Lecture 15: Type Checking (continued)

Carsten Schürmann

October 15, 2001

The system of typing rules presented in the previous lecture does only account for a few features of Tiger. Others, for example variable declarations, lvalues, assignments, mutual recursion, or type declarations are not accounted for yet. In these notes, we complete the set of typing rules. All features offered by Tiger must be reflected in the type rules. Therefore, we reconsider the design decisions from last lecture in turn and extend them appropriately.

1 Type declarations

Foremost, we address the issue of type declarations. A type declaration gives a programmer the possibility to introduce new types during runtime, so-called type constructors. Throughout this lecture we denote those type constructors by a.

$$\text{Types:} \quad \tau \quad ::= \quad \ldots \mid a$$

We have already discussed that Tiger offers two name spaces, one for functions and variables, and one for types. In order to make this requirement precise, we need to declare a second context which introduces type constructors, which we denote by $\Delta ::= \cdot \mid \Delta, a = \tau$. Naturally, Δ again can be implemented as symbol table as described in Lecture 13.

But note, that there is a conceptual difference between the standard context Γ and Δ which can be interpreted as a list of type abbreviations more than a list of type declarations. Next, we retrofit the two judgments from last lecture with Δ.

$$\begin{array}{rl} \text{a typing judgment for expressions:} & \Gamma; \Delta \vdash^{\iota} e : \tau \\ \text{and a typing judgment for declarations:} & \Gamma; \Delta \vdash^{\iota} d/e : \tau. \end{array}$$

Clearly, with changing a judgment, we have to revisit all inference rules. This however is pretty easy, because all we have to do is retrofit each judgment in the conclusion and the premisses with Δ. For the purpose of illustration, here are a few examples of updated rules:

$$\frac{\Gamma; \Delta \vdash^{\iota} e_1 : \text{int} \qquad \Gamma; \Delta \vdash^{\iota} e_2 : \text{unit}}{\Gamma; \Delta \vdash^{\iota} \text{if } e_1 \text{ then } e_2 : \text{unit}}\ \textsf{ifthen}$$

$$\frac{\Gamma;\Delta \vdash^{\iota} e_1 : \text{int} \quad \Gamma;\Delta \vdash^{1} e_2 : \text{unit}}{\Gamma;\Delta \vdash^{\iota} \text{while } e_1 \text{ do } e_2 : \text{unit}}\ \textsf{while}$$

$$\frac{\Gamma;\Delta \vdash^{\iota} e_1 : \text{int} \quad \Gamma;\Delta \vdash^{\iota} e_2 : \text{int} \quad \Gamma;\Delta \vdash^{1} e_3 : \text{unit}}{\Gamma;\Delta \vdash^{\iota} \text{for } x = e_1 \text{ to } e_2 \text{ do } e_3 : \text{unit}}\ \textsf{for}$$

How do we actually declare a new type constructor? Type declarations allow the introduction of new type constructors. Note the similarity to function declarations from last lecture.

$$\text{Declarations:} \quad d \quad ::= \quad \ldots \mid a = \tau, d$$

The typing rule for type declaration is as follows.

$$\frac{\Gamma;\Delta, a = \tau' \vdash^{\iota} d/e : \tau}{\Gamma;\Delta \vdash^{\iota} a = \tau', d/e : \tau}\ \textsf{type}\ \ (a \text{ does not occur free in } \tau)$$

Note, that the side condition is of utmost importance. Without this side condition, we could find a type for

```
let
  type a = int
  var x : a := 5
in
  x
end
```

which should be impossible, because the type constructor `a` escapes its scope. It is also noteworthy, that Tiger decides type equivalence by equivalence and *not* by structural equivalence. Two types that are declared identical in two different instances are not considered equal.

2 Variable declarations and assignments

Variables declarations declare names of a variables and their initial value. The content of a variable is mutable during execution, which makes Tiger an imperative programming language as opposed to a purely functional programming language where this is not the case. Side effects are possible in Tiger. But which assignments are legal, and which are not? The type checking rules must enforce that any assignment is conform with the type.

$$\text{Declarations:} \quad d \quad ::= \quad \ldots \mid x := e, d \mid x : \tau := e, d$$

$$\frac{\Gamma \vdash^\iota e' : \tau' \quad \Gamma, x : \tau' \vdash^\iota d/e : \tau}{\Gamma \vdash^\iota x := e', d/e : \tau}\ \mathsf{var1}\ \ (e' \neq \text{nil})$$

$$\frac{\Gamma \vdash^\iota e' : \tau' \quad \Gamma, x : \tau' \vdash^\iota d/e : \tau}{\Gamma \vdash^\iota x : \tau' := e', d/e : \tau}\ \mathsf{var2}$$

$$\frac{\Gamma(x) = \tau \quad \Gamma \vdash^\iota e : \tau}{\Gamma \vdash^\iota x := e : \text{unit}}\ \mathsf{assign}$$

nil is an element of every record type which will be discussed below. If a variable is declared and bound to nil, one must explicitly declare its type using var2. Recall that only the rightmost declaration of a variable $x : \tau$ is valid. They shadow earlier declarations.

3 Arrays

As the first example for how to introduce new types, we consider arrays. Arrays are common constructs in any kind of imperative but also functional programming language. Therefore it is also present in Tiger.

$$\begin{array}{lrcl} \text{Types:} & \tau & ::= & \ldots \mid \text{array } \tau \\ \text{Expressions:} & e & ::= & \ldots \mid a[e_1] \text{ of } e_2 \mid x[e] \end{array}$$

As invariant a is a type constructor that abbreviates array τ in Δ. The first expression shows how arrays can be created, and the second, how the program can refer to the content of a cell of an array.

$$\frac{\Delta(a) = \text{array } \tau \quad \Gamma; \Delta \vdash^\iota e_1 : \text{int} \quad \Gamma; \Delta \vdash^\iota e_2 : \tau}{\Gamma; \Delta \vdash^\iota a[e_1] \text{ of } e_2 : a}\ \mathsf{arrayI}$$

$$\frac{\Gamma(x) = \text{array } \tau \quad \Gamma; \Delta \vdash^\iota e : \text{int}}{\Gamma; \Delta \vdash^\iota x[e] : \tau}\ \mathsf{arrayE}$$

The next two rules are necessary to decide equality and inequality on arrays.

$$\frac{\Delta(a) = \text{array } \tau \quad \Gamma; \Delta \vdash^\iota e_1 : a \quad \Gamma; \Delta \vdash^\iota e_2 : a}{\Gamma; \Delta \vdash^\iota e_1 = e_2 : \text{int}}\ \mathsf{arrayeq}$$

$$\frac{\Delta(a) = \text{array } \tau \quad \Gamma; \Delta \vdash^\iota e_1 : a \quad \Gamma; \Delta \vdash^\iota e_2 : a}{\Gamma; \Delta \vdash^\iota e_1 <> e_2 : \text{int}}\ \mathsf{arrayneq}$$

4 Records

The second and last family of type constructors are records. They can be seen as datatype definitions, since fields have names. In this they differ from Standard ML where constructors are considered first-class objects. The second example discusses how record types are declared and used in Tiger.

$$\begin{array}{lrcl} \text{Types:} & \tau & ::= & \ldots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ \text{Expressions:} & e & ::= & \ldots \mid a\{l_1 = e_1 \ldots l_n = e_n\} \mid x.l \mid \text{nil} \end{array}$$

Records can be created inspected field by field during runtime. Here are the typing rules.

$$\frac{\Delta(a) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad \Gamma; \Delta \vdash^\iota e_1 : \tau_1 \quad \Gamma; \Delta \vdash^\iota e_n : \tau_n}{\Gamma; \Delta \vdash^\iota a\{l_1 = e_1, \ldots, l_n = e_n\} : a}\ \textsf{recordI}$$

$$\frac{\Gamma(x) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma; \Delta \vdash^\iota x.l_i : \tau_i}\ \textsf{recordE}\ \text{ for } 1 \leq i \leq n$$

$$\frac{\Delta(a) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma; \Delta \vdash^\iota \text{nil} : a}\ \textsf{nil}$$

Note that a typing derivation is valid only if it is unique. Occurrences of nil may introduce non-uniqueness, because they are well-typed with every record type declared in Δ, independent of the number of fields and their types. Therefore, for example, the expression nil = nil cannot be typed.

$$\frac{\Delta(a) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad \Gamma; \Delta \vdash^\iota e_1 : a \quad \Gamma; \Delta \vdash^\iota e_2 : a}{\Gamma; \Delta \vdash^\iota e_1 = e_2 : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}\ \textsf{recordeq}$$

$$\frac{\Delta(a) = \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad \Gamma; \Delta \vdash^\iota e_1 : a \quad \Gamma; \Delta \vdash^\iota e_2 : a}{\Gamma; \Delta \vdash^\iota e_1 <> e_2 : \text{int}}\ \textsf{recordneq}$$

5 Mutual recursion

We will not give all rules for mutual recursion here, but elude to one possibility how it could be established. For example, we could introduce new judgments for declarations as follows.

$$\text{A typing judgment for declarations:} \quad (\Gamma; \Delta)/d_1 \vdash^\iota d_2/e : \tau.$$

Here, d_1 presents all declarations that may be assumed recursively, d_2 presents the bodies of declarations that still need to be checked. The type checker will recurse through the list d_1. Only continuous blocks of function declarations or type declarations can be mutual recursive, and this must be reflected in the

rules. Possibly auxiliary judgments are necessary to accomplish this task. Variable declarations are checked as a type checking algorithm goes along, their declaration cannot be mutual recursive on one another. Cycles in type declarations must be recognized during type checking and reported as semantic error.

6 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 6.1 (Non-productive type definitions)

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

Example 6.2 (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 6.3 (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 Δ, we write

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

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

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

$$\frac{}{\Delta \vdash a \equiv a}\ \textsf{reflexive} \qquad \frac{\Delta \vdash a \equiv b}{\Delta \vdash b \equiv a}\ \textsf{symmetric} \qquad \frac{\Delta \vdash a \equiv b \quad \Delta \vdash b \equiv c}{\Delta \vdash a \equiv c}\ \textsf{transitive}$$

We can easily show that no record or array types can be equivalent.

Theorem 6.4 *We can show that if $\tau_1 \equiv \tau_2$ then either $\tau_1 = a$ or $\tau_2 = a$ for some type constructor a.*

Proof: By induction on the derivation of $\tau_1 \equiv \tau_2$. □

In summary, we must add a type conversion rule to Tiger, that accounts for equivalent types.

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

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

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

7 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 Δ are productive, each type possesses a uniquely defined head normal form which

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

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

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

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

Deciding if two types are equivalent is easy now.

Theorem 7.1 *For all τ_1 and τ_2 types valid in Δ, if $\mathcal{D}_1$ is a derivation of $\mathrm{nf}(\tau_1) = \tau$ and $\mathcal{D}_2$ is a a derivation of $\mathrm{nf}(\tau_2) = \tau$ for some τ 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 alias on $\mathcal{D}_1''$
there exists $\mathcal{P}$ of $\Delta \vdash a \equiv \tau_2$	by trans on $\mathcal{P}_1$ and $\mathcal{P}_2$.

□