Scribe: Lin Guo, Feng Shao | Lecturer: Andrew Myers

1 Type Equivalence

1.1 Introduction

In the C language, we can use the **typedef** command to get equivalent types as follows:

$$\text{typedef int foo} \Rightarrow \text{foo} \cong \text{int}$$

which means foo and int indicate the same type. So the two types can be used interchangeably, e.g.:

$$\text{foo x;} \quad \text{int y;} \quad \text{x = y;}$$

In other words, foo is an alias for the type int. On the other hand, two types can be equivalent based on their structure. For example, in ML we have:

$$\{\text{x: int, y: int}\} \cong \{\text{y: int, x: int}\}$$

This equation tells us the order in the record structure is unimportant, which makes sense since we get the element of a record by a term like e.x, which has nothing to do with the order of the fields. For implementation reasons, similar types in many languages(e.g. C) are not considered equivalent.

1.2 Type equivalence proof system

In general, there are multiple ways to write a single type. Thus it is important to determine when type expressions τ_1 and τ_2 represent the same type, which is written as $\vdash \tau_1 \cong \tau_2$. The typing rule that allows us to make use of equivalence is this:

$$\frac{\Gamma \vdash e : \tau_1 \quad \vdash \tau_1 \cong \tau_2}{\Gamma \vdash e : \tau_2} \text{ (equiv)}$$

This means if e has type τ_1 and τ_1 and τ_2 are equivalent types, then e is also has type τ_2 in the same context.

To obtain a complete type equivalence system, we need other rules:

$$\frac{}{\vdash \tau \cong \tau} \qquad \frac{\vdash \tau' \cong \tau}{\vdash \tau \cong \tau'} \qquad \frac{\vdash \tau_1 \cong \tau_2 \quad \vdash \tau_2 \cong \tau_3}{\vdash \tau_1 \cong \tau_3} \qquad \text{(equivalence rules)}$$

$$\frac{\vdash \tau_1 \cong \tau_1' \quad \vdash \tau_2 \cong \tau_2'}{\vdash \tau_1 * \tau_2 \cong \tau_1' * \tau_2'} \qquad \frac{\vdash \tau_1 \cong \tau_1' \quad \vdash \tau_2 \cong \tau_2'}{\vdash \tau_1 + \tau_2 \cong \tau_1' + \tau_2'} \qquad \frac{\vdash \tau_1 \cong \tau_1' \quad \vdash \tau_2 \cong \tau_2'}{\vdash \tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'} \qquad \text{(congruence rules)}$$

1.3 Name vs. Structural equivalence

There are two kinds of type equivalence: name equivalence and structural equivalence. For example, C has name equivalence, so

$$\text{struct foo\{int x; int y\}} \ncong \text{struct bar\{int z; int w\}}$$

This is because the name of a particular data structure is part of its type identity in C and obviously the name **foo**$\neq$ the name **bar**. So even if we redefine structure **bar** like:

$$\text{struct bar\{int x; int y\}}$$

The above inequation also holds.
C also allows you write a data structure without an explicit name, like struct {int x; int y}. But in fact, the compiler will fabricate a new hidden name for this structure that makes it different from every other *struct*.

We say **struct** in **C** is a *generative* type constructor because each use of **struct** generates a new hidden name that defines the identity of the type. Thus, with:

$$\begin{aligned}\mathsf{typedef\ struct\{int\ x\}\ foo;}\\ \mathsf{typedef\ struct\{int\ x\}\ bar;}\\ \mathsf{typedef\ foo\ baz;}\end{aligned}$$

We have $\mathsf{foo} \cong \mathsf{baz}$ but $\mathsf{foo} \ncong \mathsf{bar} \ncong \mathsf{baz}$.
Modula-3 has completely structural equivalence. For example:

$$\begin{aligned}\mathsf{TYPE\ foo} &= \{\mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}\}\\ \mathsf{TYPE\ bar} &= \{\mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}\}\end{aligned}$$

Then foo and bar are equivalent types. To achieve the effect of name equivalence in **Modula-3**, one can give a type a unique identity using the type constructor BRANDED. If we modify the definition of foo and bar as follows:

$$\begin{aligned}\mathsf{TYPE\ foo} &= \mathsf{BRANDED}\{\mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}\}\\ \mathsf{TYPE\ bar} &= \mathsf{BRANDED}\{\mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}\}\end{aligned}$$

Then $\mathsf{foo} \ncong \mathsf{bar}$ because BRANDED is generative.

1.4 An example of structural equivalence

For an example of nontrival equivalent types, we can imagine extending our type system with an axiom $\vdash (\tau_1 * \tau_2) \to \tau_3 \cong \tau_1 \to (\tau_2 \to \tau_3)$. This would let us curry functions in $\lambda^{\to}$, e.g.:

$$\begin{gathered}\mathsf{let\ f} = \lambda\mathsf{x{:}int.}\ \lambda\mathsf{y{:}int.e\ in\ f}\langle\mathsf{1,2}\rangle\\ \text{and}\\ \mathsf{let\ f} = \lambda\mathsf{x{:}int.}\lambda\mathsf{y{:}int.e\ in\ f(left\ e)(right\ e)}\end{gathered}$$

would be two ways to write the same function.
In fact, even if we have no such type equivalence system (or no such rule), we can transform a lambda term $e : \tau_1 \to (\tau_2 \to \tau_3)$ to a term $e' : (\tau_1 * \tau_2) \to \tau_3$ using the function *uncurry* and the typing derivation. Recall the definition of *uncurry* and *curry*:

$$\begin{aligned}uncurry_{\tau_1,\tau_2,\tau_3} &= \lambda f : \tau_1 \to (\tau_2 \to \tau_3).\lambda p : \tau_1 * \tau_2.f(\mathsf{left}\ p)(\mathsf{right}\ p)\\ curry_{\tau_1,\tau_2,\tau_3} &= \lambda f : (\tau_1 * \tau_2) \to \tau_3.\lambda g : \tau_1.\lambda h : \tau_2.f(g,\ h)\end{aligned}$$

1.5 Equivalence and type-directed translations

One way to justify the soundness of a type system extension such as type equivalence is to show that we can translate well-typed terms in the extended type system into well-typed terms in a simpler type system. This is done by giving a translation from source *type derivations* to target type derivations. Thus, the translations have the same flavor as our denotational semantics for typed languages, but the target will be a typed programming language rather than mathematical objects. A *type-directed translation* will in general consist of a type translation $\mathcal{T}[\![\tau]\!] = \tau'$ that tells how to translate source types into target types and a translation $[\![\Gamma \vdash e : \tau]\!] = e'$ that tells how to translate source type derivations into target derivations. $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]$ has the form $\Gamma' \vdash e' : \mathcal{T}[\![\tau]\!]$, but typically we don't bother to write down the typing derivation for the target-language term, just the term e' itself. We also define a function $\mathcal{G}[\![\Gamma]\!]$ that translates a valid typing context Γ to a valid target-language typing context Γ' by simply mapping all the types in the context using $\mathcal{T}[\![\cdot]\!]$.

For example, we can define a type-directed translation from the lambda calculus to itself as follows:

$$
\begin{array}{rcl}
\mathcal{T}[\![B]\!] & = & B \\
\mathcal{T}[\![\tau \to \tau']\!] & = & \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!] \\
\mathcal{G}[\![x_1{:}\tau_1, \ldots, x_n{:}\tau_n]\!] & = & x_1{:}\mathcal{T}[\![\tau_1]\!], \ldots x_n{:}\mathcal{T}[\![\tau_n]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\left[\!\!\left[\dfrac{\vdash \Gamma}{\Gamma, x{:}\tau \vdash x : \tau} \right]\!\!\right] & = & x \left(\textit{i.e.,} \quad \dfrac{\vdash \mathcal{G}[\![\Gamma]\!]}{\mathcal{G}[\![\Gamma]\!], x{:}\mathcal{T}[\![\tau]\!] \vdash x : \mathcal{T}[\![\tau]\!]} \right) \\
\left[\!\!\left[\dfrac{\Gamma \vdash e_0 : \tau \to \tau' \quad \Gamma \vdash e_1 : \tau \quad \vdash \Gamma}{\Gamma \vdash e_0\ e_1 {:} \tau'} \right]\!\!\right] & = & [\![\Gamma \vdash e_0 : \tau \to \tau']\!]\ [\![\Gamma \vdash e_1 : \tau]\!] \\
\left[\!\!\left[\dfrac{\Gamma, x{:}\tau \vdash e : \tau' \quad \vdash \Gamma}{\Gamma \vdash (\lambda x{:}\tau.\, e){:}\tau \to \tau'} \right]\!\!\right] & = & \lambda x{:}\mathcal{T}[\![\tau]\!].\, [\![\Gamma, x{:}\tau \vdash e : \tau']\!]
\end{array}
$$

In this case the type and term translations are identity functions, but they have been expressed in a way that allows them to be extended. Showing that this translation is correct involves more than the well-formedness of the translated term, of course; we also want to show that the translation is sound and adequate; that is, that operational evaluation of the source term mirrors operational evaluation of the target term. This can be shown using techniques we have seen earlier in this course, such as logical relations. If the translation is simple enough, one can appeal to simpler arguments.

Now let us consider how we might justify type equivalence via a typed translation. The essence of type equivalence is the automatic generation of an isomorphism that witnesses the equivalence by showing how to map back and forth. Therefore, for a given judgement $\vdash \tau_1 \cong \tau_2$, we can define a target-language function $\mathcal{I}[\![\vdash \tau_1 \cong \tau_2]\!] : \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!]$ that captures one half of the isomorphism pair, and a corresponding function $\mathcal{I}^\dagger[\![\vdash \tau_1 \cong \tau_2]\!] : \mathcal{T}[\![\tau_2]\!] \to \mathcal{T}[\![\tau_1]\!]$ that captures the other half. Given the function $\mathcal{I}$, we can extend the type-directed translation above in a generic way to capture use of the type equivalence rule (equiv); this corresponds to a compiler automatically introducing some coercion code $\mathcal{I}[\![\vdash \tau_1 \cong \tau_2]\!]$ to convert the value of type τ_1 to the type τ_2.

$$
\left[\!\!\left[\dfrac{\Gamma \vdash e : \tau_1 \quad \vdash \tau_1 \cong \tau_2}{\Gamma \vdash e : \tau_2} \right]\!\!\right] \quad = \quad \mathcal{I}[\![\vdash \tau_1 \cong \tau_2]\!]\ [\![\Gamma \vdash e : \tau_1]\!] : \mathcal{T}[\![\tau_2]\!]
$$

The conversion functions $\mathcal{I}[\![\cdot]\!]$ are straighforward; as we've just observed, *curry* and *uncurry* witness the equivalence used in our example:

$$
\begin{array}{rcl}
\mathcal{I}[\![\vdash \tau_1 \to (\tau_2 \to \tau_3) \cong (\tau_1 * \tau_2) \to \tau_3]\!] & = & \mathit{uncurry}_{\tau_1,\tau_2,\tau_3} \\
\mathcal{I}[\![\vdash (\tau_1 * \tau_2) \to \tau_3 \cong \tau_1 \to (\tau_2 \to \tau_3)]\!] & = & \mathit{curry}_{\tau_1,\tau_2,\tau_3}
\end{array}
$$

Defining the action of the type conversion function $\mathcal{I}[\![\cdot]\!]$ by induction on the derivation of type equivalence rules is straightforward for the basic rules that capture the congruence properties of $\cong$; the rest of the definition follows naturally.

$$
\begin{array}{rcl}
\mathcal{I}[\![\vdash \tau \cong \tau]\!] & = & \mathcal{I}^\dagger[\![\vdash \tau \cong \tau]\!] = \lambda x{:}\mathcal{T}[\![\tau]\!].\, x \\
\mathcal{I}\left[\!\!\left[\dfrac{\vdash \tau' \cong \tau}{\vdash \tau \cong \tau'} \right]\!\!\right] & = & \mathcal{I}^\dagger[\![\vdash \tau' \cong \tau]\!] \\
\mathcal{I}\left[\!\!\left[\dfrac{\vdash \tau_1 \cong \tau_2 \quad \vdash \tau_2 \cong \tau_3}{\vdash \tau_1 \cong \tau_3} \right]\!\!\right] & = & \lambda x{:}\mathcal{T}[\![\tau_1 \to \tau_3]\!].\, [\![\vdash \tau_2 \cong \tau_3]\!]\ ([\![\vdash \tau_1 \cong \tau_2]\!]\ x) \\
\mathcal{I}\left[\!\!\left[\dfrac{\vdash \tau_1 \cong \tau_1' \quad \vdash \tau_2 \cong \tau_2'}{\vdash \tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'} \right]\!\!\right] & = & \lambda x{:}\mathcal{T}[\![\tau_1 \to \tau_2]\!].\, \lambda y{:}\mathcal{T}[\![\tau_1']\!].\, \mathcal{I}[\![\vdash \tau_2 \cong \tau_2']\!]\ (x\ (\mathcal{I}^\dagger[\![\vdash \tau_1 \cong \tau_1']\!]\ y))
\end{array}
$$

2 Recursive Types

At present, our simple type system has the limitation that it cannot describe such type structures that may grow to arbitrary size, but that have a simple, regular structure. The following are two infinite-shape types for *List*

$$\begin{aligned} List_1 &= 1 + \tau + \tau \times \tau + \tau \times \tau \times \tau + ... \\ List_2 &= 1 + (\tau \times (1 + \tau \times (1 + ...))) \end{aligned}$$

There are other examples in actual programming languages,

$$\begin{aligned} &\text{in ML :} \quad \mathsf{type\ \ Alist} = \mathsf{\ empty \mid full\ of\ A} * \mathsf{Alist} \\ &\text{in C :} \quad \mathsf{struct\ \ Alist\ \{\ A\ elem;\ struct} * \mathsf{\ \ Alist;\}} \end{aligned}$$

So we need a general mechanism with which they can be defined from simpler elements, as needed. This mechanism is **recursive types**.

Consider the recursive equation specifying the infinite type *Alist* in a Modula-3 like language:

$$\mathsf{type\quad Alist\ =\ ref\{elem : A\ ,\ next : Alist\}}$$

It is similar to the equation specifying the recursive factorial function. We can **unfold** it to get $Alist_2$,

$$\mathsf{type\quad Alist_2\ =\ ref\ \{\ \ elem : A\ ,\ \ \ next : ref\{\ elem\ A\ ,\ \ \ next\ Alist\}\ \}}$$

Alist and $Alist_2$ are describing the same type shape: $Alist \cong Alist_2 \cong 1 + A \times (1 + A \times (1 + ...))$.

Define $Alist \equiv \tau$ and a function $\mathcal{F}(\tau) = 1 + A \times \tau$. Because $\tau \cong 1 + A \times \tau$, we have $\mathcal{F}(\tau) \cong \tau$. This means τ is a fixed point of $\mathcal{F}$!. Here, as there, it is convenient to make the equation of *Alist* into a proper definition by introducing an explicit recursion operator μ for types.

2.1 Recursive Type Operator

We use $\mu X.\tau$ to denote the solution to $X \cong \tau$ where τ may mention X. (Here X is a type variable.) Using it, $Alist \equiv \mu X.1 + A \times X$. Similarly, we can define many other types such as binary tree of integers and the natural numbers:

$$\begin{aligned} \textit{Binary tree of integers} \quad &: \quad IntTree \equiv \mu T.\ \ Int + T \times T \quad (IntTree \cong Int + IntTree \times IntTree) \\ \textit{Natural Numbers} \quad &: \quad Nat \equiv \mu N.\ \ 1 + N \quad (Nat \cong 1 + Nat) \\ &\quad\ \ 0 \equiv \mathsf{inl}(\#\mathsf{u}) \\ &\quad\ \ 1 \equiv \mathsf{inr}(\mathsf{inl}(\#\mathsf{u})) \\ &\quad\ \ 2 \equiv \mathsf{inr}(\mathsf{inr}(\mathsf{inl}(\#\mathsf{u}))) \\ &\qquad\quad ... \\ &\quad\ \ succ : \ Nat \to Nat \equiv \lambda n : Nat.\ \ \mathsf{inr}(n) \end{aligned}$$

2.2 Open and Closed Recursion

The μ operator introduces closed recursion: the new type variable X can only be mentioned in the scope of $\mu X.\tau$. In many languages, types are allowed to refer to one another arbitrarily. **Open** recursion occurs when the type expression is not closed. **Open** recursion requires taking a fixed point over all types in scope. For example, consider the following class definitions for Node and Edge.

$$\begin{aligned} &\mathsf{Class} \quad \mathsf{Node\{\ \ Edge[\ ]\ \ edges;\}} \\ &\mathsf{Class} \quad \mathsf{Edge\{\ \ Node\ \ from,\ \ to;\}} \end{aligned}$$

Note that Node refers to Edge and vice versa. So, we must take a fixed point when assigning their types. Thus, we have

$$\begin{aligned} \mathsf{Node} &= \mu \mathsf{N}.\ \mathsf{array}[\mathsf{N} \times \mathsf{N}] \\ \mathsf{Edge} &= \mu \mathsf{E}.\ (\mathsf{array}[\mathsf{E}] \times \mathsf{array}[\mathsf{E}]) \times (\mathsf{array}[\mathsf{E}] \times \mathsf{array}[\mathsf{E}]) \end{aligned}$$

Other mutually recursive types can b expressed using μ constructor similarly.

3 Fold and Unfold

By taking fixed points over types, we have created an issue of type equivalence in $\lambda^{\rightarrow *+}$. The difficulty is that we no longer have one unique syntactic form for a given type. Now if we suppose that $\mu X.\tau$ solves $X = \tau$, this suggests we can substitute $\mu X.\tau$ for X wherever it appears in τ . So, for the natural numbers, we get $\mu N.\ (1 + N) = \mu N.(1 + (\mu N.(1 + N)\ )\ )$ *and* $Nat = (1 + Nat) = (1 + (1 + Nat)\ )$

In order to develop a new formal definition of type equivalence, we will define the **unfolding** of type $\mu X.\tau$ to be $\tau\{\mu X.\tau/X\}$. And, we will write $\mu X.\tau \cong \tau\{\mu X.\ \tau/X\}$ to mean that $\mu X.\tau$ and its unfolding are equivalent types. Implicitly, this gives us the following notion of equivalence: type expressions are equivalent if they are fully substitutable for each other. A weaker notion of equivalence is that the types are isomorphic and the expressions must be explicitly mapped between types.

It will be more straightforward for now if we consider a recursive type and its unfolding to be isomorphic and explicitly shift values between the two types. In order to move between the recursive type $\mu X.\tau$ and the corresponding unfolding $\tau\{\mu X.\tau/X\}$, we define two new operators named fold and unfold that witness the isomorphism. We can consider them to have the following types:

$$\begin{aligned} \mathsf{fold}_{\mu X.\tau} &: \tau\{\mu X.\tau/X\} \rightarrow \mu X.\tau \\ \mathsf{unfold}_{\mu X.\tau} &: \mu X.\tau \rightarrow \tau\{\mu X.\tau/X\} \end{aligned}$$

The *unfold* operator is the elimination form that allows access to internals of value of recursive type; the *fold* operator is the corresponding introduction form. The typing rules are:

$$\frac{\Gamma \vdash e : \tau\{\mu X.\tau/X\}}{\Gamma \vdash \mathsf{fold}_{\mu X.\tau}\ \ e : \mu X.\tau}$$

$$\frac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \mathsf{unfold}\ \ e : \tau\{\mu X.\tau/X\}}$$

We extend the language as follows:

$$\begin{aligned} \tau &::= \ \ldots \mid \mu X.\tau \\ e &::= \ \ldots \mid \mathsf{fold}_{\mu X.\tau}\ e \mid \mathsf{unfold}\ e \\ v &::= \ \ldots \mid \mathsf{fold}_{\mu X.\tau}\ v \\ C &::= \ \ldots \mid \mathsf{fold}\ C \mid \mathsf{unfold}\ C \end{aligned}$$

with the new reduction:

$$\mathsf{unfold}(\mathsf{fold}_{\mu X.\tau}\ v) \longmapsto v$$

As an example of using the typing and evaluation rules, we prove that $\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ can be typed as α, where $x : \tau$ and $x : \tau \rightarrow \alpha$. Let $SA \equiv \lambda x : \tau\ \ (x\ x)$, we get that $\tau \cong \tau \rightarrow \alpha$ and $\mu T.\ T \rightarrow \alpha$ is solution to it. By unfolding,

$$(\mu T.T \rightarrow \alpha) \Longrightarrow (\mu T.T \rightarrow \alpha) \rightarrow \alpha$$

Then we get the following declarations to explicitly show the types of fold and unfold as applied to SA:

$$\begin{aligned} SA &= \mathsf{fold}_{\mu T.\ T \rightarrow \alpha}(\lambda x : \mu T.\ T \rightarrow \alpha.\ \ (\mathsf{unfold}\ x)\ x) \\ \Omega &= (\mathsf{unfold}\ \ SA)\ \ SA \end{aligned}$$

We can prove that Ω can be typed as any type α by first proving that SA can be typed as $\mu T.\ T \to \alpha$. Here is the derivation:

$$\dfrac{\dfrac{\dfrac{\{x : \mu T.\ T \to \alpha\} \vdash x : \mu T.\ T \to \alpha}{\{x : \mu T.\ T \to \alpha\} \vdash (\mathsf{unfold}\ x) : (\mu T.\ T \to \alpha) \to \alpha} \quad \{x : \mu T.\ T \to \alpha\} \vdash x : \mu T.\ T \to \alpha}{\{x : \mu T.\ T \to \alpha\} \vdash (\ (\mathsf{unfold}\ x)\ x)\ ) : \alpha}}{\dfrac{\emptyset \vdash (\lambda x : \mu T.\ T \to \alpha.\ \ (\mathsf{unfold}\ x)\ x)\ ) : (\mu T.\ T \to \alpha) \to \alpha}{\emptyset \vdash \mathsf{fold}_{\mu T.\ T \to \alpha}(\lambda x : \mu T.\ T \to \alpha.\ \ (\mathsf{unfold}\ x)\ x) : \mu T.\ T \to \alpha}}$$

Now we prove that $\Omega : \alpha$:

$$\dfrac{\dfrac{\emptyset \vdash SA : \mu T.\ T \to \alpha}{\emptyset \vdash (\mathsf{unfold}\ SA) : (\mu T.\ T \to \alpha) \to \alpha} \quad \emptyset \vdash SA : \mu T.\ T \to \alpha}{\emptyset \vdash (\mathsf{unfold}\ SA\ )\ \ SA\ : \alpha}$$

It is perhaps surprising that Ω can be given any type α. However, this is okay because nontermination is allowed by any type α.

Another example is $Nat \equiv \mu N.\ 1 + N$

$$\begin{aligned} 0 &\equiv \mathsf{inl}(\#\mathsf{u}) = \mathsf{fold}_{Nat}\ \mathsf{inl}(\#\mathsf{u}) \\ 1 &\equiv \mathsf{fold}_{Nat}\mathsf{inr}(0) \\ &\ldots \\ succ &\equiv \lambda n : Nat.\ \mathsf{fold}_{Nat}(\mathsf{inr}(n)) \end{aligned}$$

Thus, although integers are convenient, in principle we don't need to have them baked into our type system if we have recursive types!