Dynamic Typing

Questions

- What are the tradeoffs between static and dynamic types?
- Why are many languages with interpreter based implementations dynamically typed?
  - e.g. Python, Perl, Tcl, Sh, …
- Why languages that are often compiled statically typed?
  - e.g. C, C++, Java, SML, …

Warning

- There is much controversy over definitions of what it means to be a “dynamically typed” language
  - untyped?
  - uni-typed?
  - stupid and broken
- Try to focus on the underlying semantics and don’t get caught up in the “what you call it wars”
  - I’ll call it stupid and broken (Harper is only slightly more diplomatic)

Typing vs. Safety

<table>
<thead>
<tr>
<th></th>
<th>statically typed</th>
<th>untyped or “untyped”</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe</td>
<td>SML, Java,</td>
<td>Scheme, Lisp, Perl,…</td>
</tr>
<tr>
<td></td>
<td>Modula-3</td>
<td></td>
</tr>
<tr>
<td>unsafe</td>
<td>C, C++, Pascal</td>
<td>BCPL, assembly</td>
</tr>
</tbody>
</table>

What is Safety?

Safety – No unchecked runtime “errors”
What’s a runtime error?
Up to the definition of language!
Need to carefully understand what situations are considered errors to understand what it means for a language to be safe.

What is Dynamic Typing?

Language feature where “type errors” are checked at runtime
What’s a type?
Again depends on language definition?
Why do people like dynamic typing?
Dynamic Typing
Delays errors until runtime
Results in more flexible language
Allows for heterogeneous data-structures
Easier to build prototypes

(if true then 7 else "7") + 1
[true, 1, 3.4, fn x=>x]

Semantics
• Treat type errors like other checked errors
  – e.g. division by zero
• Take original dynamic semantics modify so that all “stuck” cases go to a well defined error state
• Resulting language is safe because there are no stuck states

Example: MinML
\[
\frac{v \text{ value } \ v_1 \text{ value } \ (v = \text{ fun } \ (x: \tau_1): \tau_2 \text{ is } \epsilon)}{\text{apply}(v, v_1) \mapsto \{v, v_1/f, x\} \epsilon}
\]
\[
\frac{v \text{ value } \ v_1 \text{ value } \ (v \neq \text{ fun } \ (x: \tau_1): \tau_2 \text{ is } \epsilon)}{\text{apply}(v, v_1) \mapsto \text{error}}
\]

Which is Better?
• Harper attempts to show that dynamic typing is just a “special case” of static typing
  – Shows that dynamic typing has performance costs
  – Therefore should avoid dynamic typing when possible
• Question: Do you believe Harper’s argument?

Example: MinML
\[
\frac{v_1 \text{ value } \ e_2 \mapsto e'_2}{\text{apply}(v_1, e_2) \mapsto \text{apply}(v_1, e'_2)}
\]
\[
\frac{e_1 \mapsto e'_1}{\text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)}
\]
\[
\frac{v_1 \text{ value}}{\text{apply}(v_1, \text{error}) \mapsto \text{error}}
\]
\[
\frac{\text{apply}(\text{error}, e_2) \mapsto \text{error}}{\text{apply}(\text{error}, e_2) \mapsto \text{error}}
\]

Implementation Issues
• In a statically typed language we can represent everything as simple machine words (int, code pointer, bool)
• In-order to implement the dynamic semantics for dynamic typing must tag our values so we can test at runtime what type a value is
Implementation Issues (cont.)

Expressioms \( c := x \mid v \mid o(c_1, \ldots, c_n) \mid \text{if} c_1 \text{then} c_2 \text{else} c_2 \mid \text{apply}(c_1,c_2) \)

TaggedValues \( v := \text{Int}(n) \mid \text{Bool}(true) \mid \text{Bool}(false) \mid \text{Fun}(fun(x : \mathcal{T}:\mathcal{T} : \text{is} \mathcal{C}) \)

UntaggedValues \( u := \text{true} \mid \text{false} \mid n \mid \text{fun}(x : \mathcal{T} : \text{is} \mathcal{C}) \)

Tag checking rules

\[
\begin{align*}
\text{Fun} (\text{u}) \text{ is} \text{fun} \text{u} \\
\text{Int} (\text{.}) \text{ is} \text{nt} \text{fun} \\
\text{Bool} (\text{.}) \text{ is} \text{nt} \text{fun}
\end{align*}
\]

MinML Interpreter

MinML interpreter is doing this tagging/untagging and tag checking already!

This is because this is the only way to safely implement an interpreter in SML.

If we were implementing MinML with C we could avoid runtime checks and rely on MinML static type-checker.

Even in C you'd probably add the tags for debugging!

Pay as You Go

- Dynamic typing requires tagging semantics.
- Tagging adds cost to runtime of type-error free programs.
- Violates pay-as-you-go principle.
  - Cost of a feature should only be imposed when it is used.
  - Hence dynamic typing is bad.

Harper’s Argument (cont.)

- If we add a type of tagged values to our statically typed language a programmer can choose to have dynamic typing when the programmer chooses too.
- Make tagging a programmer decision not a language designer decision use static types to do this!

Example

(* The type of tagged values. *)

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

def exception TypeError
fun checked.add (m:tagged, n:tagged):tagged =
    case (m,n) of
        (Int a, Int b) => Int (a+b)
        (_, _) => raise TypeError
Example (cont.)

```ocaml
fun checked_apply (f:tagged, a:tagged):tagged =
  case f of
  Fun g => g a
  _ => raise TypeError

val het_list : tagged list =
  [Int 1, Bool true, Fun (fn x => x)]
val f : tagged = hd(tl(tl het_list))
val x : tagged = checked_apply (f, Int 5)
```

Example: Factorial

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

Example: Optimized Factorial

```ocaml
fun opt_dyn_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 RuntimeTypeError
  end
```

Observations

- Performing tag elimination operations automatically is hard to do in a **reliable** way
- Reliable means the program knows when the compiler is smart enough to eliminate a needless check and knows what changes to the programs won’t confuse the optimizer

Summary of Harper’s Argument

- Dynamic typing is just dynamic tag checking
- Adding tags pervasively to your implementation violates pay-as-you-go
- Give programmer control by adding a tagged typed to your statically typed language
- Let programmer perform some tag checking optimizations by hand

Complaints

- Writing `tagged_add(Int 1,Int 2)` is such a pain
- What if I forget to include all the values I need to tag in my tagged type?
- When I prototype my program everything is going to start out tagged so why bother with static types in the first place, just make everything tagged by default!
Summary of My View Point

- Static types are great for
  - modularity and data abstraction (i.e. big programs)
  - optimizing compilers
    (good type info make it easier to make programs run fast)
  - documentation for people who have to maintain your code

Should Perl be Statically Typed?

- Dan’s rule of thumb for Perl/Python hacking
  - It should be less than one page
  - I want to finish the script quickly, I don’t care how long it runs
  - I never want to have to maintain it or force anyone else to
- Of course lots of real Perl/Python scripts violate all those assumptions

“Scripting” Language Myth’s

- Every “scripting” language I’ve every seen sooner or later turns into a full blown language with lots of libraries and whole applications written in them
- In reality, no program is going to be smaller than one page, performance does matter, and someone will be stuck maintaining it

Soft Typing

- Approach that tries to get the best of both worlds
- Programmer writes programs without types
- Soft-type checker detects obvious errors or inserts runtime time check when not sure
- Soft-type checker tries to minimize runtime penalty and reduced programmer burden

Soft Typing Case Study

The DIALYZER: a DIscrepancy AnaLYZer for ERlang programs

http://www.it.uu.se/research/group/hipe/dialyzer/

Apply soft-typing to large existing well tested code base for telecom switching application

Summary

- Start with a statically typed language and add dynamic typing by introducing tagged types
- Start with a dynamically typed language and add more static checking with a soft-typing

One day maybe two approaches will meet in the middle