Today's Lecture Notes for cs173

sk and dbtucker

October 18, 2000

Type checking datatype expressions

How do we type check datatype expressions? We need to prove that the body of the expression has some type t in an extended type environment, and we define this type t to be type of the expression:

$$\frac{\Gamma[\ldots] \vdash e : t}{\begin{array}{l}\Gamma \vdash \ (\texttt{datatype}\ \tau \\ \qquad [v_1\ \ (s_{11}\ \ \tau_{11})\ \ (s_{12}\ \ \tau_{12})\ \ldots] \\ \qquad [v_2\ \ (s_{21}\ \ \tau_{21})\ \ (s_{22}\ \ \tau_{22})\ \ldots] \\ \qquad e) : t\end{array}}$$

How should we extend Γ in order to type check the body? Remember, we added three kinds of functions in the body; here are examples of those functions and their types:

Contructors The function v_1 has type $\tau_{11} \times \tau_{12} \times \cdots \times \tau_{1m} \rightarrow \tau$.

Predicates The function $v_1?$ has type $\tau \rightarrow \texttt{bool}$.

Selectors The function s_{21} has type $\tau \rightarrow \tau_{21}$.

We need to extend Γ with these functions and their types in order to type check the body of the `datatype` expression:

$$\frac{\Gamma \left[\begin{array}{lll} v_1 : \tau_{11} \times \tau_{12} \times \cdots \times \tau_{1m} \rightarrow \tau & v_1? : \tau \rightarrow \texttt{bool} & s_{1i} : \tau \rightarrow \tau_{1i} \\ v_2 : \tau_{21} \times \tau_{22} \times \cdots \times \tau_{2n} \rightarrow \tau & v_2? : \tau \rightarrow \texttt{bool} & s_{2j} : \tau \rightarrow \tau_{2j} \end{array}\right] \vdash e : t}{\begin{array}{l}\Gamma \vdash \ (\texttt{datatype}\ \tau \\ \qquad [v_1\ \ (s_{11}\ \ \tau_{11})\ \ (s_{12}\ \ \tau_{12})\ \ldots] \\ \qquad [v_2\ \ (s_{21}\ \ \tau_{21})\ \ (s_{22}\ \ \tau_{22})\ \ldots] \\ \qquad e) : t\end{array}}$$

where i ranges over $1 \ldots m$ (the fields of v_1) and j ranges over $1 \ldots n$ (the fields of v_2).

For example, the following code creates a number list and computes its length:

```
(datatype nlist
  [mt]
  [cons (first num) (rest nlist)]
  (rec length : (nlist -> num)
       L
       (if (mt? (L))
           0
           (1 + (length ((rest (L))))))
    (length (cons 1 ((cons 2 ((cons 3 ((mt))))))))))
```

This expression evaluates to `3`, which has type `num`. We could also write an expression that just returns the list:

```
(datatype nlist
  [mt]
  [cons (first num) (rest nlist)]
  (cons 1 ((cons 2 ((cons 3 ((mt)))))))))
```

The result of this expression is `(cons 1 (cons 2 (cons 3 (mt))))`, which has type `nlist`. This is a problem—the type `nlist` has escaped the scope in which it is defined!

The value `(cons 1 ((cons 2 ((cons 3 ((mt)))))))` is useless outside of the datatype scope, since we don't have any way to look at values of type `nlist`. To prevent the type from escaping, we can add a restriction to the type judgement:

> **Restriction #1:** t cannot be τ.

Actually, this restriction isn't strong enough. Consider this expression:

```
(datatype nlist
  [mt]
  [cons (first num) (rest nlist)]
  (vector ((mt))))
```

The result is a vector of `nlist`s. We can perform operations on the vector, such as computing its length, but we still can't get any information from the `nlist`s. It would have been equally effective to simply return the length of the vector, if that's what we were interested in. So, we make a stronger restriction:

> **Restriction #1 (revised):** τ cannot be free in t.

Another problem arises from defining two datatypes with the same name:

```
(datatype A
  [v1 (f1 num)]
  (datatype A
    [v2 (f2 bool)]
    (f2 ((v1 (3))))))
```

Here, `v1 : num` $\rightarrow$ `A`, and `f2 : A` $\rightarrow$ `bool`, so according to the type system, `(f2 ((v1 (3))))` has type `bool`. However, `v1` really constructs values of the first `A` type, whereas `f2` consumes values of the second `A` type, so `(f2 ((v1 (3))))` *should* be a type error. We'll just say that you can't use the same name for two datatypes:

Restriction #2: τ cannot be in Γ.

There are two limitations of our datatype expression. First, we can't return values of the defined datatypes. Second, we can't create mutual type references. Most languages overcome these limitations by requiring all datatypes to be defined at the top level:

Program ::= (datatype definition)* L-expression

Variant predicates

Look at the `nlist` datatype again:

```
(datatype nlist
  [mt]
  [cons (first num) (rest nlist)]
  ...)
```

How do we evaluate predicates such as `(cons? L)`? We need to know at run time which variant `L` is. Therefore, we *tag* each value of type `nlist` with its variant type, `mt` or `cons`. In this case, we only need one bit to distinguish between the two variants (in general, you need $\log_2$(# of variants) bits). What if you have two datatypes, `A` and `B`?

```
(datatype A
  [v1 (f1 int)]
  [v2 (f2 int)]
  (datatype B
    [v3 (f3 int)]
    [v4 (f4 int)]
    ...))
```

Do you need one or two bits to tag each value? That is, do you need to distinguish variants only within their datatype, or across all datatypes?

The answer is that you only need to distinguish variants within a datatype, because variants of different datatypes will be distinguished by the type checker. For example, consider `(v1? ((v3 (6))))`. This expression will not type check because `v3` produces values of type `B`, but `v1` consumes values of type `A`.

Type safety

What do we mean when we say an interpreter is type safe? Here's the definition:

> *Type safety* is the property that no primitive operation is ever applied to values of the wrong (dynamic) type.

By primitive operations, we mean not only primitives such as addition, but operations such as function application.

Note that type safety is a property of the evaluator. In particular, a language can be type safe even if it has no static type checker (this is the case with Scheme).

We can make a table of languages based on whether they are type checked and whether they are type safe:

	checked	not checked
safe	ML, Java	Scheme
unsafe	C, C++	assembly

Without the type safety property, type soundness breaks. Therefore, the unsafe languages in the above chart do not have sound type systems. Those languages which are type checked but not type safe (e.g., C and C++) are truly insidious, since the type checker leads you to believe that your program will not perform any unsafe operations, when in fact you have no such guarantee.