Exception Handlers as Extensible Cases

Matthias Blume Umut A. Acar Wonseok Chae

Toyota Technological Institute at Chicago
{blume,umut,wchae}@tti-c.org

Abstract. Exceptions are an indispensable part of modern programming languages. They are, however, handled poorly, especially by higher-order languages such as Standard ML and Haskell: in both languages a well-typed program can unexpectedly fail due to an uncaught exception. In this paper, we propose a technique for type-safe exception handling. Our approach relies on representing exceptions as sums and assigning exception handlers polymorphic, extensible row types. Based on this representation, we describe an implicitly typed external language EL where well-typed programs do not raise any unhandled exceptions. EL relies on sums, extensible records, and polymorphism to represent exception-handling, and its type system is no more complicated than that for existing languages with polymorphic extensible records.

EL is translated into an internal language IL that is a variant of System F extended with extensible records. The translation performs a CPS transformation to represent exception handlers as continuations. It also relies on duality to transform sums into records. (The details for this translation are given in an accompanying technical report.)

We describe the implementation of a compiler for a concrete language based on EL. The compiler performs full type inference and translates EL-style source code to machine code. Type inference relieves the programmer from having to provide explicit exception annotations. We believe that this is the first practical proposal for integrating exceptions into the type system of a higher-order language.

1 Introduction

Exceptions are widely used in many languages including functional, imperative, and object-oriented ones such as SML, Haskell, C++ and Java. The mechanism enables programs to *raise* an exception when an unexpected condition arises and to *handle* such exceptions at desired program points. Although uncaught exceptions—just like stuck states—constitute unrecoverable runtime errors, their presence or absence is usually not tracked by the type system and, thus, also not included in the definition of soundness. Even strongly typed languages such as ML and Haskell which claim that "well-typed programs do not go wrong" permit uncaught exceptions. This is unfortunate, since in practice an uncaught exception can be as dangerous as a stuck state. Critical software failing due to a divide-by-zero error is just as bad as it failing due to a segmentation fault. (Famously, the Ariane 5 disaster was in part caused by a runtime exception that was not appropriately handled by its flight-control software [15].)

There have been attempts at providing some form of exception checking. For example, CLU, Modula-3, and Java allow the programmer to declare the exceptions that each function may raise and have the compiler check that declared exceptions are caught properly [16,6,10]. If these languages were to be used only in the first-order setting, the approach would work well. But this assumption would preclude first-class use of objects and function closures. Functional languages like ML or Haskell, where higher-order features are used pervasively, do not even attempt to track exceptions statically.

In the higher-order setting, latent exceptions, before they get raised, can be carried around just like any other value inside closures and objects. To statically capture these highly dynamic aspects faithfully and precisely is challenging, which perhaps explains why existing language designs do not incorporate exceptions tracking. A practical design should have the following properties:

(1) Exception types can be **inferred**[1] by the compiler, avoiding the need for prohibitively excessive programmer annotations. (2) **Exception polymorphism** makes it possible for commonly used higher-order functions such as `map` and `filter` to be used with functional arguments that have varying exception signatures. (3) Soundness implies that well-typed programs **handle all exceptions**. (4) The language is **practically implementable**. (5) Types **do not unduly burden** the programmer by being conceptually too complex.

In this paper we describe the design of a language that satisfies these criteria. We use *row types* to represent sets of exceptions and universal quantification over row type variables to provide the necessary polymorphism. As a result, our type system and the inference algorithm are no more complicated than those for extensible records and extensible cases. We implemented this design in our **MLPolyR** compiler.The proof of soundness and more implementation details can be found in the extended technical report [5].

The starting point for our work is **MLPolyR** [4], a language already featuring row polymorphism, polymorphic sum types, and extensible cases. To integrate exceptions into the type system, we consider an implicitly typed *external language* EL (Section 3) that extends λ-calculus with exceptions and extensible cases. Our syntax distinguishes between the act of establishing a new exception handler (**handle**) and that of overriding an existing one (**rehandle**). The latter can be viewed as a combination of **unhandle** (which removes an existing handler) and **handle**. As we will explain in Section 5, this design choice makes it possible to represent exception types as row types without need for additional complexity. From a usability perspective, the design makes overriding a handler explicit, reducing the likelihood of this happening by mistake.

In EL, the typing of an expression includes both a result type and an exception type. The latter describes the set of exceptions that might be raised by the expression. The typing of exception-handling constructs is analogous to that of

[1] There still will be a need for programmers to spell out types, including exception types, for example in module signatures. It is possible to avoid excessive notational overhead by choosing a syntax with good built-in defaults, e.g., shortcuts for common patterns and the ability to elide parts that can be filled in by the compiler. In our prototype, the pretty-printer for types employs such tricks (see footnote 4).

extensible first-class cases. Exception values themselves are sums, and those are also described by row types. To support exceptions, the dynamic semantics of EL evaluates every expression in an *exception context* that is an extensible record of individual evaluation contexts—one for each exception constructor.

Our implementation rests on a deterministic, type-sensitive semantics for EL based on elaboration (i.e., translation) into an explicitly typed internal language IL (Section 4). The elaboration process involves type inference for EL. IL is an extension of System F with extensible records and nearly coincides with the System F language we used in previous work on **MLPolyR**. The translation of exception constructs views an exception handler as an alternative continuation whose domain is the sum of all exceptions that could arise at a given program point. By duality, such a continuation is equivalent to a record of individual continuations, each responsible for a single exception constructor. Taking this view, establishing a handler for a new exception corresponds to functional extension of the exception handler record. Raising an exception simply projects out the appropriate continuation from the handler record and throws its payload to it. Since continuations are just functions, this is the same mechanism that underlies **MLPolyR**'s *first-class cases* feature; first-class cases are also extensible, and their dual representations are records of functions.

Previous work in this context focused on analysis tools for statically detecting uncaught exceptions [11,19,18,7,14,2]. We take advantage of our position as language designers and incorporate an exception analysis into the language and its compiler directly. The ability to adapt the language design to the analysis makes it possible to use a formalism that is simple and transparent to the programmer.

2 Motivating examples

We will now visit a short sequence of simple program fragments, roughly ordered by increasing complexity. None of the examples exhibits uncaught exceptions. The rejection of any one of them by a compiler would constitute a false positive. The type system and the compiler that we describe accept them all.

Of course, baseline functionality consists of being able to match a manifest occurrence of a raised exception with a manifestly matching handler:

```
(... raise `Neg 10 ...) handle `Neg i ⇒ ...
```

The next example moves the site where the exception is raised into a separate function. To handle this in the type system, the function type constructor $\rightarrow$ acquires an additional argument ρ representing the set of exceptions that may be raised by an application, i.e., function types have the form $\tau_1 \xrightarrow{\rho} \tau_2$. This is about as far as existing static exception trackers that are built into programming languages (e.g., Java's **throws** declaration) go.

```
fun foo x = if x<0 then raise `Neg x else ...
(... foo y ...) handle `Neg i ⇒ ...
```

But we also want to be able to track exceptions through calls of higher-order functions such as `map`, which themselves do not raise exceptions while their functional arguments might:

```
fun map f [] = []   | map f (x :: xs) = f x :: map f xs
(... map foo l ...) handle `Neg i ⇒ ...
```

Moreover, in the case of curried functions and partial applications, we want to be able to distinguish stages that do not raise exceptions from those that might. In the example of `map`, there is no possibility of any exception being raised when `map` is partially applied to the function argument; all exceptions are confined to the second stage when the list argument is supplied:

```
val mfoo = map foo
(... mfoo l ...) handle `Neg i ⇒ ...
```

Here, the result `mfoo` of the partial application acts as a data structure that carries a latent exception. In the general case, exception values can occur in any data structure. For example, the SML/NJ Library [9] provides a constructor function for hash tables which accepts a programmer-specified exception value which becomes part of the table's representation from where it can be raised, for example when an attempt is made at looking up a non-existing key.

The following example shows a similar but simpler situation. Function `check` finds the first pair in the given list whose left component does not satisfy the predicate `ok`. If such a pair exists, its right component, which must be an exception value, is raised. To guarantee exception safety, the caller of `check` must be prepared to handle any exception that might be passed along in the argument of the call:

```
fun check ((x, e) :: rest) = if ok x then check rest else raise e
  | check [] = ()
(... check [(3, `A 10), (4, `B true)] ...) handle `A i ⇒ ... | `B b ⇒ ...
```

Finally, exception values can participate in complex data flow patterns. The following example illustrates this by showing an exception `` `A `` that carries another exception `` `B `` as its payload. The payload `` `B 10 `` itself gets raised by the exception handler for `` `A `` in function `f2`, so a handler for `` `B `` on the call of `f2` suffices to make this fragment exception-safe:

```
fun f1 () = ... raise `A (`B 10) ...
fun f2 () = f1 () handle `A x ⇒ raise x
(... f2 () ...) handle `B i ⇒ ...
```

u

3 The External Language (EL)

We start by describing EL, our implicitly typed *external* language that facilitates sums, cases, and mechanisms for raising as well as handling exceptions.

Syntax of terms

Figure 1 shows the definitions of expressions e and values v. EL is **MLPolyR** [4] extended with constructs for raising and handling exceptions. We have integer constants n, variables x, injection into sum types $l\ e$, applications $e_1\ e_2$, recursive functions **fun** $f\ x = e$, and *let*-bindings **let** $x = e_1$ **in** e_2. For the purpose of comparison, we also include first-class cases $\{\ l_1\ x_1 \Rightarrow e_1, \ldots, l_n\ x_n \Rightarrow e_n\ \}$ and with their elimination form **match** e_1 **with** e_2 as well as case extension $e_1 \oplus \{l\ x \Rightarrow e_2\}$. The additions over **MLPolyR** consist of **raise** e for raising exceptions and several forms for managing exception handlers: The form e_1 **handle** $\{\ l\ x \Rightarrow e_2\ \}$ establishes a handler for the exception constructor l.

The new exception context is used for evaluating e_1, while the old context is used for e_2 in case e_1 raises l. The old context cannot already have a handler for l. The form e_1 **rehandle** $\{\, l\ x \Rightarrow e_2 \,\}$, on the other hand, overrides an existing handler for l. Again, the original exception context is restored before executing e_2. The form e_1 **handle** $\{\, x \Rightarrow e_2 \,\}$ establishes a new context with handlers for *all* exceptions that e_1 might raise. As before, e_2 is evaluated in the original context. The form e **unhandle** l evaluates e in a context from which the handler for l has been removed. The original context must have a handler for l.

To simplify and shorten the presentation, we exclude features that are unrelated to exceptions. Therefore, EL does not have records or recursive types. Adding them back into the language would not cause technical difficulties.

Operational semantics

We give an operational small-step semantics for EL as a context-sensitive rewrite system in a style inspired by Felleisen and Hieb [8]. An *evaluation context* E is essentially a term with one sub-term replaced by a hole (see Figure 2). Any closed expression e that is not a value has a unique decomposition $E[r]$ into an evaluation context E and a redex r that is placed into the hole within E.[2] Evaluation contexts in this style of semantics represent continuations. The rule for handling an exception could be written simply as $E[(E'[\mathbf{raise}\ l\ v])\ \mathbf{handle}\ \{\, l\, x \Rightarrow e\,\}] \mapsto E[e[v/x]]$, but this requires an awkward side-condition stating that E' must not also contain a handler for l. We avoid this difficulty by maintaining the exception context separately and explicitly on a per-constructor basis. This choice makes it clear that exception contexts can be seen as extensible records of continuations. However, we now also need to be explicit about where a computation re-enters the scope of a previous context. This is the purpose of restore-frames of the form $\mathbf{restore}_{\ E_{\mathbf{exn}}}\ E$ that we added to the language, but which are assumed not to occur in source expressions.[3]

An *exception context* $E_{\mathbf{exn}}$ is a record $\{l_1 = E_1, \ldots, l_n = E_n\}$ of evaluation contexts $E_1, \ldots, E_n$ labeled $l_1, \ldots, l_n$. A *reducible configuration* $(E[r], E_{\mathbf{exn}})$ pairs a redex r in context E with a corresponding exception context $E_{\mathbf{exn}}$ that represents all exception handlers that are available when reducing r. A *final configuration* is a pair $(v, \{\})$ where v is a value. Given a reducible configuration $(E[r], E_{\mathbf{exn}})$, we call the pair $(E, E_{\mathbf{exn}})$ the *full context* of r.

The semantics is given as a set of single-step transition rules from reducible configurations to configurations. A program (i.e., a closed expression) e evaluates to a value v if $(e, \{\})$ can be reduced in the transitive closure of our step relation to a final configuration $(v, \{\})$. Rules unrelated to exceptions are standard and leave the exception context unchanged. The rule for **raise** $l\ v$ selects field l of the exception context and places v into its hole.

---

[2] We omit the definition of redexes. All important redexes appear as part of our semantic rules. Non-value expressions that are not covered are stuck.

[3] There are real-world implementations of languages with exception handlers where restore-frames have a concrete manifestation. For example, SML/NJ [1] represents the exception handler as a global variable storing a continuation. When leaving the scope of a handler, this variable gets assigned the previous exception continuation.

$$
\begin{array}{lll}
\text{Terms} & e ::= n \mid x \mid l\ e \mid e_1\ e_2 \mid \textbf{fun}\ f\ x = e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \{\, l_i\ x_i \Rightarrow e_i \,\}_{i=1}^{n} \mid \textbf{match}\ e_1\ \textbf{with}\ e_2 \mid e_1 \oplus \{\, l\ x \Rightarrow e_2 \,\} \mid \\
& \quad \textbf{raise}\ e \mid e_1\ \textbf{handle}\ \{\, l\ x \Rightarrow e_2 \,\} \mid e\ \textbf{unhandle}\ l \mid e_1\ \textbf{rehandle}\ \{\, l\ x \Rightarrow e_2 \,\} \mid e_1\ \textbf{handle}\ \{\, x \Rightarrow e_2 \,\} \\
\text{Values} & v ::= n \mid \textbf{fun}\ f\ x = e \mid l\ v \mid \{\, l_i\ x_i \Rightarrow e_i \,\}_{i=1}^{n} \qquad \text{Kinds}\ \ \kappa ::= \star \mid L \qquad \text{Label sets}\ L ::= \{l_1, \ldots, l_n\} \mid \varnothing \\
\text{Types} & \tau ::= \alpha \mid \mathsf{int} \mid \tau_1 \xrightarrow{\rho} \tau_2 \mid \langle \rho \rangle \mid \langle \rho_1 \rangle \xrightarrow{\rho_2} \tau \qquad \rho ::= \alpha \mid \centerdot \mid l : \tau, \rho \qquad \theta ::= \tau \mid \rho \\
\text{Schemas} & \sigma ::= \tau \mid \forall \alpha : \kappa . \sigma \qquad \text{Typenv}\ \ \Gamma ::= \varnothing \mid \Gamma, x \mapsto \sigma \qquad \text{Kindenv}\ \Delta ::= \varnothing \mid \Delta, \alpha \mapsto \kappa
\end{array}
$$

Fig. 1. External language (EL) syntax.

$$
\begin{array}{l}
e ::= \ldots \mid \textbf{restore} \qquad E_{\textbf{exn}} ::= \{l_1 = E_1, \ldots, l_n = E_n\} \\
E ::= [] \mid l\ E \mid E\ e \mid v\ E \mid \textbf{let}\ x = E\ \textbf{in}\ e \mid E \oplus \{\, l\ x \Rightarrow e \,\} \mid \textbf{match}\ E\ \textbf{with}\ e \mid \textbf{match}\ v\ \textbf{with}\ E \mid \textbf{raise}\ E \mid \textbf{restore}\ {}_{E_{\textbf{exn}}}\ E
\end{array}
$$

Fig. 2. Evaluation contexts and exception contexts

$$
\begin{array}{rll}
(E[(\textbf{fun}\ f\ x = e)\ v], E_{\textbf{exn}}) \longmapsto (E[e[\textbf{fun}\ f\ x = e/f, v/x]], E_{\textbf{exn}}) & & (\textbf{app}) \\
(E[\textbf{let}\ x = v\ \textbf{in}\ e], E_{\textbf{exn}}) \longmapsto (E[e[v/x]], E_{\textbf{exn}}) & & (\textbf{let}) \\
(E[\{\, l_i\ x_i \Rightarrow e'_i \,\}_{i=1}^{n} \oplus \{\, l\ x \Rightarrow e \,\}], E_{\textbf{exn}}) \longmapsto (E[\{\, l_1\ x_1 \Rightarrow e'_1, \ldots, l_n\ x_n \Rightarrow e'_n, l\ x \Rightarrow e \,\}], E_{\textbf{exn}}) & & (\textbf{c/ext}) \\
(E[\textbf{match}\ l_i\ v\ \textbf{with}\ \{\ldots, l_i\ x_i \Rightarrow e_i, \ldots\}], E_{\textbf{exn}}) \longmapsto (E[e_i[v/x_i]], E_{\textbf{exn}}) & & (\textbf{match}) \\
(E[\textbf{raise}\ l_i\ v], \{\ldots, l_i = E_i, \ldots\}) \longmapsto (E_i[v], \{\}) & & (\textbf{raise}) \\
(E[e_1\ \textbf{handle}\ \{\, l\ x \Rightarrow e_2 \,\}], E_{\textbf{exn}}) \longmapsto (E[\textbf{restore}\ {}_{E_{\textbf{exn}}}\ e_1], E'_{\textbf{exn}}) & & (\textbf{handle}) \\
\quad \text{where} \quad E_{\textbf{exn}} = \{\, l_i = E_i \,\}_{i=1}^{n} & & \\
\quad \text{and} \quad E'_{\textbf{exn}} = \{l_1 = E_1, \ldots, l_n = E_n, l = E[\textbf{let}\ x = \textbf{restore}\ {}_{E_{\textbf{exn}}}\ []\ \textbf{in}\ e_2]\} & & \\
(E[e_1\ \textbf{rehandle}\ \{\, l_j\ x \Rightarrow e_2 \,\}], E_{\textbf{exn}}) \longmapsto (E[\textbf{restore}\ {}_{E_{\textbf{exn}}}\ e_1], E'_{\textbf{exn}}) & & (\textbf{rehandle}) \\
\quad \text{where} \quad E_{\textbf{exn}} = \{\, l_i = E_i \,\}_{i=1}^{n} \text{ and } E'_{\textbf{exn}} = \{\, l_i = E'_i \,\}_{i=1}^{n} \text{ and } \forall i \neq j . E'_i = E_i & & \\
\quad \text{and} \quad E'_j = E[\textbf{let}\ x = \textbf{restore}\ {}_{E_{\textbf{exn}}}\ []\ \textbf{in}\ e_2] & & \\
(E[e\ \textbf{unhandle}\ l_j], E_{\textbf{exn}}) \longmapsto (E[\textbf{restore}\ {}_{E_{\textbf{exn}}}\ e], E'_{\textbf{exn}}) & & (\textbf{unhandle}) \\
\quad \text{where} \quad E_{\textbf{exn}} = \{\, l_i = E_i \,\}_{i=1}^{n} \text{ and } E'_{\textbf{exn}} = \{\, l_i = E_i \,\}_{i=1, i \neq j}^{n} & & \\
(E[e_1\ \textbf{handle}\ \{\, x \Rightarrow e_2 \,\}], E_{\textbf{exn}}) \longmapsto (E[\textbf{restore}\ {}_{E_{\textbf{exn}}}\ e_1], E'_{\textbf{exn}}) & & (\textbf{handle all}) \\
\quad \text{where} \quad E'_{\textbf{exn}} = \{\, l_i = E[\textbf{let}\ x = l_i(\textbf{restore}\ {}_{E_{\textbf{exn}}}\ [])\ \textbf{in}\ e_2] \,\}_{i=1}^{n} \text{ (for some n)} & & \\
(E[\textbf{restore}\ {}_{E'_{\textbf{exn}}}\ v], E_{\textbf{exn}}) \longmapsto (E[v], E'_{\textbf{exn}}) & & (\textbf{restore})
\end{array}
$$

Fig. 3. Operational semantics for EL

The result, paired with the empty exception context, is the new configuration which, by construction, will have the form $(E'[\mathbf{restore}\ _{E'_{\mathbf{exn}}}\ v], \{\})$ so that the next step will restore exception context $E'_{\mathbf{exn}}$. The rules for e_1 **handle** $\{\ l\ x \Rightarrow e_2\ \}$ and e_1 **rehandle** $\{\ l\ x \Rightarrow e_2\ \}$ as well as e **unhandle** l are very similar to each other: one adds a new field to the exception context, another replaces an existing field, and the third drops a field. All exception-handling constructs augment the current evaluation context with a **restore**-form so that the original context is re-established if and when e_1 reduces to a value.

The rule for the "handle-all" construct e_1 **handle** $\{\ x \Rightarrow e_2\ \}$ stands out because it is non-deterministic. Since we represent each handled exception constructor separately, the rule must *guess* the relevant set of constructors $\{l_1, \ldots, l_n\}$. Introducing non-determinism here might seem worrisome, but we can justify it by observing that different guesses never lead to different outcomes:

Lemma 1. *If* $(e, \{\}) \mapsto^* (v, \{\})$ *and* $(e, \{\}) \mapsto^* (v', \{\})$*, then* $v = v'$*.*

The proof for this lemma uses a bi-simulation between configurations, where two configurations are related if they are identical up to records. Records may have different sets of labels, but common fields must themselves be related. It is easy to see that each step of the operational semantics preserves this relation.

However, guessing too few or too many labels can get the program stuck. Fortunately, for well-typed programs there always exists a good choice. The correct choice can be made deterministically by taking the result of type inference into account, giving rise to a type soundness theorem for EL. Type soundness is expressed in terms of a well-formedness condition $\vdash (E[e], E_{\mathbf{exn}})\ \mathsf{wf}$ on configurations. For details on its definition and the proof of soundness (progress and preservation) see the technical report [5]. Since uncaught exceptions are simply stuck configurations, the soundness theorem justifies our motto: *Well-typed* EL *programs do not have uncaught exceptions.*

Well-formed types

The type language for EL is also given in Figure 1. It contains type variables $(\alpha, \beta, \ldots)$, base types (e.g., int), constructors for function- and case types ($\rightarrow$ and $\hookrightarrow$), sum types ($\langle\rho\rangle$), the empty row type ($\mathbf{.}$), and row types with at least one typed label $(l : \tau, \rho)$. Notice that function- and case arrows take *three* type arguments: the domain, the co-domain, and a row type describing the exceptions that could be raised during an invocation. A type is either an ordinary type or a row type. Kinding judgments of the form $\Delta \vdash \tau : \kappa$ (stating that in the current kinding context Δ type τ has kind κ) are used to distinguish between these cases and to establish that types are well-formed. As a convention, wherever possible we will use meta-variables such as ρ for row types and τ for ordinary types. Where this distinction is not needed, for example for polymorphic instantiation (VAR), we will use the letter θ.

Ordinary types have kind $\star$. A row type ρ has kind L where L is a set of labels which are known not to occur in ρ. An unconstrained row variable has kind $\varnothing$. Inference rules can be found in the technical report [5]. The use of a kinding judgment in a typing rule constrains Δ and ultimately propagates

kinding information back to the LET/VAL rule where type variables are bound and kinding information is used to form type schemas.

Typing

The type τ of a closed expression e characterizes the values that e can evaluate to. From a dual point of view it describes the values that the evaluation context E must be able to receive. In our operational semantics E is extended to a full context $(E, E_{\mathbf{exn}})$, so the goal is to develop a type system with judgments that describe the full context of a given expression. Our typing judgments have an additional component ρ that describes $E_{\mathbf{exn}}$ by individually characterizing the its constituent labels and evaluation contexts. General typing judgments have the form $\Delta; \Gamma \vdash e : \tau; \rho$, expressing that e has type τ and exception type ρ. The typing environment Γ is a finite map assigning types to the free variables of e. Similarly, the kinding environment Δ maps the free type variables of τ, ρ, and Γ to their kinds.

The **typing rules** for EL are given in Figure 4. Typing is syntax-directed; for most syntactic constructs there is precisely one rule, the only exceptions being the rules for **fun** and **let** which rely on the notion of *syntactic values* to distinguish between two sub-cases. As usual, in rules that introduce polymorphism we impose the *value restriction* by requiring certain expressions to be *valuable*. Valuable expressions do not have effects and, in particular, do not raise exceptions. We use a separate typing judgment of the form $\Delta; \Gamma \vdash_{\mathsf{v}} e : \tau$ for syntactic values (VAR, INT, FUN/VAL, FUN/NON-VAL, and C). Judgments for syntactic values are lifted to the level of judgments for general expressions by the VALUE rule. The VALUE rule leaves the exception type ρ unconstrained. Administrative rules TEQ and TEQ/V deal with type equivalences $\tau \approx \tau'$, especially the reordering of labels in row types. Rules for $\tau \approx \tau'$ were described in previous work [4].

Rules unrelated to exceptions simply propagate a single exception type without change. This is true even for expressions that have more than one sub-term, matching our intuition that the exception type characterizes the exception context. For example, consider function application $e\ e'$: The rules do not use any form of sub-typing to express that the set of exceptions is the union of the three sets corresponding to e, e', and the actual application. Like Pessaux and Leroy we rely on polymorphism to collect exception information across multiple subterms. As usual, polymorphism is introduced by the LET/VAL rule for expressions **let** $x = e_1$ **in** e_2 where e_1 is a syntactic value.

The rules for handling and raising exceptions establish bridges between ordinary types and handler types (i.e., types of exception handler contexts). Exceptions themselves are simply values of sum type; the **raise** expression passes such values to an appropriate handler. Notice that the corresponding rule equates the row type of the sum with the row type of the exception context; there is no implicit subsumption here. Instead, subsumption takes place where the exception payload is injected into the corresponding sum type (DCON).

Rule HANDLE-ALL is the inverse of RAISE. The form e_1 **handle** $\{\, x \Rightarrow e_2 \,\}$ establishes a handler that catches *any* exception emanating from e_1. The exception is made available to e_2 as a value of sum type bound to variable x.

Operationally this corresponds to replacing the current exception handler context with a brand-new one, tailor-made to fit the needs of e_1. The other three constructs do not replace the exception handler context wholesale but adjust it incrementally: **handle** adds a new field to the context while retaining all other fields; **rehandle** replaces an existing handler at a specific label l with a new (potentially differently typed) handler at the same l; **unhandle** removes an existing handler. There are strong parallels between C/EXT (case extension) and HANDLE, although there are also some significant differences due to the fact that exception handlers constitute a hidden part of the context while cases are first-class values. As hinted in Section 4 and explained fully in the technical report [5], we can highlight the connection and de-emphasize the differences by translating both handlers and cases into the same representation, namely records of functions. The value-level counterpart to **rehandle** is functional case update, which we omitted for reasons of brevity. Similarly, **unhandle** corresponds to a form for narrowing cases or, dually, a form for widening a sums (not are shown here).

Whole programs are closed up to some initial basis environment Γ_0, raise no exceptions, and evaluate to int. This is expressed by a judgment $\Gamma_0 \vdash e$ program.

Polymorphic recursion vs. explicit narrowing

Pessaux and Leroy explain that their exception analysis becomes more precise if they employ inference for polymorphic recursion. Since the problem of type inference in the presence of polymorphic recursion is generally undecidable [12], we chose not to base our language design on this idea. To see the problem, consider a scenario similar to the one described by Pessaux and Leroy:

```
fun f x = (if (... raise `C() ...) then () else f x) handle `C() =>()
```

Here `f` is called in a context that requires a handler for exception `` `C ``, because `` `C `` is raised in a different sub-expression. However, exception `` `C `` will always be caught, and it would be nice to have this fact expressed in `f`'s type. Since the Pessaux/Leroy system uses *presence types* (see Section 5), it can represent and (in some cases) infer this fact. But doing so requires type inference with polymorphic recursion because `f`'s body has to be type-checked under the assumption of `f` being exception-free. Exception freedom is expressed via polymorphism.

In our language, the above example would be illegal even in the presence of polymorphic recursion. The body of `f` must be checked assuming that `f`'s exception type can be instantiated with a row containing `` `C ``. But the use of ... **handle** `` `C() `` $\Rightarrow$ ... within the same body ultimately invalidates this assumption. The problem can easily be understood in terms of the operational semantics: every recursive call of `f` wants to add another field `` `C `` to the exception context. Since fields can only be added if they are not already present, this is impossible.

Fortunately, the programmer can work around such problems by explicitly removing `` `C `` from the exception context at the site of the recursive call:

```
fun f x = (if (... raise `C() ...) then () else (f x unhandle `C))
          handle `C() =>()
```

In the concrete language design implemented by our compiler we adopt an idea of Benton and Kennedy [3] and provide a variant of the **handle**-syntax that in some cases avoids the need for **unhandle**, since it provides an explicit success branch that is evaluated under the original context:

```
fun f x = let val tmp = ... raise `C() ...
          handling `C() ⇒()
          in if tmp then () else f x
```

Polymorphic recursion and curried functions

There is one situation, however, where a limited form of type inference for polymorphic recursion is very helpful: curried functions, or—more generally—functions whose bodies are syntactic values. Our type system has a separate rule (FUN/VAL) that types the body of the function assuming unrestricted polymorphism in its exception type. Since the body is a syntactic value, this assumption is guaranteed to be valid and leads to a decidable inference problem. The point of including the rule is improved precision. Consider our `map` example from Section 2. In the absence of FUN/VAL, i.e., if FUN/NON-VAL were to be used for all functions regardless of whether or not their bodies are syntactic values, we would infer the type of `map` (using Haskell-style notation for lists types $[\tau]$) as:

$$\textbf{val map} \ : \ \forall\alpha:\star.\forall\beta:\star.\forall\gamma:\varnothing.(\alpha \xrightarrow{\gamma} \beta) \xrightarrow{\gamma} ([\alpha] \xrightarrow{\gamma} [\beta])$$

Notice the use of the same row type variable γ on all three occurrences of $\rightarrow$. The type checker fails to notice that exceptions suspended within the first argument cannot be raised during the first stage (i.e., a partial application) of `map`. This violates one of the requirements that we spelled out when we motivated the need for an exception type system. The fix is the introduction of FUN/VAL. Consider `map` once again. Since it is curried, its body—when rendered in EL—is another **fun** expression, i.e., a syntactic value. Without having performed any type inference at all we know that `map` will not raise an exception at the time of a partial application, so we can use this as an assumption when type-checking the body. To express that a function cannot raise an exception we make its type polymorphic in the exception annotation on $\rightarrow$. Indeed, FUN/VAL binds f to $\forall\alpha:\varnothing.\tau_2 \xrightarrow{\alpha} \tau$ in the typing environment for the function body. With this rule in place, `map`'s type is now inferred as we had hoped[4]:

$$\textbf{val map} \ : \ \forall\alpha:\star.\forall\beta:\star.\forall\gamma:\varnothing.\forall\delta:\varnothing.(\alpha \xrightarrow{\gamma} \beta) \xrightarrow{\delta} ([\alpha] \xrightarrow{\gamma} [\beta])$$

4 CPS and Duality: The Internal Language (IL)

In previous work we used a type-directed translation of **MLPolyR** into a variant of System F with extensible polymorphic records. Sum types and extensible cases were eliminated by taking advantage of duality [4]. We then used an adaptation of Ohori's technique of compiling record polymorphism by passing sets of *indices* which serve as witnesses for row types [17].

[4] As hinted in footnote 1, our compiler prints this type as $(\alpha \xrightarrow{\gamma} \beta) \rightarrow ([\alpha] \xrightarrow{\gamma} [\beta])$ since all elided parts (including δ) can be inferred from suitably chosen conventions.

$$\frac{\Gamma(x) = \forall\alpha_1 : \kappa_1 \ldots \forall\alpha_n : \kappa_n.\tau \qquad \forall i.\Delta \vdash \theta_i : \kappa_i}{\Delta; \Gamma \vdash_{\mathsf{v}} x : \tau[\theta_1/\alpha_1, \ldots, \theta_n/\alpha_n]}(\textsc{var}) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathsf{v}} n : \mathsf{int}}(\textsc{int}) \qquad \frac{\forall i.\Delta; \Gamma, x_i \mapsto \tau_i \vdash e_i : \tau; \rho \qquad \Delta \vdash (l_1 : \tau_1, \ldots, l_n : \tau_n, \bullet) : \varnothing}{\Delta; \Gamma \vdash_{\mathsf{v}} \{\, l_i\ x_i \Rightarrow e_i \,\}_{i=1}^{n} : \langle\, l_i : \tau_i \,\rangle_{i=1}^{n} \overset{\rho}{\hookrightarrow} \tau}(\textsc{c})$$

$$\frac{\Delta; \Gamma, f \mapsto (\forall\alpha : \varnothing.\tau_2 \overset{\alpha}{\to} \tau), x \mapsto \tau_2 \vdash_{\mathsf{v}} e : \tau \qquad \Delta \vdash \tau_2 : \star \qquad \Delta \vdash \rho : \varnothing}{\Delta; \Gamma \vdash_{\mathsf{v}} \mathbf{fun}\ f\ x = e : \tau_2 \overset{\rho}{\to} \tau}(\textsc{fun/val}) \qquad \frac{\Delta; \Gamma, f \mapsto \tau_2 \overset{\rho}{\to} \tau, x \mapsto \tau_2 \vdash e : \tau; \rho \qquad \Delta \vdash \tau_2 : \star \qquad \Delta \vdash \rho : \varnothing}{\Delta; \Gamma \vdash_{\mathsf{v}} \mathbf{fun}\ f\ x = e : \tau_2 \overset{\rho}{\to} \tau}(\textsc{fun/non-val})$$

$$\frac{\Delta : \Gamma \vdash_{\mathsf{v}} e : \tau \quad \tau \approx \tau'}{\Delta; \Gamma \vdash_{\mathsf{v}} e : \tau'}(\textsc{teq/v}) \qquad \frac{\Delta; \Gamma \vdash e : \tau; \rho \quad \tau \approx \tau' \quad \rho \approx \rho'}{\Delta; \Gamma \vdash e : \tau'; \rho'}(\textsc{teq}) \qquad \frac{\Delta; \Gamma \vdash_{\mathsf{v}} e : \tau \quad \Delta \vdash \rho : \varnothing}{\Delta; \Gamma \vdash e : \tau; \rho}(\textsc{value})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \overset{\rho}{\to} \tau; \rho \qquad \Delta; \Gamma \vdash e_2 : \tau_2; \rho}{\Delta; \Gamma \vdash e_1\ e_2 : \tau; \rho}(\textsc{app}) \qquad \frac{\alpha_1, \ldots, \alpha_n = \mathrm{FTV}(\tau_1) \setminus \mathrm{FTV}(\Gamma) \quad \Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n; \Gamma \vdash_{\mathsf{v}} e_1 : \tau_1 \qquad \Delta; \Gamma, x \mapsto \forall\alpha_1 : \kappa_1. \ldots \forall\alpha_n : \kappa_n.\tau_1 \vdash e_2 : \tau_2; \rho}{\Delta; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2; \rho}(\textsc{let/val})$$

$$\frac{\Delta; \Gamma \vdash e : \tau; \rho' \qquad \Delta \vdash (l : \tau, \rho) : \varnothing}{\Delta; \Gamma \vdash l\ e : \langle l : \tau, \rho \rangle; \rho'}(\textsc{dcon}) \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_1; \rho \qquad \Delta; \Gamma, x \mapsto \tau_1 \vdash e_2 : \tau_2; \rho}{\Delta; \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2; \rho}(\textsc{let/non-val})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \langle \rho \rangle; \rho' \qquad \Delta; \Gamma \vdash e_2 : \langle \rho \rangle \overset{\rho'}{\hookrightarrow} \tau; \rho'}{\Delta; \Gamma \vdash \mathbf{match}\ e_1\ \mathbf{with}\ e_2 : \tau; \rho'}(\textsc{match}) \qquad \frac{\Delta; \Gamma \vdash e : \langle \rho \rangle; \rho \qquad \Delta \vdash \tau : \star}{\Delta; \Gamma \vdash \mathbf{raise}\ e : \tau; \rho}(\textsc{raise})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \langle \rho_1 \rangle \overset{\rho}{\hookrightarrow} \tau; \rho' \qquad \Delta \vdash (l : \tau_1, \rho_1) : \varnothing \qquad \Delta; \Gamma, x \mapsto \tau_1 \vdash e_2 : \tau; \rho}{\Delta; \Gamma \vdash e_1\ \oplus\ \{\, l\ x \Rightarrow e_2 \,\} : \langle l : \tau_1, \rho_1 \rangle \overset{\rho}{\hookrightarrow} \tau; \rho'}(\textsc{c/ext}) \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau; l : \tau', \rho \quad \Delta; \Gamma, x \mapsto \tau' \vdash e_2 : \tau; \rho}{\Delta; \Gamma \vdash e_1\ \mathbf{handle}\ \{\, l\ x \Rightarrow e_2 \,\} : \tau; \rho}(\textsc{handle})$$

$$\frac{\Delta; \Gamma \vdash e : \tau; \rho \quad \Delta \vdash (l : \tau', \rho) : \varnothing}{\Delta; \Gamma \vdash e\ \mathbf{unhandle}\ l : \tau; l : \tau', \rho}(\textsc{unhandle}) \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau; l : \tau', \rho \quad \Delta; \Gamma, x \mapsto \tau' \vdash e_2 : \tau; l : \tau'', \rho}{\Delta; \Gamma \vdash e_1\ \mathbf{rehandle}\ \{\, l\ x \Rightarrow e_2 \,\} : \tau; l : \tau'', \rho}(\textsc{rehandle})$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau; \rho' \quad \Delta; \Gamma, x \mapsto \langle \rho' \rangle \vdash e_2 : \tau; \rho}{\Delta; \Gamma \vdash e_1\ \mathbf{handle}\ \{\, x \Rightarrow e_2 \,\} : \tau; \rho}(\textsc{handle-all}) \qquad \boxed{\frac{\varnothing; \Gamma_0 \vdash e : \mathsf{int}; \bullet}{\Gamma_0 \vdash e\ \mathsf{program}}(\textsc{program})}$$

Fig. 4. Typing rules for EL for syntactic values (top), type equivalence and lifting (2nd), basic computations (3rd), and computations involving cases or exceptions (bottom). The judgment for whole programs is shown in the framed box.

It is possible to retain this general approach to compilation even in the presence of exceptions. For this we combine the original dual transformation—which eliminates sums and cases—with continuation-passing style (CPS). While a "double-barreled" CPS makes continuations explicit and represents exception handlers as alternative continuations, we take this idea one step further and split the exception context into a record of possibly many individual continuations—one for each exception constructor that is being handled. Such a "multi-barreled" CPS provides a manifest explanation for the claim contained in the title of this paper: after CPS- and dual translation both exception handlers and extensible cases are represented identically, namely as records of functions.

The type-sensitive translation from EL into the System F-like intermediate language IL is given as rules for an extended typing judgment. Such a judgment has the form:

$$\Delta; \Gamma \vdash e : \tau; \rho \rightsquigarrow \bar{c} : (\bar{\tau}, \bar{\rho})\ \mathsf{comp}$$

Here Δ, Γ, e, τ, and ρ are EL-level environments, terms, and types which appear exactly as they do in Figure 4. The IL-term $\bar{c}$ is the result of the translation. Its type is that of a *computation* that either sends a result of type $\bar{\tau}$ (corresponding to τ) to its default continuation or invokes one of the handler continuations described by $\bar{\rho}$ (corresponding to ρ). The notation $(\bar{\tau}, \bar{\rho})\ \mathsf{comp}$ is a type synonym for $\bar{\tau}\ \mathsf{cont} \to \bar{\rho}\ \mathsf{hdlr} \to \mathsf{ans}$, with $\bar{\tau}\ \mathsf{cont}$ and $\bar{\rho}\ \mathsf{hdlr}$ themselves being further type synonyms. The details of IL and the translation of EL-types to IL-types as well as all rules for deriving translation judgments are given in the extended technical report accompanying this paper [5].

Eliminating non-determinism—reify

Recall that the **handle all** rule of the EL semantics had to guess the new exception record to be constructed. But guessing correctly is not that difficult: the constructed record merely needs to match the exception (row-) type inferred at this point. To provide runtime access to the necessary type information, we equipped IL with a special type-sensitive **reify** construct, whose role is to convert functions on (dually encoded) sums to their corresponding records of individual functions.

Since **reify** is type-sensitive, type-erasure cannot be applied to IL. This is not a problem in practice, since subsequent compilation into lower-level untyped languages via index-passing in the style of our previous work on **MLPolyR** [4] is still possible. Details can be found in our technical report [5].

The translation from EL into IL amounts to an alternative elaboration-based semantics for EL. IL is sound and the translation maps well-typed EL programs to well-typed IL programs. Since an uncaught EL exception would try to select from the top-level exception handler, which by definition is an empty record, this constitutes a proof for our motto that "well-typed programs do not have uncaught exceptions" in the setting of this alternative semantics.

5 Related work

We know of no prior work that incorporates higher-order exception tracking into the design of a programming language and its type system. Previous work in this

$$\rho ::= \alpha \mid \centerdot \mid l : \pi, \rho \qquad \pi ::= \alpha \mid \mathfrak{p}(\tau) \mid \mathfrak{a} \qquad \theta ::= \tau \mid \rho \mid \pi \qquad \kappa ::= \star \mid L \mid \circ$$

Fig. 5. Alternative type language with presence types (for τ see Figure 1)

context focused on *analysis* for identifying uncaught exceptions without attempting to incorporate such analysis into the language design itself [11,19,18,7,14,2]. While most of these attempts use flow analysis for exception tracking, some are based on types. In particular, this is true for the analysis by Pessaux and Leroy [14]. Much like we do, they also encode exception types as row types and employ row polymorphism.

Since the technical details of the Pessaux-Leroy system are similar to our work, we will now discuss the differences in some detail.

5.1 The Pessaux-Leroy type system

Pessaux and Leroy [14] describe an exception analysis based on a type system that is very similar to ours. However, there are a number of key differences which reflect the difference in purpose: The Pessaux-Leroy system is an exception *analyzer*, i.e., a separate tool that is used to fine-comb programs written in an existing language—in their case Ocaml [13]. In contrast, our exploration of the language design space itself leads us to different trade-offs. Perhaps most importantly, our type system is simpler but forces the programmer to choose among a variety of exception-handling constructs (**handle**, **rehandle**, **unhandle**).

In most dialects of ML, including Ocaml, there is only one **handle** construct which—depending on the current exception context—can act either like our **handle** or like our **rehandle**. To deal with this form of exception-handling, the Pessaux-Leroy type system uses a third kind of type called a *presence type*. In row types, labels are associated with presence types π rather than ordinary types. There are two forms of presence types: $\mathfrak{a}$ indicates that a field is absent; $\mathfrak{p}(\tau)$ expresses that the field is present and carries values of type τ. The key to the added expressive power of presence types is the fact that they can also be represented by type variables of *presence kind* ($\circ$), opening the possibility of functions being polymorphic in the presence of explicitly named fields.

Figure 5 shows the modified type language. Notice that this is an idealized version. Pessaux and Leroy use presence types only for nullary exception constructors (i.e., exceptions without "payload"). This is done in an attempt at dealing with *nested patterns*, a feature that is also absent from our language.[5] The new expression language is the same as the old one, except that we no longer need **rehandle** or **unhandle**.

New or modified typing rules are given in Figure 6. (For modified kinding see the technical report [5].) The addition of presence types results in a considerably more involved equational theory on types, since (assuming well-formedness) both ρ and $l : \mathfrak{a}, \rho$ are the same type. See rule C in Figure 6 for an example of

[5] For some programs, not using presence types for non-constant constructors reduces the precision of the analysis. The fact that Pessaux and Leroy do not report on such a loss probably indicates that such exception constructors are rare in practice.

$$\frac{\Delta;\Gamma \vdash e : \tau;\rho' \quad \Delta \vdash (l : \mathfrak{p}(\tau),\rho) : \varnothing}{\Delta;\Gamma \vdash l\ e : \langle l : \mathfrak{p}(\tau),\rho\rangle;\rho'}(\textsc{dcon}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau; l : \mathfrak{p}(\tau'),\rho \quad \Delta;\Gamma, x : \tau' \vdash e_2 : \tau;\rho \quad \Delta \vdash \pi : \circ}{\Delta;\Gamma \vdash e_1\ \mathbf{handle}\ \{\, l\ x \Rightarrow e_2\,\} : \tau; l : \pi,\rho}(\textsc{handle})$$

$$\frac{\Delta \vdash (l_1 : \mathfrak{p}(\tau_1),\ldots,l_n : \mathfrak{p}(\tau_n),\centerdot) : \varnothing \qquad \forall i.\Delta;\Gamma, x_i : \tau_i \vdash e_i : \tau;\rho}{\Delta;\Gamma \vdash_{\mathsf{v}} \{\, l_i\ x_i \Rightarrow e_i\,\}_{i=1\ldots n} : \langle\, l_i : \mathfrak{p}(\tau_i), l_j : \mathfrak{a}\,\rangle_{i=1\ldots n, j=n+1\ldots m} \xhookrightarrow{\rho} \tau}(\textsc{c})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \langle l : \pi,\rho_1\rangle \xhookrightarrow{\rho} \tau;\rho' \quad \Delta \vdash (l : \mathfrak{p}(\tau_1),\rho_1) : \varnothing \qquad \Delta;\Gamma, x : \tau_1 \vdash e_2 : \tau;\rho}{\Delta;\Gamma \vdash e_1\ \oplus\ \{\, l\ x \Rightarrow e_2\,\} : \langle l : \mathfrak{p}(\tau_1),\rho_1\rangle \xhookrightarrow{\rho} \tau;\rho'}(\textsc{c/ext})$$

Fig. 6. Alternative typing rules for EL with presence types of kind $\circ$

how this formally manifests itself in the type system. To see the difference in expressiveness between the two systems—stemming from the fact that rule C/EXT can both extend and override cases—consider a function definition such as:

$$\mathbf{fun}\ \mathtt{f}\ \mathtt{c} = c\ \oplus\ \{\text{`}A\ x \Rightarrow x+1\}$$

In our type system, the most general type schema inferred for `f` is:

$$\mathbf{val}\ \mathtt{f}\ :\ \forall\alpha : \{\text{`}A\}.\forall\gamma : \varnothing.\forall\delta : \varnothing.(\langle\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int}) \xrightarrow{\delta} (\langle\text{`}A : \mathsf{int},\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int})$$

This type restricts the argument to cases that do not already handle the `'A` constructor. With presence types, the inferred schema is:

$$\mathbf{val}\ \mathtt{f}\ :\ \forall\alpha : \{\text{`}A\}.\forall\beta : \circ.\forall\gamma : \varnothing.\forall\delta : \varnothing.(\langle\text{`}A : \beta,\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int}) \xrightarrow{\delta} (\langle\text{`}A : \mathfrak{p}(\mathsf{int}),\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int})$$

It does not restrict the argument, since all labels—including `'A`—are permitted to be present or absent, even though the type "talks" about label `'A` separately.

A low-level implementation technique based on index-passing is still possible. Presence type variables that in the IL are bound by a type abstraction turn into Boolean witness parameters. The calculation of indices itself becomes more complicated, since presence witnesses have to be taken into account.

The improved expressiveness of a type system with presence types comes at the expense of added conceptual complexity. Such complexity is perfectly acceptable when it is used internally by an analysis tool, but it may be undesirable in a type system that programmers confront directly in everyday use of the language.

A more serious problem with presence types is the loss of a certain degree of functional abstraction. Let `h` be a polymorphic function of type $\forall\alpha : \varnothing.\forall\gamma : \varnothing.(\langle\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int}) \xrightarrow{\gamma} \mathsf{int}$. Now consider the following two definitions:

$$\begin{array}{l}\mathbf{fun}\ \mathtt{f}\ \mathtt{c} = \mathtt{h}\ \mathtt{c}\\ \mathbf{fun}\ \mathtt{g}\ \mathtt{c} = \mathtt{h}\ (c\ \oplus\ \{\text{`}A\ x \Rightarrow x+1\})\end{array}$$

Using presence types, the inferred signatures are:

$$\begin{array}{l}\mathbf{val}\ \mathtt{f}\ :\ \forall\alpha : \varnothing.\forall\gamma : \varnothing.(\langle\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int}) \xrightarrow{\gamma} \mathsf{int}\\ \mathbf{val}\ \mathtt{g}\ :\ \forall\alpha : \{\text{`}A\}.\forall\beta : \circ.\forall\gamma : \varnothing.(\langle\text{`}A : \beta,\alpha\rangle \xhookrightarrow{\gamma} \mathsf{int}) \xrightarrow{\gamma} \mathsf{int}\end{array}$$

The type schemas for `f` and `g` look very different, yet their respective sets of instantiations coincide precisely. In traditional denotational models of types they *are* the same. Should we not consider the two schemas equivalent then? The

obvious difference between them seems to be just an artifact of type inference. It arises due to the absence of case extension in `f`, while case extension is used in `g`. Clearly, an *implementation detail* has leaked into the inferred signature.

An obvious idea is to try and "normalize" `g`'s type to make it equal to that of `f`. However, doing so would break index-passing compilation! Since `g` performs functional extension of a case, it needs to know the index of label `'A`. But the index-passing transformation—when working with the signature of `f`—would not generate such an index argument. Thus, not only has an implementation detail leaked into the signature, this leakage is actually essential to the compilation technology. Under the rules that we use for EL, function `g` is typed as:

$$\textbf{val}\ \texttt{g}\ :\ \forall\alpha : \{`A\}.\forall\gamma : \varnothing.(\langle\alpha\rangle \overset{\gamma}{\hookrightarrow} \mathsf{int}) \overset{\gamma}{\to} \mathsf{int}$$

Here `g`'s type is not the same as that of `f`—neither syntactically nor denotationally. Indeed, `f` and `g` have different interfaces, since the latter does not accept arguments that already handle `'A`. While this seems like a restriction on where `g` can be used, there is always an easy workaround when this becomes a problem: the programmer can explicitly narrow the intended argument by removing `'A` at the call site. Applied to the problem of handling exceptions, the same reasoning inspired us to the inclusion of **rehandle** and **unhandle** in the language.

Forcing the programmer to be explicit about such things can be considered both burden and benefit. By being explicit about widening or narrowing, a program documents more clearly what is going on. We hope that practical experience with the language will tell whether or not this outweighs the disadvantages.

6 Conclusions

We have shown the design of a higher-order programming language that guarantees freedom from uncaught exceptions. Using a type system—as opposed to flow- or constraint-based approaches—for tracking exceptions provides a straightforward path for integration into a language design. Moreover, our type system is simpler than the one used by Pessaux and Leroy for exception analysis, because we are able to avoid presence types by trading them for language constructs that explicitly manipulate the shape of the exception handler context.

Our language is sound, and soundness includes freedom from uncaught exceptions. We formalized our language and also provided an elaboration semantics from the implicitly typed external language EL into an explicitly typed internal form IL that is based on System F. The elaboration performs CPS transformation as well as dual translation, eliminating exceptions, handlers, sums, as well as first-class cases. The translation supports our intuition that the act of establishing a new exception handler is closely related to the one of extending first-class cases. The latter had recently been described in the context of our work on **MLPolyR** [4]. Our new IL is almost the same as the original internal language, a fact that enabled us to reuse much of the existing implementation machinery. The only major addition to our IL is a type-sensitive **reify** construct that accounts for "catch-all" exception handlers.

We have adapted the existing index-passing translations into low-level code to work with our system. Our prototype compiler translates a concrete language

modeled after EL into PowerPC machine code. It shares most of its type inference engine with the previous compiler for **MLPolyR**.

Index-passing, while providing motivation for this work, is not the driving force behind the actual language design. Instead, our distinction between **handle** and **rehandle** keeps the type system simpler and may have certain software-engineering benefits. The same considerations would apply even if the target language provided native support for, e.g., extensible sums. As we have explained in Section 5, based on the more complicated Pessaux-Leroy type system, witness-passing can also be used for implementing an ML-like dual-purpose **handle**-construct.

References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, August 1991.
2. Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI '07*, pages 15–26, New York, NY, USA, 2007. ACM.
3. Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001.
4. Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 2006. ACM.
5. Matthias Blume, Umut A. Acar, and Wonseok Chae. Exception handlers as extensible cases. U.Chicago, Computer Sci. Tech. Report TR-2008-03, February 2008.
6. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report Research Report 31, DEC SRC, 1988.
7. M. Fähndrich, J. Foster, J. Cu, and A. Aiken. Tracking down exceptions in Standard ML programs. Technical Report CSD-98-996, UC Berkeley, February 1998.
8. Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
9. Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library.* Cambridge University Press, 2002.
10. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series.* Addison Wesley, 2000.
11. J. Guzmán and A. Suárez. An extended type system for exceptions. In *ACM SIGPLAN Workshop on ML and its Applications*, 1994.
12. Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.
13. X. Leroy. *The Objective Caml System.* `http://caml.inria.fr/ocaml`, 1996.
14. Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.
15. J. L. Lions. Ariane 5, flight 501 failure, report by the inquiry board. `http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html`, 1996.
16. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* SPRINGER, New York, NY, 1981.
17. Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.
18. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Sci. Comput. Program.*, 31(1), 1998.
19. Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in sml programs. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 98–113, London, UK, 1997. Springer-Verlag.