A type system and logic for exceptions

John Longley

We will give two type systems for a simply typed lambda calculus with exceptions. In both, the type system keeps track of the set of exceptions that a program could possibly raise, and indeed we obtain type safety results saying that programs cannot raise any exceptions not declared by their type.

The first system, called $\mathcal{E}_1$, will allow us to infer principal types very easily for all terms that are well-typed in the ordinary sense, This version may be of interest in its own right from the point of view of programming language design and compiling techniques. The second version, called $\mathcal{E}_2$, will be slightly more restrictive — its purpose is to exclude certain programs that are problematic from a semantic point of view and get in the way of natural reasoning principles. We will then show how to give a simple and fairly clean program logic for $\mathcal{E}_2$ based on a denotational semantics for this language.

1 A safe type system for exceptions

1.1 The basic system

To start with, we define a simply typed language with a fixed set E of non-parameterized exceptions (we will relax these assumptions later). We let e range over E and ζ range over finite subsets of E. (Maybe represented by lists in an implementation.) We write ζ, ζ' for $\zeta \cup \zeta'$, and ζ, e or e, ζ for $\zeta \cup \{e\}$.

Starting with a bunch of *ground types* γ, we define the syntax of simple types as usual, and define *e-types* to be types with an associated set of exceptions.

$$\begin{array}{rrcl} \text{Types:} & \sigma & ::= & \gamma \mid \sigma_1 \to \sigma_2 \\ \text{E-types:} & \tau & ::= & \sigma[\zeta] \\ \text{Contexts:} & \Gamma & ::= & x_1 : \sigma_1, \ldots, x_n : \sigma_n \end{array}$$

We write $\sigma[\,]$ for $\sigma[\emptyset]$. A *ground context* is one in which all the σ_i are ground types.

We will consider only terms M given by the following grammar. We assume we are given some collection of typed constants $c : \sigma$, including at least one constant of each ground type.

$$M \ ::= \ c \mid x \mid M_1\, M_2 \mid \texttt{fn}\, x : \sigma \texttt{ => } M_1 \mid \texttt{raise}_\sigma e \mid M_1 \texttt{ handle } e \texttt{ => } M_2$$

We give rules for typing judgements $\Gamma \vdash M : \tau$. The idea is that if $\vdash M : \sigma[\zeta]$, then $C[M]$ cannot raise an uncaught exception not in ζ for any context $C[-]$ that does not itself raise exceptions. (As an analogy, we may think of ζ as something like a set of "free variables" that may appear in M.) This idea will be made precise later.

The typing rules for our system $\mathcal{E}_1$ are as follows:

Constants
$$\frac{}{\Gamma \vdash c : \sigma[\,]} \quad c : \sigma$$

Variables
$$\frac{}{\Gamma \vdash x : \sigma[\,]} \quad \Gamma(x) = \sigma$$

Promotion
$$\frac{\Gamma \vdash M : \sigma[\zeta]}{\Gamma \vdash M : \sigma[\zeta, \zeta']}$$

Application
$$\frac{\Gamma \vdash M : (\sigma \to \sigma')[\zeta] \qquad \Gamma \vdash N : \sigma[\zeta]}{\Gamma \vdash MN : \sigma'[\zeta]}$$

Abstraction
$$\frac{\Gamma, x : \sigma \vdash M : \sigma'[\zeta]}{\Gamma \vdash \texttt{fn}\, x : \sigma \texttt{ => } M : (\sigma \to \sigma')[\zeta]}$$

Raising
$$\frac{}{\Gamma \vdash \texttt{raise}_\sigma e : \sigma[e]}$$

Handling
$$\frac{\Gamma \vdash M : \sigma[\zeta, e] \qquad \Gamma \vdash N : \sigma[\zeta]}{\Gamma \vdash M \texttt{ handle } e \texttt{ => } N : \sigma[\zeta']} \quad \zeta' = \begin{cases} \zeta \text{ if } \sigma \text{ ground} \\ \zeta, e \text{ otherwise} \end{cases}$$

The reason for the difference between the two handling rules is that if M is of ground type, an exception handler can be thought of as a "binder" for the exception e, which can never leak out of M, whereas if M is e.g. of function type, it may be that some term MP raises e even if M doesn't, so this use of handling cannot be regarded as binding e. (We don't strictly need the side-condition in the second rule for handling, but it helps to cut down on multiple derivations.)

Examples

Suppose `nat` and `natlist` are ground types. In the presence of a few suitable PCF-style constants, we could define closed terms:

$$\begin{aligned}
\vdash \texttt{map} &: ((\texttt{nat} \to \texttt{nat}) \to (\texttt{natlist} \to \texttt{natlist}))[\,] \\
\vdash \texttt{succ} &: (\texttt{nat} \to \texttt{nat})[\,] \\
\vdash \texttt{pred} &: (\texttt{nat} \to \texttt{nat})[e] \\
\vdash \texttt{rev} &: (\texttt{natlist} \to \texttt{natlist})[\,] \\
\vdash \texttt{ten} &: \texttt{natlist}[\,]
\end{aligned}$$

(We intend that `ten` is `[0,...,9]`.) Then the following are all principal typing judgements in $\mathcal{E}_1$. We assume $e' \neq e$, and omit some type annotations for readability.

$$\begin{aligned}
\vdash \texttt{map succ} &: (\texttt{natlist} \to \texttt{natlist})[\,] \\
\vdash \texttt{map pred} &: (\texttt{natlist} \to \texttt{natlist})[e] \\
\vdash (\texttt{map pred}) \texttt{ handle } e \texttt{ => rev} &: (\texttt{natlist} \to \texttt{natlist})[e] \\
\vdash \texttt{map pred ten} &: \texttt{natlist}[e] \\
\vdash (\texttt{map pred ten}) \texttt{ handle } e \texttt{ => ten} &: \texttt{natlist}[\,] \\
\vdash (\texttt{map pred ten}) \texttt{ handle } e \texttt{ => raise } e &: \texttt{natlist}[e] \\
\vdash \texttt{map pred } (\texttt{raise } e') &: \texttt{natlist}[e, e'] \\
\vdash (\texttt{map pred } (\texttt{raise } e')) \texttt{ handle } e \texttt{ => ten} &: \texttt{natlist}[e'] \\
\vdash \texttt{fn } f \texttt{ => fn } l \texttt{ => map } f \ (\texttt{rev } l) &: ((\texttt{nat} \to \texttt{nat}) \to \\
&\quad (\texttt{natlist} \to \texttt{natlist}))[\,] \\
\vdash \texttt{fn } f \texttt{ => } ((f\ 5) \texttt{ handle } e \texttt{ => } 3) &: ((\texttt{nat} \to \texttt{nat}) \to \texttt{nat})[] \\
\vdash (\texttt{fn } f \texttt{ => } ((f\ 5) \texttt{ handle } e \texttt{ => } 3)) \texttt{ pred} &: \texttt{nat}[e]
\end{aligned}$$

Note that in the last example, our type system does *not* capture the fact that the term can never raise e. A more sophisticated type system might be able to capture this, but probably at the cost of significant added complication.

These examples illustrate an important point about our system: we only need to mention exceptions in connection with parts of our code that explicitly use them — the presence of exceptions doesn't complicate the types of exception-free code.

Another useful observation is that we may write "cleaned up" versions of existing programs that eliminate exceptions from their type. For example, if $\vdash M : (\sigma \to \texttt{nat})[e]$, for example, then $\vdash \texttt{fn } x : \sigma \texttt{ => } (M \texttt{ handle } e \texttt{ => } 0) : (\sigma \to \texttt{nat})[\,]$. However, it is unfortunately not possible to write a higher-order operation F such that $\vdash FM : (\sigma \to \texttt{nat})[\,]$

Properties

Our type system has the following static properties:

Substitution lemma: if $\Gamma, x : \sigma' \vdash M : \sigma[\zeta]$ and $\Delta \vdash N : \sigma'[\zeta']$, then $\Gamma, \Delta \vdash M[N/x] : \sigma[\zeta, \zeta']$.

Principal e-types: if $\Gamma \vdash M : \sigma$ in the usual type system (with exceptions erased), there is a unique smallest ζ such that $\Gamma \vdash M : \sigma[\zeta]$ in $\mathcal{E}_1$. In this case we call $\sigma[\zeta]$ the principal e-type for M in Γ.

Type inference: given Γ and M, it is decidable (in linear time) whether M is typable relative to Γ, and if so the principal e-type can be easily inferred (again in linear time).

It is moreover fairly clear that one could give a variant of the above system with ML-style polymorphism in which the type annotations on `fn` and `raise` expressions could be dropped, and that by combining the usual Hindley-Milner type inference procedure with the procedure for $\mathcal{E}_1$, we would have inferrable principal e-types in this system.

Suppose now we are given a set of reduction rules on closed terms for various constants, of the form

$$c\, M_1 \ldots M_n \longrightarrow N$$

We had better assume these rules are type-safe: if $\vdash c\, M_1 \ldots M_n : \tau$ then $\vdash N : \tau$. Given such a set of rules, it is straightforward to define a single-step (e.g. call-by-value) reduction relation $\longrightarrow$ for well-typed closed terms of $\mathcal{E}_1$. We say a closed term V is a *value* if it cannot be further reduced: we write $M \Rightarrow V$ if $M \longrightarrow^* V$ and V is a value.

Our system then has the following dynamic properties:

Subject reduction: if $\vdash M : \tau$ and $M \longrightarrow N$, then $\vdash N : \tau$.

E-type safety: if $\vdash M : \sigma[\zeta]$ and $M \longrightarrow V$, then $V : \sigma[\zeta]$; in particular, if V has the form `raise` e then $e \in \zeta$.

Let us say a closed term M *semantically has type* $\sigma[\zeta]$, and write $\models M : \sigma[\zeta]$, if for every term context $C[-]$ such that $x : \sigma[\,] \vdash C[x] : \sigma'[\zeta']$, if $C[M] \Rightarrow$ `raise` e then $e \in \zeta \cup \zeta'$. (In particular, if $\zeta' = \emptyset$ then $e \in \zeta$.) We can now state a stronger form of type safety.

E-type soundness: if $\vdash M : \sigma[\zeta]$ then $\models M : \sigma[\zeta]$.

Of course the converse would be too much to expect of any reasonable type system, as it not in general decidable whether a given program is semantically of a given type. Indeed, by comparison with other type systems for exceptions that have been proposed, our system is probably somewhat

coarse. However, the following property shows that we can always recover from any unwanted slackness in the assigned type by wrapping a term in a suitable "coercion" context.

Type disinfection: For any e-type $\sigma[\zeta]$ and any $\zeta' \subseteq \zeta$, there is a context $C[-]$ such that

- if $\vdash M : \sigma[\zeta]$ then $\vdash C[M] : \sigma[\zeta']$;
- if moreover $\models M : \sigma[\zeta']$, then M and $C[M]$ are observationally equivalent.

The reason for this is rather trivial. If $\sigma = \sigma_1 \to \cdots \to \sigma_r \to \gamma$, we may take $C[-]$ to be

$$\texttt{fn}\ x_1 : \sigma_1, \ldots, x_r : \sigma_r\ \texttt{=>}\ (([-]x_1 \ldots x_r)\ \texttt{handle}\ e_1, \ldots, e_r\ \texttt{=>}\ c)$$

where c is a constant of type γ. (We have used some obvious syntactic sugar here for nested `fn` and `handle` expressions.)

This means that any spurious exceptions that appear in the type of M can be prevented from infecting the rest of the program. Of course, it is then the programmer's responsibility to ensure that the exceptions in $\zeta \backslash \zeta'$ can never arise from M (or if they do arise they are appropriately handled).

We now extend our basic system in various directions to take account of some more advanced uses of exceptions.

1.2 Exceptions as first-class values

Suppose we wish to include a type `exn`, such that exceptions are actually values of type `exn`:

$$\sigma\ ::=\ \gamma \mid\ \sigma \to \sigma \mid \texttt{exn}$$

We can do this by adding a new rule for exception constants and generalizing the rule for `raise`:

$$\text{Exception constants} \qquad \frac{}{\Gamma \vdash e : \texttt{exn}[e]}$$

$$\text{Exception raising} \qquad \frac{\Gamma \vdash M : \texttt{exn}[\zeta]}{\Gamma \vdash \texttt{raise}_\sigma M : \sigma[\zeta]}$$

The idea is that the e-type of M now keeps track of not only what exceptions a program $C[M]$ might *raise*, but also what exceptions it might *be*. The appropriate semantic notion of typing is now: $\models M : \sigma[\zeta]$ iff for all $C[-]$ of suitable type, if $C[M] \Rightarrow \texttt{raise}_\sigma e$ or $C[M] \Rightarrow e$ then $e \in \zeta$. With this in mind, all the previously stated properties of our system go through, except that in the statement of type disinfection we should require that the type σ does not involve `exn`.

1.3 Wildcard handlers

Extending the system to allow wildcard handlers is also straightforward. Let us extend the syntax of terms with the form:

$$M_1 \texttt{ handle \_=> } M_2$$

We may give the following typing rules for such terms, covering ground and non-ground types respectively:

Anonymous wildcards

$$\frac{\Gamma \vdash M : \gamma[\zeta] \qquad \Gamma \vdash N : \gamma[\zeta']}{\Gamma \vdash M \texttt{ handle \_=> } N : \gamma[\zeta']}$$

$$\frac{\Gamma \vdash M : \sigma[\zeta] \qquad \Gamma \vdash N : \sigma[\zeta']}{\Gamma \vdash M \texttt{ handle \_=> } N : \sigma[\zeta, \zeta']} \quad \sigma \text{ non-ground}$$

If we have the type `exn`, we might also wish to allow terms of the form:

$$M_1 \texttt{ handle } x \texttt{ => } M_2$$

where the variable x may appear in M_2. The corresponding typing rule for this construct is as follows.

Variable wildcards

$$\frac{\Gamma \vdash M : \sigma[\zeta] \qquad \Gamma, x : \texttt{exn} \vdash N : \sigma[\zeta']}{\Gamma \vdash M \texttt{ handle } x \texttt{ => } N : \sigma[\zeta, \zeta']}$$

Note that in this case a single rule serves for ground and non-ground types, since in the ground case we can no longer exclude ζ from the e-type of the resulting expression.

All the previously stated properties still hold for the system with these extensions.

1.4 Recursive types

Suppose we extend our class of types with type variables α and recursively defined types:

$$\rho \ ::= \ \alpha \mid \gamma \mid \rho_1 \to \rho_2 \mid \texttt{exn} \mid \mu\alpha.\rho_1$$

From a practical point of view this language for recursive types is severely restricted due to the absence of product and sum types, but it serves to illustrate the treatment of exceptions. We will let ρ range over general types and use σ for *closed* types, i.e. those containing no free type variables. As a minor technicality, all types of the form $\mu\alpha_1.\ldots.\mu\alpha_k.\gamma$ should also be considered to be ground types.

We also add constructor and destructors for the recursive types.

No problems, except that type disinfection no longer holds. Example with streams. If we want this, we need to introduce a new expression form

$$\texttt{Fix}_{\sigma}\, x : \sigma \texttt{ => } M_1$$

and a bit of extra machinery for typechecking these. (Ordinary ML-style recursive function declarations can be treated as sugar for declarations involving `Fix`.) However, this machinery doesn't show up in the terms themselves, the typing judgements, or the properties of the system, which all go through as previously stated.

1.5 Local exceptions

Now suppose we wish to consider a language where the set of exceptions is extensible by means of local exception declarations:

$$\texttt{let exception}\, e \,\texttt{in}\, M_1 \,\texttt{end}$$

Can keep the same notion of e-type, but need to modify our contexts to include information about the exceptions currently in scope. Distinguish between exception *names* e, and unique exception *identifiers* i (need this because of possibility of nested declarations of exceptions with the same name).

Wildcard rules given earlier need to be changed. (Condition on Γ - not LF-friendly!)

Rules all work quite easily, and neatly have the effect of precluding *anonymous exceptions* (i.e. exceptions propagating outside their scope). Although this restricts the set of typable terms (e.g. relative to ML), it would seem to us to be an entirely good thing!

(There's probably a hack we can do to allow anonymous exceptions if we insist. Reasoning about when two anonymous exceptions are equal would be hard though!)

Reduction rules: slightly subtle, as exception identifiers may be renamed to avoid capture. This gives the correct behaviour even for some surprisingly tricky programs. [Give examples.]

Property: no anonymous exceptions.

Type safety and all the rest then go through with no problem.

1.6 Parameterized exceptions

No problem with exceptions parameterized by ground types (easy modifications needed).

Exceptions parameterized by non-ground types (including `exn`?!) are harder because they might themselves raise further exceptions (or the same exception recursively!) At this point we do need a more complex notion of e-type. Forests of exceptions rather than just sets. Example to show this.

Still don't quite know how to cope with really evil recursion examples (which would seem to require an infinitely deep forest...) Use regular languages over exceptions to generalize finite forests?

A hard example:

```
exception e of (unit -> unit) ;
let fun f () = raise e f ;
    fun G f = f () handle e g => G g
in
    G f
end ;  (* diverges! *)
```

What stops this from being given e-type unit[]?

Anyway, this pretty much takes care of all uses of exceptions in ML!

1.7 Comparison with other type systems

Examples where other systems give finer types than ours. Discussion.

2 A type system for a program logic

New notion of e-types: $[\xi]\sigma[\zeta]$. (e1-types or e2-types when we need to distinguish them.) The typing rules for our system $\mathcal{E}_2$ are obtained by enriching those of $\mathcal{E}_1$ as follows.

Constants
$$\frac{}{\Gamma \vdash c : []\sigma[]} \quad c : \sigma$$

Variables
$$\frac{}{\Gamma \vdash x : []\sigma[]} \quad \Gamma(x) = \sigma$$

Promotion
$$\frac{\Gamma \vdash M : [\xi]\sigma[\zeta]}{\Gamma \vdash M : [\xi']\sigma[\zeta']} \quad \xi \subseteq \xi',\ \zeta \subseteq \zeta'$$

Application
$$\frac{\Gamma \vdash M : [\xi](\sigma \to \sigma')[\zeta] \qquad \Gamma \vdash N : [\xi]\sigma[\zeta]}{\Gamma \vdash MN : [\xi]\sigma'[\zeta]}$$

Abstraction
$$\frac{\Gamma, x : \sigma \vdash M : [\xi]\sigma'[\zeta]}{\Gamma \vdash \mathtt{fn}\, x : \sigma \,\mathtt{=>}\, M : [\xi](\sigma \to \sigma')[\zeta]}$$

Raising
$$\frac{}{\Gamma \vdash \mathtt{raise}_\sigma e : []\sigma[e]}$$

Handling
$$\frac{\Gamma \vdash M : [\xi]\sigma[\zeta, e] \qquad \Gamma \vdash N : [\xi]\sigma[\zeta]}{\Gamma \vdash M\ \mathtt{handle}\ e \,\mathtt{=>}\, N : [\xi']\sigma[\zeta']}$$

$$\text{where } \xi' = \begin{cases} \xi \text{ if } \Gamma \text{ ground} \\ \xi, e \text{ otherwise,} \end{cases} \quad \zeta' = \begin{cases} \zeta \text{ if } \sigma \text{ ground} \\ \zeta, e \text{ otherwise} \end{cases}$$

2.1 Adding other features

What other uses of exceptions can we deal with in the logic?

Type `exn` OK (surprisingly).

Wildcard handlers mostly OK (subject to the above restriction).

Inductive types OK I think. (Modelling the `Fix` stuff might be tricky - check!)

Local exceptions OK. (Refine restriction somewhat.) But anonymous exceptions ruled out! Important example: the `catch` program. Even testing equality for exceptions is OK if we don't have non-ground parameterized exceptions (not allowed in ML).

Ground parameterized exceptions OK. Non-ground ones? Hmmm.

2.2 A variant for open terms

3 A logic for exceptions

We'll first consider just the basic version of the language — no type `exn`, wildcard handlers, local exceptions or other fancy stuff. We assume we have a constant

$$\texttt{Fix}_\sigma : (\sigma \to \sigma) \to \sigma$$

for each non-ground σ.

Syntax of formulae:

$$\phi ::= M \equiv N \mid \mathit{dfd}\, M \mid \mathit{false} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \forall x : \tau.\phi_1 \mid \exists x : \tau.\phi_1$$

Typing rules: Need e-contexts. First give typing rules for terms in e-contexts. Typing for formulae is obvious. In $M \equiv N$, the two sides must have the same e-type (though of course implicit promotions are OK). Assume

Sugar: Define `bot` using `Fix` of identity. Write *proper* M for $\mathit{dfd}\, M \wedge \exists x : [\xi]\sigma[\,].\, x \equiv M$. Also write $M \not\equiv N$ for $M \equiv N \Rightarrow \mathit{false}$.

Inference rules: as usual in classical FOL. Equality is reflexive and substitutive.

Operational interpretation?

Axioms / reasoning principles. Here's everything we can think of; there may be some redundancy.

The following are analogues or simple adaptations of axioms we have in the functional fragment. (All of these are represented except function extensionality.)

- Equations arising from any δ-rules: OK for all e-types.
- Usual induction principles for types $[]\sigma[]$.
- `bot` is the unique undefined element for any e-type.
- `fn` expressions are proper.
- The following versions of beta and eta rules:

```
proper N ==> (fn x:t => M)N  ==  M[N/x]
proper M ==> M = (fn x => M x)
```

- Fix and Fix-min as usual.

The following new axioms pertain specifically to exceptions.

- Somehow, all instances of $\Gamma \vdash$ `raise` $e \not\equiv$ `raise` e' whenever $\Gamma(e) \neq \Gamma(e')$. (Hard in Isabelle. OK for small examples, e.g. just two exceptions!)
- Equations arising from reduction rules, e.g.

```
(raise e) handle e => N == N
M =/= raise e ==> M handle e'=> N == M
(raise e) N  ==  raise e
proper M ==> M (raise e) == raise e
```

  plus similar propagation rules for each δ-rule in the system. (Can maybe treat these uniformly somehow?)

- ```
  M == N  <==> promote M == promote N
  dfd M   <==> dfd (promote M)     (* derivable *)
  ```
- For ground and first order types:

```
forall x:[xi]s[zeta]. exists y:[]s[zeta]. y == x.
```

- ```
  forall x:[]gamma[zeta,e].
      x == raise e \/ exists y:[]gamma[zeta]. y == x
  ```
- First order extensionality. Here f, g can have any type $[\xi](\gamma \to \sigma)[\zeta]$ (γ must be ground but σ can be anything.)

```
dfd f /\ dfd g /\ (forall x:[]gamma[]. f x == g x)
    ==> f == g
```

- Surprising higher order quasi-extensionality axiom: it's enough that f, g agree on all arguments that can raise just one exception. Here f, g can have any type $[\xi](\sigma \to \sigma')[\zeta]$.

```
dfd f /\ dfd g /\ (forall x:[]sigma[\xi,e]. f x == g x)
    ==> f == g
```

  Here e can be any exception not in ξ — doesn't matter whether $e \in \zeta$ or not. (I believe this is quite similar to something called "Howe's principle"?)

  (Formalizing just the $\xi = \emptyset$ case would probably be enough to be going on with.)

There are other more jazzy axioms we could add, but the above should be enough for practical purposes. (Axioms for "catch": enough to prove everything?)

Examples to try

- Using quasi-extensionality, we should be able to prove things like $f\,0 * 2 \equiv f\,0 + f\,0$. (Good example!)
- Prove that `fn` f `=>` $f\,0 + f\,1 \not\equiv$ `fn` f `=>` $f\,1 + f\,0$.
- Harder: prove that `(map f) o (map f) == map (f o f)`. How many of the previous list examples go through?
- Should do an example where an exception propagates up through layers of recursion before it is handled. A really nice example would be

```
fun delete' x [] = raise e
  | delete' x (y::l) = if x=y then l else y::delete' x l
fun delete1 x l = delete' x l handle e => l

fun delete0 x [] = []
  | delete0 x (y::l) = if x=y then l else y::delete0 x l
```

  Show that `delete0`, `delete1` are equal.

- An example of type disinfection?

What goes wrong if we try to base a logic on $\mathcal{E}_1$ rather than $\mathcal{E}_2$? We couldn't easily state a correct form of quasi-extensionality. Other examples:

For $F : [\,](($`nat` $\to$ `nat`$) \to$ `nat`$)[\zeta]$ we have

$$F(\texttt{fn}\ x \texttt{ => raise }\ e) \equiv n \ \Rightarrow\ \forall g.\ Fg \equiv n$$

(Challenge: can we prove this? Not with current axioms.) This wouldn't work if an arbitrary ξ were allowed.

"Alpha conversion" for exceptions. E.g. if $C[x : \gamma] : []\gamma'[\zeta']$ then

$$C[\texttt{raise}\ e]\ \texttt{handle}\ e => N\ \equiv\ C[\texttt{raise}\ e']\ \texttt{handle}\ e' => N$$

or something (needs to be made more precise). Would fail if C could handle e or e'.

Other nice properties that break, e.g. F being an SR functional? Observational equivalences?

4 Denotational semantics

Denotational model. Include explicit promotions in syntax. Denotational interpretation. Coherence. Agreement with operational interpretation. Soundness of logic.

5 Conclusion

How we arrived at this. Value of denotational semantics to guide the design of a language and choice of restrictions.

Our system $\mathcal{E}_1$ is a very simple lightweight type system for exceptions which for practical purposes is probably almost as good as existing systems.

Our $\mathcal{E}_2$ has been motivated by logical concerns. But perhaps the information it gives might be of interest for programmers/language designers/compiler writers somehow?