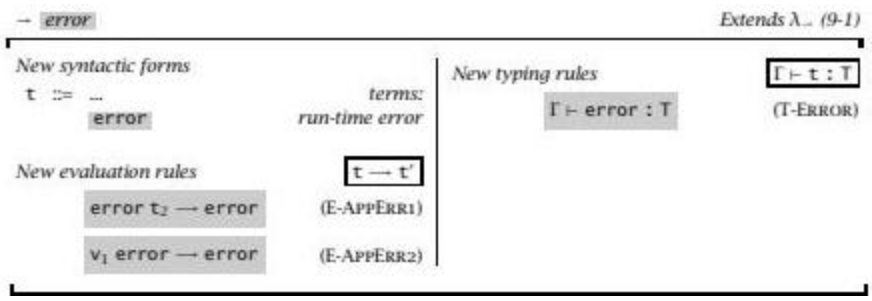
14.1 Raising Exceptions

Let's dive into exception handling by adding a term called error to the simply typed lambda-calculus. When evaluated, error aborts the term it's in.

Here's the formal definition:



The image shows the introduction of a new syntactic form "error," representing a run-time error, along with new evaluation rules (E-AppErr1) and (E-AppErr2) and a typing rule (T-Error).

Here's a breakdown of how error works:

- Operational Semantics: The key is how "abnormal termination" is formalized. error itself becomes the result of an aborted program. The rules E-AppErr1 and E-AppErr2 define this behavior.
    - E-AppErr1: If error is encountered while reducing the left-hand side of an application to a value, the result is immediately error.
    - E-AppErr2: If error occurs while reducing the argument of an application, the application is abandoned, and the result is error.
- Important Observations:
    - error is not a value, only a term.
    - This ensures no overlap between the E-AppAbs and E-AppErr2 rules.
    - Evaluation remains deterministic, avoiding ambiguity.
- Typing Rule (T-Error):
    - The error form can have any type.
    - Example: In `(λx:Bool.x) error`, it has type `Bool`.
    - Example: In `(λx:Bool.x) (error true)`, it has type `Bool→Bool`.

This flexibility can complicate typechecking algorithms because it violates the unique type property. Solutions include:

- Using a minimal type `Bot` (in languages with subtyping).
- Using a polymorphic type `∀X.X` (in languages with parametric polymorphism).

Both tricks represent infinitely many possible types for error compactly.

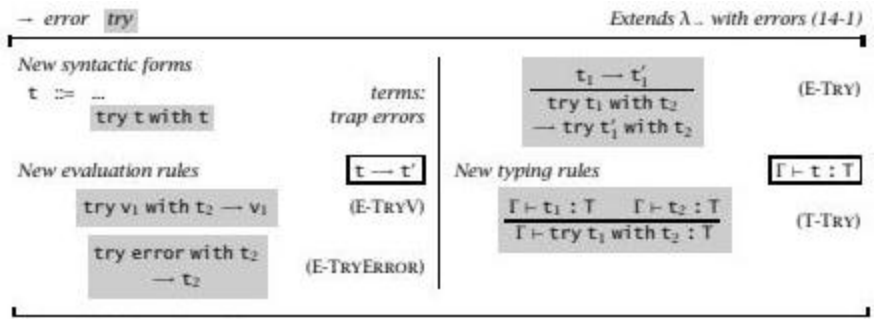
14.1.1 Theorem [Progress]

The progress property is refined to account for the new normal form, error.

> Suppose t is a closed, well-typed normal form. Then either t is a value or t = error.

14.2 Handling Exceptions

Instead of just aborting, let's add exception handlers.



> The image shows the extension of lambda calculus with errors, including new syntactic forms, evaluation rules, and typing rules for handling errors.

The expression `try t1 with t2` means:

> Evaluate `t1`, but if it aborts, evaluate the handler `t2` instead.

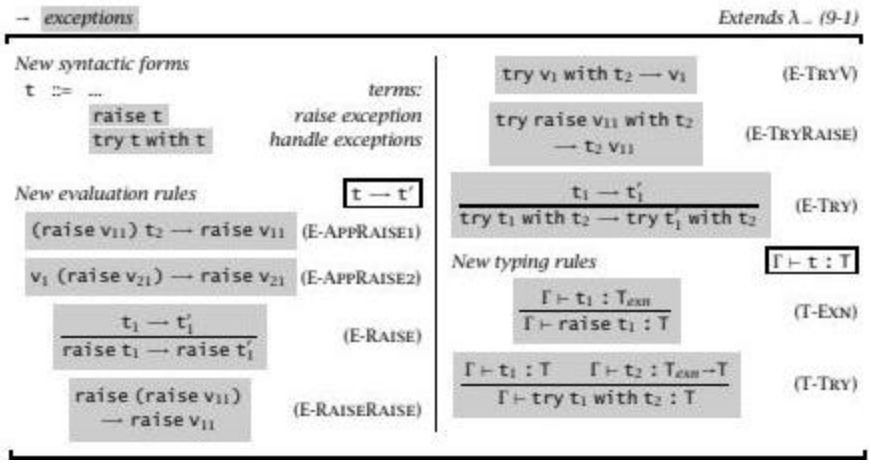
The evaluation rules are:

- E-TryV: If `t1` reduces to a value `v1`, the `try` is discarded.
- E-TryError: If `t1` results in `error`, replace the `try` with `t2` and continue.
- E-Try: Until `t1` is a value or `error`, keep evaluating it and leave `t2` alone.

The typing rule for `try` requires that the main body `t1` and the handler `t2` have the same type `T`, which is also the type of the `try`.

14.3 Exceptions Carrying Values

To provide more context about what went wrong, let's enrich our exception handling to carry values along with the exception.



The image displays the extensions to a lambda calculus system, adding support for exceptions with values, including new syntactic forms, evaluation rules, and typing rules.

Key Changes:

- raise t: replaces the atomic term error, allowing us to pass extra information `t` to the handler.
- The handler `t2` in `try t1 with t2` now takes the extra information as an argument.
- Type of the value carried by the exception is `Texn`.

Evaluation Rules:

- E-TryRaise: Passes the extra information carried by a `raise` from the body `t1` to the handler `t2`.
- E-AppRaise1 and E-AppRaise2: Propagate exceptions through applications (similar to E-AppErr1 and E-AppErr2).
- E-RaiseRaise: Propagates exceptions that may occur while evaluating the extra information.
- E-TryV: Discards the `try` if the main body reduces to a value.
- E-Try: Evaluates the body of the `try` until it becomes a value or a `raise`.

Typing Rules:

- T-Raise: The extra information must have type `Texn`. The whole `raise` can have any type `T`.
- T-Try: The handler `t2` must be a function that takes the extra information of type `Texn` and yields a result of the same type as `t1`.

Alternatives for the type `Texn`:

1. **Nat**: Corresponds to the `errno` convention.
2. **String**: Allows more descriptive messages but may require parsing.
3. **Variant Type**:

   Example: `Texn = <divideByZero: Unit, overflow: Unit, fileNotFound: String, fileNotReadable: String, ... >` Allows handlers to distinguish exceptions using a `case` expression.

4. **Extensible Variant Type**: ML adopts this, using a single extensible variant type called `exn`.
   - The ML declaration `exception l of T` adds a new tag `l` to `Texn`.
   - ML Syntax:
     - `raise l(t) def = raise (<l=t> as Texn)`
     - `try t with l(x) → h def = try t with λe:Texn. case e of <l=x> ⇒ h | _ ⇒ raise e`
5. **Java Classes**: Uses classes instead of extensible variants. `Throwable` class is used for exceptions.
   - New exceptions are declared as subclasses of `Throwable`.
   - Exceptions (subclasses of `Exception`) can be caught and recovered from.
   - Errors (subclasses of `Error`) indicate serious conditions that should terminate execution.
   - Java requires methods to declare which exceptions they might raise.