Chapter 7

Advanced Type Systems

> Type structure is a syntactic discipline for enforcing levels of abstraction. ... What computation has done is to create the necessity of formalizing type disciplines, to the point where they can be enforced mechanically.
>
> — John C. Reynolds
> *Types, Abstraction, and Parametric Polymorphism* [?]

This chapter examines some advanced type systems and their uses in functional and meta-languages. Specifically, we consider the important concepts of *parametric polymorphism*, *subtyping*, and *intersection types*. Their interaction also leads to new problems which are subject of much current research and beyond the scope of these notes.

Each language design effort is a balancing act, attempting to integrate multiple concerns into a coherent, elegant, and usable language appropriate for the intended application domain. The language of types and its interaction with the syntax on the one hand (through type checking or type reconstruction) and the operational semantics on the other hand (through type preservation and a progress theorem) is central. Types can be viewed in various ways; perhaps the most important dichotomy arises from the question whether types are an inherent part of the semantics, or if they merely describe some aspects of a program whose semantics is given independently. In one view types are *intrinsic* to the terms in a language, in another they are *extrinisic*.

This is related to the issue whether a language is *statically* or *dynamically typed*. In a statically typed language, type constraints are checked at compile-time, before programs are executed; only well-typed programs are then evaluated. It also carries the connotation that types are not maintained at run-time, the type-checking having guaranteed that they are no longer necessary. In a dynamically typed language, types are instead tested at run-time in order to prevent meaningless operations (such as trying to compute **fst** z).

The semantics of Mini-ML we have presented it so far is essentially untyped, that is, the operational semantics is given for untyped expressions. Since the language satisfies type preservation, a type expresses a property of untyped programs. A program may have multiple types, since it may satisfy multiple properties expressible in the type system. Typing was intended to be static: programs are checked for validity and then evaluated. On the other hand, the semantics of LF (and, by extension, Elf) is intrinsic: only well-typed objects represent terms or derivations. Furthermore, types occur explicitly in objects and are present at run-time, since computation proceeds by searching for an object of a given type. Nonetheless, typing is static in the sense that only well-typed queries can be executed.

Which criteria should be used to evaluate the relative merits of type systems? Language design is an art as well as a craft, so there are no clear and unambiguous rules. Nevertheless, a number questions occur frequently in the comparison of type systems.

Generality. How many meaningful programs are admitted? For example, the typing discipline of Mini-ML prohibits self-application $(\lambda x.\ x\,x)$, even though this function could be meaningfully applied to many arguments, including, for example, the identity function.

Accuracy. To what level of detail can we describe or prescribe the properties of programs? For example, in Mini-ML we can express that f represents a partial function from natural numbers to natural numbers. We can not express that it always terminates or that it maps even numbers to odd numbers.

Decidability. Is it decidable if a given expression is well-typed? In a statically typed language this is important, since the compiler should be able to determine if a given input program is meaningful and either execute or reject it.

Often, generality, accuracy, and decidability conflict: if a system is too general or too accurate it may become undecidable. Thus these design goals must be traded off against each other. Other criteria help in making such choices.

Brevity. How much type information must be provided by the programmer? It is desirable for the programmer to be allowed to omit some types if there is a canonical way of reconstructing the missing information. Conversely, decidability can often be recovered for very general type systems by requiring many type annotations. In the case of Mini-ML, no type information is necessary in the input language; in Elf, a program consists almost entirely of types.

Elegance. Complex, convoluted type systems may be difficult to use in practice, since it may be hard for the programmer to predict which expressions will be considered valid. It also becomes more difficult for the type-checker to give adequate feedback about the source of a type error. Furthermore, a type system engenders a certain discipline or style of programming. A type system that is hard to understand endangers the elegance and simplicity and therefore the usability of the whole programming language.

Efficiency. Besides the theoretical property of decidability, the practical question of the efficiency of type checking must also be a concern. The Mini-ML type-checking problem is decidable, yet theoretically hard because of the rule for **let name**. In practice, type checking for ML (which is a significant extension of Mini-ML along many dimensions) is efficient and takes less than 10% of total compilation time except on contrived examples. The situation for Elf is similar.

We return to each of these critera in the following sections.

7.1 Parametric Polymorphism

Parametric polymorphism, like many other type constructs, can be viewed from a number of perspective. In keeping with the development in Chapter **??** we start with the extrinsic view: types are assigned to typeless expressions. We generalize the language of types (but *not* the language of expressions) to include types of the form $\forall\alpha.\,\tau$. An expression should have this type if it has all instance types $[\tau'/\alpha]\tau$. As usual, we express this parametrically in the meta-language: e has type $\forall\alpha.\,\tau$ if e has type τ where α is a parameter.

$$\text{Types} \quad \tau \quad ::= \quad \ldots \mid \forall\alpha.\,\tau$$

Now the rule which introduces the new form of type in the conclusion. As in natural deduction (see Chapter **??**), we therefore call this an *introduction rule*.

$$\frac{\Delta \triangleright e : \tau}{\Delta \triangleright e : \forall\alpha.\,\tau}\,\mathsf{tp\_alli}^{\alpha}$$

We also refer to this rule as *universal generalization*. The deduction of the premise must be parametric in α. In particular, α may not already occur in Δ, which would lead to a system that does not satisfy type preservation or progress (see Exercise **??**).

Since the premise is indeed parametric in α, we can substitute an arbitrary type τ' for α to obtain another valid typing for e from the same assumptions Δ.

$$\frac{\Delta \triangleright e : \forall\alpha.\, \tau}{\Delta \triangleright e : [\tau'/\alpha]\tau}\,\textsf{tp\_alle}$$

Since this rule eliminates the universal quantifier from the judgment in the premise, it is called an *elimination rule*. We also refer to it as *universal instantiation*.

It is easy to extend our LF representation of the language and the typing rules to encompass parametric polymorphism as defined above. Note that the language of expressions and evaluation only change in that we eliminate **let name**. Once again we use the idea of higher-order abstract syntax, this time at the level of Mini-ML types, in order to represent the variable binder $\forall\alpha$.

```
all : (tp -> tp) -> tp.

tp_alli : of E (all [a] T a)
           <- ({a:tp} of E (T a)).

tp_alle : {T':tp} of E (T T')
           <- of E (all [a] T a).
```

Note once again how application `T T'` is used to represent substitution $[\tau'/\alpha]\tau$. It is also worth noting the explicit quantifier on `T'` in the last clause. In LF, this clause would read

$$\begin{array}{lll} \textsf{tp\_alle} & : & \Pi E{:}\textsf{exp}.\ \Pi T{:}\textsf{tp} \to \textsf{tp}. \\ & & \Pi T'{:}\textsf{tp}.\ \textsf{of}\ E\ (\textsf{all}\ (\lambda a{:}\textsf{tp}.\ T\ a)) \to \textsf{of}\ E\ (T\ T'). \end{array}$$

The reason to make the quantifier on T' explicit in Elf (rather than leaving it implicit as the quantifiers on E and T) is because there is no occurrence of T' as an argument to a constant. As a result, T' would often remain ambiguous when the constant `tp_alle` is used. In technical terms, T' has no *strict* occurrence in the body of the clause, which generally means that T' should be explicitly quantified.

Unfortunately, the typing rules for Mini-ML with parametric polymorphism in extrinsic form are no longer syntax-directed. The rules tp_alli and tp_alle are always potentially applicable, since the expression e does not change. As a result, the LF signature can no longer be used for type inference. It is quite likely that type inference for this language is no longer decidable. Although we currently do not have a proof of this fact, two closely related systems mentioned later have been shown to be undecidable.

As a simple example, consider the identity function **lam** $x.\ x$ which can be seen to have type $\forall\alpha.\ \alpha \to \alpha$.

$$\dfrac{\dfrac{\dfrac{}{x{:}\alpha \triangleright x : \alpha}\ \textsf{tp\_var}}{\cdot \triangleright \mathbf{lam}\ x.\ x : \alpha \to \alpha}\ \textsf{tp\_lam}}{\cdot \triangleright \mathbf{lam}\ x.\ x : \forall\alpha.\ \alpha \to \alpha}\ \textsf{tp\_alli}^{\alpha}$$

This derivation is represented in Elf by

```
_ : of (lam [x] x) (all [a] a => a)
  = tp_alli [a:tp] tp_lam [x:exp] [u:of x a] u.
```

Note that we can no longer rely on the operational semantics to construct the typing derivation, so we give it explicitly as an *anonymous definition*.

Another example is self-application, **lam** $x.\ x\ x$, a function that could not be typed in the formulation of Mini-ML in Chapter **??**. However, with the use of parametric polymorphism we can give it type

$$(\forall\alpha.\ \alpha \to \alpha) \to (\forall\alpha.\ \alpha \to \alpha)$$

with the following derivation, where we use the abbreviation $one = \forall\alpha.\ \alpha \to \alpha$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}one \triangleright x : one}\ \textsf{tp\_var}}{x{:}one \triangleright x : one \to one}\ \textsf{tp\_alle} \qquad \dfrac{}{x{:}one \triangleright x : one}\ \textsf{tp\_var}}{x{:}one \triangleright x\ x : one}\ \textsf{tp\_app}}{\cdot \triangleright \mathbf{lam}\ x.\ x\ x : one \to one}\ \textsf{tp\_lam}$$

Again, we can check this derivation in Twelf.

```
one : tp = all [a] a => a.
_ : of (lam [x] app x x) (one => one)
  = tp_lam [x] [u:of x one] tp_app u (tp_alle one u).
```

However, this is not the only type for self-application (see Exercise **??**).

The proof of type preservation needs to be modified since inversion on the typing derivation no longer yields a unique rule (the expression e does not change in the rules for polymorphic generalization and instantiation). We account for this by using a nested (or lexicographic) induction on the evaluation and the typing derivation.

Theorem 7.1 (Type Preservation with Extrinsic Parametric Polymorphism)
If $e \hookrightarrow v$ *and* $\cdot \triangleright e : \tau$ *then* $\cdot \triangleright v : \tau$.

Proof: By nested induction on the structure of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{P} :: \cdot \triangleright e : \tau$. That is, we can appeal to the induction hypothesis for a smaller evaluation and arbitrary typing derivation, or an identical evaluation and smaller typing derivation.

Case: $\mathcal{D}$ is arbitrary and

$$\mathcal{P} = \cfrac{\begin{array}{c}\mathcal{P}_1 \\ \cdot \triangleright e : \tau_1\end{array}}{\cdot \triangleright e : \forall \alpha.\ \tau_1}\ \mathsf{tp\_alli}^{\alpha}.$$

Then

$\cdot \triangleright v : \tau'$ — By i.h. on $\mathcal{D}$ and $\mathcal{P}_1$
$\cdot \triangleright v : \forall \alpha.\ \tau_1$ — By rule $\mathsf{tp\_alli}^{\alpha}$

Case: $\mathcal{D}$ is arbitrary and

$$\mathcal{P} = \cfrac{\begin{array}{c}\mathcal{P}_1 \\ \cdot \triangleright e : \forall \alpha.\ \tau_1\end{array}}{\cdot \triangleright e : [\tau'/\alpha]\tau_1}\ \mathsf{tp\_alle}.$$

Then

$\cdot \triangleright v : \forall \alpha.\ \tau_1$ — By i.h. on $\mathcal{D}$ and $\mathcal{P}_1$
$\cdot \triangleright v : [\tau'/\alpha]\tau_1$ — By rule $\mathsf{tp\_alle}$

Cases: In all other cases for $\mathcal{D}$ we can now exclude the possibility that $\mathcal{P}$ ends in tp_alli or tp_alle which is already handled above. Therefore we literally copy all cases in the original proof of type preservation for Mini-ML (Theorem 2.5).

□

Before we move on to other formulations of parametric polymorphism, we consider the relationship between the original formulation of Mini-ML in which polymorphism was captured by a **let name**-construct, and the second one above in which polymorphism is explicit in the type. It is clear from the example of self-application, **lam** $x.\ x\ x$ that the system above permits some terms that are not typable using **let name**.

In order to show that explicit polymorphism is strictly more general, we would have to show that every use of **let name** can be eliminated. One way to do this without changing the operational behavior is to simply reduce **let name** $x = e_1$ **in** e_2 to $[e_1/x]e_2$. However, this explodes program size and is not modular. Can we replace every occurrence of **let name** by an appropriate form of **let val**, taking advantage of explicit polymorphism? The answer if affirmative. The key is again is the notion of parametric judgment. For example, if $\Delta \triangleright e : \alpha \rightarrow \alpha$ where α does not occur in Δ, then $\Delta \triangleright e : \tau \rightarrow \tau$ for all types τ. It turns out that Mini-ML without explicit polymorphism has the property that each expression e that is well-typed has a *principal type* τ with the following properties:

1. $\Delta \triangleright e : \tau$, that is, τ is a type of e. Let $\alpha_1, \ldots, \alpha_n$ be the free type variables of τ that do not occur in Δ.

2. $\Delta \triangleright e : [\sigma_1/\alpha_1, .\, , \sigma_n/\alpha_n]\tau$ for any types $\sigma_1, \ldots, \sigma_n$.

3. If $\Delta \triangleright e : \sigma$ then $\sigma = [\sigma_1/\alpha_1, .\, , \sigma_n/\alpha_n]\tau$ for some $\sigma_1, \ldots, \sigma_n$.

Part 2 follows from part 1 directly since the judgment $\Delta \triangleright e : \tau$ is parametric in $\alpha_1, \ldots, \alpha_n$. Part 3 is a deeper property which requires a relatively complex development for its proof which is beyond the scope of this book. Briefly, the lines of reasoning are as follows. The rules are syntax-directed, but they do not determine the types uniquely. Instead they impose some equational constraints on the types. For example, in an expression $e_1\ e_2$, the argument type of e_1 must match the type of the argument, e_2. Such equations always have a *most-general solution*: every solution to equational constraints can be obtained as an instance of this most general solution. Technically, this problem is called *unification*, the property above is the existence of *most general unifiers*. For more on this problem of typing, see [**?**].

7.2 Intrinsic Polymorphism

Parametric polymorphism, like many other type constructs, can be viewed from a number of perspectives. In the preceding section we viewed types as extrinsic to expressions; here we view them as intrinsically part of the terms. Rather than a pure type assignment $\Delta \triangleright e : \tau$ where a given expression e may have many different types, our main judgment $\Delta \triangleright M : \tau$ has the property that the term M determines its type uniquely. This requires that terms M mention some types to remove ambiguity, for example, in the identity function $\mathbf{lam}\, x.\, x$.

For parametric polymorphism this means that polymorphic functions are explicitly parameterized over a type. We write such an abstraction as $\Lambda\alpha$, as in $\Lambda\alpha.\, \mathbf{lam} x{:}\alpha.\, x$ for the polymorphic identity function. It takes two arguments: first a type τ then an argument v of type τ. For the sake of symmetry we must then also have a means to explicitly apply a polymorphic function to a type. Write this this as $M\ [\tau]$.

We then have the rules

$$\frac{\Delta \triangleright M : \tau}{\Delta \triangleright \Lambda\alpha.\ M : \forall\alpha.\ \tau}\,\mathsf{tp\_alli}^{\alpha}$$

$$\frac{\Delta \triangleright M : \forall\alpha.\ \tau}{\Delta \triangleright M\ [\tau'] : [\tau'/\alpha]\tau}\,\mathsf{tp\_alle}$$

The rest of the language can be extended to enforce uniqueness of types via a mode analysis of the inference rules for type assignment. For example, in the rule

$$\frac{\Delta, x{:}\tau_1 \rhd e : \tau_2}{\Delta \rhd \mathbf{lam}\, x.\ e : \tau_1 \to \tau_2}\,\mathsf{tp\_lam}$$

the type τ_1 that is added to the context Δ is not uniquely determined by e. Therefore, we need to decorate the **lam**-abstraction with τ_1.

$$\frac{\Delta, x{:}\tau_1 \rhd M : \tau_2}{\Delta \rhd \mathbf{lam}\, x{:}\tau_1 M : \tau_1 \to \tau_2}\,\mathsf{tp\_lam}$$

These considerations yield the following language of terms.

$$\begin{array}{lllll}
\text{Terms} & M & ::= & \mathbf{z} \mid \mathbf{s}\ M \mid (\mathbf{case}\ M_1\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2 \mid \mathbf{s}\ x \Rightarrow M_3) & \textit{Natural numbers} \\
& & & \mid \langle M_1, M_2\rangle \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M & \textit{Pairs} \\
& & & \mid \mathbf{lam}\ x{:}\tau.\ M \mid M_1\ M_2 & \textit{Functions} \\
& & & \mid \mathbf{let\ val}\ x = M_1\ \mathbf{in}\ M_2 & \textit{Definitions} \\
& & & \mid \mathbf{fix}\ x{:}\tau.\ M & \textit{Recursion} \\
& & & \mid x & \textit{Variables} \\
& & & \mid \Lambda\alpha.\ M \mid M\ [\tau] & \textit{Polymorphism}
\end{array}$$

Note that the introduction of **let name** is not only redundant (in the sense that it solves the same problem as explicit polymorphism), but that it no longer works correctly: the typing rule for **let name** relied on the property that the same expression can have more than one type. We omit the full set of typing rules, which are completely straightforward modifications of the type assignment rules for expressions.

The two examples above now have the following form:

$$\begin{array}{rcl}
\Lambda\alpha.\ \mathbf{lam}\, x{:}\alpha.\ x & : & \forall\alpha.\ \alpha \to \alpha \\
\mathbf{lam}\, x{:}\forall\alpha.\ \alpha \to \alpha.\ x\ [\forall\alpha.\ \alpha \to \alpha]\ x & : & (\forall\alpha.\ \alpha \to \alpha) \to (\forall\alpha.\ \alpha \to \alpha)
\end{array}$$

Before we consider the operational semantics, we prove the uniqueness of types. As always, we include renaming of bound variables when considering equality. Here the variables are type variables α and $\forall\alpha$ is the only binding construct.

Theorem 7.2 (Uniqueness of Intrinsic Polymorphic Types)
If $\Delta \rhd M : \tau$ and $\Delta \rhd M : \sigma$ then $\tau = \sigma$.

Proof: By simultaneous induction over the structure of $\mathcal{P} :: \Delta \rhd M : \tau$ and $\mathcal{Q} :: \Delta \rhd M : \sigma$. Note that we already assume that the context Δ is the same on both sides. We show only one case.

Case:

$$\mathcal{P} = \quad \dfrac{\begin{array}{c}\mathcal{P}_2\\ \Delta, x{:}\tau_1 \triangleright M_2 : \tau_2\end{array}}{\Delta \triangleright \mathbf{lam}\, x{:}\tau_1.\, M_2 : \tau_1 \to \tau_2}\,\textsf{tp\_lam}$$

$$\text{and} \quad \mathcal{Q} = \quad \dfrac{\begin{array}{c}\mathcal{Q}_2\\ \Delta, x{:}\sigma_1 \triangleright M_2 : \sigma_2\end{array}}{\Delta \triangleright \mathbf{lam}\, x{:}\sigma_2.\, M_2 : \sigma_1 \to \sigma_2}\,\textsf{tp\_lam}$$

Then

$\mathbf{lam}\, x{:}\tau_1.\, M_2 = M = \mathbf{lam}\, x{:}\sigma_1.\, M_2$	$\mathcal{P}$ and $\mathcal{Q}$ end in M
$\tau_1 = \sigma_1$	by equality
$(\Delta, x{:}\tau_1) = (\Delta, x{:}\sigma_1)$	by equality
$\tau_2 = \sigma_2$	by i.h. on $\mathcal{P}_2$ and $\mathcal{Q}_2$
$\tau_1 \to \tau_2 = \sigma_1 \to \sigma_2$	by equality

□

The implementation of this proof in Twelf is somewhat awkward, since we cannot access the internal (judmental) notion of equality in the framework. Therefore we define a structural equality on types. We show below the implementation of the fragment containing only function types. The reader may consult the on-line material for the remaining cases. First, type equality.

```
eq : tp -> tp -> type.                          %name eq E e.
eq_arrow : eq (arrow T1 T2) (arrow S1 S2)
            <- eq T1 S1
            <- eq T2 S2.
```

We need a lemma stating that type equality is reflexive, which follows by induction on the structure of the type. Note that there is a dependency between the input and output argument of `refl`.

```
refl : {T:tp} eq T T -> type.
%mode refl +T -E.
refl_arrow : refl (arrow T1 T2) (eq_arrow E2 E1)
              <- refl T1 E1
              <- refl T2 E2.
%terminates T (refl T _).
```

Next we prove the uniqueness of the types, by simultaneous induction on P and Q.

```
% Uniqueness of types
tpu : of* E T -> of* E S -> eq T S -> type.
%mode tpu +P +Q -E.

tpu_lam : tpu (tp_lam* P2) (tp_lam* Q2) (eq_arrow E2 E1)
           <- refl T1 E1
           <- ({x:term*} {u:of* x T1}
                 tpu u u E1 -> tpu (P2 x u) (Q2 x u) E2).
tpu_app : tpu (tp_app* P2 P1) (tp_app* Q2 Q1) E1
           <- tpu P1 Q1 (eq_arrow E1 E2).

%terminates [P Q] (tpu P Q _).
```

The operational semantics extends quite easily from the type assignment system; only the case of type abstraction and application require some consideration. It may be possible, for example, to evaluate underneath a type abstraction with the rule

$$\frac{M \hookrightarrow V}{\Lambda\alpha.\ M \hookrightarrow \Lambda\alpha.\ V}$$

where the evaluation of the premise would be parametric in α. However, just as we only evaluate closed expressions, it seems most consistent to evaluate only closed terms. This is because only the meaning of closed terms is given unambiguously. Since types are intrinsic to terms, this would naturally preclude both free type and free term variables. We therefore instead use the rule

$$\frac{}{\Lambda\alpha.\ M \hookrightarrow \Lambda\alpha.\ M}\ \textsf{ev\_tlam}$$

Type application is straightforward.

$$\frac{M \hookrightarrow \Lambda\alpha.\ M' \qquad [\tau/\alpha]M' \hookrightarrow V}{M\ [\tau] \hookrightarrow V}\ \textsf{ev\_tapp}$$

Here we use V for term values which are slightly different from expression values v in that they include a case for type abstractions.

$$\begin{array}{lrcll} \text{Term Values} & V & ::= & \mathbf{z} \mid \mathbf{s}\ V & \textit{Natural numbers} \\ & & & \mid \langle V_1, V_2 \rangle & \textit{Pairs} \\ & & & \mid \mathbf{lam}\ x.\ M & \textit{Functions} \\ & & & \mid \Lambda\alpha.\ M & \textit{Polymorphism} \end{array}$$

We relate terms to expression via erasure and write $|M| = e$. The definition is obvious, except for the cases of type abstraction and application, where we define

$$\begin{array}{rcl} |\Lambda\alpha.\ M| & = & |M| \\ |M\ [\tau]| & = & |M| \end{array}$$

Note that erasure forgets structure: many different terms erase to the same expression. However, there is a bijection between typing derivations of terms and expressions. This is implicit in the proof of the following theorem which asserts the correctness of erasure.

Theorem 7.3 (Relation between Extrinsic and Intrinsic Typing)

1. *If* $\Delta \triangleright e : \tau$ *then there is an* M *such that* $|M| = e$ *and* $\Delta \triangleright M : \tau$.
2. *If* $\Delta \triangleright M : \tau$ *and* $|M| = e$ *then* $\Delta \triangleright e : \tau$.

Proof: In each direction by a proof over the structure of the given derivation. □

Both erasure and its correctness proof are easy to implement and elided here. They can be found in[1].

Unfortunately, the translation developed above does not preserve the operational semantics. A simple counterexample is the term

$$\triangleright\ \Lambda\alpha.\ \mathbf{fix}\ x{:}\alpha.\ \alpha : \forall a\alpha.\ \alpha$$

It evaluates to itself in one step while its erasure, $\mathbf{fix}\ x.\ x$ does not have a value. At some cost in elegance, this can be repaired by inserting vacuous abstractions and applications into the translation. Assuming we have the unit type 1 (see **??**) and the unit element $\langle\,\rangle$, we change erasure to

$$\begin{array}{rcll} |\Lambda\alpha.\ M| & = & \lambda x.\ |M| & x \text{ not free in } M \\ |M\ [\tau]| & = & |M|\ \langle\,\rangle & \end{array}$$

However, the translation is no longer a bijection because the vacuous variable x can be assigned an arbitrary type. Furthermore, an expression such as $\mathbf{fix}\ x.\ x$ at type $\forall\alpha.\ \alpha$ is not in the image of the translation function.

Another solution is to restrict universal generalization on expression or type abstraction on terms to values. This is an instance of the so-called *value restriction* which has been adopted for Standard ML [**?**] for related reasons. The modified rules are

$$\frac{\Delta \triangleright v : \tau}{\Delta \triangleright v : \forall\alpha.\ \tau}\ \mathsf{tp\_alli}^{\alpha} \qquad \frac{\Delta \triangleright V : \tau}{\Delta \triangleright \Lambda\alpha.\ V : \forall\alpha.\ \tau}\ \mathsf{tp\_tlam}^{\alpha}$$

Under these restricted rules we have a strong correspondence theorem illustrated by the following picture.

[1] *[poly/intrinsic.elf]*

$$\begin{tikzcd} e \arrow[r, "e \hookrightarrow v"] & v \\ M \arrow[u, "e = |M|"] \arrow[r, "M \hookrightarrow V"'] & V \arrow[u, "v = |V|"'] \end{tikzcd}$$

This decomposes into two theorems. The first is an instance of *coherence*. We think of e only as a shorthand for an intrinsically typed term M. In that case the meaning of e is ambiguous, since different typing derivations for e yield different terms M which evaluate to different answers V. Coherence states that it does not matter how we reconstruct the missing types in e, because whatever answers V we get by this path erase to the same v. This independence of the answer from a typing derivation is the meaning of coherence; this is particularly simple instance of the idea.

The second theorem shows that an untyped operational semantics is adequate: we do not need to carry types at run-time. If we erase M to obtain e and then evaluate e we obtain the same value as if we evaluated M according to the typed semantics and then erased the types from the answer. Since the types in the answer are not obvservable (in values they can appear only under **lam**-abstractions which are not observable), the erased value v is observably equivalent to V.

The proof of these theorems is left as Exercise **??** Note that the value restriction is critical for this correspondence.

7.3 Indexed Representation of Intrinsic Typing

Intrinsic types are considered to be an integral part of terms and a term that is not well typed is considered meaningless. For example, in an intrinsically typed system we would not ask the question if $\mathbf{fst}\,\langle \mathbf{z}, (\lambda x.\ \mathbf{snd}\,\mathbf{z})\rangle$ has a value, because it is not well typed. In an extrinsically typed system we may note that it is not well-typed, but we can nonetheless state that it evaluates to $\mathbf{z}$.

In that sense our development above falls somewhat short of a fully intrinsically typed language: term M contain types, but the evaluation judgment still could be applied to terms that do not type check. The type of a term is unique if it exists, but it need not exist.

The logical framework allows us to explore an even stronger version of intrinsic typing were the types are directly part of the representation. Because of its hidden complexity, such formulations are not particularly popular in informal presentation. We would annotate a term with its (unique) type and the presentation of the syntax itself carries this type. When a term M intrinsically has type τ, we write M^τ. The syntax for such terms (showing only the functional fragment) would be:

$$
\begin{array}{lll}
\text{Terms} & M^\tau & ::= \mathbf{z}^{\mathbf{nat}} \mid (\mathbf{s}\ M^{\mathbf{nat}})^{\mathbf{nat}} \mid (\mathbf{case}\ M_1^{\mathbf{nat}}\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2^\tau \mid \mathbf{s}\ x^{\mathbf{nat}} \Rightarrow M_3^\tau)^\tau \\
& & \mid \langle M_1^\tau, M_2^\sigma\rangle^{\tau\times\sigma} \mid (\mathbf{fst}\ M^{\tau\times\sigma})^\tau \mid (\mathbf{snd}\ M^{\tau\times\sigma})^\sigma \\
& & \mid (\mathbf{lam}\ x^\tau.\ M^\sigma)^{\tau\to\sigma} \mid (M_1^{\sigma\to\tau}\ M_2^\sigma)^\tau \\
& & \mid (\mathbf{let\,val}\ x^\sigma = M_1^\sigma\ \mathbf{in}\ M_2^\tau)^\tau \\
& & \mid (\mathbf{fix}\ x^\tau.\ M^\tau)^\tau \\
& & \mid x^\tau \\
& & \mid (\Lambda\alpha.\ M^\tau)^{\forall\alpha.\ \tau} \mid (M^{\forall\alpha.\ \sigma}\ [\tau])^{[\tau/\alpha]\sigma}
\end{array}
$$

The implementation of this formulation in LF is particularly elegant, since we can index terms by there type. The adequacy theorem for the representation than asserts a compositional bijection between well-typed terms in Mini-ML and well-typed LF objects of the representation type.

```
term : tp -> type.  %name term M x.

z'    : term nat.
s'    : term nat -> term nat.
case' : term nat -> term T -> (term nat -> term T) -> term T.

pair' : term T -> term S -> term (T * S).
fst'  : term (T * S) -> term T.
snd'  : term (T * S) -> term S.

lam'  : (term T -> term S) -> term (T => S).
app'  : term (S => T) -> term S -> term T.

letv' : term S -> (term S -> term T) -> term T.

fix'  : (term T -> term T) -> term T.

tlam' : ({a:tp} term (T a)) -> term (all [a] T a).
tapp' : term (all [a] S a) -> {T:tp} term (S T).
```

This form of indexed representation of terms has a number of advantages when it is possible. For one, no explicit type-checking rules need to be implemented: the framework implementation itself will accomplish this. In practice, however, since the type indices remain implicit, this means the framework will have to perform a complicated form of type reconstruction. Sometimes it will not be able to decide whether the term can be typed. For example, the anonymous definition

```
one = all [a] a => a.
_ = lam' [x] app' (tapp' x one) x.
```

yields the error

```
Typing ambiguous -- unresolved constraints
T1 one = all ([a:tp] T1 a) => T2.
```

This means we have to supply more type information. The analysis of intrinsic typing in Section 7.2 shows what is necessary for types to be unique. This translates to a corresponding criterion on the Elf representation. Here it means that if we provide the type of the variable x, all other types are determined.

```
one = all [a] a => a.
_ = lam' [x:term one] app' (tapp' x one) x.
```

Another advantage is that some theorems, such as Mini-ML type preservation, are now also intrinsic to the representation and do not need to be proved separately. We declare

```
eval' : term T -> term T -> type.
%mode eval' +M -V.
```

so that type-checking the declarations for `eval'` will automatically verify type preservation. Informally, the corresponding judgment could be written as $M^\tau \hookrightarrow V\tau$, capturing the same constraint. Since `T` is an implicit argument, the evaluation rules are textually the same as for expressions in Mini-ML, although the result of reconstruction is different. As an example we show the rule for application and its reconstruction.

```
ev'_app : eval' (app' M1 M2) V
           <- eval' M1 (lam' M1')
           <- eval' M2 V2
           <- eval' (M1' V2) V.

ev'_app :
   {T1:tp} {T2:tp} {M1':term T2 -> term T1} {V2:term T2} {V:term T1}
      {M2:term T2} {M1:term (=> T2 T1)}
      eval' T1 (M1' V2) V -> eval' T2 M2 V2
         -> eval' (=> T2 T1) M1 (lam' T2 T1 M1')
         -> eval' T1 (app' T2 T1 M1 M2) V.
```

One should keep in mind that such an indexed representation is not always possible. The **let name**-construct from the original formulation of Mini-ML is an example. The way it encodes polymorphism relies on multiple types for the same expression, which cannot be encoded directly in indexed form. Systems that incorporate subtyping provide another class of examples that may be difficult or impossible to represent in an indexed manner.