# Lecture 16: Type Equivalence

Carsten Schürmann

October 15, 2001

## 1 Type equivalence

When shall we consider two type to be equivalent? In class we had long discussions about how to do it, here is what the Tiger semantics dictates. Somewhat surprisingly, Tiger requires a mixture between name equivalence and structural equivalence. With *name equivalence*, two types are considered equal if and only if their names are the same. If a programmer declares

```
type a = int
type b = int
```

type `a` is equal to `a`, and `a` is not equal to `b`. The same holds for records and arrays. With *structural equivalence*, we would consider `a` and `b` to be equivalent, and so are the two following declarations.

```
type c = {x:int, y:int}
type d = {x:int, y:int}
```

*Alias types* introduce new names for old types — with the special property that two type aliases are always considered equivalent. What this means is that in declaration

```
type e = c
type f = e
```

`e`, `f`, and `c` are considered equivalent. Note, that the use of alias types brings additional problems. There may be cycles in alias type definitions. Cycles are allowed as long the cycle contains at least one record or array type. We call those type definitions *productive*. The goal is that every type definition in Tiger must be productive.

**Example 1.1 (Non-productive type definitions)**

```
type a = b
type b = c
type c = a
```

**Example 1.2 (Productive type definition)**

```
type a = {head : int, tail : b}
type b = c
type c = a
```

But how can we decide if a list of type declarations is productive or not? The answer is that we need to look at all type constants that are being defined simultaneously. Using substitution where one type constructor is replaced by an equivalent one, we try to boil a list of type declarations down until it contain $a = a$. If it does, we found a cycle, and the list of of type declarations is non-productive.

Formally, we write $[a/b]d$ for replacing all type constructors $b$ by $a$ in $d$. Moreover, we write

$$\vdash d \text{ prod}$$

for the judgment that expresses that $d$ is productive. It is defined by the following for rules.

$$\frac{}{\vdash \cdot \text{ prod}}\ \textsf{p\_base} \qquad \frac{\vdash ([a/b]d_1, [a/b]d_2) \text{ prod}}{\vdash (d_1, b = a, d_2) \text{ prod}}\ \textsf{p\_alias}\ \text{ for } a \neq b$$

$$\frac{\vdash (d_1, d_2) \text{ prod}}{\vdash (d_1, b = \{l_1 : \tau_1 \ldots l_n : \tau_n\}, d_2) \text{ prod}}\ \textsf{p\_record}$$

$$\frac{\vdash (d_1, d_2) \text{ prod}}{\vdash (d_1, b = \text{array } \tau, d_2) \text{ prod}}\ \textsf{p\_array}$$

Note, that there is no rule with $d_1, a = a, d_2$ in the conclusion. For reasons mentioned above, these are exactly the cases we want to exclude from our consideration.

**Example 1.3 (Productivity)** The list of declarations $d = (a = \{h : \text{int}, t : b\}, b = c, c = a, \cdot)$ is productive.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \cdot \text{ prod}}\ \textsf{p\_base}}{\vdash (b = a, \cdot) \text{ prod}}\ \textsf{p\_alias}}{\vdash (b = c, c = a, \cdot) \text{ prod}}\ \textsf{p\_alias}}{\vdash (a = \{h : \text{int}, t : b\}, b = c, c = a, \cdot) \text{ prod}}\ \textsf{p\_record}$$

Tiger's semantics specifies a mixture of name equivalence and type aliases. We do not use structural type equivalence at all. As judgment to express the equivalence of two types in $\Delta$, we write

$$\Delta \vdash \tau_1 \equiv \tau_2.$$

Its meaning is defined by the following set of inference rules, where $\tau$ is a type and $a, b, c$ are type constructors.

$$\frac{}{\Delta \vdash \text{int} \equiv \text{int}}\ \textsf{eq\_int} \qquad \frac{}{\Delta \vdash \text{string} \equiv \text{string}}\ \textsf{eq\_string}$$

$$\frac{\Delta(a) = \tau}{\Delta \vdash a \equiv \tau}\ \textsf{eq\_alias}$$

$$\frac{}{\Delta \vdash a \equiv a}\ \textsf{eq\_ref} \qquad \frac{\Delta \vdash a \equiv b}{\Delta \vdash b \equiv a}\ \textsf{eq\_sym} \qquad \frac{\Delta \vdash a \equiv b \quad \Delta \vdash b \equiv c}{\Delta \vdash a \equiv c}\ \textsf{eq\_trans}$$

We can easily show that no two record or array types can be equivalent. In summary, we must add a type conversion rule to Tiger, that accounts for equivalent types.

$$\frac{\Gamma; \Delta \vdash^{\iota} e : \tau_1 \qquad \Delta \vdash \tau_1 \equiv \tau_2}{\Gamma; \Delta \vdash^{\iota} e : \tau_2}\ \textsf{conv}$$

Furthermore, we must augment the original let rule from the previous lecture by the requirement that $d$ is productive.

$$\frac{\Gamma \vdash^{\iota} d/e : \tau \qquad \vdash d\ \text{prod}}{\Gamma \vdash^{\iota} \text{let } d \text{ in } e \text{ end} : \tau}\ \textsf{let}$$

# 2 A Computable Criterion

For practical purposes, however, we would like a computable criterion on types that decides if two types are equivalent or not. There is one, which is called the normal form of a type and which is simply a type constructor. Given that all declarations in $\Delta$ are productive, each type possesses a uniquely defined head normal form which

$$\frac{\Delta(a) = b \qquad \text{nf}(b) = \tau}{\text{nf}(a) = \tau}\ \textsf{nf\_alias}$$

$$\frac{\Delta(a) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\text{nf}(a) = a}\ \textsf{nf\_record} \qquad \frac{\Delta(a) = \text{array } \tau}{\text{nf}(a) = a}\ \textsf{nf\_array}$$

$$\frac{\Delta(a) = \text{int}}{\text{nf}(a) = \text{int}}\ \textsf{nf\_int1} \qquad \frac{\Delta(a) = \text{string}}{\text{nf}(a) = \text{string}}\ \textsf{nf\_string1}$$

$$\frac{}{\text{nf}(\text{int}) = \text{int}}\ \textsf{nf\_int2} \qquad \frac{}{\text{nf}(\text{string}) = \text{string}}\ \textsf{nf\_string2}$$

Deciding if two types are equivalent is easy now.

**Theorem 2.1** *For all $\tau_1$ and $\tau_2$ types valid in $\Delta$, if $\mathcal{D}_1$ is a derivation of $\text{nf}(\tau_1) = \tau$ and $\mathcal{D}_2$ is a a derivation of $\text{nf}(\tau_2) = \tau$ for some $\tau$ then $\tau_1 \equiv \tau_2$.*

**Proof:** By simultaneous induction on the definition of $\mathcal{D}_1, \mathcal{D}_2$. We illustrate the proof by one case:

**Case:** $\mathcal{D}_1 = \dfrac{\begin{array}{cc}\mathcal{D}_1' & \mathcal{D}_1'' \\ \Delta(a) = b & \mathrm{nf}(b) = \tau\end{array}}{\mathrm{nf}(a) = \tau}\ \mathsf{nf\_alias}$:

| | |
|---|---|
| there exists $\mathcal{P}_1$ of $\Delta \vdash b \equiv \tau_2$ | by induction hypothesis on $\mathcal{D}_1''$ |
| there exists $\mathcal{P}_2$ of $\Delta \vdash a \equiv b$ | by $\mathsf{alias}$ on $\mathcal{D}_1''$ |
| there exists $\mathcal{P}$ of $\Delta \vdash a \equiv \tau_2$ | by $\mathsf{trans}$ on $\mathcal{P}_1$ and $\mathcal{P}_2$. |

□