510: Programming Languages

Static and Dynamic Typing

David Walker

Fall, 2002

Static Typing

MinML is a **statically typed** language.

- Type system is based on an analysis of the **program**.
- Type errors are caught at **compile time**.

The canonical forms lemma relates the static type to the dynamic value.

Static Typing

As we have seen, we still must deal with **run-time errors**.

- Division by zero.
- Exceeding array bounds (if we had arrays).

So why bother with static type errors? Why not make them all dynamic errors?

Advantages of Static Typing

Mistakes are caught as **early** as possible.

- At **build time**, not **application time**.

Types serve as **statically checkable invariants**.

- Types are a formal “comment” stating an expectation.

- Compiler can check these requirements.

Advantages of Dynamic Typing

Mistakes are caught as **late** as possible.

- At **execution time**, not at **build time**.
- Don't have to hassle with the type checker.

Dynamic typing affords greater flexibility.

- **Flexibility**: `if true then 17 else false` "has type" `int`.
- **Heterogeneity**: `[true,17,"17"]` is a valid list.

Dynamic vs. Static Typing

Which approach is better?

- Static better than dynamic? (ML)

- Dynamic better than static? (Lisp, Scheme)

First, let's look at what is meant by **dynamic typing**.

Then we'll give a (possibly surprising) answer!

Dynamic Typing

Two main ideas:

- Do away with the static semantics entirely. If a program parses, it is acceptable for execution.

- Employ run-time checks to ensure safety. Type errors occur at run-time, just like any other errors.

How can we formalize this?

Run-Time Type Errors

Assume we've already augmented the semantics with `error` transitions.

- Errors are propagated appropriately.

- All checked errors yield `error`.

Run-Time Type Errors

Add new rules that make type errors into checked errors.

For example,

$$\frac{v \ \mathsf{value} \qquad v_1 \ \mathsf{value} \qquad (v \neq \mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\, \mathtt{is}\, e\, \mathtt{end})}{\mathtt{apply}(v, v_1) \mapsto \mathtt{error}}$$

The "side condition" checks for the case that v is not a function.

Run-Time Type Errors

This seems simple enough, **but** ...

- The side conditions on the rules should be formalized, since they correspond to run-time checks.

- It is not immediately clear how to check the type of a value at execution time (*e.g.*, booleans and integers are just words).

Tagging

These rules assume that you can determine the **form** of a value at run-time!

- In general, this is unrealistic: values are just bits! (*e.g.*, booleans and integers are both words).

- But we can **tag** values with their shape so that run-time checks are possible.
  - Takes **space** for the tag.
  - Takes **time** to apply the tag and to check it.

Tagging

Grammar for dynamic MinML:

$$
\begin{array}{llcl}
\textit{Expressions} & e & ::= & x \mid v \mid o(e_1, \ldots, e_n) \mid \\
 & & & \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \mid \\
 & & & \texttt{apply}(e_1, e_2) \\
 & & & \\
\textit{TaggedValues} & v & ::= & \texttt{Int}(n) \mid \\
 & & & \texttt{Bool(true)} \mid \texttt{Bool(false)} \mid \\
 & & & \texttt{Fun(fun}\, x\, \texttt{(}y\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end)} \\
 & & & \\
\textit{UntaggedValues} & u & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \\
 & & & \texttt{fun}\, x\, \texttt{(}y\texttt{:}\tau_1\texttt{):}\tau_2\, \texttt{is}\, e\, \texttt{end}
\end{array}
$$

NB: Untagged values are for “internal” use only!

Tagging

The tag of a value is **not** its type!

- Just says "this is a function", not "this is of type `int`$\rightarrow$`int`".

- Nevertheless, the tag is sometimes called (incorrectly) the **run-time type** of the value.

It is important to distinguish the **tag** from the **type**!

Tag Checking

Introduce rules for $v\,\mathsf{is\_int}\,u$ and $v\,\mathsf{isnt\_int}$:

$$\frac{}{\texttt{Int(}n\texttt{)}\ \mathsf{is\_int}\ n}$$

$$\frac{}{\texttt{Bool(\_)}\ \mathsf{isnt\_int}}$$

$$\frac{}{\texttt{Fun(\_)}\ \mathsf{isnt\_int}}$$

Tag Checking

Rules for addition:

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \qquad v_1 \,\mathsf{is\_int}\, n_1 \quad v_2 \,\mathsf{is\_int}\, n_2 \quad (n = n_1 + n_2)}{+(v_1, v_2) \mapsto \mathtt{Int}(n)}$$

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \quad v_1 \,\mathsf{isnt\_int}}{+(v_1, v_2) \mapsto \mathtt{error}}$$

$$\frac{v_1 \ \mathsf{value} \quad v_2 \ \mathsf{value} \quad v_1 \,\mathsf{is\_int}\, n_1 \quad v_2 \,\mathsf{isnt\_int}}{+(v_1, v_2) \mapsto \mathtt{error}}$$

Tag Checking

Introduce judgements $v \, \mathsf{is\_fun} \, u$ and $v \, \mathsf{isnt\_fun}$.

$$\frac{(g = \mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\ \mathtt{is}\ e\ \mathtt{end})}{\mathtt{Fun(}g\mathtt{)}\ \mathsf{is\_fun}\ g}$$

$$\frac{}{\mathtt{Bool(\_)}\ \mathsf{isnt\_fun}}$$

$$\frac{}{\mathtt{Int(\_)}\ \mathsf{isnt\_fun}}$$

Tag Checking

Rules for application:

$$\frac{v_1 \ \text{value} \qquad v \ \text{is\_fun} \ \texttt{fun} \ f \ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2 \ \texttt{is} \ e \ \texttt{end}}{\texttt{apply(}v, v_1\texttt{)} \mapsto \{v, v_1/f, x\}e}$$

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

Safety of Dynamic Typing

Dynamic MinML is obviously type safe!

- There is only one "type", which must be preserved by evaluation. (Every well-formed expression is well-typed.)

- Tag checking ensures that we may always make progress, perhaps to `error`.

Dynamically typed languages are "trivially" type safe!

The Cost of Dynamic Typing

Tagging data values uses time and space!

- Space for the tag itself.
- Time to apply tags, check tags, and recover untagged values.

The overheads are significant!

- Arithmetic operations in a loop repeatedly tag and untag values.
- Difficult to hoist checks outside of the loop.

Pay As You Go

The cost applies **even if** the program is statically well-typed!

- Static checker **proves** that some tag checks are not necessary.
- No means to express “raw” operations on untagged data.

Violates the **pay-as-you-go** principle of language design!

- Pay only for features you actually use.
- Dynamic checking imposes a global overhead.

Static vs. Dynamic, Revisited

So which is better, static or dynamic?

- Do we need the flexibility of dynamic typing?

- Is the overhead significant in practice?

- How soon do we wish to report errors?

- Should ill-typed programs be executable?

Static Subsumes Dynamic

Lots of energy has been wasted on this debate!

Dynamic typing is a **mode of use** of static typing!

The ideas are not **opposed**, but rather are completely **compatible**!

Tagged values are a **type**!

Dynamic Typing as Static Typing

Adding dynamic typing to MinML:

- Add a new type `tagged` of tagged values.
  - Operations to apply a tag to a value.
  - Operations to check tags.
- If you want dynamic typing, use the (static) type `tagged`.

Statically Typed Tagging

Typing rules for tagging values:

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{Int}(e) : \texttt{tagged}}$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{Bool}(e) : \texttt{tagged}}$$

$$\frac{\Gamma \vdash e : \texttt{tagged}{\rightarrow}\texttt{tagged}}{\Gamma \vdash \texttt{Fun}(e) : \texttt{tagged}}$$

Note that these rules **preclude** misapplication of tags!

Statically Typed Tagging

Primitive operations for checking tags.

$$\frac{\Gamma \vdash e : \texttt{tagged}}{\Gamma \vdash \texttt{as\_bool}(e) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : \texttt{tagged}}{\Gamma \vdash \texttt{as\_int}(e) : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \texttt{tagged}}{\Gamma \vdash \texttt{as\_fun}(e) : \texttt{tagged}{\rightarrow}\texttt{tagged}}$$

These are essentially **downcasts** that can fail at run-time.

Statically Typed Tagging

Dynamic semantics of downcasting:

$$\frac{}{\texttt{as\_int}(\texttt{Int(}n\texttt{)}) \mapsto n}$$

$$\frac{}{\texttt{as\_int}(\texttt{Bool(}v\texttt{)}) \mapsto \texttt{error}}$$

$$\frac{}{\texttt{as\_int}(\texttt{Fun(}v\texttt{)}) \mapsto \texttt{error}}$$

(Similarly for the other downcasts.)

Heterogeneity, Revisited

Heterogeneous lists:

```
[Int 1, Bool true, Fun (fn x:tagged => x)]
    : tagged list
```

Heterogeneous conditionals:

```
if true then (Int 1) else (Bool true)
    : tagged
```

Notice that dynamic checks are **required** whenever you use an element of a heterogeneous list or the result of a heterogeneous conditional!

Dynamic vs. Static Typing

Here's dynamic typing within SML.

```
datatype tagged =
  Int of int |
  Bool of bool |
  Fun of tagged -> tagged
exception TypeError
fun checked_add (Int m, Int n) = Int (m+n)
  | checked_add _ = raise TypeError
fun checked_apply (Fun f, v) = f v
  | checked_apply (_, _) = raise TypeError
```

Dynamic vs. Static Typing

A fully dynamic looping function:

```
fun dyn_fact (n : tagged) =
  let fun loop (n, a) =
      case n
        of Int m =>
           (case m
              of 0 => a
               | m =>
                  loop (checked_sub (n, 1),
                        checked_mult (n, a)))
         | _ => raise RuntimeTypeError
  in loop (n, Int 1)
  end
```

Dynamic vs. Static Typing

In a static framework you can **hoist** checks out of loops.

```
fun checked_fact (n : tagged) =
    let fun loop (0, a) = a
          | loop (n, a) = loop (n-1, n*a)
     in case n
          of Int m => Int (loop (m, 1))
           | _ => raise TypeError
    end
```

Summary

Statically typed languages **subsume** dynamically typed languages.

- Dynamic typing can be a **good** thing.

- But it is not the **only** thing: **pay-as-you-go**.