Static Typing

Thus far we have only considered **statically typed** languages.

- 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.

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.

Dynamic vs. Static Typing

Which approach is better?

- Static better than dynamic? (ML, Scala, Java)
- 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

Augment the semantics with an \( \varepsilon \) err judgement.

Add new rules that make type errors into checked errors.

For example,

\[
\frac{
\varepsilon \ \text{val} \quad \varepsilon \ \text{val} \quad (e \neq \lambda(x.d))
}{\text{ap}(\varepsilon, \varepsilon) \ \text{err}}
\]

The “side condition” checks for the case that \( e \) is not a function.

Class Labels

These rules assume that you can determine the \textbf{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 label values with their shape (or \textbf{class}) so that run-time checks are possible.
  – Takes \textbf{space} for the label.
  – Takes \textbf{time} to apply the label and to check it.

Dynamically Typed PCF

We will consider \( \mathcal{L}(\text{dys}) \), a dynamically type version of \( \mathcal{L}(\text{nat} \rightarrow) \) (PCF). Here is a grammar for \( \mathcal{L}(\text{dys}) \):

\[
\begin{array}{lll}
\text{Category} & \text{Item} & \text{Abstract} \\
\text{Expr} & d & : \ = \ x, \ \text{num}(m), \ \text{zero}, \ \text{succ}(d), \ \text{if}z(d, d_0, d_1), \ \text{fun}(\lambda(x.d)), \ \text{dap}(d_1, d_2), \ \text{fix}(x.d) \\
\text{Concrete} & & : \ = \ x, \ \text{m}, \ \text{zero}, \ \text{succ}(d), \ \text{if}z(d, d_0, d_1), \ \text{fun}(\lambda(x.d)), \ \text{dap}(d_1, d_2), \ \text{fix}(x.d) \\
\end{array}
\]

\textbf{NB}: Values are labelled with their \textbf{class} – “num” or “fun” – there are no unlabelled values!

\textbf{NB}: Successor is now an \textbf{elimination} form acting on values of class “num” rather than an introduction form for numbers.

\textbf{NB}: There are no class labels in the concrete syntax! Parser must insert them when passing to abstract syntax, since they are necessary to the dynamic semantics of \( \mathcal{L}(\text{dys}) \).
Class Labels

\( \mathcal{L}(\text{dyn}) \) is not just \( \mathcal{L}(\text{aat} \rightarrow \text{nat}) \) without types! Concrete syntax gives that appearance, but abstract syntax reveals the difference: class labels are needed in \( \mathcal{L}(\text{dyn}) \) expressions but not in \( \mathcal{L}(\text{aat} \rightarrow \text{nat}) \).

The class of a value is not its type!

- Just says “this is a function”, not “this is of type \( \text{aat} \rightarrow \text{nat} \)”. 
- Nevertheless, the class is sometimes called (incorrectly) the run-time type of the value.

It is important to distinguish the class from the type!

Dynamic Semantics for \( \mathcal{L}(\text{dyn}) \)

Introduce rules for values – fully evaluated (closed) expressions:

\[
\begin{align*}
\text{num}(\text{nat} \rightarrow \text{num}) & \rightarrow \text{val} \\
\text{fun}(\lambda \text{x.d}) & \rightarrow \text{val}
\end{align*}
\]

Introduce rules for checking the class of a numeric value, and its negation:

\[
\begin{align*}
\text{num}(\text{nat}) & \rightarrow \text{num} \\
\text{fun}(\lambda \text{x.d}) & \rightarrow \text{num} \\
\text{num}(\text{nat}) & \rightarrow \text{num}
\end{align*}
\]

Second argument of the first judgement is not an expression of \( \mathcal{L}(\text{dyn}) \) but just a special piece of syntax used internally in later dynamic semantics rules.

Dynamic Semantics for \( \mathcal{L}(\text{dyn}) \)

Dynamic semantics rules for zero and successor:

\[
\begin{align*}
\text{zero} & \rightarrow \text{num}(\text{nat}) \\
\text{succ}(d) & \rightarrow \text{num}(\text{nat})
\end{align*}
\]

Simultaneous inductive definition of \( d \rightarrow d' \) and \( d \rightarrow \text{err} \)

NB: Third rule labelled value \( \text{num}(\text{nat}) \) is bound to \( x \) to preserve invariant that variables are bound to \( \mathcal{L}(\text{dyn}) \) expressions.
Dynamic Semantics for $L^{\text{dyn}}$

Rules for application and general recursion:

\[ d_1 \mapsto d'_1 \]

\[ \text{dap}(d_1; d_2) \mapsto \text{dap}(d'_1; d_2) \]

\[ d_1 \text{ is fun } \lambda x.d \]

\[ \text{dap}(d_1; d_2) \mapsto [d_2/x]d \]

\[ d_1 \text{ isn’t fun } \]

\[ \text{dap}(d_1; d_2) \mapsto \text{err} \]

\[ \text{fix}(x.d) \mapsto [\text{fix}(x.d)/x]d \]

Safety of Dynamic Typing

Dynamically typed languages are “trivially” type safe!

- There is only one “type”, which must be preserved by evaluation. (Every well-formed expression is well-typed.)
- Class checking ensures that we may always make progress, although there are more opportunities for run-time errors.

**Theorem 1**

If $d \text{ ok}$, then either $d \text{ val}$, or $d \text{ err}$, or there exists $d'$ such that $d \mapsto d'$.

The Cost of Dynamic Typing

Labelling data values uses time and space:

- Space for the label itself.
- Time to apply labels, check labels, and recover unlabelled values.

The overheads are significant:

- Operations in a loop repeatedly label and unlabel 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 class checks are not necessary.
- No means to express “raw” operations on unlabelled 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!

Labelled values are a type!
Dynamic Typing as Static Typing

Adding dynamic typing to \( L[\text{nat} \rightarrow] \) (PCF):

- Add a new type \( \text{dyn} \) of labelled values.
  - Operations to apply a label to a value.
  - Operations to check class.
- If you want dynamic typing, use the (static) type \( \text{dyn} \).

Hybrid Typing: Syntax

We consider the language \( L[\text{nat} \rightarrow \text{dyn}] \) which extends the syntax of \( L[\text{nat} \rightarrow] \) (PCF) with:

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>( \tau )</td>
<td>( \text{dyn} )</td>
<td>( \text{dyn} )</td>
</tr>
<tr>
<td>Expr</td>
<td>( e )</td>
<td>( \text{new}[(c)] : \text{dyn} ) ( \text{line} ) ( \text{cast}[(c)] : \text{dyn} )</td>
<td>( \text{new}[(c)] : \text{dyn} ) ( \text{cast}[(c)] : \text{dyn} )</td>
</tr>
<tr>
<td>Class</td>
<td>( l )</td>
<td>( \text{num} )</td>
<td>( \text{fun} )</td>
</tr>
</tbody>
</table>

Type \( \text{dyn} \) represents the type of labelled values. Cast operation takes a class (indicated by label), not a type (which is always \( \text{dyn} \)).

Hybrid Typing: Static Semantics

Static semantics for \( L[\text{nat} \rightarrow \text{dyn}] \) extend the typing rules for \( L[\text{nat} 

\begin{align*}
\Gamma & \vdash e : \text{sat} \\
\Gamma & \vdash \text{new}[(c)] : \text{dyn} \\
\Gamma & \vdash e : \text{parr}(\text{dyn}, \text{dyn}) \\
\Gamma & \vdash \text{new}[(c)] : \text{dyn}
\end{align*}

Note that these rules preclude misapplication of labels!

Hybrid Typing: Dynamic Semantics

Dynamic semantics of \( L[\text{sat} \rightarrow \text{dyn}] \) add rules for downcasting:

\begin{align*}
e & \rightarrow e' \\
\text{new}[(c)] & \rightarrow \text{new}[(c)] \\
\text{cast}[(c)] & \rightarrow \text{cast}[(c)] \\
\text{new}[(c)] & \rightarrow \text{new}[(c)] \\
\text{cast}[(c)[\text{new}[(c)]]] & \rightarrow e \\
\text{new}[(c) \land \not \exists \not \exists] & \rightarrow \text{new}[(c)] \\
\text{cast}[(c)[\text{new}[(c)]]] & \rightarrow \text{err}
\end{align*}

Safety for Hybrid PCF

Lemma 2 (Canonical Forms)
If \( e : \text{dyn} \) and \( e \) val then \( e = \text{new}[(c')] \) for some \( c' \) val. If \( l = \text{num} \) then \( e' : \text{sat} \) and if \( l = \text{fun} \) then \( e' : \text{parr}(\text{dyn}, \text{dyn}) \).

Theorem 3 (Safety)
The language \( L[\text{sat} \rightarrow \text{dyn}] \) is safe:

1. If \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).
2. If \( e : \tau \), then either \( e \) val, or \( e \) err, or there exists \( e' \) such that \( e \rightarrow e' \).
Dynamic vs. Static Typing

Here's dynamic typing within SML.

```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
```

Heterogeneity, Revisited

Heterogeneous lists:

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

Heterogeneous conditionals:

```sml```
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!

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.