16 Polymorphic Types

poly-mor-phic: able to assume different forms

Webster's Dictionary

Some functions execute in a way that's independent of the data type on which they operate. Some data structures are structured in the same way regardless of the types of their elements.

As an example, consider a function to concatenate linked lists in Tiger. We first define a linked-list data type, and then an `append` function:

```
type elem = int
type intlist = {head: elem, tail: intlist}

function append(a: intlist, b: intlist) : intlist =
  if a=nil
  then b
  else intlist{head= a.head, tail= append(a.tail,b)}
```

There's nothing about the code for the `intlist` data type or the `append` function that would be any different if the `elem` type were `string` or `tree` instead. We might like `append` to be able to work on any kind of list.

A function is *polymorphic* (from the Greek *many+shape*) if it can operate on arguments of different types. There are two main kinds of polymorphism:

Parametric polymorphism. A function is *parametrically polymorphic* if it follows the same algorithm regardless of the type of its argument. The Ada or Modula-3 *generic* mechanism, C++ *templates*, or ML *type schemes* are examples of parametric polymorphism.

Overloading. A function identifier is *overloaded* if it stands for different algorithms depending on the type of its argument. For example, in most languages

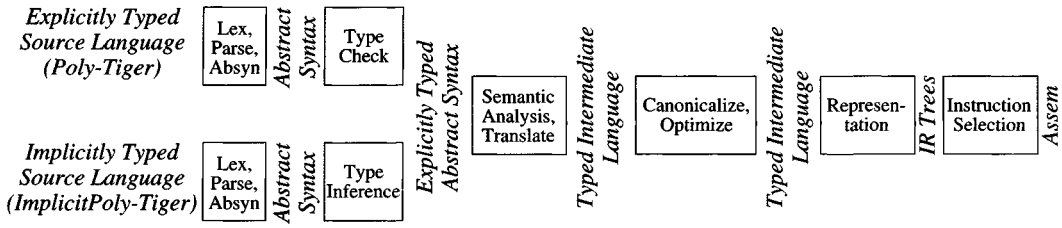


FIGURE 16.1. Compiler phases for polymorphic languages.

+ is overloaded, meaning integer addition on integer arguments and floating-point addition (which is quite a different algorithm) on floating-point arguments. In many languages, including Ada, C++, and Java, programmers can make overloaded functions of their own.

These two kinds of polymorphism are quite different – almost unrelated – and require different implementation techniques.

16.1 PARAMETRIC POLYMORPHISM

A polymorphic function $f(x : t)$ takes some parameter x of type t, where t can be *instantiated* at different actual types. In an *explicit* style of parametric polymorphism, we pass the type as an argument to the function, so we write something like $f\langle t\rangle(x : t)$, and a function call might look like $f\langle$`int`$\rangle(3)$ or $f\langle$`string`$\rangle$("`three`") In a language with *implicit* parametric polymorphism, we simply write the definition as $f(x)$, and the call as $f(3)$ or f("`three`") – the type-parameter t is unstated. Reasonable programming languages can be designed either way.

Even when compiling an implicitly typed language, it makes sense to use an explicitly typed language as the intermediate representation, as shown in Figure 16.1. One of the biggest advantages of an *explicitly typed intermediate language* is that the intermediate representation is type-checkable, unlike the `Tree` language described in Chapter 7. This means that after each optimization phase the type-checker can be run again – not to debug the program being compiled, but to debug the compiler!

Trust, but verify. A typed intermediate form allows the recipient of a partially compiled (and optimized) program to check it for safety and security; this is an important principle of *Web applets*. A Java program, for example,

can be compiled into an intermediate form called *Java virtual machine byte codes*; the semi-compiled program is transmitted to a user's machine, where it can still be type-checked by a *byte-code verifier*. Then the byte-code program is either interpreted or translated to native machine code. Type-checking of byte codes (or other transmitted intermediate representation) means that the user of an applet does not have to take it on faith that the applet will not subvert security by violating the type system. But Java does not have parametric polymorphism; in this chapter we will show a typed intermediate form for a polymorphic programming language.

To investigate polymorphism, let us make an explicitly polymorphic language Poly-Tiger, and an implicitly polymorphic language ImplicitPoly-Tiger, both based on the functional language Fun-Tiger described in Chapter 15. The explicitly typed abstract syntax for *both* languages will be similar to Poly-Tiger.

AN EXPLICITLY TYPED POLYMORPHIC LANGUAGE

Poly-Tiger is the Tiger language described in Appendix A, but with different syntax for declarations and types, and two new kinds of expressions, as shown in Grammar 16.2. Like Fun-Tiger, Poly-Tiger has function types *ty*`->`*ty* (as shown in the *function type, multi-argument function type,* and *no-argument function type* rules of Grammar 16.2), and function-calling syntax $exp_f(exp_1, \ldots, exp_n)$ that allows exp_f to be an expression, not just an identifier.

For Poly-Tiger we add several new kinds of types *ty*. A *polymorphic type* `poly<a>`T can behave like type T for any `a`. For example, `poly<a>a->a` is the type of a function that, for any `a`, can take a parameter of type `a` and return a result of type `a`.

We also need a way to build polymorphic data structures. Therefore we add new rules for *parametric type constructor* and *type construction*: the declaration **type** *id tyvars* = *ty* declares a parameterized type *id*; any type variables occurring in the right-hand-side *ty* must be from the explicit type parameters *tyvars*.

With this, we can make the "list of anything:"

```
type list<e> = {head: e, tail: list<e>}
```

Note that `list` is not a type – it is a *type constructor* (*tycon* for short), declared by the programmer as shown here. But for any type T, `list<`T`>` is a type.

ty	→ *id*	Type identifier
ty	→ *ty* -> *ty*	Function type
ty	→ (*ty* {, *ty* }) -> *ty*	Multi-argument function type
ty	→ ( ) -> *ty*	No-argument function type
ty	→ **poly** *tyvars ty*	Polymorphic type
ty	→ *ty tyargs*	Type construction
tyvars	→ < *id* {, *id*} >	Formal type parameters
tyvars	→ ϵ	No type parameters
tyargs	→ < *ty* {, *ty*} >	Type arguments
tyargs	→ ϵ	No type arguments
vardec	→ **var** *id* : *ty* := *exp*	Variable declaration with type
tydec	→ **type** *id tyvars* = *ty*	Parametric type constructor
tydec	→ **type** *id tyvars* = `array of` *ty*	Array type expression
tydec	→ **type** *id tyvars* = { *tyfields* }	*(these braces stand for themselves)*
tyfields	→ ϵ	Empty record type
tyfields	→ *id* : *ty* { , *id* : *ty* }	Record type fields
fundec	→ **function** *id tyvars* (*tyfields*) = *exp*	Polymorphic procedure declaration
fundec	→ **function** *id tyvars* (*tyfields*) : *id* = *exp*	Polymorphic function declaration
	Restriction: *tydec* must not be nested inside the body of a *fundec*.	
exp	→ ...	*all the Tiger expressions, plus ...*
exp	→ *exp tyargs* (*exp* {, *exp*})	Function call with instantiation
exp	→ *type-id tyargs* {*id*=*exp*{ , *id*=*exp*}}	Record creation with type instantiation

GRAMMAR 16.2. Syntax rules for Poly-Tiger.

To construct a record from a polymorphic record-type constructor, the *record creation with type instantiation* rule requires a type argument before the record-field initializers. Thus, for example, we can make a `list<int>` record by

```
list<int>{head=4,tail=nil}
```

The *function call with instantiation* allows us to call a polymorphic function. Now we are prepared to write a polymorphic `append` function:

```
type list<e> = {head: e, tail: list<e>}

function append<e>(a: list<e>, b: list<e>) : list<e> =
  if a=nil
  then b
  else list<e>{head= a.head, tail= append<e>(a.tail,b)}
```

ty → **Nil** | **Int** | **String** | **Unit**
ty → **Record**((*sym*, *ty*)*list*, *unique*)
ty → **Array**(*ty*, *unique*)
ty → **Name**(*sym*, *ty*)

(a) Monomorphic

ty → **Nil**
ty → **App**(*tycon*, *ty list*)
ty → **Var**(*tyvar*)
ty → **Poly**(*tyvar list*, *ty*)
tycon → **Int** | **String** | **Unit** | **Arrow**
tycon → **Array** | **Record**(*fieldname list*)
tycon → **TyFun**(*tyvar list*, *ty*)
tycon → **Unique**(*tycon*, *unique*)

(b) Polymorphic

FIGURE 16.3. `Types` module. (a) Summary of Program 5.7. (b) With new **App**,**Var**,**Poly** types, and with type constructors.

The type of `append` is `poly<e>(list<e>,list<e>)->list<e>`. Now let's build a list of two 4's:

```
var one4  : list<int> := list<int>{head=4,tail=nil}
var two4s : list<int> := append<int>(one4,one4)
```

We can even build a list of int-lists:

```
list<list<int>>{head=two4s, tail=nil}
```

POLYMORPHIC TYPE-CHECKING

Type-checking for a polymorphic language is not as straightforward as for a monomorphic language. Before embarking on an implementation, we must be clear about what the typing rules are.

For Tiger, we used a `Types` module (Program 5.7) to describe monomorphic types. To describe the types of Poly-Tiger, we use a new `Types` module as summarized in Figure 16.3.

There are some important differences. We now have an **App** type to apply a *tycon* (such as `list`) to type arguments (such as `<int>`). To simplify the internal representation of types, we think of `int` as a type constructor that takes zero type arguments; so even though it's not written this way in Poly-Tiger syntax, internally we represent it as **App**(**Int**, [ ]). The two-argument **Arrow** tycon represents functions, so that $a \rightarrow b$ is represented as **App**(**Arrow**, [a, b]).

$$subst(\mathbf{Var}(\alpha), \{\beta_1 \mapsto t_1, \ldots, \beta_k \mapsto t_k\}) = \begin{cases} t_i & \text{if } \alpha \equiv \beta_i \\ \mathbf{Var}(\alpha) & \text{otherwise} \end{cases}$$

$$subst(\mathbf{Nil}, \sigma) = \mathbf{Nil}$$

$$subst(\mathbf{App}(\mathbf{TyFun}([\alpha_1, \ldots, \alpha_n], t), [u_1, \ldots, u_n]), \ \sigma) =$$
$$subst(subst(t, \{\alpha_1 \mapsto u_1, \ldots, \alpha_n \mapsto u_n\}), \ \sigma)$$

$$subst(\mathbf{App}(tycon, [u_1, \ldots, u_n]), \ \sigma) = \quad \text{where } tycon \text{ is not a } \mathbf{TyFun}$$
$$\mathbf{App}(tycon, [subst(u_1, \sigma), \ldots, subst(u_n, \sigma)])$$

$$subst(\mathbf{Poly}([\alpha_1, \ldots, \alpha_n], u), \sigma) = \mathbf{Poly}([\gamma_1, \ldots, \gamma_n], subst(u', \sigma))$$
where $\gamma_1, \ldots, \gamma_n$ are new variables not occurring in σ or in u
and $u' = subst(u, \{\alpha_1 \mapsto \mathbf{Var}(\gamma_1), \ldots, \alpha_n \mapsto \mathbf{Var}(\gamma_n)\})$

ALGORITHM 16.4. Rules for substitution of type variables. The third clause shows that when we encounter a **TyFun**, we expand it out before continuing the substitution; it would also have been possible to carry the substitution into the body of the **TyFun** if we took care about scopes as shown in the **Poly** clause. We could avoid applying *subst* twice in the **Poly** clause by composing the two substitutions; see Exercise 16.4.

Substitution. The type-constructor **TyFun**($[\alpha_1, \ldots, \alpha_n], t$) represents a *type function*. Type t may mention $\alpha_1 \ldots \alpha_n$, and the meaning of any **App** type using this *tycon* is found by expanding t, substituting actual type arguments for the formal parameters α_i. The rules for this substitution are given in Algorithm 16.4.

The basic idea of substitution is that substituting t_1 for β_1 in a type expression t replaces all the occurrences of β_1 within t by t_1. But this must be done subject to scope rules to avoid *variable capture*. If the type t is **Poly**($[\beta_1]$, `list<`β_1`>`), then the formal parameter $[\beta_1]$ of the **Poly** is a new variable whose scope is the body of the **Poly** (e.g. `list<`β_1`>`), and no substitution for β_1 should occur within that scope.

The problem of variable capture for expressions is described in Section 15.4 (page 321). Type substitution requires α-conversion in the same way as term (i.e., expression) substitution. To avoid capture, we introduce the new variables $[\gamma_1, \ldots, \gamma_n]$ in the **Poly** rule of Algorithm 16.4.

Equivalence of types. Given the declarations

```
type number = int
type transformer<e> = e -> e
```

then `number` means just the same as `int`, and `transformer<int>` means the same as `int->int`; we can freely substitute the definitions of these types for their uses. This is called *structural equivalence of types.*

The internal representation of these type constructors is

`number` $\mapsto$ **Int**
`transformer` $\mapsto$ **TyFun**([e], **App**(**Arrow**, [**Var**(e), **Var**(e)]))

In Poly-Tiger we wish to preserve the "record distinction" rule of the Tiger language (page 513), that each record-type declaration creates a "new" type. This is called *occurrence equivalence of types.*[1] That is, given the declarations

```
type pair<a>    = {first: a, second: a}
type twosome<a> = {first: a, second: a}
```

the types `pair<int>` and `twosome<int>` are *not* the same. Structural versus occurrence equivalence is a decision that each language designer must make; record types in ML, for example, use structural equivalence.

In ordinary Tiger, **Record** types are distinguished by a `unit ref` field called `unique`. But in a polymorphic language, we may need to copy record descriptions when we apply them to arguments. In the following program,

```
let type pair<z> = {first: z, second: z}
    function f(a: pair<int>) : pair<int> = a
 in f
end
```

the first line creates a new *type constructor* `pair`, not a new type. We want `pair<int>` (the parameter type) to be the same type as `pair<int>` (the result type); but `pair<string>` must be recognized as a different type.

To express the internal structure of the `pair` type-constructor, we could write,

$$tycon_{\mathrm{p}} = \textbf{TyFun}([z], \textbf{App}(\textbf{Record}([first, second]), [\textbf{Var}(z), \textbf{Var}(z)]))$$

but this would not distinguish `pair` from `twosome`. Therefore, we wrap a **Unique** tycon around it:

$$tycon_{\mathrm{pair}} = \textbf{Unique}(tycon_{\mathrm{p}}, q_{323})$$
$$tycon_{\mathrm{twosome}} = \textbf{Unique}(tycon_{\mathrm{p}}, q_{324})$$

The tags q_{323} and q_{324} distinguish `pair` types from `twosome` types.

[1] Sometimes this has been called *name equivalence*, but in fact it is the occurrence of a definition that "generates" the new type, not the name to which it is bound.

$unify(\mathbf{App}(tycon_1, [t_1, \ldots, t_n]), \mathbf{App}(tycon_1, [u_1, \ldots, u_n])) =$ if $tycon_1$ is **Int**, **String**, **Unit**, **Arrow**, **Array**, or **Record**$([id_1, \ldots, id_n])$
  $unify(t_1, u_1)$; $\ldots$ $unify(t_n, u_n)$
$unify(\mathbf{App}(\mathbf{TyFun}([\alpha_1, \ldots, \alpha_n], u), [t_1, \ldots, t_n]), t) =$
  $unify(subst(u, \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n\}), t)$
$unify(t, \mathbf{App}(\mathbf{TyFun}([\alpha_1, \ldots, \alpha_n], u), [t_1, \ldots, t_n])) =$
  $unify(t, subst(u, \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n\}))$
$unify(\mathbf{App}(\mathbf{Unique}(u, z), [t_1, \ldots, t_n]), \mathbf{App}(\mathbf{Unique}(u', z'), [t'_1, \ldots, t'_n])) =$
  if $z \neq z'$ then *error*;
  $unify(t_1, t'_1)$; $\cdots$ $unify(t_n, t'_n)$
$unify(\mathbf{Poly}([\alpha_1, \ldots, \alpha_n], u), \mathbf{Poly}([\alpha'_1, \ldots, \alpha'_n], u')) =$
  $unify(u, subst(u', \{\alpha'_1 \mapsto \mathbf{Var}(\alpha_1), \ldots, \alpha'_n \mapsto \mathbf{Var}(\alpha_n)\}))$
$unify(\mathbf{Var}(\beta), \mathbf{Var}(\beta)) = OK$
$unify(\mathbf{Nil}, \mathbf{App}(\mathbf{Record}(\ldots), \ldots)) = OK$
$unify(\mathbf{App}(\mathbf{Record}(\ldots), \ldots), \mathbf{Nil}) = OK$
$unify(t, u) = error$ *in all other cases*

ALGORITHM 16.5. Testing for equivalence of types. This function may print an error message but has no other side effect on the global state. It is called *unify* because – when we extend it to do type inference in an implicitly typed language – it will not only check that two types are the same but will *make* them the same if possible, modifying global state.

$expand(\mathbf{App}(\mathbf{TyFun}([\alpha_1, \ldots, \alpha_n], u), [t_1, \ldots, t_n])) =$
  $expand(subst(u, \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n\}))$
$expand(\mathbf{App}(\mathbf{Unique}(tycon, z), [t_1, \ldots, t_n])) = expand(\mathbf{App}(tycon, [t_1, \ldots, t_n]))$
$expand(u) = u$ *in all other cases*

ALGORITHM 16.6. Expanding a type to see its internal structure.

Testing type equivalence. In type-checking, we often need to test whether one type is equivalent to another. To test equivalence of types containing **App**(**TyFun** $\ldots, \ldots$) we may need to expand the **TyFun**, substituting actual parameters for formal parameters. But to compare types containing **App**(**Unique**$(tycon, z), \ldots$) we should not expand *tycon*, but instead test the uniqueness mark z. The *unify* function (Algorithm 16.5) tests types for equivalence. Where *error* is reached, a type-checking error message will need to be printed for the user.

$transdec(\sigma_v, \sigma_t,$ `type` id $=$ `array of` $ty) =$
    $z \leftarrow$ `newunique()`
    $(\sigma_v,\ \sigma_t + \{id \mapsto \textbf{Unique}(\textbf{TyFun}([\,], \textbf{App}(\textbf{Array}, [transty(\sigma_t, ty)])), z)\})$
$transdec(\sigma_v, \sigma_t,$ `type` id `<`a`>` $=$ `array of` $ty) =$
    $\beta \leftarrow$ `newtyvar()`
    $z \leftarrow$ `newunique()`
    $tyc \leftarrow \textbf{TyFun}([\beta], \textbf{App}(\textbf{Array}, [transty(\sigma_t + \{a \mapsto \textbf{Var}(\beta)\},\ ty)]))$
    $(\sigma_v,\ \sigma_t + \{id \mapsto \textbf{Unique}(tyc, z)\})$
$transdec(\sigma_v, \sigma_t,$ `type` $id = ty) =$
    $(\sigma_v,\ \sigma_t + \{id \mapsto \textbf{TyFun}([\,], transty(\sigma_t, ty))\})$
$transdec(\sigma_v, \sigma_t,$ `type` id `<`a`>` $= ty) =$
    $\beta \leftarrow$ `newtyvar()`
    $(\sigma_v,\ \sigma_t + \{id \mapsto \textbf{TyFun}([\beta], transty(\sigma_t + \{a \mapsto \textbf{Var}(\beta)\},\ ty))\})$

$transty(\sigma_t,\ id) =$ if $\sigma_t(id)$ is a 0-argument *tycon*
    $\textbf{App}(\sigma_t(id), [\,])$
$transty(\sigma_t,\ id) =$ if $\sigma_t(id)$ is a *ty* (that is, *id* is a type variable)
    $\sigma_t(id)$
$transty(\sigma_t,\ id\langle u_1, \ldots, u_k\rangle) =$ $\sigma_t(id)$ must be a k-argument *tycon*
    $\textbf{App}(\sigma_t(id), [transty(u_1), \ldots, transty(u_k)])$
$transty(\sigma_t,\ ty_1$ `->` $ty_2) =$ $\textbf{App}(\textbf{Arrow}, [transty(\sigma_t,\ ty_1), transty(\sigma_t,\ ty_2)])$
$transty(\sigma_t,$ `poly<`a`>` $ty) =$
    $\beta \leftarrow$ `newtyvar()`
    $\textbf{Poly}([\beta], transty(\sigma_t + \{a \mapsto \textbf{Var}(\beta)\}, ty))$

ALGORITHM 16.7. Nonrecursive type declarations. Shown here are a few of the translation rules.

Expansion of Unique types. We do look inside the definition of a **Unique** type when we need some operation that requires its internal structure. For **Array** types, this means subscripting; for **Record** types, this means field selection or record construction. The function *expand* (Algorithm 16.6) illustrates where **TyFun** and **Unique** types must be expanded to expose internal structure.

Translation of types. Algorithm 16.7 shows how the syntax of Poly-Tiger type declarations is translated into the internal representation of the new `Types` module. The type environments σ_t map identifiers to *tycons*, except

that a type-variable identifier is mapped to a *ty*; type variables are introduced into the environment for explicit type parameters of polymorphic functions, `poly` types, and parameterized type constructors.

Like the ML code on page 118, Algorithm 16.7 does not handle recursive types. The way to handle recursive type declarations is just as described on pages 119–121: process the headers first, then the bodies. For ordinary Tiger, the first pass of processing a recursive declaration creates `NAME` types whose right-hand sides were not filled in until the second pass. For Poly-Tiger, the "hole" that gets filled in on the second pass must be in the **Unique** type-constructor, not in the **TyFun**.

Type-checking. Algorithm 16.8 shows some of the rules for type-checking declarations and expressions. To type-check `function f<z>(x:`t_1`):`t_2`=`e_1, we create a new type variable β and insert the binding `z` $\mapsto$ β into the type environment for the processing of the function parameters and body, so that uses of `z` will be correctly recognized. Then we create a **Poly** type whose formal parameter is β.

To type-check a function-call `f<int>(`e_3`)`, we look up `f` in the variable-environment σ_v to get **Poly**($[\beta]$, **App**(**Arrow**, $[t_1, t_2]$)), and substitute `int` for β in t_1 and t_2. Then we check that e_3 has type t_1, and return type t_2 as the type of the entire function-call.

To type-check a record-creation `list<int>{head=x,tail=y}`, we look up `list` in the type-environment σ_t to get a *tycon*, translate `int` in σ_t to get t_1, and then make the type of the new record, t_r = **App**(*tycon*, t_1). Then we can get the type of the `head` field from t_r and make sure that it unifies with the type of `x` (and similarly for `tail`).

16.2 TYPE INFERENCE

In order to make polymorphic programming easier, some polymorphic languages – notably ML and Haskell – don't require the programmer to write down all the types. Instead, the type-checker infers the types. For example, consider the `append` function from page 347, written without all the `<types>` – except that polymorphic record-type declarations must still be written with all type parameters spelled out in full:

$\sigma_{t0} = \{\texttt{int} \mapsto \textbf{App}(\textbf{Int}, [\,]), \ldots\}$ — Initial type environment

transdec$(\sigma_v, \sigma_t,$ `function` f<z>$(x : t_1) : t_2 = e_1) =$ — Function declaration

$\beta \leftarrow$ `newtyvar()`
$\sigma_t' \leftarrow \sigma_t + \{z \mapsto \textbf{Var}(\beta)\}$
$t_1' \leftarrow transty(\sigma_t', t_1)$; $t_2' \leftarrow transty(\sigma_t', t_2)$
$\sigma_v' \leftarrow \sigma_v + \{f \mapsto \textbf{Poly}([\beta], \textbf{App}(\textbf{Arrow}, [t_1', t_2']))\}$
$\sigma_v'' \leftarrow \sigma_v' + \{x \mapsto t_1'\}$
$t_3' \leftarrow transexp(\sigma_v'', \sigma_t', e_1)$
$unify(t_2', t_3')$
(σ_t, σ_v')

transdec$(\sigma_v, \sigma_t,$ `var` $id : ty = exp) =$ — Variable declaration

$t \leftarrow transty(\sigma_t, ty)$
$unify(t,\ transexp(\sigma_t, \sigma_v, exp))$
$(\sigma_t, \sigma_v + \{id \mapsto t\})$

transexp$(\sigma_v, \sigma_t, id) = \sigma_v(id)$ — Identifier expression

transexp$(\sigma_v, \sigma_t, e_1 + e_2) =$ — Integer addition

$unify(transexp(\sigma_v, \sigma_t, e_1), \textbf{App}(\textbf{Int}, [\,]))$;
$unify(transexp(\sigma_v, \sigma_t, e_2), \textbf{App}(\textbf{Int}, [\,]))$;
App(**Int**, [ ])

transexp$(\sigma_v, \sigma_t, e_1$<ty>$(e_2)) =$ — Function call with instantiation

$t \leftarrow transty(\sigma_t, ty)$
$t_f \leftarrow transexp(\sigma_v, \sigma_t, e_1)$
$t_e \leftarrow transexp(\sigma_v, \sigma_t, e_2)$
check that $expand(t_f) = \textbf{Poly}([\beta], \textbf{App}(\textbf{Arrow}, [t_1, t_2]))$
$unify(t_e, subst(t_1, \{\beta \mapsto t\}))$
$subst(t_2, \{\beta \mapsto t\})$

transexp$(\sigma_v, \sigma_t, rcrd$<$ty$>$\{fld_1 = e_1\}) =$ — Record creation

$t_r \leftarrow \textbf{App}(\sigma_t(rcrd), [transty(\sigma_t, ty)])$
check that $expand(t_r) = \textbf{App}(\textbf{Record}([fld_1]), [t_f])$
$unify(t_f, transexp(\sigma_v, \sigma_t, e_1))$
t_r

ALGORITHM 16.8. Type-checking expressions of Poly-Tiger. Shown here are a few of the type-checking rules; not shown are the cases for recursive or multi-argument functions, multiple type arguments, or multiple record fields.

```
type list<e> = {head: e, tail: list<e>}

function append(a, b) =
  if a=nil
  then b
  else list{head= a.head, tail= append(a.tail,b)}
```

This style is more concise to write and perhaps easier to read, but how does a compiler infer the types? First, it inserts place-holders α, β, γ, δ where types will be required:

```
function append(a: α, b: β) : γ =
  if a=nil
  then b
  else list<δ>{head= a.head, tail= append(a.tail,b)}
```

Now, because of the expressions `a.head` and `a.tail` it knows that `a` must be a list,[2] so α = `list<`η`>` for some η. Because `b` can be returned as the result of `append`, it knows that $\beta = \gamma$. The `else` clause returns `list<`δ`>`, so γ = `list<`δ`>`. Finally, because of `head=a.head`, $\delta = \eta$. Applying these equations to the `append` code, the compiler has:

```
function append(a: list<δ>, b: list<δ>) : list<δ> =
  if a=nil
  then b
  else list<δ>{head= a.head, tail= append(a.tail,b)}
```

It has not relied on any particular property of δ, so this `append` function should work for any type δ. We express this by *generalizing* over δ, making it an explicit type parameter of the `append` function:

```
function append<d>(a: list<d>, b: list<d>) : list<d> =
  if a=nil
  then b
  else list<d>{head= a.head, tail= append<d>(a.tail,b)}
```

and now we have a function identical (modulo renaming of `d`) to the one shown on page 347. The next few pages will explain this type inference and generalization algorithm in detail.

[2]This follows only if there are no other record types with `head` or `tail` fields; the issue will be discussed later in this section.

AN IMPLICITLY TYPED POLYMORPHIC LANGUAGE

The *Hindley-Milner type inference* algorithm takes a polymorphic program written without explicit type annotations, and converts it to an explicitly typed program. To explain the algorithm we can use a language ImplicitPoly-Tiger, which is like Poly-Tiger but without explicit type annotations on function parameters (Grammar 16.9). This language is meant to model some of the important concepts of the ML programming language.

Unlike Poly-Tiger, function declarations in ImplicitPoly-Tiger don't list the types of their arguments and are not written with a *tyargs* list. Also, users cannot write explicit **poly** types in ImplicitPoly-Tiger, although these can be inferred internally by the type inferencing algorithm. Finally, function calls and record creations are written without any type-argument list.

However, type declarations (such as the declaration of `list<e>`) *are* written with explicit type parameters and explicitly typed record fields.

Translation of ImplicitPoly-Tiger. Algorithm 16.7 for translating Poly-Tiger types and type declarations also applies to ImplicitPoly-Tiger, although we do not need the last rule (for `poly` types). But type-checking function declarations and expressions is rather different, so we cannot use Algorithm 16.8 but use Algorithm 16.10 instead.

ALGORITHM FOR TYPE INFERENCE

Type inference uses an internal representation of types similar to the one shown in Figure 16.3b, but with two additional kinds of *ty*s, the first of which is the *type metavariable*:

$$ty \rightarrow \textbf{Meta}(metavar)$$

A metavariable is not like an ordinary **Var** type variable that is bound by a **Poly**; it is just a placeholder for an unknown type that we expect to infer. In the `append` program on page 355, the Greek letters α, β, γ, δ are metavariables; we solve for the values of α, β, γ, but δ is left undetermined. That means we can convert δ to an ordinary variable `d` that is bound by a **Poly** in the type of `append`.

As Algorithm 16.10 shows, when type-checking `function` $f(x) = e_1$ we make up new metavariables t_x (to stand for the type of x) and t_2 (to stand for the return-type of f). Then we use *unify* to derive the relationship between the metavariables, in the following way.

ty, *tyvars*, *tyargs*, *tydec*		*All as in Poly-Tiger (Grammar 16.2), but no* `poly` *types*
vardec	→ **var** *id* := *exp*	Variable declaration without type
fundec	→ **function** *id* (*formals*) = *exp*	Function declaration
	Restriction: *tydec* must not be nested inside the body of a *fundec*.	
formals	→ *id* { , *id* }	Formal parameter list without types
formals	→	Empty parameter list
exp	→ ...	*all the Tiger expressions, plus ...*
exp	→ *exp* (*exp* {, *exp*})	Function call with implicit type instantiation
exp	→ *type-id* {*id*=*exp*{, *id*=*exp*}}	Record creation with implicit type instantiation

GRAMMAR 16.9. Syntax rules for ImplicitPoly-Tiger.

The type-checker maintains a global environment σ_m mapping metavariables to their instantiations. On page 355 when we learn that $\alpha =$ `list<`η`>`, we add the binding $\alpha \mapsto$ **App**(`list`, [**Meta**(η)]) into σ_m.

Most implementations do not implement σ_m as a lookup table; instead, each **Meta**(α) has an additional field that starts out empty; when an instantiation is done, instead of adding the binding $\alpha \mapsto t$ to a table, a pointer to t is stored into the **Meta** record.

The *unify* function of Algorithm 16.5 needs new clauses to handle **Meta** types; these clauses access and update the global environment `sigma_m`:

unify(**Meta**(α), t) =
    if $\alpha \in$ domain(σ_m)
        then *unify*($\sigma_m(\alpha)$, t)
    else if $t \equiv$ **App**(**TyFun** ...)
        then *unify*(**Meta**(α), expand **TyFun** type as usual)
    else if $t \equiv$ **Meta**(γ) and $\gamma \in$ domain(σ_m)
        then *unify*(**Meta**(α), $\sigma_m(\gamma)$)
    else if $t \equiv$ **Meta**(α)
        then *OK*
    else if **Meta**(α) occurs in t
        then *error*
    else $\sigma_m \leftarrow \sigma_m + \{\alpha \mapsto t\}$; *OK*
unify(t, **Meta**(α)) = where t is not a **Meta**
    unify(**Meta**(α), t)

If the metavariable α has already been instantiated to some type u, then we just unify u with t. Otherwise, we instantiate α to t, except that we never instantiate α to α.

The clause "if **Meta**(α) occurs in t" is called the *occurs check*, and it avoids creating circular types: we do not want to instantiate α = `list<`α`>`, because this is not permitted in ImplicitPoly-Tiger's type system.

Other functions that deal with types – such as *subst* and *expand* – need to "see through" instantiated metavariables:

$$
\begin{array}{ll}
\textit{subst}(\textbf{Meta}(\alpha), \sigma) = & \textit{expand}(\textbf{Meta}(\alpha)) = \\
\quad \text{if } \alpha \in \text{domain}(\sigma_m) & \quad \text{if } \alpha \in \text{domain}(\sigma_m) \\
\quad\quad \text{then } \textit{subst}(\sigma_m(\alpha), \sigma) & \quad\quad \text{then } \textit{expand}(\sigma_m(\alpha)) \\
\quad\quad \text{else } \textbf{Meta}(\alpha) & \quad\quad \text{else } \textbf{Meta}(\alpha)
\end{array}
$$

Generalization and instantiation. In the translation of `function` $f(x) = e_1$, we may end up with a type with *free metavariables*. That is, the type t_f may be something like **App**(**Arrow**, [**Meta**(α), **Meta**(α)]) where α is not instantiated in σ_m. In such a case, we would like to *generalize* this type to

Poly([a], **App**(**Arrow**, [**Var**(a), **Var**(a)]))

so that f can be applied to any type of argument. But we must be careful. Consider the program

```
function randomzap(x) =
  let function f(y) = if random() then y else x
   in f
  end
```

While type-checking `randomzap`, we recursively call *transexp* on the `let` expression, which type-checks the declaration of f. At this point, the type of `x` in σ_v is **Meta**(α), and the type of f turns out to be **App**(**Arrow**, [**Meta**(α), **Meta**(α)]). But we *cannot* generalize – `f` cannot take arguments of arbitrary type, only arguments whose type is the same as that of `x`. The point is that α appears in the current environment σ_v, in the description of `x`'s type.

Therefore the algorithm for *generalize* is:

generalize(σ_v, t) =
  let $\alpha_1, \ldots, \alpha_k$ be the metavariables that appear in t
    but do not appear anywhere in σ_v
    (when searching σ_v we must interpret metavariables
    by looking them up in σ_m and searching the result)
  for $i \leftarrow 1$ to k
    let $a_i \leftarrow$ `newtyvar()`
    $\sigma_m \leftarrow \sigma_m + \{\alpha_i \mapsto \textbf{Var}(a_i)\}$
  return **Poly**([$a_1, \ldots, a_k$], t)

$\sigma_{t0} = \{\texttt{int} \mapsto \textbf{App}(\textbf{Int}, [\,]), \ldots\}$ — Initial type environment

transdec$(\sigma_v, \sigma_t,$ `function` $f(x) = e_1) =$ — Function declaration

$t_x \leftarrow \textbf{Meta}(\texttt{newmetavar}())$

$t_2 \leftarrow \textbf{Meta}(\texttt{newmetavar}())$

$t_f \leftarrow \textbf{App}(\textbf{Arrow}, [t_x, t_2])$

$t_3 \leftarrow \mathit{transexp}(\sigma_v + \{f \mapsto t_f, x \mapsto t_x\}, \sigma_t, e_1)$

$\mathit{unify}(t_2, t_3)$

$t'_f \leftarrow \mathit{generalize}(\sigma_v, t_f)$

$(\sigma_v + \{f \mapsto t'_f\},\ \sigma_t)$

transdec$(\sigma_v, \sigma_t,$ `var` *id*:=*exp*$) =$ — Variable declaration with implicit type

$t \leftarrow \mathit{transexp}(\sigma_t, \sigma_v, exp)$

check that t is not **Nil**

if *id* is assigned to anywhere in its scope

$t' \leftarrow \textbf{Poly}([\,], t)$

else $t' \leftarrow \mathit{generalize}(t)$

$(\sigma_t, \sigma_v + \{id \mapsto t\})$

transexp$(\sigma_v, \sigma_t, id) =$ — Occurrence of a variable

instantiate$(\sigma_v(id))$

transexp$(\sigma_v, \sigma_t, e_1 + e_2) =$ — same as for Poly-Tiger

$\mathit{unify}(\mathit{transexp}(\sigma_v, \sigma_t, e_1), \textbf{App}(\textbf{Int}, [\,]))$

$\mathit{unify}(\mathit{transexp}(\sigma_v, \sigma_t, e_2), \textbf{App}(\textbf{Int}, [\,]))$

$\textbf{App}(\textbf{Int}, [\,])$

transexp$(\sigma_v, \sigma_t, f(e)) =$ — Function call

$t_f \leftarrow \mathit{transexp}(\sigma_v, \sigma_t, f)$

$t_e \leftarrow \mathit{transexp}(\sigma_v, \sigma_t, e)$

$t_2 \leftarrow \textbf{Meta}(\texttt{newmetavar}())$

$\mathit{unify}(t_f, \textbf{App}(\textbf{Arrow}, [t_e, t_2]))$

t_2

ALGORITHM 16.10. Hindley-Milner algorithm for type-checking expressions in ImplicitPoly-Tiger.

Note that *all* functions will be given **Poly** types – but a monomorphic function will have type **Poly**([ ], ...) which is polymorphic only in a trivial sense.

The converse of generalization is instantiation. Where a polymorphic variable is used, we replace the bound type variables with metavariables:

instantiate(**Poly**($[a_1, \ldots, a_k], t$)) =
    for $i \leftarrow 1$ to k
        let $\alpha_i \leftarrow$ `newmetavar()`
    return *subst*($t, \{a_1 \mapsto$ **Meta**$(\alpha_1), \ldots, a_k \mapsto$ **Meta**$(\alpha_k)\}$)

We perform instantiation at each use of a polymorphic function. At one call site, the bound variable a_1 may be replaced with a metavariable α that unifies with `int`; at another call site, the same a_1 may be replaced with a metavariable β that unifies with `string`. But at a particular call site, all the uses of α must be consistent with each other.

For example, the `randomzap` function has type `poly<a> a->(a->a)`. It can be used in the following way:

```
let var i0 := randomzap(0)
    var s0 := randomzap("zero")
 in i0(3)+size(s0("three"))
end
```

which might return any of the following: 3+5, 0+5, 3+4, or 0+4. The first occurrence of `randomzap` is instantiated with the type $\alpha \rightarrow (\alpha \rightarrow \alpha)$, where α is a metavariable. But all the α's must unify: when type-checking `randomzap(0)` we unify α with `int`, which instantiates α; when type-checking `i0(3)` we unify α with `int`, but since α is already instantiated to `int` this is just a check that α is used consistently; then when type-checking `i0(3)+...` we again unify α with `int`. Similarly, the second occurrence of `randomzap` is instantiated to $\beta \rightarrow (\beta \rightarrow \beta)$, and the β's are all unified with `string` (`"zero"`, `"three"`, and the argument of `size`).

Updatable variables. Given a variable declaration `var a :=` *exp*, we generalize the type of *exp* in determining the type of `a`. This is sound as long as there are no assignments to `a` other than this initialization. But *polymorphic references* (i.e., assignable variables of polymorphic type) are a problem. The following program should not be allowed to type-check:

```
let function identity(x) = x
    function increment(i) = i+1
    var a := identity
 in a := increment; a("oops")
end
```

and one way of assuring this is to avoid generalizing the types of updatable variables; this restriction on polymorphic references is implemented in the *Variable declaration with implicit type* clause of Algorithm 16.10.

transdec$(\sigma_v, \sigma_t,$ `type` *id* `<a>` $=$ $\{fld : ty\}$ $) =$ Parametric record declaration
  $z \leftarrow$ `newunique()`
  $\beta \leftarrow$ `newtyvar()`
  $\sigma_t' \leftarrow \sigma_t + \{a \mapsto \mathbf{Var}(\beta)\}$
  $t \leftarrow transty(\sigma_t', ty)$
  $t_{id} \leftarrow \mathbf{Unique}(\mathbf{TyFun}([\beta], \mathbf{App}(\mathbf{Record}([fld]), [t])), z)$
  $t_f \leftarrow \mathbf{Poly}([\beta], \mathbf{Field}(t_{id}, t))$
  $(\sigma_v,\ \sigma_t + \{id \mapsto t_{id}, fld \mapsto t_f\})$
transexp$(\sigma_v, \sigma_t, rcrd\{fld_1 = e_1\}) =$ Record creation
  check that $\sigma_t(rcrd) = \mathbf{Unique}(\mathbf{TyFun}([\alpha_1, \ldots, \alpha_k], t_r), z)$ (perhaps $k = 0$)
  for $i \leftarrow 1$ to k let $x_i \leftarrow$ **Meta**(`newmetavar()`)
  $t_r' \leftarrow subst(t_r, \{\alpha_1 \mapsto x_1, \ldots, \alpha_k \mapsto x_k\})$
  check that expand$(t_r') = \mathbf{App}(\mathbf{Record}([fld_1]), [t_f])$
  $unify(t_f', transexp(\sigma_v, \sigma_t, e_1))$
  t_r
transexp$(\sigma_v, \sigma_t, e.id) =$ Field selection
  $t_e \leftarrow transexp(\sigma_v, \sigma_t, e)$
  $t_2 \leftarrow$ **Meta**(`newmetavar()`)
  $t_f \leftarrow instantiate(\sigma_v(id))$
  $unify(t_f, \mathbf{Field}(t_e, t_2))$
  t_2

ALGORITHM 16.11. Type-checking records and fields in ImplicitPoly-Tiger.

RECURSIVE DATA TYPES

In Tiger and its variants, record types may be *recursive*; a language must have some form of recursive types in order to build lists and trees. Recursive types pose their own challenges for type inference: we might ask, for example, whether the following parameterized types are equivalent:

```
type list<a>     = {head: a, tail: list<a>}
type sequence<a> = {head: a, tail: sequence<a>}
```

We have sidestepped this question by the *record distinction rule*, saying that record types declared at different places are different – a simple and neat solution, overall. The **Unique** tycons are used to make this distinction, as explained earlier in the chapter.

Global record fields. In the Tiger language, different record types may use the same field names; when compiling `p.x`, the record type of `p` is known before looking up field `x`.

But in ImplicitPoly-Tiger, an expression such as `p.x` must be type-checked when we don't yet know the type of `p` – all we may have is a metavariable for `p`'s type. Therefore, we require that field names (such as `x`) have global scope.

The type-checking rules for record declarations, record-creation expressions, and field selection, are shown in Algorithm 16.11. We require a new kind of *ty* (to be added to Figure 16.3b, along with **Meta** types):

$$ty \rightarrow \textbf{Field}(ty, ty)$$

where $\textbf{Field}(t_{id}, t)$ means a field of type t in a record of type t_{id}. The globalization of field names is unpleasant, because it means that the same field name cannot be easily used in two different record types (one will hide the other); but it is the price to be paid for automatic type inference of field selection in Tiger. The ML language solves the problem in a different way, with global data constructors in its datatypes instead of global field names in its record types; so in ML the same data constructor name cannot be used in two different types (or one will hide the other).

A **Field** such as `head` is polymorphic: its type is

$$\textbf{Poly}([\beta], \textbf{Field}(\textbf{App}(\texttt{list}, [\beta]), \beta))$$

indicating that it selects a value of type β from a record of type `list<`β`>`. When a field is used in an expression, it must be *instantiated* just like a polymorphic function is.

THE POWER OF HINDLEY-MILNER TYPES

The polymorphic `append` function can be written in ImplicitPoly-Tiger, a language that uses the Hindley-Milner type system – or it can be written in Poly-Tiger, whose type system is equivalent to second-order lambda calculus. But in fact Poly-Tiger is strictly more expressive than ImplicitPoly-Tiger; the following Poly-Tiger function has no equivalent in ImplicitPoly-Tiger:

```
function mult(m: poly<a>((a->a)->(a->a)),
              n: poly<b>((b->b)->(b->b)))
                                  : poly<c>(c->c)->(c->c) =
        let function g<d>(f: d->d) : d->d =
                m<d->d>(n<d>(f))
         in g
        end
```

The reason is that the function `mult` has formal parameters `m` and `n` that are explicitly polymorphic. But the only place that the Algorithm 16.10 introduces a `poly` type is in *generalization*, which only occurs at function declarations, never at formal parameters.

On the other hand, any ImplicitPoly-Tiger program can be translated directly into Poly-Tiger.[3] Algorithm 16.10 can be augmented to perform this translation as it does type-checking; it would then fit in the "Type Inference" box of Figure 16.1.

No Hindley-Milner-style type inference algorithm exists – or can exist – for a Poly-Tiger-like type system (one that can handle functions such as `mult`). If we wish to use the full power of second-order lambda calculus we will have to write our types out in full. It is not clear whether the extra expressive power of Poly-Tiger is necessary, or whether it outweighs the convenience of type inference available in implicitly typable languages. ML and Haskell, which use implicit polymorphic types, have been quite successful as general-purpose functional programming languages in the research community; no explicitly typed polymorphic language has caught on in the same way. But explicitly typed languages are becoming the state of the art in *intermediate representations* for polymorphic languages.

16.3 REPRESENTATION OF POLYMORPHIC VARIABLES

After type-checking, the program can be translated into an explicitly typed intermediate language similar in structure to Poly-Tiger. Canonicalization (translation into functional intermediate form, as described in Section 19.7) and optimization (as in Chapters 17–19) can be performed on this typed IR.

Finally, we must prepare for instruction selection. The basic problem that must be solved for polymorphic languages is that the compiler cannot know the type – and size – of the data held in a polymorphic variable.

[3] Well, almost any; see Exercise 16.7.

Consider the `append` function, rewritten with variables `x` and `y` to model compiler-generated temporaries:

```
function append<e>(a: list<e>, b: list<e>) : list<e> =
  if a=nil
  then b
  else let var x : e := a.head
           var y : list<e> := a.tail
       in list<e>{head=x, tail=append<e>(y,b)}
       end
```

What is the type of `x`? If it is a pointer (i.e., a record or array type), then it should serve as a root for garbage collection; if is an integer, then it should not. If it is a double-precision floating-point number (in a Poly-Tiger enhanced with floating-point types) then it is eight bytes long; if it is a pointer, it is (typically) four bytes.

It is obvious why the compiler needs to know the size of a value such as `a.head`: it must copy the data from the `a` record to the newly created `list` record, and it needs to know how many bits to copy. But the garbage-collection issue is also important: if the allocation call that creates the new `list` finds that a garbage collection is necessary, then all the local variables and compiler temporaries of `append` are *roots* of garbage collection (see Section 13.7). If `a.head` is an integer i, then attempting to trace reachable data from heap-address i will lead to disaster.[4]

But the compile-time type of `a.head` is simply `e`, which is a type variable that will be instantiated differently at different call sites of `append`. How can the machine code deal with different types and sizes of data for `a.head`?

There are several solutions to this problem:

Expansion: Don't generate code for the general-purpose `append<e>` function; instead, for each different type at which `e` is instantiated, generate an append function specific to that type.

Boxing, tagging: Make sure that every value is the same size (typically one word) and can serve as a root of garbage collection.

Coercions: Allow differently typed values to be different sizes, but coerce them into a uniform-size representation when they are passed to polymorphic functions.

Type-passing: Allow differently typed values to be different sizes, and pass type information to polymorphic functions along with the values so that the polymorphic functions know how to handle them.

[4]Only a *conservative* collector (see page 290) doesn't need to be told which fields are pointers.

Each of these is a complete solution that can handle every case – some compilers use only expansion, others use only boxing/tagging, and so on. But a compiler can best optimize programs by using tagging for some types, coercions for other types, type-passing to handle other cases, and expansion where convenient.

The next few sections describe these techniques in more detail.

EXPANSION OF POLYMORPHIC FUNCTIONS

A simple approach to polymorphism is to inline-expand all polymorphic functions until everything is monomorphic. This is the way that Ada generics and C++ templates work. The advantage of this approach is a simple and efficient compilation model; the disadvantage is that functions are replicated, which can cause code bloat.

Section 15.4 describes the inline expansion of functions: the body of a function is copied, but with the actual parameters of the call substituted for the formal parameters of the definition. This technique works just as well when the parameters are types instead of values.

Given a function definition

$$\texttt{function}\ f\langle z_1, \ldots, z_k\rangle(x_1 : t_1, \ldots, x_n : t_n) : t_r = e$$

and a function call $f\langle u_1, \ldots, u_k\rangle(a_1, \ldots, a_n)$, we can replace the call by

$$\texttt{let function}\ f(x_1 : t_1', \ldots, x_n : t_n') : t_r' = e'\ \texttt{in}\ f(a_1, \ldots, a_n)\ \texttt{end}$$

where $t_i' = subst(t_i, \{z_1 \mapsto u_1, \ldots, z_k \mapsto u_k\})$ and e' is formed by replacing any occurrence of a z_i within e by u_i.

This works very straightforwardly for nonrecursive functions: we just do it bottom-up, so that in the case of function f the expression e would have been processed already, so that it contains no polymorphic function definitions or calls to polymorphic functions. But if e contains a recursive call to f, then we have two cases to consider:

1. The call to f within e is of the form $f\langle z_1, \ldots, z_k\rangle(\ldots)$, where the actual type parameters of the call match the formal type parameters of f's definition. In this case, we can just delete the parameters as we rewrite e to use the monomorphic function f introduced by the `let` described above. This is, in fact, a very common case; *all* recursive function-calls introduced by Algorithm 16.10 follow this pattern.
2. The recursive call to f has different actual type parameters.

The latter situation is called *polymorphic recursion*, and is illustrated by the following program:

```
let function blowup<e>(i:int, x:e) : e =
     if i=0 then x
     else blowup<list<e>>(i-1,list<e>{head=x,tail=nil}).head
 in blowup<int>(N, 0)
end
```

The `blowup` function will be called at N different types: `int`, `list<int>`, `list<list<int>>`, and so on. No finite amount of inline expansion of `blowup` can cover all possible values of N.

Because ImplicitPoly-Tiger (and languages like ML and Haskell) do not permit polymorphic recursion, this kind of blowup cannot occur, so complete expansion into monomorphic code is possible. But total inline-expansion will not work for Poly-Tiger; and in languages where it does work, there's still a difficulty with separate compilation: often we'd like to compile a function where it is declared, not recompile it at every place where it's called.

FULLY BOXED TRANSLATIONS

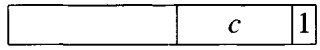
Another way to solve the polymorphic-variable problem is to use *fully boxed* representations, in which all values are the same size, and each value describes itself to the garbage collector. Usually we put each value in one word; when its natural representation is too large to fit, we allocate a record, and use the pointer as the word. This technique is called *boxing*, and the pointer is a *boxed* value. As described in Section 13.7, the record representing the boxed value usually starts with a descriptor indicating how large the record is and whether it contains pointers.

The basic Tiger compiler described in Chapters 2–12 represents everything in one word, but the data objects do not describe themselves to the garbage collector. The garbage-collection descriptor format described at the end of Chapter 13 does better, but still cannot not support polymorphism.

Boxing and tagging. An integer value fits into a word but doesn't describe itself to the garbage collector, so it must also be boxed. In this case, we create a one-word record (preceded by a descriptor, as usual) to hold the integer, and the boxed value is the pointer to this record.

To compile arithmetic such as $c \leftarrow a + b$ on boxed values requires that a be fetched from its box (*unboxed*), b be fetched, the addition performed, then a new record be allocated to hold the boxed value c. This is quite expensive.

For values (such as characters) whose natural representation is *smaller* than one word, there is an alternative to boxing called *tagging*. Suppose, for example, that all record pointers on a byte-addressed machine are aligned at multiple-of-4 boundaries; then any word ending with a 1 bit will be recognizably not a pointer. So we can represent character values by shifting them left and adding 1:



To compile $c \leftarrow a+b$ on tagged values requires that a be shifted right (*untagged*), b be shifted right, the addition performed, then c be shifted left and incremented (*tagged*). This is much cheaper than allocating a new (garbage-collectable) record. In fact, many of the shifts cancel out (see Exercise 16.8).

Tagging is so much cheaper than boxing that many implementations of polymorphic languages use tagging for their ordinary integer variables, with the disadvantage that integers cannot use the full word size of the machine because one bit must be reserved for the tag.

COERCION-BASED REPRESENTATION ANALYSIS

The problem with full boxing is that the entire program uses (expensive) boxed representations, even in places where the programmer has not used any polymorphic functions. The idea of *coercion-based representation analysis* is to use unboxed (and untagged) representations for values held in monomorphic variables, and boxed (or tagged) representations for values held in polymorphic variables. That way the monomorphic parts of the program can be very efficient, with a price paid only when polymorphic functions are called.

In Poly-Tiger, or in an ImplicitPoly-Tiger program that the type-checker has converted into Poly-Tiger, the conversion from unboxed to boxed values must occur at the call to a polymorphic function. Consider the definition of f as

$$\texttt{function}\ f\texttt{<}a\texttt{>}(x : a) : a = \ldots x \ldots x \ldots$$

with some call-site that calls $f\texttt{<int>}(y)$, where y is an integer variable. The type of y is `int`, and the type of x is a, which is a polymorphic type variable. In this case we can convert from `int` to "polymorphic" by boxing y.

The type of the formal parameter will always be at least as general as the type of the actual parameter; that is, the actual type can be derived from the formal type by substitution. Based on this substitution, the compiler will always be able to construct a sort of boxing function appropriate for the task.

Type	Representation	How to wrap and unwrap	
int	32-bit word	wrap_{int}	allocate 1-word record
		$\text{unwrap}_{\text{int}}$	fetch 1 word
char	8 bits	$\text{wrap}_{\text{char}}$	shift left and add 1
		$\text{unwrap}_{\text{char}}$	shift right
float	64 bits	$\text{wrap}_{\text{float}}$	allocate 8-byte record
		$\text{unwrap}_{\text{float}}$	fetch 8 bytes
(t_1, t_2)	t_1 (n words) catenated with t_2 (m words)	$\text{wrap}_{\text{tuple}}$	allocate $(n+m)$-word record
		$\text{unwrap}_{\text{tuple}}$	fetch $n+m$ words
$a \to b$	2-word closure: code pointer and env. pointer	$\text{wrap}_{\text{closure}}$	allocate 2-word record
		$\text{unwrap}_{\text{closure}}$	fetch 2 words
$\{\texttt{a} : t_1, \texttt{b} : t_2\}$	pointer to record	$\text{wrap}_{\text{record}}$	leave it as is
		$\text{unwrap}_{\text{record}}$	leave it as is
string	pointer to characters	$\text{wrap}_{\text{string}}$	leave it as is
		$\text{unwrap}_{\text{string}}$	leave it as is

TABLE 16.12. Wrapping and unwrapping of primitive types.

Table 16.12 shows how to *wrap* values of primitive types (int, char, string) by boxing or tagging.

Records by value or by reference? A Tiger-language record, like a Java object or a C pointer-to-struct, allows several kinds of operations:

1. Is a value there or is it nil?
2. Is it the same record (by pointer-equality) as that one?
3. What are the field values?
4. Let me update one of the fields!

But a C struct or ML record value has no pointer and cannot be "nil." Therefore, only operation (3) applies. The essential difference is between a *reference* and a *pure value*. The importance of this for representation analysis is that we can take advantage of concept (3) especially when (1), (2), and (4) don't get in the way: we can *copy* the record into a different format – we can keep a two-word C struct or ML record in two registers if we want to, and pass the record as a function-parameter in registers. That is, representation analysis can do more on pure values than it can on references.

To give an example of a record-style language construct that is a pure value, I have introduced *tuple* types into Table 16.12. The type (t_1, t_2), for example, is the type of two-element tuples whose first element has type t_1 and whose second has type t_2. Tuples are just like records without field names.

However, I have given these tuples a pure-value semantics: one cannot test a tuple for `nil`, update a field of a tuple value, or test pointer-equality on tuples. This is a design choice; the fact that records have field names and tuples do not is actually irrelevant to the distinction between references and pure values.

Recursive wrapping. For structured types such as (`int`, `char`) or (`int` → `char`), primitive wrapping (as shown in Table 16.12) can convert the value into a single boxed word. But this is not enough, as shown by this example:

```
let function get(l) = l.head.1
    function dup(x) = list{head=x,tail=list{head=x,tail=nil}}
    var tuple = (3,'a')
 in extract(dup(tuple))
end
```

If we primitive-wrap `tuple` by making a *boxed* tuple containing an *unboxed* integer and an *untagged* character, then the polymorphic function `get` will be directly handling the unboxed integer – which is not allowed.

To solve this problem we *recursively wrap:* we first wrap the components (bottom-up), then build a tuple of wrapped types. When a function is to be wrapped, we must recursively wrap its argument and result. Recursive wrapping is summarized in Table 16.13. Note that to recursively wrap a function `f` of type (`int` → `char`), we make a new function `f` that takes a boxed argument, *unwraps* the argument, applies `f`, and wraps the result. Thus, the recursive definition of `wrap` relies on `unwrap` for function-parameters, and vice versa.

These primitive wrapping functions will suffice when the actual parameter is a polymorphic variable. But when the type of the formal parameter is something like $a \rightarrow int$ or (int, a) where a is a polymorphic type variable, then we cannot simply box the entire actual parameter. Instead, we must make an unboxed function (or tuple, respectively) whose argument (or component, respectively) is boxed.

Let us use • as the symbol for a type variable holding a boxed value. We can do this because coercion-based analysis doesn't care *which* type variable has been used: they all have the same, boxed, representation.

Type	How to wrap and unwrap	
(t_1, t_2)	$\text{wrap}_{(t_1,t_2)}(x)$	$\text{wrap}_{\text{tuple}}(\text{wrap}_{t_1}(x.1), \text{wrap}_{t_1}(x.2))$
	$\text{unwrap}_{(t_1,t_2)}(x)$	$y = \text{unwrap}_{\text{tuple}}(x);\ (\text{unwrap}_{t_1}(y.1), \text{unwrap}_{t_1}(y.2))$
$t_1 \to t_2$	$\text{wrap}_{t_1 \to t_2}(f)$	$\text{wrap}_{\text{closure}}$(`let function fw(a)=` wrap_{t_2}(`f(`unwrap_{t_1}`(a))) in fw end`)
	$\text{unwrap}_{t_1 \to t_2}(f)$	`let function fu(a) =` unwrap_{t_2}(`f)(`wrap_{t_1}`(a)))` `in fu end`
{a : t_1, b : t_2}	$\text{wrap}_{\{a:t_1,b:t_2\}}(r)$	$\text{wrap}_{\text{record}}(r)$
	$\text{unwrap}_{\{a:t_1,b:t_2\}}(r)$	$\text{unwrap}_{\text{record}}(r)$

TABLE 16.13. Recursive wrapping and unwrapping of structured types.

Type of actual	Type of formal	Transformation
y : •	•	y
y : int	•	$\text{wrap}_{\text{int}}(y)$
y : char	•	$\text{wrap}_{\text{char}}(y)$
$y : (t_1, t_2)$	•	$\text{wrap}_{(t_1,t_2)}(y)$
$y : (t_1, t_2)$	$(t_1, \bullet)$	$(y.1, \text{wrap}_{t_2}(y.2))$
$y : (t_1, t_2)$	(•, •)	$(\text{wrap}_{t_1}(y.1), \text{wrap}_{t_2}(y.2))$
$f : t_1 \to t_2$	•	$\text{wrap}_{t_1 \to t_2}(f)$
$f : t_1 \to t_2$	$\bullet \to t_2$	`let function fw(a)= f(`unwrap_{t_1}`(a)) in fw end`
$f : t_1 \to t_2$	$\bullet \to \bullet$	`let function fw(a)=`wrap_{t_2}`(f(`unwrap_{t_1}`(a))) in fw end`

TABLE 16.14. Wrapping for partially polymorphic formal parameters.

To convert variable y from type $(int, char)$ to $(int, \bullet)$ we must create a new record whose first component is copied from the first component of y, and whose second component is $\text{wrap}_{\text{char}}$ applied to the second component of y.

To wrap a function $f : t_1 \to t_2$ into a boxed •, we recursively wrap f into a single pointer as shown in Table 16.13. But when the formal parameter is $x : \bullet \to t_2$, then that won't do: the called function expects a function-closure, not a box, and the return-value must be t_2, not a box. The compiler must construct a new function as shown in Table 16.14.

When a polymorphic function returns a result into a monomorphic con-

text, the return value must be unwrapped. If the result is completely polymorphic, then we can use an unwrapper from Table 16.12 or 16.13. But if it is something like $(t_1, t_2 \rightarrow \bullet)$, then we must pick it apart, unwrap *some* of the components, and rebuild it; the process is complementary to the one shown in Table 16.14.

Performance advantage. Coercion-based representation analysis relies on the fact that in typical programs, instantiation of polymorphic variables (which is where the coercions must be inserted) is rarer than ordinary execution. Representation analysis is particularly useful for programs that heavily use floating point numbers (which need lots of boxing/unboxing in the fully boxed translation scheme) or other data types for which tagging or boxing is expensive.

PASSING TYPES AS RUN-TIME ARGUMENTS

Another approach to the implementation of polymorphism is to keep data always in its natural representation. A function f with a polymorphic formal parameter x will tolerate different representations of x depending on the type of the actual parameter. To do this, f must be told the actual type of each instance.

Descriptions of the types of actual parameters can be passed exactly in those places where Poly-Tiger (or, equivalently, second-order lambda calculus) passes a type parameter. Consider the `randomzap` function as an example; its representation in Poly-Tiger is

```
function randomzap<a>(x:a) : a =
  let function f(y:a) : a = if random() then y else x
   in f
  end
```

and a sample use of it is

```
let var i0 := randomzap<int>(0)
    var s0 := randomzap<string>("zero")
 in i0(3)+size(s0("three"))
end
```

So far we have seen three ways to deal with the parameter `<a>`: substitute for it *(expansion of polymorphic functions)*, ignore it *(fully boxed translations)*, or treat it as a black box *(coercion-based representation analysis)*. But the most explicit thing we could do is to take it at face value: that is, to pass at run time a description of the type `a`.

The compiler can build run-time descriptions of types, not so different from the data structures summarized in Figure 16.3b. The primitive types can be statically bound to specific labels (such as `L_int_type`). Then the function `randomzap<a>(x:a)` can actually be translated as something like,

```
function randomzap(a:type,x:a) : a =
  let function f(y:a) : a =
        let var s = sizeof(a)
         in if random()
                then copy s bytes from y to result
                else copy s bytes from x to result
        end
   in f
  end
```

The type-description `a` is a free variable of the inner function `f`, and must be handled using closures as described in Section 15.2. The code in the `then`-clause inside `f` that moves a value of type `a` from its argument `y` to the return-value register must examine `a` to see how many words to move, and from what kind of register.

An interesting aspect of type passing is that the descriptions of types can also be used in the garbage-collector interface. The data does not need to describe itself using boxing, because each function knows the type of all its variables – and the polymorphic functions know which variables describe the types of *other* variables. Type passing also enables the introduction of a *typecase* facility (see Table 14.6 on page 301).

Type passing has certain implementation challenges. Descriptions of types must be constructed at run time, as in the `append` function (page 347), which receives a type-description `e` and must construct the description `list<e>`. One must take care that constructing these descriptions does not become too costly. Also, a polymorphic function must treat its variables differently depending on what the type parameters say, and this can become costly.

16.4 RESOLUTION OF STATIC OVERLOADING

Some languages permit *overloading:* different functions of the same name but different argument types. The compiler must choose between function-bodies based on the types of the actual parameters. This is sometimes known as *ad hoc polymorphism*, as opposed to the *parametric polymorphism* described in the previous sections.

Static overloading is not difficult to implement. When processing the declaration of an overloaded function f, the new binding b_n must not hide the old definitions $b_1, \ldots, b_{n-1}$. Instead, the new binding maps f to a list of different implementations, $f \mapsto [b_1, \ldots, b_n]$. Depending on the language semantics, it may be necessary to give an error message if b_n has identical parameter types to one of the b_i.

Then, when looking up f in a place where it is called with actual parameters, the types of the actual parameters will determine which of the bindings b_i should be used.

Some languages allow functions of identical argument types (but different result type) to be overloaded; some languages allow forms of dynamic overloading; see the Further Reading section.

FURTHER READING

One of the first "polymorphic" languages was Lisp [McCarthy 1960], which has no static (i.e., compile-time checkable) type system at all. Consequently, the fully boxed implementation of data was used, so that the data could describe itself to the run-time type-checker as well as to the garbage collector.

The Hindley-Milner type system was invented for combinatory logic by Hindley [1969] and for the ML programming language by Milner [1978]. Similarly, second-order lambda calculus was invented for logic by Girard [1971] and for programming by Reynolds [1974]. Harper and Mitchell [1993] show how programs using Hindley-Milner types can be translated into second-order lambda calculus (e.g., how ImplicitPoly-Tiger can be automatically translated into Poly-Tiger). Mitchell [1990] explains the theoretical aspects of polymorphic type systems.

The first programming language to use implicit parametric polymorphism was ML, which was originally the *M*eta*L*anguage of the Edinburgh theorem prover [Gordon et al. 1978] but was later developed into a general-purpose programming language [Milner et al. 1990]. Cardelli [1984] describes a fully boxed implementation of ML. Leroy [1992] describes coercion-based representation analysis, Shao and Appel [1995] describe a variant that does recursive wrapping only when necessary, and Shao [1997] shows a more general scheme that combines coercion-based and type-passing styles, and also works on explicitly typed languages such as Poly-Tiger. Harper and Morrisett [1995] and Tolmach [1994] describe type-passing style.

Type inference for ML takes exponential time in the worst case [Mairson 1990], but in practice it runs quickly: the particular arrow-type structures that cause the worst-case behavior have not been observed in real programs. When polymorphic recursion is permitted, then type inference is no longer a computable problem [Henglein 1993; Kfoury et al. 1993].

In the Ada programming language [Ada 1980], the *generic* mechanism allows a function (in fact, an entire package) to be parameterized over types; but full type-checking is done at each call site after the generic is applied to actual parameters, and the *expansion* technique of implementation must be used.

Overloading. Ada allows different functions *with the same parameter types* to be overloaded, as long as the result types are different. When the output of such a function is an argument to another overloaded identifier, then there may be zero, one, or many possible interpretations of the expression; the Ada semantics say that the expression is legal only if there is exactly one interpretation. Aho et al. [1986, section 6.5] discuss this issue and give a resolution algorithm. But Ada-style overloading has not been widely imitated in recent language designs, perhaps because it can confuse the programmer.

Dynamic overloading allows different implementations of a function to be chosen based on the *run-time* type of an actual parameter; another name for this is *overriding*, and it is a fundamental concept of object-oriented programming (see Chapter 14). *Type classes* in the Haskell language allow overloading and parametric polymorphism to interact in a useful and expressive way [Hall et al. 1996].

EXERCISES

16.1 Show the steps in type-checking the declaration of `append` on page 347 using Algorithm 16.8.

***16.2** Algorithm 16.8 shows how to type-check the declarations and calls of single-argument functions and records. Write a version (in the same notational style) that covers the cases where there are multiple type arguments, multiple value arguments, and multiple record fields; that is, complete these clauses:

$$transdec(\sigma_v, \sigma_t, \texttt{function}\ f\langle z_1, \ldots, z_k\rangle(x_1 : t_1, \ldots, x_n : t_n) : t_r = e_{\text{body}}) =$$
$$transexp(\sigma_v, \sigma_t, e_f\langle t_1, \ldots, t_k\rangle(e_1, \ldots, e_n)) =$$
$$transexp(\sigma_v, \sigma_t, rcrd\langle t_1, \ldots, t_k\rangle\{fld_1 = e_1, \ldots, fld_n = e_n\}) =$$

16.3 Show the results of calling *unify* on the following pairs of arguments: Is the result *OK* or *error*? What bindings are added to σ_m? The symbols $\alpha, \beta, \ldots$ stand for **Meta** types, and in each case σ_m is initially empty.

a. (α, `int` $\rightarrow \gamma$) and (`int` $\rightarrow \gamma$, β).

b. $\alpha \rightarrow \alpha$ and α.

c. (`list<`β`>`, $\alpha \rightarrow$ `string`) and (`list<`α`>`, $\beta \rightarrow$ `string`).

d. $\alpha \rightarrow \alpha$ and `int` $\rightarrow$ `string`.

e. ($\alpha, \alpha, \alpha, \beta, \beta$) and ($\delta$, `int`, γ, γ, δ).

***16.4** Show an algorithm for composing two substitutions. That is, given σ_1 and σ_2, construct σ_{12} such that $subst(t, \sigma_{12}) = subst(subst(t, \sigma_1), \sigma_2)$ for any type t. Then show how Algorithm 16.4 can be written more efficiently.

***16.5** Show the steps in type-inferencing the declaration of `randomzap` (page 358) followed by the expression `randomzap(0)`, using Algorithm 16.10.

16.6 Translate the following declarations from ImplicitPoly-Tiger to Poly-Tiger.

a.
```
type list<e> = {head: e, tail: list<e>}
function map(f,l) =
       if l=nil then nil
       else list{head=f(l.head),tail=map(f,l.tail)}
```

b.
```
type list<e> = {head: e, tail: list<e>}
function fold(f,z) =
  let function helper(l) =
       if l=nil then z else f(l.head,helper(l.tail))
   in helper
  end
function add(i,j) = i+j
var x := fold(add,0)(list{head=3,tail=
                          list{head=5,tail=nil}})
```

***16.7** There is a difficulty in translating the following program from ImplicitPoly-Tiger to Poly-Tiger:

```
let function f(s) = let function g(y) = y
                     in print(s);  g
                    end
    var f1 := f("beep")
 in size(f1("three"))+f1(3)
end
```

a. What integer does this expression return, and what gets printed during the evaluation of the expression?

b. Apply Algorithm 16.10 to this program to demonstrate that it is well typed in ImplicitPoly-Tiger. (Remember that `print` takes a string argument; it is not polymorphic.) **Hint:** The type of `f` is **Poly**([z], **App**(**String**, [ ]) → (**Var**(z) → **Var**(z))).

c. Show how the declaration `function f(s)=...` is translated to Poly-Tiger. **Hint:** Because the type of `f` is **Poly**([z], ...), the explicitly typed function should begin `function f<z>(s:string) = ...`

d. Translate `var f1 := f("beep")`. **Hint:** Somewhere in that process you'll have `f<t>("beep")`; but where does `t` come from? The usual solution is to make up a new function

```
var f1: ? = let function h<t>(): ? = f<t>("beep") in h end
```

but I'll let you fill the omitted types. The function `h` has a type parameter but no value parameter!

e. Translate the expression `size(f1("three"))+f1(3)`.

f. What gets printed during the evaluation of the translated expression? **Hint:** It's not the same as what's printed in part (a).

One way to avoid this problem [Wright 1995] is to restrict the implicitly polymorphic language (such as ML or ImplicitPoly-Tiger) such that expressions containing top-level effects (such the function call `f("beep")`) must not be polymorphic.

16.8 On a 32-bit machine, let us represent *pointers* with the last two bits 00 (because they point to word-aligned data), and *integers* with the last bit 1. The 31-bit integer value x will be represented as $x' = 2x + 1$. Show the best sequence of ordinary machine instructions to implement each of the following expressions on tagged values. In each case, compare the length of this instruction sequence with the instructions needed to implement the same expression on 32-bit *untagged* values.

a. $c' \leftarrow a' + b'$

b. $c' \leftarrow a' + 3$

c. $c' \leftarrow a' \times b'$.

d. A conditional branch on $a' < b'$.

e. The basic block $c' \leftarrow a' \times b'$; $s' \leftarrow s' + c'$ with c' dead afterwards.

f. Now suppose you wish to have the computer's *overflow* flags set if any computation computes a value that cannot fit in a 31-bit signed integer. But your computer only calculates this for its 32-bit calculations. Analyze each of the instruction sequences above to see whether it sets the *overflow* appropriately in all cases.