Supplementary Notes on Dynamic Typing

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 20
November 5, 2002

So far, we have been working with type systems were all checking was static, and types were not needed at run-time. In this lecture we investigate the consequences of loosening this assumption. A language in which types are used at run-time is called *dynamically typed*. Examples of dynamically typed languages are Lisp, Scheme, and Java. There is an excellent treatment of dynamic types in [Ch. 24]. We emphasize a few complementary points here.

We begin with the extreme point of no static type-checking at all. We simply take a program written in MinML and run it without type-checking it first. Clearly, this would violate progress since a term such as $\mathtt{apply}(1, 1)$ is neither a value, nor can it make a step. In order to obtain a sensible language, we need to extend our computational rules to check for such stuck states and raise a run-time type error. We show here only the rules for function application. First, the usual rules, written (implicitly) under the assumption of type-correctness.

$$\frac{e_1 \mapsto e_1'}{\mathtt{apply}(e_1, e_2) \mapsto \mathtt{apply}(e_1', e_2)} \qquad \frac{v_1 \ \mathsf{value} \quad e_2 \mapsto e_2'}{\mathtt{apply}(v_1, e_2) \mapsto \mathtt{apply}(v_1, e_2')}$$

$$\frac{(v_1 = \mathtt{fun}(f.x.e_1)) \quad v_2 \ \mathsf{value}}{\mathtt{apply}(v_1, v_2) \mapsto \{v_2/x\}\{v_1/f\}e_1}$$

Next, the rules to handle the case of a run-time type error. The first one signals the actual error, the others propagate the error outward.

$$\frac{v_1 \text{ value} \quad (v_1 \neq \texttt{fun}(f.x.e_1)) \quad v_2 \text{ value}}{\texttt{apply}(v_1, v_2) \mapsto \texttt{error}}$$

$$\frac{}{\texttt{apply}(\texttt{error}, e_2) \mapsto \texttt{error}} \qquad \frac{v_1 \text{ value}}{\texttt{apply}(v_1, \texttt{error}) \mapsto \texttt{error}}$$

The rest of the operational semantics should be extended in a similar way. In particular, there need to be additional rules for primitive operations and other elimination forms (conditionals, projections, case expressions, etc.) in the case of a dynamic type error.

In the resulting language, we can write and execute expressions such as

```
if true then 1 else λx. x  ↦  1
[1, true, λx. x]  value
(hd (tl (tl [1, true, λx. x]))) 3 ↦* 3
```

where we used ML-style notations for lists.

With these additions, we can recover preservation and progress. We assume here that we still want to statically compile the program, so free variables are still not permitted (e must be closed, that is, $FV(e) = \{\}$). Preservation is somewhat trivialized.

Theorem 1 (Preservation)
If e is closed and $e \mapsto e'$ then e' is closed.

It is also posssible to talk about expressions e that happen to be well-typed. In that case, evaluation preserves types, since we have only added rules to the operational semantics in a conservative way.

Theorem 2 (Progress)
If e is closed then either

(i) e value*, or*

(ii) $e = \texttt{error}$*, or*

(iii) $e \mapsto e'$ *for some* e'*.*

The main conceptual cost of dynamic typing is the inability to discover errors early: code that does not happen to be executed can have lurking bugs that would be obvious to a type-checker. However, there is also an

implementation cost. We cannot simply compile a program with the same strategy as may be possible for a statically typed language, because we must be able to perform the run-time type checks. Fortunately, this is not as bad as it sounds, since we do not need to have the precise type of a function, we only need to know where it is in fact a function or not.

In order to make tags explicit, we can change the syntax of our language; see [Ch. 24] for the details. Here we show that we can make such tags explicit in a statically typed language. The small price we pay for this is that now the user program or library will have to do some tag-checking. But this means we can tag-check precisely where necessary, instead of everywhere in the program.

In ML, we would write such a tagged datatype as

```
datatype tagged =
    Int of int
  | Bool of bool
  | Fun of tagged -> tagged -> tagged
```

Note that we do not explicitly represent bound variables, as they are mapped to bound variables in ML. This is an application of the idea of higher-order abstract syntax.

As a reminder, here is how we would write this type using plain recursive types instead.

$$\mathsf{tagged} = \mu t.\mathsf{int} + \mathsf{bool} + (t \to t \to t)$$

Instead of named tags, as in the datatype declaration above, we use compositions of inl and inr as tags.

Now the elimination forms become functions on tagged representations. We show one primitive operator, conditional, and application.

```
exception TypeError
fun checkedMult (Int(n), Int(m)) = Int(n*m)
  | checkedMult _ = raise TypeError
fun checkedIf (Bool(true), e1, e2) = e1 ()
  | checkedIf (Bool(false), e1, e2) = e2 ()
  | checkedIf _ = raise TypeError
fun checkedApply (v1 as Fun(g), v2) = g v1 v2
  | checkedApply _ = raise TypeError
```

Heterogeneous lists now become lists of tagged data. This is in fact exactly the same as the representation in a purely dynamically typed language, except that the tagging is visible to the programmer. For example, the following are all well-typed and execute as expected:

```
val hetList : tagged list =
  [Int 1, Bool true, Fun (fn _ => fn x => x)];
val f : tagged = hd(tl(tl(hetList)));
val x : tagged = checkedApply (f, Int(3));
```

The non-terminating self-application example can also be written quite easily using checked application. Note that even though functions can in principal be recursive in our encoding, we do not use this feature here to give a correct implementation of the dynamically typed $(\lambda x.x\,x)\,(\lambda x.x\,x)$.

```
val omega : tagged =
  Fun (fn _ => fn x => checkedApply (x, x));
checkedApply (omega, omega);
```

As expected, this last expression diverges.

It is also interesting to consider if we can perhaps use subtyping to allow us to write heterogenous lists. For example, if we had a universal type $\top$ that includes all values, one might expect

```
[1, true, λx. x] : ⊤ list
```

In order to see if this is indeed the case, we consider the laws that $\top$ should satisfy. First, every value should have type $\top$. Second every type should be a subtype of $\top$.

$$\frac{v \text{ value}}{\Gamma \vdash v : \top} \qquad \frac{}{\tau \leq \top}$$

There are no other rules regarding $\top$. Clearly, it is necessary to require v to be a value in the first rule in order to save the progress theorem.

Now, indeed, we have

```
[1, true, λx. x] : ⊤ list
```

However, we find we cannot *use* such a list in a non-trivial way. For example

```
val hetList : ⊤ list = [1, true, λx. x];
val f : ⊤ = hd(tl(tl(hetList)));
```

Now the application `f 3` would not be well-typed, because `f` is not known to be a function, only a value of type $\top$.

In order to use `f`, we must introduce a *downcast* operator into the language, that allows us to check explicitly *at run-time* if a given value has a specified type, and raise an error otherwise. We could then write

```
((int -> int)f) 3
```

and it would type-check. Of course, at this point we realize we haven't made any progress, because the function still must be tagged with its type in order to verify the correctness of the downcast at run-time. In fact, we would need to keep more information to verify the precise form of the function's type, rather than just the information that it is a function.

Note that his form of downcast is very different from an unsafe version of cast where $(\tau)e$ will be treated as if it has type τ, regardless of the actual type of the value of v. In a language like C this cannot be repaired, because data are not tagged and no run-time checking of tags is possible.

What would be the coercion interpretation of the $\top$ type? Recall that intersection types are interpreted as pairs. If we think of $\top$ as a 0-ary intersection, it should logically be interpretated as a 0-ary product, namely the unit type.

$$\overline{\lambda x.\langle\,\rangle : \tau \leq \top}$$

Intuitively, this is also meaningful: the coercion from any type to $\top$ is unique (the constant function) and coherence is preserved. Knowing that $v : \top$ carries no information.

This points out an important property of the coercion interpretation of subtyping: since we run the program *after* all coercions have been applied, any term that was assigned type $\top$ via a subtyping coercion may not be executed at all! This is nonetheless consistent since essentially only values have type $\top$ directly. But this means that we cannot implement downcasting in a coercion interpretation of subtyping. Again, intuitively this make sense: since a coercion from τ to σ where $\tau \leq \sigma$ loses information,

we cannot in general recover an element of the original type τ if we try to downcast a value of type σ.

On the other hand, under the subset interpretation of subtyping, a coercion is useless, since all coercions will be the identity: if a value of type τ *is* a value of type σ then there is no need to apply a coercion. In that case downcasting as a run-time operation that may fail makes sense: in a downcast $(\tau)e$ for $e : \sigma$ we evaluate e, and then verify if it has type τ. The latter operation will, of course, require tags or some other method to check that a given value has a specified type at run-time.

These observations help to explain why objected-oriented languages such as Java, which rely heavily on downcasting, have a subset interpretation of subtypes that arise from subclassing. In such languages objects are tagged with their class, which makes it quite efficient to implement downcasts or the related `instanceOf` operation. Of course, it is critical that viewing an object of a class as an element of the superclass does *not* apply a coercion, dropping extra fields, because later downcasts could not undo this damage.

Also, in Java there is a class `Object` which is a superclass of any other class. This almost corresponds to $\top$, except that Java also has primitive types such as `int` or `float`. Given that Java performs coercions on `int` and `float` we could summarize the situation as coercive subtyping on primitive types and subset subtyping on objects.