Typed Lambda Calculus and Exception Handling

Dan Zingaro

`zingard@mcmaster.ca`

McMaster University

Untyped Lambda Calculus

- Goal is to introduce typing rules that are realistic, and satisfy the preservation and progress theorems
- First attempt: claim that all lambda-abstractions are of the same type, called $\rightarrow$
- This would assert that $(\lambda x.x)$, and $(\lambda x.\lambda y.xy)$ are of the same type
- If we claim this is true, what does the type become under application?
- May become a value (first example), or another function (second example). More information necessary.

Family of Types

- Replace $\rightarrow$ with $T_1 \rightarrow T_2$, representing functions that take type T_1 as argument and return type T_2
- This is of course what we see in functional languages; for example, Ocaml's filter function:
- `filter : ('a -> bool) -> 'a list -> 'a list`
- The $\rightarrow$ constructor is right-associative, so filter takes a function mapping alpha to boolean, and a list of alphas, and returns a list of alphas

Type of Argument

- How do we know which type of argument a function expects?
- Option 1 : type annotations...
- `let addFive x : int = x + 5`
- Option 2: infer the type...
- Since x is added to 5, we know something about x without the annotation - namely that it must be a number
- Type annotations are used in TAPL
- The type that the function returns is the type of its body, using the additional annotation from the head

Rule for Abstractions

This gives the first attempt for the abstraction rule in the typing relation.

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2}$$

- In words: “if, under the assumption that x has type T1, we can deduce that t2 has type T2, then we know that $\lambda x : T_1 . t_2$ has type $T_1 \rightarrow T_2$”
- This only helps us with the assumption given for the type of the head of the inner-most lambda abstraction, though
- For example, if we had $(\lambda x : \text{int}.\lambda y : \text{int}.x)$, and we wanted the type of the inner function, we’d be asking for the type of $(\lambda y.x)$ by knowing just the fact that y has type int

Typing Contexts

- A typing context Γ is a sequence of variables and their types, containing the free variables of a term
- Now the $(\lambda x : \text{int}.\lambda y : \text{int}.x)$ example is trivial: the type of x can just be looked up
- Refined typing rule for abstractions is the same as before, plus Γ:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

Other Typing Rules

Variables:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Application (if t_1 has type $X \rightarrow Y$, and we apply it to something of type X, we get type Y):

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

Typing Example

- We'll show that $(\lambda x : \text{Bool}.x)\text{true}$ has type Bool in the empty context
- First, under context $[x : \text{Bool}]$, we know that $x : \text{Bool}$ (wow!)
- Next we use the abstraction rule: We know that under the empty context concatenated with $[x : \text{Bool}]$, we have $x : \text{Bool}$ from above
- Therefore under the empty context, we have $\vdash \lambda x : \text{Bool}.x : \text{Bool} \rightarrow \text{Bool}$
- Finally, since True is of type Bool, applying $(\lambda x : \text{Bool}.x)$ to true yields a Bool, by the application rule

Easy Exercises

- What's the type of $f(\text{if false then true else false})$ under context $[f : \text{Bool} \to \text{Bool}]$?
- Find a context so that $f\,x\,y$ has type Bool.

Inversion Lemma

- By reversing the typing relation, we have some basic statements of the form “if t has any type, then its type is …”. A few examples:
- If $\Gamma \vdash x : R$, then $x : R \in \Gamma$
- If $\Gamma \vdash t_1 t_2 : R$, then there is some type T_{11} such that $\Gamma \vdash t_1 : T_{11} \to R$ and $\Gamma \vdash t_2 : T_{11}$
- If $\Gamma \vdash \mathsf{true} : R$, then $R = \mathsf{Bool}$

Other Lemmas

- We also have the following (just like for integers and booleans):
- A term whose free variables are all in the context Γ, has at most one type
- Progress theorem for closed terms (not open ones): if its well-typed, then its a value or can take an evaluation step
- Types are preserved under substitution: if $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash (x \leftarrow s)t : T$
- I.E., if x and s have the same type, and we replace x for s in t, then t still has the same type
- Preservation of types: if its well-typed and it takes a step, its still well-typed

Erasure

- In compilers, type annotations are used during typechecking and code generation, but not at runtime (I.E. they get dropped from generated code)
- This takes our typed lambda calculus back to untyped form (using an erasure function), prior to evaluation
- Erasure and evaluation commute: we get the same thing if we erase and then evaluate (like above), or we evaluate and then erase (carry type info along to the end, then get rid of it and print the result)

Exception Handling

- Two possibilities for what to do when a function can’t execute:
- Return a value of type Option, which contains None or Some. Caller has to check the return of every function (*puke*)
- Use exceptions, which may abort the program entirely, or be caught by an exception handler somewhere along the call stack

Exceptions that Terminate the Program

- Assume that programs that terminate exceptionally result in the `error` term
- We add a new syntactic construct:
- $t ::=$ error
- New evaluation rules (when error is evaluated, it aborts evaluation of its enclosing term):
- error $t_2 \longrightarrow$ error
- v_1 error $\longrightarrow$ error
- Finally, a new typing rule:
- $\Gamma \vdash$ error : T

Some Considerations

- Why isn't error a Value?
- Consider term $(\lambda x : \text{nat}.0)$error
- If error is a value, then we can perform a beta-reduction to arrive at 0, or use the second rule above to result in error; nondeterministic choice!
- The typing rule allows error to have any type at all, so it can be inserted in any context
- What about the type preservation theorem?

Handling Exceptions

- Activation records are popped off the call stack until an exception handler is encountered; evaluation continues at this point
- We introduce a new syntactic construct try t with t, where the first subexpression is evaluated and, if it aborts with an exception, the second is evaluated instead

Evaluation Rules

- When we have a value in the “try”, we can throw everything else out:
- try v_1 with $t_2 \longrightarrow v_1$
- When “trying” an error, we start evaluating the “with”:
- try error with $t_2 \longrightarrow t_2$
- Otherwise, we keep evaluating the “try”:

$$\frac{t_1 \longrightarrow t_1'}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{ try } t_1' \text{ with } t_2}$$

Typing Rule

- Both branches of the “try” have to be the same type, and this is the type of the whole construct:

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 : T}$$

Exceptions with Values

- With the previous mechanisms, all we know is that something bad happened
- Its useful to send some meaningful information along with the exception, so the handlers have more information for how to deal with it
- The type of these values will be denoted T_{exn}
- This now closely approximates Ocaml exceptions, for example:
- `exception Result of string;;`

Evaluation Rules

- Replace `error` with `raise`, which takes a term; the rules are then similar to those for error:
- $(\text{raise } v_{11})t_2 \longrightarrow \text{raise } v_{11}$
- $v_1(\text{raise } v_{21}) \longrightarrow \text{raise } v_{21}$
- $\text{raise}(\text{raise } v_{11}) \longrightarrow \text{raise } v_{11}$

$$\frac{t_1 \longrightarrow t_1'}{\text{raise } t_1 \longrightarrow \text{raise } t_1'}$$

Remaining Evaluation Rules

- Like before, we have:
- try v_1 with $t_2 \longrightarrow v_1$
- When we reach an exception, we pass the value to the exception handler:
- try raise v_{11} with $t_2 \longrightarrow t_2\, v_{11}$

$$\frac{t_1 \longrightarrow t_1'}{\text{try } t_1 \text{ with } t_2 \longrightarrow \text{try } t_1' \text{ with } t_2}$$

Typing Rules

Raise, like error, can have any type, as long as the value is T_{exn}:

$$\frac{\Gamma \vdash t_1 : T_{\text{exn}}}{\Gamma \vdash \text{raise } t_1 : T}$$

The "with" has to have the same type as "try", after the value is passed:

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T_{\text{exn}} \to T}{\Gamma \vdash \text{try } t_1 \text{ with } t_2 \to T}$$

What's this $T_{mathrmexn}$?

- Type Nat (error codes)
- Type String (more descriptive info, but may require parsing)
- A variant type (but requires fixed set of choices)
- Extensible variant type (lets user add exceptions; ML)
- Similar idea in Java, where new exceptions are classes extending Throwable

Exception Example

When is an exception thrown in the below example? What does the exception handler do?

sub $= (\lambda x.$if $x <= 0$ then raise "negative result" else $x - 1)$

safeSub $= (\lambda x.$try sub x with λy : string$.0)$

safeSub -3