

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.

1

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?

2

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.

3

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" `nat`.
- **Heterogeneity**: `[true, 17, "17"]` is a valid list.

4

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!

5

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?

6

Run-Time Type Errors

Augment the semantics with an $e \text{ err}$ judgement.

Add new rules that make type errors into checked errors.

For example,

$$\frac{e \text{ val} \quad e_1 \text{ val} \quad (e \neq \lambda(x.d))}{\text{ap}(e; e_1) \text{ err}}$$

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

7

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).

8

Class Labels

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 **label** values with their shape (or **class**) so that run-time checks are possible.
 - Takes **space** for the label.
 - Takes **time** to apply the label and to check it.

9

Dynamically Typed PCF

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

Category	Item	Abstract	Concrete
Expr	d	x	x
		$\text{num}(\bar{n})$	\bar{n}
		zero	zero
		$\text{succ}(d)$	$\text{succ}(d)$
		$\text{ifz}(d; d_0; x.d_1)$	$\text{ifz } d\{\text{zero} \Rightarrow d_0 \mid \text{succ}(x) \Rightarrow d_1\}$
		$\text{fun}(\lambda(x.d))$	$\lambda x.d$
		$\text{dap}(d_1; d_2)$	$d_1(d_2)$
	$\text{fix}(x.d)$	$\text{fix } x \text{ is } d$	

10

Dynamically Typed PCF

NB: Values are labelled with their **class** – "num" or "fun"; there are no unlabelled values!

NB: Successor is now an **elimination** form acting on values of class "num" rather than an introduction form for numbers.

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 $\mathcal{L}\{\text{dyn}\}$.

11

Class Labels

$\mathcal{L}\{\text{dyn}\}$ is **not** just $\mathcal{L}\{\text{nat} \rightarrow\}$ 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 $\mathcal{L}\{\text{nat} \rightarrow\}$.

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

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

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

12

Static Semantics for $\mathcal{L}\{\text{dyn}\}$

Static semantics is trivial since there is only one type.

$$x_1 \text{ ok}, \dots, x_n \text{ ok} \vdash d \text{ ok}$$

says expression d is well-formed with free variables among those listed in the hypothesis.

13

Dynamic Semantics for $\mathcal{L}\{\text{dyn}\}$

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

$$\frac{\text{num}(\bar{n}) \text{ val}}{\text{fun}(\lambda(x.d)) \text{ val}}$$

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

$$\frac{\text{num}(\bar{n}) \text{ is_num } \bar{n}}{\text{fun}(_) \text{ isnt_num}}$$

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.

14

Dynamic Semantics for $\mathcal{L}\{\text{dyn}\}$

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

$$\frac{\text{fun}(\lambda(x.d)) \text{ is_fun } \lambda(x.d)}{\text{num}(_) \text{ isnt_fun}}$$

Again, 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.

15

Dynamic Semantics for $\mathcal{L}\{\text{dyn}\}$

Dynamic semantics rules for zero and successor:

$$\frac{}{\text{zero} \mapsto \text{num}(z)}$$

$$\frac{d \mapsto d'}{\text{succ}(d) \mapsto \text{succ}(d')}$$

$$\frac{d \text{ is_num } \bar{n}}{\text{succ}(d) \mapsto \text{num}(s(\bar{n}))}$$

$$\frac{d \text{ isnt_num}}{\text{succ}(d) \text{ err}}$$

Simultaneous inductive definition of $d \mapsto d'$ and $d \text{ err}$

16

Dynamic Semantics for $\mathcal{L}\{\text{dyn}\}$

Dynamic semantics rules for ifz:

$$\frac{d \mapsto d'}{\text{ifz}(d; d_0; x.d_1) \mapsto \text{ifz}(d'; d_0; x.d_1)}$$

$$\frac{d \text{ is_num } z}{\text{ifz}(d; d_0; x.d_1) \mapsto d_0}$$

$$\frac{d \text{ is_num } s(\bar{n})}{\text{ifz}(d; d_0; x.d_1) \mapsto [\text{num}(\bar{n})/x]d_1}$$

$$\frac{d \text{ isnt_num}}{\text{ifz}(d; d_0; x.d_1) \text{ err}}$$

NB: In third rule, labelled value $\text{num}(\bar{n})$ is bound to x to preserve invariant that variables are bound to $\mathcal{L}\{\text{dyn}\}$ expressions.

17

Dynamic Semantics for $\mathcal{L}_{\{\text{dyn}\}}$

Rules for application and general recursion:

$$\frac{d_1 \mapsto d'_1}{\text{dap}(d_1; d_2) \mapsto \text{dap}(d'_1; d_2)}$$

$$\frac{d_1 \text{ is_fun } \lambda(x.d)}{\text{dap}(d_1; d_2) \mapsto [d_2/x]d}$$

$$\frac{d_1 \text{ isnt_fun}}{\text{dap}(d_1; d_2) \text{ err}}$$

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

18

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

19

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.

20

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.

21

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?

22

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**!

23

Dynamic Typing as Static Typing

Adding dynamic typing to $\mathcal{L}\{\text{nat} \rightarrow\}$ (PCF):

- Add a new type `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 `dyn`.

24

Hybrid Typing: Syntax

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

Category	Item	Abstract	Concrete
Type	τ	::= <code>dyn</code>	<code>dyn</code>
Expr	e	::= <code>new[l](e)</code> <code>cast[l](e)</code>	<code>!e</code> <code>e?l</code>
Class	l	::= <code>num</code> <code>fun</code>	<code>num</code> <code>fun</code>

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

25

Hybrid Typing: Static Semantics

Static semantics for $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ extend the typing rules for $\mathcal{L}\{\text{nat} \rightarrow\}$ (PCF) with rules for labelling values:

$$\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{new}[\text{num}](e) : \text{dyn}}$$

$$\frac{\Gamma \vdash e : \text{parr}(\text{dyn}; \text{dyn})}{\Gamma \vdash \text{new}[\text{fun}](e) : \text{dyn}}$$

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

26

Hybrid Typing: Static Semantics

Further extend the typing rules for PCF with rules for checking labels:

$$\frac{\Gamma \vdash e : \text{dyn}}{\Gamma \vdash \text{cast}[\text{num}](e) : \text{nat}}$$

$$\frac{\Gamma \vdash e : \text{dyn}}{\Gamma \vdash \text{cast}[\text{fun}](e) : \text{parr}(\text{dyn}; \text{dyn})}$$

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

27

Hybrid Typing: Dynamic Semantics

Dynamic semantics of $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ add rules for downcasting:

$$\frac{e \text{ val}}{\text{new}[l](e) \text{ val}}$$

$$\frac{e \mapsto e'}{\text{new}[l](e) \mapsto \text{new}[l](e')}$$

$$\frac{e \mapsto e'}{\text{cast}[l](e) \mapsto \text{cast}[l](e')}$$

$$\frac{\text{new}[l](e) \text{ val}}{\text{cast}[l](\text{new}[l](e)) \mapsto e}$$

$$\frac{\text{new}[l'](e) \text{ val} \quad l \neq l'}{\text{cast}[l](\text{new}[l'](e)) \text{ err}}$$

28

Safety for Hybrid PCF

Lemma 2 (Canonical Forms)

If $e : \text{dyn}$ and $e \text{ val}$ then $e = \text{new}[l](e')$ for some class l and $e' \text{ val}$. If $l = \text{num}$ then $e' : \text{nat}$ and if $l = \text{fun}$ then $e' : \text{parr}(\text{dyn}; \text{dyn})$.

Theorem 3 (Safety)

The language $\mathcal{L}\{\text{nat dyn} \rightarrow\}$ is safe:

1. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.
2. If $e : \tau$, then either $e \text{ val}$, or $e \text{ err}$, or there exists e' such that $e \mapsto e'$.

29

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
```

30

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!

31

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, Int 1),
                           checked_mult (n, a)))
          | _ => raise RuntimeError
    in loop (n, Int 1)
  end
```

32

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
```

33

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**.

34