geqslant \bot).\, \sigma) & = \forall\alpha \cdot \mathsf{ft}(\sigma) \\ \mathsf{ft}(\forall(\alpha \geqslant \sigma_1).\, \sigma_2) & = \forall\alpha \cdot (\mathsf{ft}(\sigma_1) \to \alpha) \to \mathsf{ft}(\sigma_2) \end{array}$$

Figure 8. MLF types to system-F types.

3.2 Translation of rigid bounds

Figure 8 defines the translation of MLF types to System F types. The translation works on normal forms to discard trivial bounds such as monotype bounds or unbound quantifiers. The translation as given is very satisfactory as the only evidence passed is for non-trivial flexible polymorphic bounds – exactly those bounds that were needed to allow type inference with first-class polymorphism in the first place.

Rigid bounds are simply inlined to get the corresponding System F type. Unfortunately, this prevents us also from defining a direct type directed translation on the standard MLF rules. In particular, since the equivalence and abstraction relation can work under rigid bounds (through Eq-Context and A-Context), it is no longer the case that when two types are equivalent, that their System F types are equivalent, i.e. $\sigma_1 \equiv \sigma_2$ does not imply $\sigma_1^* = \sigma_2^*$! Take for example the lists $xs_1 :: \forall(\gamma = \forall\alpha\beta.\, \alpha \to \beta \to \alpha).\, [\gamma]$ and $xs_2 :: \forall(\gamma = \forall\beta\alpha.\, \alpha \to \beta \to \alpha).\, [\gamma]$. The term

$$\mathbf{if}\ \mathit{True}\ \mathbf{then}\ xs_1\ \mathbf{else}\ xs_2$$

is well-typed in MLF since the types of xs_1 and xs_2 are equivalent,

$$\begin{array}{l} \forall(\gamma = \forall\alpha\beta.\, \alpha \to \beta \to \alpha).\, [\gamma] \\ \quad \equiv \{\ \textsc{Eq-Context}, \textsc{Eq-Comm}\ \} \\ \forall(\gamma = \forall\beta\alpha.\, \alpha \to \beta \to \alpha).\, [\gamma] \end{array}$$

However, according to Figure 8, the corresponding System F types are not equal, and we end up with an ill-typed System F term! Nevertheless, not using an evidence translation for rigid bounds is most important in practice to avoid many 'trivial' coercions between equivalent types and we consider this a serious weakness of the earlier approach by Leijen and Löh (2005). Furthermore, as we will see in Section 4, the direct translation of rigid bounds enables a variation of MLF that never passes any evidence which in turns enables us gain insight in the relation between boxy type inference and MLF.

Thus, the challenge is to set up the typed translation in such a way that the above situation is prevented and where rigid bounds never need evidence translation, and we are going to tackle it head on in the following sections.

3.3 Canonical equivalence and abstraction

Both the equivalence and abstraction relation can be applied under rigid bounds through their context rules (Eq-Context and A-Context). Therefore, to ensure that the System F type of a rigid bound never changes, we need to define both a restricted equivalence and restricted abstraction relation where equivalent types have equal System F translations.

The only rule that prevents equal System F types is Eq-Comm when quantifiers are rearranged. As we see later, we cannot just remove the rearrangement rule since abstraction must be able to rearrange rigid binders. Fortunately, since the System F translation inlines rigid bounds, rearranging rigid binders does not change the translated System F type and such rearrangement is sound.

Definition 3.3.a (*Canonical equivalence*): We define *canonical equivalence*, written as $\equiv^{\mathsf{c}}$, as equal to the MLF equivalence relation of Figure 4, but the EQ-COMM rule is restricted to rearrangement of rigid binders:

CEQ-COMM

$$\frac{\alpha_1 \neq \alpha_2 \quad \alpha_1 \notin \mathsf{ftv}(\sigma_2) \quad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \ \forall(\alpha_1 = \sigma_1)(\alpha_2 \diamond \sigma_2).\, \sigma \equiv^{\mathsf{c}} \forall(\alpha_2 \diamond \sigma_2)(\alpha_1 = \sigma_1).\, \sigma}$$

Similarly, we define canonical abstraction as an extension of canonical equivalence.

Definition 3.3.b (*Canonical abstraction*): Canonical abstraction, written as $\mathrel{\sqsubseteq\!\!\!\!-}^{\mathsf{c}}$, is equal to the MLF abstraction relation in Figure 5 but the rule A-EQUIV is restricted to canonical equivalence:

$$\text{CA-EQUIV}\ \frac{(Q)\ \ \sigma_1 \equiv^{\mathsf{c}} \sigma_2}{(Q)\ \ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-}^{\mathsf{c}} \sigma_2}$$

Canonical equivalence satisfies many properties of MLF equivalence, for example $\mathsf{nf}(\sigma) \equiv^{\mathsf{c}} \sigma$ holds. Moreover, canonically equivalent types have equal System F types. To make this notion precise, we first define a System F substitution extraction.

Definition 3.3.c (*Substitution extraction*): The extraction of a monotype substitution from a prefix Q, written as Q^{θ}, is defined as:

$$\begin{array}{lll}
(\varnothing)^{\theta} & = id & \\
(\alpha \diamond \sigma, Q)^{\theta} & = [\alpha \mapsto \tau] \circ Q^{\theta} & \text{iff } \mathsf{nf}(\sigma) = \tau \\
(\alpha \diamond \sigma, Q)^{\theta} & = Q^{\theta} & \text{otherwise}
\end{array}$$

Similarly, we can also define the System F substitution extracted from a prefix Q, written as $Q^{\ominus}$:

$$\begin{array}{lll}
(\varnothing)^{\ominus} & = id & \\
(\alpha = \sigma, Q)^{\ominus} & = [\alpha \mapsto \sigma^*] \circ Q^{\ominus} & \\
(\alpha \geqslant \sigma, Q)^{\ominus} & = [\alpha \mapsto \tau^*] \circ Q^{\ominus} & \text{iff } \mathsf{nf}(\sigma) = \tau \\
(\alpha \geqslant \sigma, Q)^{\ominus} & = Q^{\ominus} & \text{otherwise}
\end{array}$$

Note that $(Q_1 Q_2)^{\ominus} = Q_1^{\ominus} \circ Q_2^{\ominus}$ for any well formed prefix $Q_1 Q_2$. Futhermore, the domain of the substitution is a subset of domain of the prefix: $\mathsf{dom}(Q^{\ominus}) \subseteq \mathsf{dom}(Q)$. Using System F substitutions, we can now state our main theorems of canonical equivalence and abstraction:

Theorem 3.3.d (*Canonically equivalent types have equal System F types*):

$$(Q)\ \ \sigma_1 \equiv^{\mathsf{c}} \sigma_2 \ \Rightarrow\ Q^{\ominus}(\sigma_1^*) = Q^{\ominus}(\sigma_2^*)$$

Theorem 3.3.e (*Canonical abstraction has equal System F types*):

$$(Q)\ \ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-}^{\mathsf{c}} \sigma_2 \ \Rightarrow\ Q^{\ominus}(\sigma_1^*) = Q^{\ominus}(\sigma_2^*)$$

Both theorems are proved by straightforward induction over the rules of canonical equivalence and abstraction.

3.4 Canonical instance with System F translation

The canonical instance relation, written as $\sqsubseteq^{\mathsf{c}}$, is defined in Figure 9 and Figure 10. There are two differences from normal MLF instantiation (defined in Figure 6): rule CI-ABSTRACT uses canonical abstraction, and rule CI-COMM is reintroduces the ability to rearrange quantifiers with flexible bounds. Furthermore, the canonical instantiation now derives a System F witness term for the instantiation.

The derivation $(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \leadsto \mathsf{f}$ states that σ_2 is a canonical instance of σ_1 under prefix Q, where the derived System F witness f has type $\sigma_1^* \to \sigma_2^*$, i.e. it instantiates a term with System F type σ_1^* into a term with type σ_2^*.

To describe such witness functions conveniently, we use the $\bullet$ notation. An expression e with a hole $\bullet$ stands for a function $\lambda x \cdot e$ where $\bullet$ is replaced by the fresh variable x. For example,

$$\text{CI-TRANS}\quad \frac{(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \leadsto \mathsf{f}_1 \quad (Q)\ \ \sigma_2 \sqsubseteq^{\mathsf{c}} \sigma_3 \leadsto \mathsf{f}_2}{(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_3 \leadsto \mathsf{f}_2\ (\mathsf{f}_1\ \bullet)}$$

$$\text{CI-PREFIX}\quad \frac{(Q, \alpha \diamond_{\mathsf{v}} \sigma)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \leadsto \mathsf{f} \quad \alpha \notin \mathsf{dom}(Q)}{(Q)\ \ \forall(\alpha \diamond_{\mathsf{v}} \sigma).\, \sigma_1 \sqsubseteq^{\mathsf{c}} \forall(\alpha \diamond_{\mathsf{v}} \sigma).\, \sigma_2}$$
$$\leadsto \mathsf{gen}[\forall(\alpha \diamond_{\mathsf{v}} \sigma).\, \sigma_2]\ (\mathsf{f}\ (\mathsf{inst}[\forall(\alpha \diamond_{\mathsf{v}} \sigma).\, \sigma_1]\ \bullet))$$

$$\text{CI-CONTEXT}\quad \frac{(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \leadsto \mathsf{f}}{(Q)\ \ \forall(\alpha \geqslant_{\mathsf{v}} \sigma_1).\, \sigma \sqsubseteq^{\mathsf{c}} \forall(\alpha \geqslant_{\mathsf{w}} \sigma_2).\, \sigma}$$
$$\leadsto \mathsf{gen}[\forall(\alpha \geqslant_{\mathsf{w}} \sigma_2).\, \sigma]$$
$$(\mathbf{let}\ \mathsf{v}\ \mathsf{e} = \mathsf{w}\ (\mathsf{f}\ \mathsf{e})\ \mathbf{in}\ \mathsf{inst}[\forall(\alpha \geqslant_{\mathsf{v}} \sigma_1).\, \sigma]\ \bullet)$$

Figure 9. General rules for canonical instance with evidence translation

$$\text{CI-BOTTOM}\quad (Q)\ \ \bot \sqsubseteq^{\mathsf{c}} \sigma \leadsto \bullet\ \sigma^*$$

$$\text{CI-ABSTRACT}\quad \frac{(Q)\ \ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-}^{\mathsf{c}} \sigma_2}{(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \leadsto \bullet}$$

$$\text{CI-HYP}\quad \frac{\forall(\alpha \geqslant_{\mathsf{v}} \sigma) \in Q}{(Q)\ \ \sigma \sqsubseteq^{\mathsf{c}} \alpha \leadsto \mathsf{v}\ \bullet}$$

$$\text{CI-RIGID}\quad (Q)\ \ \forall(\alpha \geqslant_{\mathsf{v}} \sigma_1).\, \sigma_2 \sqsubseteq^{\mathsf{c}} \forall(\alpha = \sigma_1).\, \sigma_2$$
$$\leadsto \mathsf{gen}[\forall(\alpha =_{\mathsf{v}} \sigma_1).\, \sigma_2]\ (\mathsf{inst}[\forall(\alpha \geqslant_{\mathsf{v}} \sigma_1).\, \sigma_2]\ \bullet)$$

CI-COMM

$$\frac{\alpha_1 \neq \alpha_2 \quad \alpha_1 \notin \mathsf{ftv}(\sigma_2) \quad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \ \forall(\alpha_1 \geqslant_{\mathsf{v}} \sigma_1)(\alpha_2 \geqslant_{\mathsf{w}} \sigma_2).\, \sigma \sqsubseteq^{\mathsf{c}} \forall(\alpha_2 \geqslant_{\mathsf{w}} \sigma_2)(\alpha_1 \geqslant_{\mathsf{v}} \sigma_1).\, \sigma}$$
$$\leadsto \mathsf{gen}[\forall(\alpha_2 \geqslant_{\mathsf{w}} \sigma_2).\, \forall(\alpha_1 \geqslant_{\mathsf{v}} \sigma_1).\, \sigma]$$
$$(\mathsf{inst}[\forall(\alpha_1 \geqslant_{\mathsf{v}} \sigma_1).\, \forall(\alpha_2 \geqslant_{\mathsf{w}} \sigma_2).\, \sigma]\ \bullet)$$

Figure 10. Canonical instance with evidence translation

$$\begin{array}{lcl}
\bullet & = & \lambda x \cdot x \\
\bullet\ \sigma^* & = & \lambda x \cdot x\ \sigma^* \\
\lambda y \cdot \bullet & = & \lambda x \cdot \lambda y \cdot x
\end{array}$$

The rule CI-BOTTOM instantiates the bottom type ($\bot$) to an arbitrary type σ. The evidence term thus needs to instantiate a term of type $\forall\alpha{\cdot}\alpha$ to σ^*, which corresponds to applying the type σ^* in System F, and the derived evidence is $\bullet\ \sigma^*$.

Rule CI-ABSTRACT is also straightforward. Due to Theorem 3.3.e, we know that σ_1 and σ_2 have equal System F types and that the term stays the same, i.e. the evidence is the identity: $\bullet$.

The evidence for rule CI-HYP needs to transform a type σ^* into α – but how should it do that if nothing is known about σ? The solution to this problem is to annotate each flexible bound $\forall(\alpha \geqslant_{\mathsf{v}} \sigma)$ in the prefix with a witness term v of type $\sigma^* \to \alpha$. One can easily check that these terms are always in scope (just like the type variables in the prefix). The rule CI-HYP simply applies the witness term to transform σ^* to α, i.e. $\mathsf{v}\ \bullet$.

Most of the other rules make use of two helper functions, $\mathsf{gen}[\cdot]$ and $\mathsf{inst}[\cdot]$, defined in Figure 11. These functions introduce evidence terms v for flexible bounds and remove trivial bounds.

The function $\mathsf{gen}[\forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2]$ takes a System F term of type σ_2^* and generalizes it to a term of type $(\forall(\alpha \diamond \sigma_1).\, \sigma_2)^*$. The $\mathsf{gen}[\cdot]$ function ensures that in all cases the type variable α and the evidence term v of type $\sigma_1^* \to \alpha$ are defined and in scope. For convenience we use $\mathbf{let}$ bindings to define v, even though System F does not contain them, and one can read those $\mathbf{let}$ bindings as a substitution on witness terms. For rigid and trivial bounds, $\mathsf{gen}[\cdot]$ just substitutes σ_1 for α, and the evidence term v becomes the identity.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{gen}[\forall(\alpha \diamond_{\mathsf{v}} \sigma).\, \sigma_0] :: \sigma_0^* \to (\forall(\alpha \diamond \sigma).\, \sigma_0)^*} \\
\multicolumn{2}{l}{\mathsf{gen}[\forall(\alpha =_{\mathsf{v}} \sigma).\, \sigma_0]} \\
\quad = (\mathbf{let}\ \mathsf{v\ e = e}\ \mathbf{in}\ \bullet)[\alpha \mapsto \sigma^*] & \\
\multicolumn{2}{l}{\mathsf{gen}[\forall(\alpha \geqslant_{\mathsf{v}} \sigma).\, \sigma_0]} \\
\quad = (\mathbf{let}\ \mathsf{v\ e = e}\ \mathbf{in}\ \bullet)[\alpha \mapsto \sigma^*] & \text{iff } \alpha \notin \mathsf{ftv}(\sigma_0) \\
\quad = (\mathbf{let}\ \mathsf{v\ e = e}\ \mathbf{in}\ \bullet)[\alpha \mapsto \sigma^*] & \text{iff } \mathsf{nf}(\sigma_0) = \alpha \\
\quad = (\mathbf{let}\ \mathsf{v\ e = e}\ \mathbf{in}\ \bullet)[\alpha \mapsto \sigma^*] & \text{iff } \mathsf{nf}(\sigma) = \tau \\
\quad = \Lambda\alpha \cdot \mathbf{let}\ \mathsf{v\ e = e}\ \alpha\ \mathbf{in}\ \bullet & \text{iff } \mathsf{nf}(\sigma) = \bot \\
\quad = \Lambda\alpha \cdot \lambda \mathsf{v} : \sigma^* \to \alpha \cdot \bullet & \text{otherwise}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{inst}[\forall(\alpha = \sigma).\, \sigma_0]} \\
\quad = \bullet & \\
\multicolumn{2}{l}{\mathsf{inst}[\forall(\alpha \geqslant_{\mathsf{v}} \sigma).\, \sigma_0]} \\
\quad = \bullet & \text{iff } \alpha \notin \mathsf{ftv}(\sigma_0) \\
\quad = \bullet & \text{iff } \mathsf{nf}(\sigma_0) = \alpha \\
\quad = \bullet & \text{iff } \mathsf{nf}(\sigma) = \tau \\
\quad = \bullet\ \alpha & \text{iff } \mathsf{nf}(\sigma) = \bot \\
\quad = \bullet\ \alpha\ \mathsf{v} & \text{otherwise}
\end{array}
$$

Figure 11. Canonical type generalization and instantiation

For an unconstrained bound $\forall(\alpha \geqslant \bot)$, a type lambda generalizes over α. For a non-trivial flexible bound, we not only generalize over the type α, but also bind the evidence v as an argument.

Dually, the function $\mathsf{inst}[\forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2]$ takes a System F term of type $(\forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2)^*$ and instantiates it to σ_2^* (or σ_1^* if $\mathsf{nf}(\sigma_2) = \alpha$), where it is assumed that α and v are in scope and where v has type $\sigma_1^* \to \alpha$. One can easily check that this is the case for all the uses of $\mathsf{inst}[\cdot]$. For rigid and trivial bounds nothing has to be done. For an unconstrained bound we instantiate the type argument to α, and for a non-trivial flexible bound we instantiate the type argument and pass in the evidence function v as an argument.

With these helper functions, the evidence translation for the other rules is straightforward. Rule CI-RIGID uses $\mathsf{inst}[\cdot]$ to instantiate the flexible quantifier and immediately applies $\mathsf{gen}[\cdot]$ with a rigid bound, which substitutes α for σ_1^*, and binds v to the identity. Another interesting rule is CI-CONTEXT. Since the bounded types themselves are instantiated, the types of the witness functions v and w are not equal and we need to build an *witness transformer*: using the evidence $\mathsf{f} :: \sigma_1^* \to \sigma_2^*$, we can define the old witness $\mathsf{v} :: \sigma_1^* \to \alpha$ in terms of the new witness $\mathsf{w} :: \sigma_2^* \to \alpha$, as $\mathsf{v} = \mathsf{w} \circ \mathsf{f}$.

To define soundness of the System F translation, we need to be able to refer to the witness terms in the prefix from the System F type environment, and we define the following operation to extract such environment from the prefix.

Definition 3.4.a (*Sytem-F environment extraction*): The environment extraction from a prefix Q is written as Q^Γ and defined as:

$$
\begin{array}{ll}
(\varnothing)^\Gamma & = \varnothing \\
(\alpha = \sigma, Q)^\Gamma & = Q^\Gamma \\
(\alpha \geqslant_{\mathsf{v}} \sigma, Q)^\Gamma & = \mathsf{v} : \sigma^* \to \alpha, Q^\Gamma
\end{array}
$$

Using environment extraction, we can state that the derived System F term is well-typed in System F. The typing relation of System F ($\Gamma \vdash^{\mathsf{F}} \mathsf{e} : \sigma$) is standard and we omit it here.

Theorem 3.4.b (*Soundness of System F translation*): If two types σ_1 and σ_2 are in a canonical instance relation under a prefix Q with a witness f, then the witness f is well typed in System F under the type environment extracted from Q, with type $\sigma_1^* \to \sigma_2^*$:

$$(Q)\ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \rightsquigarrow \mathsf{f} \;\Rightarrow\; Q^\ominus(Q^\Gamma) \vdash^{\mathsf{F}} \mathsf{f} : Q^\ominus(\sigma_1^* \to \sigma_2^*)$$

$$\text{CT-VAR}\quad \frac{x : \sigma \in \Gamma}{(Q)\ \Gamma \vdash^{\mathsf{c}} x : \sigma \rightsquigarrow \mathsf{x}}$$

$$\text{CT-APP}\quad \frac{\begin{array}{c}(Q)\ \Gamma \vdash^{\mathsf{c}} e_1 : \tau_2 \to \tau_1 \rightsquigarrow \mathsf{e}_1 \\ (Q)\ \Gamma \vdash^{\mathsf{c}} e_2 : \tau_2 \rightsquigarrow \mathsf{e}_2\end{array}}{(Q)\ \Gamma \vdash^{\mathsf{c}} e_1\ e_2 : \tau_1 \rightsquigarrow \mathsf{e}_1\ \mathsf{e}_2}$$

$$\text{CT-FUN}\quad \frac{(Q)\ \Gamma, x : \tau_1 \vdash^{\mathsf{c}} e : \tau_2 \rightsquigarrow \mathsf{e}}{(Q)\ \Gamma \vdash^{\mathsf{c}} \lambda x.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda(\mathsf{x} : \tau_1^*) \cdot \mathsf{e}}$$

$$\text{CT-LET}\quad \frac{\begin{array}{c}(Q)\ \Gamma \vdash^{\mathsf{c}} e_1 : \sigma_1 \rightsquigarrow \mathsf{e}_1 \\ (Q)\ \Gamma, x : \sigma_1 \vdash^{\mathsf{c}} e_2 : \sigma_2 \rightsquigarrow \mathsf{e}_2\end{array}}{\begin{array}{c}(Q)\ \Gamma \vdash^{\mathsf{c}}\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_2 \\ \rightsquigarrow (\lambda(\mathsf{x} : \sigma_1^*) \cdot \mathsf{e}_2)\ \mathsf{e}_1\end{array}}$$

$$\text{CT-GEN}\quad \frac{(Q, \alpha \diamond_{\mathsf{v}} \sigma_1)\ \Gamma \vdash^{\mathsf{c}} e : \sigma_2 \rightsquigarrow \mathsf{e} \quad \alpha \notin \mathsf{ftv}(\Gamma)}{\begin{array}{c}(Q)\ \Gamma \vdash^{\mathsf{c}} e : \forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2 \\ \rightsquigarrow \mathsf{gen}[\forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2]\ \mathsf{e}\end{array}}$$

$$\text{CT-INST}\quad \frac{(Q)\ \Gamma \vdash^{\mathsf{c}} e : \sigma_1 \rightsquigarrow \mathsf{e} \quad (Q)\ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \rightsquigarrow \mathsf{f}}{(Q)\ \Gamma \vdash^{\mathsf{c}} e : \sigma_2 \rightsquigarrow \mathsf{f\ e}}$$

$$\text{CT-ANN}\quad \frac{\begin{array}{c}(Q)\ \Gamma \vdash^{\mathsf{c}} e : \sigma_1 \rightsquigarrow \mathsf{e} \\ (Q)\ \sigma \sqsupseteq^{\mathsf{c}} \sigma_1\end{array}}{(Q)\ \Gamma \vdash^{\mathsf{c}} (e :: \sigma) : \sigma \rightsquigarrow \mathsf{e}}$$

Figure 12. Type rules with evidence translation

3.5 Canonical type rules with System F translation

Figure 12 defines canonical type rules for MLF that derive a corresponding System F term. The type rules are equivalent to the corresponding MLF type rules (see Figure 2) except for the use of canonical instance in CT-INST and canonical abstraction in CT-ANN. The expression $(Q)\ \Gamma \vdash^{\mathsf{c}} e : \sigma \rightsquigarrow \mathsf{e}$ states that under a prefix Q and type environment Γ, the expression e is well typed with type σ, and e is a corresponding System F term with type σ^*.

The type directed translation is straightforward. Rule CT-VAR simply returns x, since the condition $x : \sigma \in \Gamma$ implies that x has type σ^* under the System F environment Γ^*, (i.e. $\Gamma^* \vdash^{\mathsf{F}} \mathsf{x} : \sigma^*$). In CT-APP, e_1 and e_2 both have monotypes and their System F types are equivalent, and we can directly apply the corresponding System F terms too. The same holds for CT-FUN that directly translates into a corresponding lambda expression. Rule CT-LET is interesting as the System F translation uses a polymorphic lambda expression since System F does not contain **let** bindings.

Generalization in rule CT-GEN uses the $\mathsf{gen}[\cdot]$ function defined in Figure 11 to generalize the term e with type σ_2^* into $(\forall(\alpha \diamond_{\mathsf{v}} \sigma_1).\, \sigma_2)^*$. The instantiation rule (CT-INST) uses the witness function f of type $\sigma_1^* \to \sigma_2^*$, derived with canonical instance to transform the term e to a term of type σ_2^*. Finally, type annotations in CT-ANN use canonical abstraction. By Theorem 3.3.e, their System F types are equal and no translation is necessary.

Theorem 3.5.a (*Canonical type inference derives well typed System F terms*): If under a prefix Q and type environment Γ an expression e is well typed with type σ and a translated System F term e, then there exists a System F derivation such that under a System F type environment consisting of the environment extraction of Q (namely Q^Γ) and the translated environment Γ^*, the term e is well typed with type σ^*.

$$(Q)\ \Gamma \vdash^{\mathsf{c}} e : \sigma \rightsquigarrow \mathsf{e} \;\Rightarrow\; Q^\ominus(Q^\Gamma\Gamma^*) \vdash^{\mathsf{F}} Q^\ominus(\mathsf{e}) : Q^\ominus(\sigma^*)$$

Note that under an empty prefix this simplifies to:

$$()\ \Gamma \vdash^{\mathsf{c}} e : \sigma \rightsquigarrow \mathsf{e} \;\Rightarrow\; \Gamma^* \vdash^{\mathsf{F}} \mathsf{e} : \sigma^*$$

3.6 Soundness

By carefully changing the original equivalence, abstraction, and instance relation, we were able to derive a type directed and sound System F translation for instantiation terms. But at the same time we are no longer doing MLF instantiation but canonical instantiation, and we would like to establish that the new relations are sound and complete with respect to the MLF relations. Clearly, the canonical relations are sound with respect to MLF:

Theorem 3.6.a (*Canonical equivalence, abstraction, and instance are sound*):

$$\begin{array}{l}
(Q)\ \ \sigma_1 \equiv^{\mathsf{c}} \sigma_2 \ \Rightarrow\ (Q)\ \ \sigma_1 \equiv \sigma_2 \\
(Q)\ \ \sigma_1 \mathrel{\sqsubset\!\!\!\!-}^{\mathsf{c}} \sigma_2 \ \Rightarrow\ (Q)\ \ \sigma_1 \mathrel{\sqsubset\!\!\!\!-} \sigma_2 \\
(Q)\ \ \sigma_1 \sqsubseteq^{\mathsf{c}} \sigma_2 \ \Rightarrow\ (Q)\ \ \sigma_1 \sqsubseteq \sigma_2
\end{array}$$

Proof of Theorem 3.6.a: This is immediate since all the new relations are sub-relations of the MLF relations. Equivalence includes all the rules of canonical equivalence, and allows rearrangement of flexible binders. Canonical abstraction equals MLF abstraction but restricts equivalence to canonical equivalence. Finally, canonical instance includes an extra rule (CI-COMM) to rearrange flexible binders, but this is also included in MLF instantiation by EQ-COMM (via A-EQUIV and I-ABSTRACT). □

As a corollary, it follows that the canonical type rules are also sound since the only difference with the MLF type rules is the use of canonical instance and abstraction.

Theorem 3.6.b (*Canonical type inference is sound*):

$$(Q)\,\Gamma \vdash^{\mathsf{c}} e : \sigma \leadsto \mathsf{e} \ \Rightarrow\ (Q)\,\Gamma \vdash e : \sigma$$

3.7 Completeness

Completeness of the type rules cannot be established directly since the canonical relations are not complete with respect to the MLF relations. Take for example the following equivalent types:

$$(Q)\ \ \forall\alpha\beta.\,\alpha \to \beta \to \alpha \equiv \forall\beta\alpha.\,\alpha \to \beta \to \alpha$$

These types are equivalent by using EQ-COMM to rearrange the flexible binders. Since canonical equivalence restricts rearrangement to rigid binders, the above types are *not* canonically equivalent (as it should, since the corresponding System F types are different). This seems a fatal flaw of our new rules: incompleteness would imply that our new rules can only be used to type a subset of the programs accepted by MLF.

Fortunately, the rules are incomplete only with respect to the order of the quantifiers. We can show that the canonical relations are complete with respect to the MLF relations under specific invariants where rigid types are in *canonical form*. In the next section we formalize this notion and establish a completeness theorem.

3.7.1 Canonical form

Figure 13 defines the *canonical form* of a type σ, written as $\mathsf{can}(\sigma)$. The canonical form is a rearrangement of the normal form, where the position of a quantifier is uniquely determined by the occurrences of the bound type variable. Canonical forms satisfy the following properties:

Properties 3.7.a
i. $(Q)\ \ \sigma_1 \equiv \sigma_2 \ \Leftrightarrow\ Q^{\theta}(\mathsf{can}(\sigma_1)) = Q^{\theta}(\mathsf{can}(\sigma_2))$
ii. $\mathsf{ftv}(\mathsf{can}(\sigma)) = \mathsf{ftv}(\sigma)$

As a corrollary of Property 3.7.a.i, we have under an unconstrained prefix that equivalent types have equal canonical forms:

$$\sigma_1 \equiv \sigma_2 \ \Leftrightarrow\ \mathsf{can}(\sigma_1) = \mathsf{can}(\sigma_2)$$

The rearranged form makes use of the insertion function $\mathsf{ins}(\cdot,\cdot)$ to insert quantifiers in their canonical position. Insertion of into a monomorphic type or $\bot$ has no effect, but otherwise quantifiers may be rearranged. The side condition of the last insertion case can always be satisfied by simple alpha renaming. The relative order between quantifiers is now uniquely determined by the *position* of their bound variable in the type, written as $\alpha\#\sigma$, and is defined in Figure 14.

$$\mathsf{can}(\sigma) = \mathsf{rf}(\mathsf{nf}(\sigma))$$

$$\begin{array}{ll}
\mathsf{rf}(\tau) & = \tau \\
\mathsf{rf}(\bot) & = \bot \\
\mathsf{rf}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \mathsf{ins}(\alpha\diamond\mathsf{rf}(\sigma_1), \mathsf{rf}(\sigma_2))
\end{array}$$

$$\begin{array}{l}
\mathsf{ins}(\alpha\diamond\sigma, \tau) \quad\ \ = \forall(\alpha\diamond\sigma).\,\tau \\
\mathsf{ins}(\alpha\diamond\sigma, \bot) \quad\ = \bot \\
\mathsf{ins}(\alpha_1\diamond\sigma_1, \forall(\alpha_2\diamond\sigma_2).\,\sigma) \quad \mathsf{iff}\ \alpha_1 \neq \alpha_2 \wedge \alpha_2 \notin \mathsf{ftv}(\sigma_1) \\
\quad = \textbf{if } \alpha_1\#(\forall(\alpha_2\diamond\sigma_2).\,\sigma) < \alpha_2\#\sigma \\
\qquad\quad \textbf{then } \forall(\alpha_2\diamond\sigma_2).\,\mathsf{ins}(\alpha_1\diamond\sigma_1, \sigma) \\
\qquad\quad \textbf{else } \ \forall(\alpha_1\diamond\sigma_1)(\alpha_2\diamond\sigma_2).\,\sigma
\end{array}$$

Figure 13. Canonical and rearranged form

$$\alpha\#\sigma = \mathsf{max}(\mathsf{occurences}(\alpha,\sigma) \cup \{0\})$$

$$\begin{array}{ll}
\mathsf{occurences}(\alpha,\sigma) & = \{\, o \mid \mathsf{select}(\mathsf{skeleton}(\sigma), o) = \alpha \} \\
\mathsf{select}(\bot, 1) & = \bot \\
\mathsf{select}(\alpha, 1) & = \alpha \\
\mathsf{select}(c\ \tau_1\ ...\ \tau_n, 1) & = c \\
\mathsf{select}(c\ \tau_1\ ...\ \tau_n, i.o) & = \mathsf{select}(\tau_i, o) \\
\mathsf{skeleton}(\tau) & = \tau \\
\mathsf{skeleton}(\bot) & = \bot \\
\mathsf{skeleton}(\forall(\alpha\diamond\sigma_1).\,\sigma_2) & = \mathsf{skeleton}(\sigma_2)[\alpha \mapsto \mathsf{skeleton}(\sigma_1)]
\end{array}$$

Figure 14. The position of type variable

The position of a type variable in a type is defined as the maximum of all its occurrences of in the type, or 0 if the type variable is unbound. This case never occurs for a canonical form since the rearranged form is taken over the normal form which discards unbound quantifiers.

To define the occurrences of a type, we first define the skeleton of a type ($\mathsf{skeleton}(\sigma)$) as the full inlining of all its bounds. For example

$$\mathsf{skeleton}(\forall(\beta \geqslant \forall\gamma.\,\alpha \to \gamma).\,\beta \to \alpha) = (\alpha \to \bot) \to \alpha$$

The selection function $\mathsf{select}(\cdot,\cdot)$ takes such skeleton and a *position*, and returns an element of that skeleton. For example

$$\begin{array}{l}
\mathsf{select}((\alpha \to \bot) \to \beta, 1.1.1) = \alpha \\
\mathsf{select}((\alpha \to \bot) \to \beta, 2.1) \quad\ = \beta \\
\mathsf{select}((\alpha \to \bot) \to \beta, 1.2.1) = \bot
\end{array}$$

The occurrences of a type variable is simply the set of all positions of the type variable in the skeleton, for example

$$\begin{array}{ll}
\mathsf{occurences}(\alpha, \forall(\beta \geqslant \forall\gamma.\,\alpha \to \gamma).\,\beta \to \alpha) & = \{1.1.1, 2.1\} \\
\mathsf{occurences}(\beta, \forall(\beta \geqslant \forall\gamma.\,\alpha \to \gamma).\,\beta \to \alpha) & = \varnothing \\
\mathsf{occurences}(\beta, \beta \to \alpha) & = \{1.1\}
\end{array}$$

During insertion, we have to be careful never to insert beyond dependent binders. In particular, we need to ensure that when quantifiers are rearranged (in the **then** branch of the last case of insertion) that $\alpha_1 \notin \mathsf{ftv}(\sigma_2)$. Indeed, together with the side conditions on insertion these are exactly the conditions under which we can apply EQ-COMM. Fortunately, the position of type variables respects dependence:

Lemma 3.7.b (*Insertion respects dependencies*): The definition of the position of a type variable respects dependence between binders.

$$\alpha_1 \# (\forall(\alpha_2 \diamond \sigma_2).\, \sigma) < \alpha_2 \# \sigma \;\Rightarrow\; \alpha_1 \notin \mathsf{ftv}(\sigma_2)$$

Proof of Lemma 3.7.b: First note that if $\alpha_2 \notin \mathsf{ftv}(\sigma)$, then the position $\alpha_2 \# \sigma = 0$, and the position of α_1 cannot be smaller. Otherwise, if $\alpha_2 \in \mathsf{ftv}(\sigma)$, we can prove Lemma 3.7.b by contradiction. Assume that $\alpha_1 \in \mathsf{ftv}(\sigma_2)$ **(1)**, where $\alpha_2 \# \sigma$ equals $o_1.1$. In that case, the set of occurrences of α in $\forall(\alpha_2 \diamond \sigma_2).\, \sigma$, namely $\mathsf{occurences}(\alpha, \forall(\alpha_2 \diamond \sigma_2).\, \sigma)$, contains an occurrence for α with the form $o_1.o_2.1$, which is larger or equal to $o_1.1$ (since 0 cannot be part of an occurrence). Since the position is the maximum of the occurrences, we have $\alpha \# (\forall(\alpha_2 \diamond \sigma_2).\, \sigma) \geqslant \alpha_2 \# \sigma$. By contradiction, this invalidates the assumption (1), and implies $\alpha_1 \notin \mathsf{ftv}(\sigma_2)$. □

3.7.2 Completeness under canonical form

Using canonical forms, we can now state a completeness theorem for canonical equivalence.

Theorem 3.7.c (*Completeness of canonical equivalence*): Whenever two types are equivalent, then their canonical forms are also canonically equivalent:

$$(Q)\;\; \sigma_1 \equiv \sigma_2 \;\Rightarrow\; (Q)\;\; \mathsf{can}(\sigma_1) \equiv^{\mathsf{c}} \mathsf{can}(\sigma_2)$$

This can be proved by straightforward induction over the rules of equivalence.

For canonical abstraction, completeness does not hold directly since CI-HYP abstracts over rigid types in the prefix. If those types are not in canonical form the rule may not apply. To remedy this situation we are going to define the notion of *weak canonical form*.

Definition 3.7.d (*Weak canonical form*): A type σ is in weak canonical form if all its rigid bounds are in canonical form. We write σ^{c} for types in weak canonical form. A prefix Q is in weak canonical form if all its quantifiers are in weak canonical form, and write Q^{c} for such prefix.

The following properties hold for types in weak canonical form:

Properties 3.7.e
i. $(Q^{\mathsf{c}})\;\; \sigma^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \mathsf{can}(\sigma)$, a type in weak canonical form instantiates to its canonical form, and conversely:
ii. $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma) \sqsubseteq^{\mathsf{c}} \sigma^{\mathsf{c}}$.

Using weak canonical form, we can now state a compleness theorem for canonical abstraction:

Theorem 3.7.f (*Completeness of canonical abstraction*): Whenever two types are in an abstraction relation under a prefix Q, then their canonical forms are in canonical abstraction relation under a prefix Q^{c} in weak canonical form:

$$(Q)\;\; \sigma_1 \mathrel{\sqsubseteq\!\!\!\!\!-} \sigma_2 \;\Rightarrow\; (Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{can}(\sigma_2)$$

For this theorem to hold, it is essential that canonical equivalence can rearrange rigid binders. Take for example the following abstraction:

$$\begin{array}{l} \forall(\alpha = \sigma_{id}).\, \forall(\gamma \geqslant \sigma_{id}).\, \forall(\beta = \sigma_{id}).\, [\beta] \to [\gamma] \to [\alpha] \\ \;\mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{A-PREFIX, A-CONTEXT, A-HYP, and EQ-MONO } \} \\ \forall(\alpha = \sigma_{id}).\, \forall(\gamma \geqslant \sigma_{id}).\, [\alpha] \to [\gamma] \to [\alpha] \end{array}$$

Alas, although the first type is in canonical form, the second one is not. This happens because abstraction can change the occurrences of rigidly bound types through A-HYP. Therefore, rearrangement of rigid binders is necessary to derive canonical forms.

Proof of Theorem 3.7.f: By induction over the rules of abstraction:
Case A-EQUIV: We have $(Q)\;\; \sigma_1 \equiv \sigma_2$, and therefore by Theorem 3.7.c, $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \equiv^{\mathsf{c}} \mathsf{can}(\sigma_2)$. By rule CA-EQUIV we can now derive $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{can}(\sigma_2)$.
Case A-HYP: We have $(\alpha = \sigma) \in Q$, and by therefore $(\alpha = \mathsf{can}(\sigma)) \in Q^{\mathsf{c}}$ since Q^{c} is in weak canonical form. By rule CA-HYP it follows that $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \alpha$.
Case A-CONTEXT: By induction, we have $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{can}(\sigma_2)$ **(1)**. Since $\mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) = \mathsf{rf}(\mathsf{nf}(\forall(\alpha = \sigma_1).\, \sigma))$ we proceed by case analysis over the normal form:
subcase $\alpha \notin \mathsf{ftv}(\sigma)$: In this case, it follows directly that $\mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) = \mathsf{can}(\sigma) = \mathsf{can}(\forall(\alpha = \sigma_2).\, \sigma)$.
subcase $\mathsf{nf}(\sigma_1) = \tau_1$ **(2)**: First we note that due to (2) and (1), $\mathsf{can}(\sigma_2) = \tau_2$ **(3)** for some monotype τ_2. We can now derive under a prefix Q:

$$\begin{array}{ll} \mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) & = \{ (2) \} \\ \mathsf{can}(\sigma)[\alpha \mapsto \tau_1] & \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{CA-EQUIV, CEQ-MONO } \} \\ \forall(\alpha = \tau_1).\, \mathsf{can}(\sigma) & \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{CA-CONTEXT, CA-EQUIV, (2) } \} \\ \forall(\alpha = \mathsf{can}(\sigma_1)).\, \mathsf{can}(\sigma) & \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{CA-CONTEXT, (1) } \} \\ \forall(\alpha = \mathsf{can}(\sigma_2)).\, \mathsf{can}(\sigma) & \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{CA-CONTEXT, CA-EQUIV, (3) } \} \\ \forall(\alpha = \tau_2).\, \mathsf{can}(\sigma) & \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \{ \text{CA-EQUIV, CEQ-MONO } \} \\ \mathsf{can}(\sigma)[\alpha \mapsto \tau_2] & = \{ (3) \} \\ \mathsf{can}(\forall(\alpha = \sigma_2).\, \mathsf{can}(\sigma)) & \end{array}$$

subcase $\mathsf{nf}(\sigma) = \alpha$: We have $\mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) = \mathsf{can}(\sigma_1)$ **(4)**, and $\mathsf{can}(\sigma_2) = \mathsf{can}(\forall(\alpha = \sigma_2).\, \sigma)$. By EQ-REFL, CA-EQUIV, (1), and transitity, we can derive $(Q^{\mathsf{c}})\;\; \mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{can}(\forall(\alpha = \sigma_2).\, \sigma)$.
subcase otherwise: In this case $\mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma)$ equals $\mathsf{ins}(\forall(\alpha = \mathsf{can}(\sigma_1)), \mathsf{can}(\sigma_2))$ where the bound $\forall(\alpha = \mathsf{can}(\sigma_1))$ is inserted into $\mathsf{can}(\sigma_2)$. But by using CEQ-COMM to rearrange the rigid binder we can bring it back to the front and we can derive $(Q^{\mathsf{c}})\;\; \mathsf{can}(\forall(\alpha = \sigma_1).\, \sigma) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \forall(\alpha = \mathsf{can}(\sigma_1)).\, \mathsf{can}(\sigma)$. By CA-CONTEXT and (1), we know that $(Q^{\mathsf{c}})\;\; \forall(\alpha = \mathsf{can}(\sigma_1)).\, \mathsf{can}(\sigma) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \forall(\alpha = \mathsf{can}(\sigma_2)).\, \mathsf{can}(\sigma)$. Finally, we can again apply CEQ-COMM repeatedly to derive $(Q^{\mathsf{c}})\;\; \forall(\alpha = \mathsf{can}(\sigma_2)).\, \mathsf{can}(\sigma) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{ins}(\forall(\alpha = \mathsf{can}(\sigma_2)), \mathsf{can}(\sigma))$ where $\mathsf{ins}(\forall(\alpha = \mathsf{can}(\sigma_2)), \mathsf{can}(\sigma))$ equals $\mathsf{can}(\forall(\alpha = \sigma_2).\, \sigma)$.
Case A-PREFIX: Similar to A-CONTEXT.
Case A-TRANS: Follows directly by induction and CA-TRANS. □

The above completeness theorems are rather weak as they require the types to be in canonical form. For canonical instantiation we have a stronger result where only rigid bounds need to be in canonical form:

Theorem 3.7.g (*Completeness of canonical instance*): When two types are in an instance relation under a prefix Q, then their weak canonical forms are also in a canonical instance relation under the same prefix in weak canonical form:

$$(Q)\;\; \sigma_1 \sqsubseteq \sigma_2 \;\Rightarrow\; (Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \sigma_2^{\mathsf{c}} \rightsquigarrow \mathsf{f}$$

Proof of Theorem 3.7.g: By induction over the instance relation.
Case I-BOTTOM: We have $(Q)\;\; \bot \sqsubseteq \sigma$. Using CI-BOTTOM, we can derive $(Q^{\mathsf{c}})\;\; \bot \sqsubseteq^{\mathsf{c}} \sigma^{\mathsf{c}}$.
Case I-ABSTRACT: We have $(Q)\;\; \sigma_1 \mathrel{\sqsubseteq\!\!\!\!\!-} \sigma_2$. As a consequence of Theorem 3.7.f, $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \mathrel{\sqsubseteq\!\!\!\!\!-}^{\mathsf{c}} \mathsf{can}(\sigma_2)$ holds **(1)**. By Property 3.7.e.i and Property 3.7.e.ii, we can also derive $(Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \mathsf{can}(\sigma_1)$ **(2)**, and $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_2) \sqsubseteq^{\mathsf{c}} \sigma_2^{\mathsf{c}}$ **(3)**. Combining (1), (2), and (3), using CI-TRANS, we can derive $(Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \sigma_2^{\mathsf{c}}$.
Case I-HYP: We assume $(\alpha \geqslant \sigma) \in Q$. This implies $(\alpha \geqslant \sigma^{\mathsf{c}}) \in Q^{\mathsf{c}}$, and by CI-HYP, $(Q^{\mathsf{c}})\;\; \sigma^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \alpha$.
Case I-RIGID: We have $(Q)\;\; \forall(\alpha \geqslant \sigma_1).\, \sigma \sqsubseteq \forall(\alpha = \sigma_1).\, \sigma)$. By Property 3.7.e.i, $(Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \mathsf{can}(\sigma_1)$ **(4)**. The weak canonical form of $\forall(\alpha \geqslant \sigma_1).\, \sigma$ is $\forall(\alpha \geqslant \sigma_1^{\mathsf{c}}).\, \sigma^{\mathsf{c}}$. Using CI-CONTEXT and (4), we instantiate to $\forall(\alpha \geqslant \mathsf{can}(\sigma_1)).\, \sigma^{\mathsf{c}}$. By CI-RIGID, this instantiates to $\forall(\alpha = \mathsf{can}(\sigma_1)).\, \sigma^{\mathsf{c}}$ which equals $(\forall(\alpha = \sigma_1).\, \sigma_2)^{\mathsf{c}}$.
Case I-TRANS: Immediate by induction and CI-TRANS.
Case I-PREFIX: Immediate by induction and CI-PREFIX.
Case I-CONTEXT: Immediate by induction and CI-CONTEXT. □

Using the completeness theorems on the canonical relations, we can finally state a completeness theorem for the canonical type rules.

Theorem 3.7.h (*Canonical type inference is complete*): If we can infer a type σ for an expression in e under a certain prefix Q and environment Γ, there exists a canonical type deriviation where σ, Q, and Γ are in weak canonical form:

$$(Q)\;\Gamma \vdash e : \sigma \;\Rightarrow\; (Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma^{\mathsf{c}} \;\rightsquigarrow \mathsf{e}$$

Proof of Theorem 3.7.h: By induction of the inference rules.
Case T-VAR: We have $(x : \sigma) \in \Gamma$, and therefore, $(x : \sigma^{\mathsf{c}}) \in \Gamma^{\mathsf{c}}$. By rule CT-VAR, we can derive $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} x : \sigma^{\mathsf{c}}$.
Case T-APP: Immediate by induction.
Case T-FUN: Immediate by induction.
Case T-LET: Immediate by induction.
Case T-GEN: By induction and Property 3.7.a.ii, we have $\alpha \notin \mathsf{ftv}(\Gamma^{\mathsf{c}})$, and either $(Q^{\mathsf{c}}, \alpha = \mathsf{can}(\sigma_1))\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma_2^{\mathsf{c}}$ or $(Q^{\mathsf{c}}, \alpha \geqslant \sigma_1^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma_2^{\mathsf{c}}$. By rule CT-GEN, we can derive $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \forall(\alpha = \mathsf{can}(\sigma_1)).\,\sigma_2^{\mathsf{c}}$ or $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \forall(\alpha \geqslant \sigma_1^{\mathsf{c}}).\,\sigma_2^{\mathsf{c}}$ where both derived types are in weak canonical form.
Case T-INST: We have $(Q)\;\Gamma \vdash e : \sigma_1$ **(1)**, and $(Q)\;\; \sigma_1 \sqsubseteq \sigma_2$ **(2)**. By the induction hypothesis and (1), we have $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma_1^{\mathsf{c}}$ **(3)**. By Theorem 3.7.g and (2), we can conclude $(Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \sigma_2^{\mathsf{c}}$, and by CT-INST and (3), we have $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma_2^{\mathsf{c}}$.
Case T-ANN: We have $(Q)\;\Gamma \vdash e : \sigma_1$ **(4)**, and $(Q)\;\; \sigma \mathrel{-\!\!\!\!\sqsubseteq} \sigma_1$ **(5)**. By the induction hypothesis and (4), we have $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \sigma_1^{\mathsf{c}}$. Due to Property 3.7.e.i, we know that $(Q^{\mathsf{c}})\;\; \sigma_1^{\mathsf{c}} \sqsubseteq^{\mathsf{c}} \mathsf{can}(\sigma_1)$, and we can use CT-INST to derive $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} e : \mathsf{can}(\sigma_1)$ **(6)**. As a consequence of Theorem 3.7.f and (5), we have $(Q^{\mathsf{c}})\;\; \mathsf{can}(\sigma_1) \mathrel{-\!\!\!\!\sqsubseteq}^{\mathsf{c}} \mathsf{can}(\sigma_2)$ (assuming the type annotation is converted to canonical form), and we can use CT-ABS with (6) to derive $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} (e :: \sigma_1) : \mathsf{can}(\sigma_1)$. Using CI-INST and Property 3.7.e.ii, we can instantiate to $(Q^{\mathsf{c}})\;\Gamma^{\mathsf{c}} \vdash^{\mathsf{c}} (e :: \sigma_1) : \sigma_1^{\mathsf{c}}$. □

3.8 Type inference

MLF has an effective type inference algorithm that infers principal types. It is beyond the scope of this paper to discuss type inference in detail, but we remark that the standard MLF inference algorithm can be used to infer types to canonically, as long as we maintain the invariants required for completeness (Theorem 3.7.h). In particular, type annotations must be normalized to canonical form, and the prefix, type environment, and inferred types must be in weak canonical form. Ensuring that types are in weak canonical form is easy to do by requiring in the update algorithm (Le Botlan 2004, page 123) that rigid bounds are only updated with types in canonical form, rearranging binders as necessary. No further changes are required. The translation to System F types can be done similarly to Leijen and Löh (2005) but simplified to remove translation for rigid bounds.

4. A restriction to System F types

We have defined an elegant type directed translation from MLF to System F, where only polymorphic flexible bounds require extra evidence and type annotations are needed only on lambda bound arguments that are used polymorphically. Even though this is probably the best we can hope for, it is interesting to consider a more restricted version.

In particular, the (implicit) introduction of evidence arguments for polymorphic flexible bounds may be undesirable in practice. Take for example our example from Section 2.1:

$$\mathbf{let}\ ids = [id]\ \mathbf{in}\ (polyL\ ids, ids +\!\!+ [inc])$$

where ids has type $\forall(\alpha \geqslant \sigma_{id}).\,[\alpha]$. In the System F translation, the ids value is transformed into a function that takes an evidence argument of type $\sigma_{id}^{*} \to \alpha$. This means that a programmer cannot assume that **let** bound values are shared since they could be translated into functions! The same situation occurs with type classes too, and Haskell introduced the *monomorphism restriction* that rejects unannotated values that require evidence translation.

$$\text{F-LET}\quad \frac{\begin{array}{c}(Q)\;\Gamma \vdash^{\mathsf{c}} e_1 : \sigma_1 \\ \forall\sigma_0.\mathbf{if}\ (Q)\;\Gamma \vdash e_1 : \sigma_0\ \mathbf{then}\ (Q)\;\; \sigma_1 \sqsubseteq \sigma_0 \\ (Q)\;\; \sigma_1 \sqsubseteq^{\mathsf{c}} \mathsf{ftype}(\sigma_1) \\ (Q)\;\Gamma, x : \mathsf{ftype}(\sigma_1) \vdash^{\mathsf{c}} e_2 : \sigma_2\end{array}}{(Q)\;\Gamma \vdash^{\mathsf{c}}\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_2}$$

$$\text{F-ANN}\quad \frac{(Q)\;\Gamma \vdash^{\mathsf{c}} e : \sigma_1 \quad (Q)\;\; \sigma \mathrel{-\!\!\!\!\sqsubseteq}^{\mathsf{c}} \sigma_1 \quad \sigma \text{ is an F-type}}{(Q)\;\Gamma \vdash^{\mathsf{c}} (e :: \sigma) : \sigma}$$

Figure 15. New type rules for MLF$^{=}$

$$\begin{array}{ll}
\mathsf{ftype}(\sigma) = \mathsf{ftype_n}(\mathsf{nf}(\sigma)) & \\
\mathbf{where} & \\
\quad \mathsf{ftype_n}(\tau) & = \tau \\
\quad \mathsf{ftype_n}(\bot) & = \bot \\
\quad \mathsf{ftype_n}(\forall(\alpha = \sigma_1).\,\sigma_2) & = \forall(\alpha = \sigma_1).\,\mathsf{ftype_n}(\sigma_2) \\
\quad \mathsf{ftype_n}(\forall(\alpha \geqslant \bot).\,\sigma) & = \forall(\alpha \geqslant \bot).\,\mathsf{ftype_n}(\sigma) \\
\quad \mathsf{ftype_n}(\forall(\alpha \geqslant \forall Q.\,\tau).\,\sigma) & = \mathsf{ftype_n}(\forall Q.\,\sigma[\alpha \mapsto \tau])
\end{array}$$

Figure 16. Force to F-Type

4.1 Rigid MLF

We consider a restriction of MLF, called "Rigid MLF" (MLF$^{=}$), that never requires evidence translation. To achieve this goal, all bound values in MLF$^{=}$ are restricted to standard System F types, i.e. types without polymorphic flexible bounds, and we call such types *F-types*

Definition 4.1.a (*F-types (Le Botlan 2004)*): A type in normal-form is an *F-type* if and only if all its flexible bounds are of the form $\forall(\alpha \geqslant \bot)$. A type is an *F-type* if an only if its normal form is an F-type.

Types that are not F-types are exactly those with polymorphic flexible bounds, i.e. a type $\forall(\alpha \geqslant \sigma_1).\,\sigma_2$ where σ_1 is not equivalent to either a monotype τ or $\bot$.

Definition 4.1.b (MLF$^{=}$): We define MLF$^{=}$ as a restriction of MLF where both type annotations and **let** bound values are restricted to F-types.

In practice, this means that a programmer always writes standard F-type annotations and never MLF types with flexible bounds. Furthermore, the restriction of type annotations to F-types effectively restricts lambda bound arguments to F-types, and together with the restriction of **let** bound values to F-types this means that no evidence translation is ever necessary.

Figure 15 gives the new type rules for **let** expressions and type annotations where we have left out the System F translation for clarity. The type annotation rule F-ANN just requires that type annotations have an F-type. The rule for **let** expressions is more involved. We cannot just require that the inferred type for the **let** binding is an F-type since we would lose principal typings. For example, the term $[id]$ can have two F-types, namely $[\forall\alpha.\,\alpha \to \alpha]$, and $\forall\alpha.\,[\alpha \to \alpha]$ where neither is an instance of the other.

Instead, the rule F-LET forces the inferred type to an F-type using the function $\mathsf{ftype}(\sigma)$ defined in Figure 16. This function systematically translates a shallow type to an F-type. The first four cases of $\mathsf{ftype_n}(\cdot)$ are easy: monomorphic types, $\bot$, rigid bounds, and unconstrained bounds are already in the correct form.

The last case deals with non-trivial flexible polymorphic bounds. A concrete example of this case happens for the term $[id]$ with type $\forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, [\beta]$. There are two possible strategies to convert such term to an F-type. First, we can instantiate and lift the quantifiers outside the bound, giving $\forall\alpha.\, [\alpha \to \alpha]$, which we call *variant HM*. This type is what the Hindley-Milner system would infer for this term, and therefore we chose this strategy for the definition of $\mathsf{ftype}_\mathsf{n}(\cdot)$. Another possibility is to keep the type polymorphic and instantiate to a rigid bound $\forall(\beta = \forall\alpha.\, \alpha \to \alpha).\, [\beta]$, which we call *variant F*. We did not choose this variant since we feel that compatability with Hindley-Milner is more important.

It is not enough to just force the inferred type for the **let** bound expression to an F-type. Since both variant HM and variant F can be obtained by instantiation, we can always use the instantiation rule to derive either variant, losing principal typings again. The solution is simple though: in rule F-LET we specify that the type derived for the **let** expression must be the most general type and cannot be an instance thereof. This most general type is then forced to an F-type. Requiring **let** bindings to most general types is not new and similar solutions are explored for example by Leroy and Mauny (1993), Garrigue and Rémy (1999), and Vytiniotis et al. (2006).

Standard MLF type inference can be used to infer types for MLF$^=$ where **let** bound values are of course forced to F-types and type annotations restricted to F-types. Since lambda bound arguments are therefore F-types, this implies that rigid bounds are always F-Types too. Types where all rigid bounds are F-types are called *shallow types*. Standard type inference works for MLF$^=$ because all types in MLF$^=$ are shallow, and Le Botlan (2004, Section 10.2) showed that standard MLF type inference is sound and complete for shallow terms, deriving shallow principal types.

4.2 Expressiveness of MLF$^=$

In MLF$^=$ our original example

$\mathbf{let}\ ids = [id]\ \mathbf{in}\ (polyL\ ids, ids \mathbin{+\!\!+} [inc])$ -- rejected

is no longer accepted. All bound expressions now have simple F-types, and *ids* gets the standard Hindley-Milner type $\forall\alpha.\, [\alpha \to \alpha]$ and cannot be passed to *polyL* which expects a list of polymorphic identity functions. Of course, the expression $ids \mathbin{+\!\!+} [inc]$ *is* well-typed. However, we could type the *polyL ids* application if we add a type annotation though:

$\mathbf{let}\ ids = [id] :: [\forall\alpha.\, \alpha \to \alpha]\ \mathbf{in}\ polyL\ ids$

This is well-typed since the annotation is a valid F-type and is not influenced by the $\mathsf{ftype}(\cdot)$ coercion in F-LET. Of course, now the application $ids \mathbin{+\!\!+} [inc]$ would be ill-typed. We can type check both applications if we inline the **let** binding and no longer try to share the *ids* value:

$(polyL\ [id], [id] \mathbin{+\!\!+} [inc])$

In order to type check this example, the type inferencer still assigns most general types with flexible polymorphic bounds to intermediate terms like $[id]$. The reader might worry that this still leads to evidence translation at runtime. However, all bound values have F-types and we can *never abstract over non F-types*. In practical terms this means that all evidence is always locally known and all remaining evidence translation can always be optimized away.

Having intermediate terms with non F-types is very important since it removes the need to annotate impredicative instantiations, and in practice that reduces the number of required type annotations significantly. For example, we did not need to annotate the application $polyL\ [id]$ even though the list is impredicatively instantiated. The *only* type annotations necessary in MLF$^=$ are (1) on lambda bound arguments that are *used* polymorphically, and (2) on **let** bound expressions that contain flexible polymorphic types that should stay polymorphic. Admittedly, the second condition is harder to explain. On the bright side, MLF$^=$ stays fully compatible with Hindley-Milner and in practice few of such annotations are necessary.

5. Related work

There have been many proposed extensions to standard Hindley-Milner type inference to support higher-ranked or impredicative types. Jones (1997) describes a system that retains a stratification between monomorphic types and type schemes and embed polymorphic types inside type constructors, where constructor application correspond to type abstraction, and pattern matching to type application.

Odersky and Läufer (1996) describe a type system for higher-ranked types where higher-ranked arguments need to be annotated with their type. Peyton Jones et al. (2007) extend this work where the type rules propagate known type information to reduce the annotation burden. Dijkstra (2005) describes a further extension that supports impredicative types and has bidirectional propagation of type annotations. In earlier work, Garrigue and Rémy (1999) describe a system where polymorphic types are embedded inside monotypes where polytypes are marked whether they are annotated or declared.

MLF (Le Botlan and Rémy 2003) goes beyond System F types to support type inference with first-class polymorphism. Several variants of MLF and their relation to other systems are explored in (Le Botlan and Rémy 2007). Rémy and Yakobowski (2007) present an efficient graph based inference algorithm for MLF that has the same complexity as normal Hindley-Milner type inference.

The work most closely related to this article is by Leijen and Löh (2005), who describe how MLF can be used with the theory of qualified types (Jones 1994) and give an algorithm that translates MLF terms to System F where evidence is passed for all bounded polymorphic types, including rigidly bound types.

5.1 A comparison with boxy types

Recently, Vytiniotis et al. (2006) introduce boxy type inference where inferred and annotated types are elegantly distinguished through boxes in the types. Just like MLF, boxy types support first-class polymorphism. Moreover, it has proved its value in practice since it has been implemented in version 6.6 of the Glasgow Haskell Compiler (GHC).

Directly comparing boxy types and MLF is difficult since their formulations differ substantially, and since MLF is strictly more expressive than boxy types (due to flexible polymorphic bounds). MLF$^=$ on the other hand is just *as expressive* as boxy types where all values can be given System F types. The set of programs accepted by boxy types is exactly the same as for MLF$^=$ modulo type annotations. A interesting way of comparing the systems would therefore be to study how type checking behaves under common small program transformations, for example, if $f\ x$ is well typed, is $apply\ f\ x$ also well typed?

Let us first look at inlining of **let** bindings: if $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ is accepted, is the inlining $e_2[x \mapsto e_1]$ also accepted?

binding type	HM	MLF	MLF$^=$	boxy
monomorphic	✓	✓	✓	✓
polymorphic	✓	✓	✓	×
higher-rank	-	✓	✓	×
impredicative	-	✓	✓	×

In the above table HM stands for Hindley-Milner type inference. We make a distinction for each kind of binding type, where "polymorphic" stands for a rank-1 polymorphic Hindley-Milner type. In Hindley-Milner, MLF, and MLF$^=$ we can always inline a let binding and still have a well-typed program. Surprisingly, this does not hold for boxy type inference – not even for rank-1 types. This is because the boxy type system relies crucially on type generalization at $\mathbf{let}$ bindings. Let's assume that ids has type $[\forall\alpha.\,\alpha \to \alpha]$, and that we have a function $choose$ of type $\forall\alpha.\,\alpha \to \alpha \to \alpha$. Now take the following program:

$\mathbf{let}\ f = choose\ [\,]\ \mathbf{in}\ f\ ids$

This program is well-typed in all systems (except HM of course) where the type of f is $\forall\alpha.\,[\alpha] \to [\alpha]$ (i.e. a standard Hindley-Milner type). However, if we inline the binding to $choose\ [\,]\ ids$, the boxy type system rejects this program since it fails to generalize the intermediate $choose\ [\,]$ term (and annotations do not help‘).

We now take a look at the inverse of inlining: if $e_2[x \mapsto e_1]$ is accepted, is the abstraction $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ also accepted?

binding type	HM	MLF	MLF$^=$	boxy
monomorphic	✓	✓	✓	✓
polymorphic	✓	✓	✓	✓
higher-rank	-	✓	ann	ann
impredicative	-	✓	×	×

In the above table 'ann' means that the abstracted $\mathbf{let}$ binding might need an annotation. Of all systems, only MLF allows this transformation. HM, MLF$^=$, and boxy types allow it for standard Hindley-Milner types but higher-rank types may need an annotation. This happens specifically when a function returns a higher order function. Suppose g has type $(Int \to (\forall\alpha.\,\alpha \to \alpha)) \to Int$, then the expression $g\ (\lambda x.id)$ is well typed in MLF, MLF$^=$, and boxy types. The abstraction however is only accepted in MLF:

$\mathbf{let}\ f = \lambda x.id\ \mathbf{in}\ g\ f$

In both boxy types and MLF$^=$ we need to annotate the binding. Without annotation, both systems assign the type $\forall\alpha\beta.\,\alpha \to (\beta \to \beta)$ which is not polymorphic enough. The binding f must be annotated with the type $\forall\beta.\,\beta \to (\forall\alpha.\,\alpha \to \alpha)$,

Impredicative types cannot be abstracted in general in MLF$^=$ and boxy types. Whenever such type requires a non-trivial flexible polymorphic bound in MLF, both MLF$^=$ and boxy types fail to type it and with good reason: without evidence translation, there is not even a possible System F translation (See Section 3.1).

The final transformation we study is the $apply$ transformation: if $e_1\ e_2$ is well typed, is $apply\ e_1\ e_2$ also accepted?

	HM	MLF	MLF$^=$	boxy
apply	✓	✓	✓	ann

In Section 2 we argued that this a particularly important transformation since it enables general abstraction over polymorphic values. Unfortunately, this property does not hold for boxy types as impredicative instantiation needs a type annotation. Take for example the application of $runST\ (return\ 1)$ where $runST$ has type $\forall s.\,ST\ s\ \alpha$. To type $apply\ runST\ (return\ 1)$, we need to annotate the full instantiation type of $apply$:

$$(apply :: ((\forall s.\,ST\ s\ Int) \to Int) \to (\forall s.\,ST\ s\ Int) \to Int)$$
$$runST\ (return\ 1)$$

A heavy burden indeed!

6. Conclusion and future work

We presented a type directed translation of MLF to System F terms. This is important in practice in order to compile MLF typed programs efficiently. In particular, when MLF is extended with qualified types it is essential to have a translation scheme to System F that can accomodate evidence translation for qualified types, and we hope that the presented translation increases adoption of MLF. Even though MLF is very attractive as an inference system for first-class polymorphism, it is a drawback that users are exposed more complicated MLF types. Inspired by MLF$^=$, we recently discovered a type system for first-class polymorphism, called HMF, which is much simpler than MLF. It uses just regular System F types and is still very expressive. For example, if $e_1\ e_2$ is well typed, then $apply\ e_1\ e_2$ is well typed too. We are currently studying the properties of this system (Leijen 2007).

References

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Universiteit Utrecht, November 2005.

Jaques Garrigue and Didier Rémy. Semi-expicit first-class polymorphism for ML. *Journal of Information and Computation*, 151:134–169, 1999.

J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

Mark P. Jones. First-class polymorphism with type inference. In *Proceedings of the Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1997.

Mark P. Jones. *Qualified types in Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

Didier Le Botlan. *ML^F: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, INRIA Rocquencourt, May 2004. Available in English.

Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

Didier Le Botlan and Didier Rémy. Recasting MLF. Technical Report 6228, INRIA, Rocquencourt, June 2007. URL `http://gallium.inria.fr/~remy/project/mlf`.

Daan Leijen. HMF: simple type inference for first-class polymorphism. Work in progress, July 2007.

Daan Leijen and Andres Löh. Qualified types for MLF. In *The International Conference on Functional Programming (ICFP'05)*. ACM Press, September 2005.

Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

Robert Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, 1978.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23th ACM Symp. on Principles of Programming Languages (POPL'96)*, pages 54–67, January 1996.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In *TLDI '07: Proc. of the 2007 ACM SIGPLAN int. workshop on Types in languages design and implementation*, pages 27–38, 2007.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: type inference for higher-rank types and impredicativity. In *The International Conference on Functional Programming (ICFP'06)*, September 2006.