Static Typing

MinML is a \textit{statically typed} language.

- Type system is based on an analysis of the \textit{program}.
- Type errors are caught at \textit{compile time}.

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

Advantages of Static Typing

Mistakes are caught as \textit{early} as possible.

- At \textit{build time}, not \textit{application time}.
- Types serve as \textit{statically checkable invariants}.
- Types are a formal "comment" stating an expectation.
- Compiler can check these requirements.

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 \textit{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

Add new rules that make type errors into checked errors.

For example,

\[
\begin{align*}
\text{val} & \quad \text{val} \\
\text{v} & \neq \text{fun} \ (x: \tau_b); \tau_c \text{ end} \\
\text{apply}(v, v) & \mapsto \text{error}
\end{align*}
\]

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.

Grammar for dynamic MinML:

\[
\begin{align*}
\text{Expressions} & \quad e \ ::= \ x \mid v \mid \text{of} (r_1, \ldots, r_n) \mid \text{if} \ v \ \text{then} \ r_1 \ \text{else} \ r_2 \ \text{fi} \\
\text{Tagged Values} & \quad v \ ::= \ \text{Int}(n) \mid \text{Bool}(true) \mid \text{Bool}(false) \\
\text{Untagged Values} & \quad u \ ::= \ \text{true} \mid \text{false} \mid n \mid \text{fun} \ (y: \tau_1); \tau_2 \text{ is end}
\end{align*}
\]

**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→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

**Rules for addition**

\[
\begin{align*}
& \text{Int}(n) \text{ is int } n \\
& \text{Bool}(\text{)} \text{ is int } \text{error} \\
& \text{Fun}(\text{)} \text{ is int } \text{error}
\end{align*}
\]

**Rules for application**

\[
\begin{align*}
& \text{Fun}(\text{fun } f (x: \tau_b): \tau_c \text{ is } e \text{ end}) \text{ is fun } f (x: \tau_b): \tau_c \text{ is } e \text{ end} \\
& \text{Bool}(\text{)} \text{ is int } \text{error} \\
& \text{Fun}(\text{)} \text{ is int } \text{error}
\end{align*}
\]

### 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:
\[
\begin{align*}
\Gamma & \vdash e : \text{int} \\
\Gamma & \vdash \text{Int}(e) : \text{tagged} \\
\Gamma & \vdash e : \text{bool} \\
\Gamma & \vdash \text{Bool}(e) : \text{tagged} \\
\Gamma & \vdash e : \text{tagged- tagged} \\
\Gamma & \vdash \text{Fun}(e) : \text{tagged}
\end{align*}
\]
Note that these rules preclude misapplication of tags!
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

Statically Typed Tagging

Primitive operations for checking tags.

\[
\Gamma \vdash x : \text{tagged} \\
\Gamma \vdash \text{as\_int}(x) : \text{int} \\
\Gamma \vdash \text{as\_bool}(x) : \text{bool} \\
\Gamma \vdash \text{as\_fun}(x) : \text{tagged} \rightarrow \text{tagged}
\]

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

Heterogeneity, Revisited

Heterogeneous lists:

\[\text{[Int 1, Bool true, Fun (fn x:tagged => x)]: tagged list}\]

Heterogeneous conditionals:

\[\text{if true then (Int 1) else (Bool true): tagged}\]

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

Dynamic vs. StaticTyping

A fully dynamic looping function:

fun dyn_fact (n : tagged) =
  let fun loop (n, a) =
    case n
      of Int m => loop (m, Int 1)
      | _ => raise RunTimeTypeError
  in loop (n, [Int 1])
end

Statically Typed Tagging

Dynamic semantics of downcasting:

\[
\text{as\_int}(\text{Int}(n)) \rightarrow n \\
\text{as\_int}(\text{Bool}(n)) \rightarrow \text{error} \\
\text{as\_int}(\text{Fun}(n)) \rightarrow \text{error}
\]

(Similarly for the other downcasts.)

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 (p, a) = loop (p-1, p*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.