An Extended Type System for Exceptions*

Juan Carlos Guzmán and Ascánder Suárez
Universidad Simón Bolívar – Departamento de Computación
Valle de Sartenejas, Caracas 1080, Venezuela
Email: {jcguzman,suarez}@usb.ve

Abstract

We present in this paper an extension to the ML type system by which it is possible to statically estimate all untrapped exceptions that can be raised by executing a program. This type system can handle polymorphic information on exceptions. A principal extended type exists and can be computed for any well-typed expression.

1 Introduction

Several programming languages such as ML[12, 15, 9] facilitate the handling of exceptional computation such as reading past end-of-file, dividing by zero, or taking the head of an empty list. The handling of such events can be more convenient, and more modular by using the raise and try constructs provided by the language. Rather than servicing the exceptional event where it happens (e.g., division by zero) or introducing code to avoid the execution of such an event, the ML user can escape from computations by raising exceptions which can be caught in a dynamically enclosing computation.

However, ML does not provide any mechanism for controlling the universe of possible exceptions that can happen, and manually keeping track of this at a given program point can prove to be tedious for first-order computations, and difficult and error prone for the higher-order case. An uncaught exception aborts the whole computation and is usually a result of a programming bug (failure to catch that exception).

Modula 3, another language that facilitates the handling of exceptional events accepts annotations in function type signatures for the purpose of identifying the untrusted exceptions within the code segment, but it does so without any verification of the validity of these annotations.

We propose in this paper a method for statically determining the uncaught exceptions for ML expressions. This is done as an extension to the Hindley-Milner type system of the language. Intuitively, this extension consists on annotating the functional type constructor (→) with the set of possible exceptions that may be raised by applying this function. Also, each expression is tagged with the set of exceptions it may raise when evaluated.

The division function raises the “division by zero” exception (/) if the divisor is zero. The type of the curried version would be:

\[
/ : \langle \text{num} \rightarrow \text{num} \cup_{\text{ } \cup} \text{num}, \emptyset \rangle
\]

being its type the first component of the tuple, and its “exceptions when evaluated” the second component—evaluating the division function cannot cause any exception to be raised.

Partially applying the function (/ 1) still cannot raise any exception. This fact is reflected in its type:

\[
\langle 1 \rangle : \langle \text{num} \cup_{\text{ } \cup} \text{num}, \emptyset \rangle
\]

However, fully applying the function may raise the “division by zero” exception:

\[
\langle / 1 0 \rangle : \langle \text{num}, \emptyset \rangle
\]

If that expression is enclosed within another that captures the “division-by-zero” exception, then the resulting value has no exception:

\[
\text{try} \langle / 1 0 \rangle \text{ with } / \rightarrow 0 : \langle \text{num}, \emptyset \rangle
\]

We follow the approach used in in [6], and further developed in [5] for the purpose of controlling mutations of state in single-threaded computations. Also, [7] uses this method to perform a liveliness analysis of functional programs. Following their convention, we will call use, or use to the annotations to the functional constructor, and liveness to the exceptions possibly raised by the evaluation of expressions.

The proposed technique has the following features:

1. simple—just annotations to types,
2. intuitive—using types as information carriers makes it easy to comprehend the scope of the annotations,
3. reasonably precise for first-, as well as for higher-order—is as precise as it can be for an analysis that does not have access to expression values,
4. polymorphic on types and annotations—not all the uses of a variable need to have the same annotations—and
5. completely reconstructible—the extended type inferencer can reconstruct the most general extended type without requiring any additional information from the user.

*In the Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida, June 25-26, 1994. INRIA, Rapport de Recherche No. 2065 BP 105 - 78153 Le Chesnay Cedex, France
For presentation purposes, we introduce our technique to a representative subset of the ML language. However, the reader should be assured that the extension to the full language is straightforward.

In Section 2 we establish the working language as well as its type language. Section 3 presents the inference rules for the extended type system. In Section 4 we provide several examples of the extended types. In Section 5, we discuss pragmatic issues concerning this type system. In Section 6 we explain how type reconstruction is done. Section 7 surveys other work on the area. Finally, a proof of soundness of the type system can be found in the Appendix.

2 Preliminaries

In this paper we limit ourselves to a representative subset of the ML language. Each construct included shows a specific feature of the model. We have included the traditional λ-calculus constructs, as well as let-expressions, a conditional, a fix point operator, and mechanisms for raising, and capturing exceptions. The syntax for language follows:

**Definition 1 (The Language)** Let $V$ and $\Sigma$ be a sets of identifiers used as variable names and exception names respectively and $C$ be a set of constants. The set $E$ of expressions of the language is defined as:

$$E = x \mid c \mid e_1 e_2 \mid \lambda x.e \mid \text{let } x = c_1 \text{ in } e_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{fix } x = e \mid \text{raise } e \mid \text{try } e_1 \text{ with } e \rightarrow e_2$$

where $x \in V$, $c \in C$, $e_1, e_2 \in E$, $\lambda x.e \in E$, $x \in V$, $e \in E$, $b \in \{\text{true}, \text{false}\}$, $c_1, c_2 \in E$, $x \in V$, $e \in E$, $\Sigma$ is the set of exception names.

We can safely assume the domains of variable identifiers and exception names to be disjoint. Also, there is no provision for "exception variables". All these decisions are present in ML, and are based on pragmatic considerations.

In well-typed ML programs, each expression can be assigned a type in such a way that the type of the whole program can be proved from the assertions about the type of all its constituent expressions. In this paper we extend the notion of type to include the set of all possible exceptions that the program may raise. In order to consider higher-order expressions, we allow parts of the exception set to be unknown—hence the introduction of exception set variables. The operations on sets will be union (noted as "$+$") and set difference (noted as "$-$"). Union is used to combine the ability to raise exceptions of subexpressions, while difference is used to note that an exception has been handled.

**Definition 2 (Exception Uses)** Let $UV$ be a set of identifiers which will be used as variables. The set $U$ of use expressions is defined as:

$$U = s \mid \chi \mid u_1 + u_2 \mid u \rightarrow s \mid s \in P(\Sigma) \mid \chi \in UV \mid u_1, u_2 \in U$$

Note that in set difference, the subtrahend must be an enumeration of exceptions. This corresponds to the restriction that an exception must be explicitly named in order to be caught. Also, the introduction of use variables only make sense for the higher-order cases. Also note that the exception set $u_1 + u_2$ is considered equal to $u_2 + u_1$ since they denote the same set.

In order to consider higher-order functions, types of functions will be decorated in this system with use information. Notice that this is the only change required to the type system.

**Definition 3 (Types)** Let $TC$ be a set of type constants and $TV$ be a set of type identifiers. The set $T$ of type expressions is defined as:

$$T = c \mid \alpha \mid (\tau_1, \ldots, \tau_n) . c \mid \tau_1, \ldots, \tau_n \in T, c \in TC \mid \tau_1 \rightarrow \tau_2 \mid \tau_1, \tau_2 \in T, u \in U$$

Types behave as use information carriers. The type $\tau_1 \rightarrow \tau_2$ will read as "a function from $\tau_1$ to $\tau_2$ allowing $u$." We write $\tau_1 \rightarrow \tau_2$ when the use of the type is a variable not occurring elsewhere in the text.

Type schemes are generalizations of types on both type variables and use variables.

**Definition 4 (Type Schemes)** The set $S$ of type schemes is defined as:

$$S = \forall \alpha. \sigma \mid \forall \chi. \sigma \mid \chi \in UV, \sigma \in S \mid \tau \mid \tau \in T$$

Type schemes are also called polymorphic types, and specify a family of types each of which can be obtained by appropriate instantiation of all the quantified variables of the scheme. Type schemes are always shallow (all their quantifiers appear at the outermost of the expression, so the relative positions of the type and use variables are irrelevant). Therefore, we will write a type scheme as

$$\forall \alpha_1, \ldots, \alpha_n, \chi_1, \ldots, \chi_m \tau$$

to denote the scheme whose bound variables are

$$\alpha_1, \ldots, \alpha_n, \chi_1, \ldots, \chi_m$$

**Definition 5 (Type Environment)** A type environment $\Gamma : V \rightarrow (S \times U)$ is a partial mapping from identifiers to type schemes. We write

$$\Gamma \Rightarrow x: (\sigma, u) \text{ if } \Gamma x = (\sigma, u)$$

The mapping where all identifiers are undefined is denoted by $\emptyset$:

$$\emptyset \Rightarrow x: \bot$$

Environments are expanded by using the operator $+$

$$+ : (V \rightarrow (S \times U)) \rightarrow (V \times S \times U) \rightarrow V \rightarrow S$$

$$(\Gamma + \{ x: (\sigma, u) \}) x := (\sigma, u)$$

$$(\Gamma + \{ x_1: (\sigma_1, u) \}) x_2 := \Gamma x_2 \text{ if } x_1 \neq x_2$$
\[
\begin{align*}
\Gamma &\vdash x : (\sigma, u) & \text{Inst}(\sigma, \tau) \\
\Gamma &\vdash x : (\tau, u) & \text{(Var)} \\
\Gamma + \{ x : (\tau_1, u_1) \} &\vdash e : (\tau_2, u_2) & \text{Abs} \\
\Gamma &\vdash \lambda x.e : (\tau_1 \rightarrow \tau_2, u) & \text{(Cond)} \\
\Gamma &\vdash b : \langle \text{bool}, u \rangle & \Gamma &\vdash e_1 : (\tau_1, u_1) & \Gamma &\vdash e_2 : (\tau_2, u_2) \\
\Gamma &\vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : (\tau_1 u + u_1 + u_2) & \text{(Try)} \\
\Gamma &\vdash \text{raise } e : (\tau_1, u + \{ e \}) & \text{(Raise)} \\
\Gamma &\vdash e_1 : (\tau_1, u_1) & \Gamma &\vdash \text{let } x = e_1 \text{ in } e_2 : (\tau_2, u_1 + u_2) & \text{(Let)} \\
\Gamma &\vdash e_1 : (\tau_1 \rightarrow \tau_2, u_1) & \Gamma &\vdash e_2 : (\tau_1, u_2) & \text{(App)} \\
\Gamma &\vdash e_1 e_2 : (\tau_2, u + u_1 + u_2) & \\
\Gamma &\vdash \text{fix } f : (\tau_1, u) & \Gamma &\vdash f : (\tau_1, u) & \text{(Fix)} \\
\Gamma &\vdash e : (\tau_1, u) & \Gamma &\vdash e_1 : (\tau_1, u) & \Gamma &\vdash e_2 : (\tau_2, u_2) \\
\Gamma &\vdash \text{try } e_1 \text{ with } e_2 : (\tau_1 (u_1 - \{ e \}) + u_2) & \text{(Cond)} \\
\end{align*}
\]

**Figure 1: Type System with Uses**

**Definition 6 (Type Substitution)** Let \( \sigma \) be a type scheme, \([\tau/\alpha] \) a substitution on type variable \( \alpha \), and \([u/x] \) a substitution on use variable \( u \). Then \( [\tau/\alpha] \sigma \) is the type scheme obtained by replacing all free occurrences of \( \alpha \) in \( \sigma \) by \( \tau \). Similarly, \( [u/x] \sigma \) is the type scheme obtained by replacing all free occurrences of \( x \) in \( \sigma \) by \( u \).

**Definition 7 (Type Instantiation)** Let

\[
\sigma = \forall \alpha_1 . . . \forall \alpha_n . \chi_1 . . . \chi_m \cdot \rho
\]

\( \tau \) is an instance of \( \sigma \) if

\[
\tau = \tau[\tau_1/\alpha_1, . . . , \tau_n/\alpha_n, \chi_1/\chi_1, . . . , \chi_m/\chi_m]
\]

for appropriate types \( \tau_i \), and exceptions sets \( u_j \), for \( i = 1, . . . , n \), and \( j = 1, . . . , m \).

**Definition 8 (Generalization)** Let \( \tau, \sigma, \Gamma, \) and \( u \) be a type, type scheme, type environment, and use, respectively. We say that \( \sigma \) is a generalization of \( \tau \) according to \( \Gamma \) and \( u \) if \( \tau \) is an instance of \( \sigma \), i.e.

\[
\sigma = \forall \alpha_1 . . . \forall \alpha_n . \chi_1 . . . \chi_m \cdot \rho
\]

\[
\tau = \tau[\tau_1/\alpha_1, . . . , \tau_n/\alpha_n, \chi_1/\chi_1, . . . , \chi_m/\chi_m]
\]

and no \( \alpha_i, \chi_j \) appear free in \( \Gamma \), or \( u \) for \( i = 1, . . . , n \), and \( j = 1, . . . , m \).

### 3 Extended Type System

In this section we propose an extension based of the Hindley-Milner type system for the language ML in which function types are decorated with use information. Following is a justification for each rule, detailing the extensions while largely ignoring the basics of type inference. The type system rules are shown on Figure 1.

**Var** Like in the standard type system, a variable's type and use is an instance of the type associated with it in the environment. Due to the fact that ML is strict, upon reduction of any ML expression, variables are replaced by values, which cannot produce any exception. Note that a use is retrieved from the environment. This makes a conservative assumption.

**Abs** Evaluating the abstraction does not raise any exception. However any exception that can be raised in the body of the abstraction can potentially be raised in any context where the abstraction is applied. This is encoded in the system by annotating the function type constructor with the potential use of the body. Note that the environment of the body is augmented with a type and use assignment for the bound variable (\( \tau_1 \) and \( u_1 \)). Further, another use \( u \) is paired to the type of the abstraction.

**Cond** Conditionals just require that both branches of the decision have the same type. Again, this requirement may lead to equalize the extended parts of the types of both branches of the conditional. This construct can have an exceptional behavior if either of its branches or its condition does.

**App** In a function application, any exception raised when evaluating the function or the argument can be raised in the application. In addition, any exception that can be raised when the function is applied (i.e., the use set on top of the function’s type) can also be raised. Note that the requirement that the function’s formal argument’s type be identical to the actual argument’s type takes a new meaning for extended types— their use information, if any, must be made equal. This may seem too restrictive, but, in fact, it is not. It just states that the information collected on the environment the function is applied can be used to restrict further the type of the function.

**Fix** The fix point operator just enforces that function’s extended type and the body’s extended type unify. It can raise any exception that its body can.

**Try** Try catches exception \( x \) from its body returning \( e_2 \) in that case, but cannot catch any exception raised while evaluating \( e_2 \).

**Raise** Raise simply raises an exception, thus its abstract use reflects that restriction. Also, it does not im-
pose any restriction on the type of the resulting expression.

**Let**  Let just generalizes the type of \( c_1 \) during the typing of \( c_2 \), thus giving the polymorphic behavior on types and exception values.

There is a proof in the Appendix of the adequacy of this type system to the problem of statically determining the exceptions that a program can raise.

Some of the rules just presented have the ability of introducing unrestricted uses, which will typically be a use variable, or the empty set, or a more complex use. Given the partial order introduced by type instantiation, the choice that would lead to a more general type would always be a free variable if possible. Thus, the type of the identity function could be

\[
\forall \alpha \forall \alpha \alpha \xrightarrow{\lambda} \alpha
\]

We will omit uses that correspond to variables that appear only once. Therefore, we will write its type as

\[
\forall \alpha \alpha \rightarrow \alpha
\]

Figure 2 shows the extended types for a selected group of constants.

| \{+, -, *\}: num \rightarrow num \rightarrow num |
| \rightarrow num \rightarrow num |
| \rightarrow num |
| \rightarrow num |
| head : \forall \alpha \alpha \alpha \rightarrow \alpha |
| tail : \forall \alpha \alpha \alpha \rightarrow \alpha |
| map : \forall \alpha \beta \chi : \alpha \rightarrow \beta \rightarrow \beta |
| map2 : \forall \alpha \beta \chi : \alpha \rightarrow \beta \rightarrow \beta |
| Exceptions: |
| / \text{ division by zero} |
| d \text{ negative number} |
| h \text{ head of an empty list} |
| t \text{ tail of an empty list} |
| i \text{ invalid argument} |

Figure 2: Extended Type for Selected Constants

### 4 Examples

The type system presented in the previous section can discriminate among expressions of the same base type according to their ability to raise exceptions. To illustrate this fact, let us present two versions of a function that receives four parameters: \( a, b, c, \) and \( d \), and produces

\[
\frac{a/b}{c/d}
\]

The first version performs the computation only after it has received all four parameters

\[
F_1 : \lambda a b c d (a/b) / (c/d)
\]

The second version, on the other hand, performs the computation \( a/b \) as soon as \( a \) and \( b \) become available:

\[
F_2 : \lambda \lambda C d.v / (c/d) (a/b)
\]

Both functions have the same base type, but they differ on where they raise the "divide by zero" exception: \( F_1 \) can raise it only after reading all four arguments, while \( F_2 \) can raise the exception after the second, and after the fourth argument. This difference is reflected in their extended types:

\[
F_1 : \langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

\[
F_2 : \langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

However interesting it is to have the type system discriminate between them, it is certainly desirable that both functions be somehow "compatible". In fact, they are. The extension to the type system ensures that if two expressions can be given the same base type, then they can both be given the same extended type. In this case, both functions can be given the type of \( F_2 \), shown above. Therefore, expressions such as

\[
e_1 \equiv [F_1; F_2] \quad \text{and} \quad e_2 \equiv \text{if true then } F_1 \quad \text{else} \quad F_2
\]

have types

\[
\tau_1 \equiv \langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

and

\[
\langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

But, in those expressions, the types of \( F_1 \), and \( F_2 \) have to be precisely \( \tau_2 \).

Further, let \( F_3 \), and \( F_4 \) compute a mathematically equivalent computation

\[
\frac{(a/b) \times d}{c}
\]

in fashions similar to \( F_1 \), and \( F_2 \):

\[
F_3 : \lambda a b c d (a/b) \times d/c
\]

\[
F_4 : \lambda a b (\lambda c. (\lambda d. (d/c)) (v/c)) (a/b)
\]

Their types are

\[
F_3 : \langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

\[
F_4 : \langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

and a type common to \( F_3 \) and \( F_4 \) is

\[
\langle \text{num} \rightarrow \text{num} \rightarrow \text{num} \rightarrow \text{num}, \emptyset \rangle
\]

### Higher-Order Functions

Higher-order functions can be polymorphic on their exceptions. Such is the case of the function

\[
F = \lambda f \times f (f x)
\]

whose type is

\[
\langle (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha), \emptyset \rangle
\]

130
The type of this expression reflects the fact that \( F \) does not have any knowledge of what exceptions can be produced by the application of \( f \) to \( x \). In fact, no such exceptions are handled by \( F \); thus they will remain raised as a result of \( F \)'s application to its arguments. Figure 3 shows all the details of the deduction of \( F \)'s extended type.

Another interesting example that shows how the system behaves with higher-order functions is the expression

\[ K \equiv \lambda f. g \quad h \quad x. f \quad g \quad (f \quad h \quad x) \]

where

\[ \tau = \text{num} \frac{(\xi)}{(\cdot)} \text{num} \]

The exception set \( \{ f, \xi \} \) represents the only possible exceptions that can produce expression \( A \).

**Polymorphism**

Let-expressions are the means of introducing type polymorphism to ML. It seems natural to utilize this construct to also obtain polymorphism on exceptions. In particular, in the example for higher-order functions presented in the previous section. In particular, in the last example, we found that in order to correctly type function \( A \), we had to choose particular instances of the types of \( F, G, H \) and \( K \).

On the other hand, the following program

\[
\begin{align*}
& \text{let } F = \lambda f. x. f \; (f \; x) \\
& \quad \text{and } G = \lambda x. 2 / x \\
& \quad \text{and } H = \lambda x. \log x \\
& \quad \text{in } (\lambda x. F \; x) \; (\lambda x. F \; H \; x)
\end{align*}
\]

generalizes the types of \( F, G, \) and \( H \) while instantiating the type of each of their occurrences. Therefore, in this program, the type of the body of the let expression is

\[ \langle \text{num} \frac{(\xi)}{(\cdot)} \text{num} \times \text{num} \frac{(\cdot)}{(\cdot)} \text{num} \rangle \]

If we had not used the let expression the resulting type would have been

\[ \langle \text{num} \frac{(\xi)}{(\cdot)} \text{num} \times \text{num} \frac{(\cdot)}{(\cdot)} \text{num} \rangle \]

Note that polymorphism on exceptions is just the ability of a function to be able to carry different exceptions in different occurrences. At each occurrence, however, all the exceptions that were deduced from its definition have to belong to the exception set.

The above example shows that for some higher-order expressions, unless they are introduced using a let construct, their polymorphism on exceptions—as well as on types—is lost. This decision fits nicely into ML, since the same construct introduces both kinds of polymorphism.

**Sharing Morphisms**

The handling of exceptions in ML makes it adequate to implement sharing morphisms in the language. It is usually the case that when homomorphisms such as variable substitution is performed on a term, a new term is generated even in the case that the resulting term equals the original one—in this case, the term is completely copied.

The function \( \text{map} \) is an example of such a behavior:

\[
\begin{align*}
& \text{let } \text{map} \; f = \text{fun } \begin{cases} \\
& [] \rightarrow [] \\
& (x @ l) \rightarrow f \; x @ \text{map} \; f \; l
\end{cases}
\end{align*}
\]

The function \( \text{map} \; i \), where \( i \) is the identity function, copies its list argument even though its elements are returned unchanged. Its extended type (Figure 2) conveys the information that \( \text{map} \) does not capture any exception raised by the application functional argument \( f \).

A sharing homomorphism, on the other hand, returns the argument—not a copy—if the argument is not altered by the function. A sharing homomorphism can be implemented in either of these techniques:
• pair the result with a boolean value—effectively signalling whether the result is the very argument, or
• define an exception, say \( Id \), that is to be raised whenever the result of a function is the same as its argument. The calling function then receives a result if it is different from the argument, and an exception if they are equal.

In both cases, the calling function has enough information to decide whether to return the argument depending on the results of its calls. However, the latter solution is preferred because it conveys this information more naturally.

The implementation of a sharing version of \( \text{map} \) follows

\[
\begin{aligned}
\text{exception } \text{Id} \\
\text{let } \text{share } f a = \\
& \text{try } f a \text{ with } \text{Id} \rightarrow a \\
\text{let } \text{mapshare } f = \\
& \text{fun } [\rightarrow \text{raise } \text{Id} \\
& (x : \text{x}) \rightarrow \\
& \text{try } f x \rightarrow \text{share } (\text{mapshare } f) x \text{ with } \text{Id} \rightarrow x : \text{mapshare } f x \\
\text{let } \text{mapsh } f x s = \text{share } (\text{mapshare } f) x s
\end{aligned}
\]

In the above programs, \text{share} is a function that handles \( \text{Id} \) by returning the argument—in fact, preventing the exception from escaping further. It is called in contexts where the result is to be combined in a new structure (although the result itself can be shared with the previous structure). The function \text{mapshare} actually processes the list. Note that the empty list is shared, and note how (\text{mapshare} \ f) is protected from raising \( \text{Id} \)—in this context, \((f \ x)\) did not raise any exception, thus the new structure needs to be consed to the result of \((f \ x)\). If, on the other hand, \((f \ x)\) raised \( \text{Id} \), then \((\text{mapshare} \ f)\) is not protected, since it would in fact imply that the current cons cell should be shared. Finally, \text{mapsh} just manages the case when the whole data structure can be shared, effectively sharing it.

With respect to the typing, the extended type of \text{share} is

\[
\text{share} : \forall \alpha. (\alpha \cong \alpha) \rightarrow \alpha \rightarrows[\text{Id}] \alpha, \varnothing
\]

which indicates that \text{share} explicitly handles the \( \text{Id} \) exception. The extended type of \text{mapshare} is

\[
\text{mapshare} : \forall \alpha. (\alpha \cong \alpha) \rightarrow \alpha \rightarrows[\text{Id}] \alpha, \varnothing
\]

since it can raise \( \text{Id} \), and the extended type of \text{mapsh} is

\[
\text{mapsh} : \forall \alpha. (\alpha \cong \alpha) \rightarrow \alpha \rightarrows[\text{Id}] \alpha, \varnothing
\]

since it handles the \( \text{Id} \) exception through the call to \text{share}.

5 Pragmatic Considerations

In a glance through the CAML-Light documentation [9] the reader will notice that the description of each function includes its type, a short English description of what it does, and the list of exceptions raised when fully applied. This illustrates the importance of exceptions when describing interfaces. CAML allows for modules to be defined, along with their interfaces, but, unfortunately, CAML interface mechanism does not allow for the association of functions to the exceptions they potentially raise.

However important the handling of exceptions can be in typing programs, or providing interfaces, several issues need to be addressed in order to accept a mechanism to help us in this regard. Among them are:

• is it really necessary to control exceptions?,
• how easy is it to read annotated types and understand what they mean?,
• how easy is it to write annotated types?,
• how much overhead does this new analysis impose?

As with many other analyses, the real value of the analysis is not realized from toy programs, such as those presented here, but from large programs. Large programs tend to have many exceptions. Would it not be annoying to have a type system report that the main function may have a variety of exceptions? On the other hand, exceptions should be handled, and thus, there should not be any untended exceptions. Good programming style indicates that exceptions should be treated as soon as they can be.

There are exceptions that the user may choose not to handle, such as an out of memory exception. In this case, it would certainly be very bothersome to have the type system report such an exception for any expression. To solve this case, we propose a new ML directive

\[
\text{#ignore exc_list}
\]

that eliminates the list of exceptions exc_list from the reported exceptions.

In some cases, there can be exceptions that the user is certain they will never be raised, therefore, the user does not trap them. To illustrate this, consider the function \text{map2} whose extended type signature appears in Figure 2. This function is similar to \text{map}, but acts on functions of two arguments. It requires that the lists of first, and second arguments be of equal lengths, raising the exception in otherwise (in stands for “invalid argument”). The expression

\[
(f x \rightarrow \text{map2 } (\text{pre fix } \ast ) x x) \ [1, 2, 3, 4]
\]

has type \( \langle \text{num list}, \{i\} \rangle \) whereas, in practice, the \( i \) exception will not be raised. The type system is unable to detect and/or verify that the exception is bogus in this case, and thus, it will report it. In ML, the user may choose to explicitly handle the expression, but it would not express the fact that the expression should not happen:

\[
(f x \rightarrow \text{try map2 } (\text{pre fix } \ast ) x x \text{ with } i a \rightarrow \ldots) \ [1, 2, 3, 4]
\]

has type \( \langle \text{num list}, \varnothing \rangle \) (assuming ... does not raise any exception).
Further, the user may use a new exception, such as stop, and write the program as

```haskell
#ignore stop

(fun x → try map2 (prefix *) x x
   with ia → raise stop)
[1, 2, 3, 4]
```

which also has the above type.

A more expressive solution is to introduce a new ML expression

```
exp ignoring excalist
```

to provide a behavior such as the one just presented. It should be noted that the compilation of the ignoring-construct should include the raising of another exception that cannot be captured. Otherwise, the expression would not be sound. With in mind, the raising of an ignored exception should abort the whole program. Thus, the expression

```
((fun x → try map2 (prefix *) x x) ignoring ia)
[1, 2, 3, 4]
```

has type `(num list, ε)` which is indeed quite reasonable.

Another issue that needs to be addressed is about the concrete syntax of the extension. The syntax presented so far follows the convention introduced in [4] of annotating the arrows with the abstract use that happens when the functional is applied. This is great for pedagogical purposes, but cannot be implemented as is in textual programming languages. Modula 3 associates exceptions to types using the keyword raises [1]

```
type raises exciset
```

FX [3] uses the infix operator "*" to separate the type from the effects of exceptions. We propose to use "*" for this purpose. Thus, the extended type signature

```
ε : τ | u1 + · · · + un
```

means that ε has type τ and can raise exceptions u1,...,un (n ≥ 0). The following conventions aid the readability of the extension:

1. "*" has lower operator precedence than "+", but more precedence than any other type constructor.
2. the exceptions clause is optional. Omitting it indicates no exceptions are raised.
3. For user-generated types, "*" indicates the user makes no claims on the exceptions raised by the expression (Computer generated types can always indicate the inferred exceptions).

The above conventions favor first order types as well as the types of nested abstractions, that usually appear when programming with equational groups. It does not necessarily favor combinations that raise exceptions and are of functional type.

Following the conventions, the type of map can be written as

```
(α → β, ε) → α list → β list, ε
```

The type of fold is

```
(α → β, ε1, ε2) → α list → β list, ε1 + ε2
```

The type of fun x y = (fun v z t = v/(z/t))(x/y) is num → num → num → num → num.

6 Type Reconstruction

The type reconstruction algorithm needed to infer the extended types presented in this paper is a straightforward modification to the Hindley-Milner algorithm. The only complication is how to maintain the set of possible exceptions handled by any expression, and how to refine those sets with the restrictions implied by the inference rules.

As explained in Section 2, the only operations allowed on exception sets are union, and set difference where the subtrahend must be an enumeration of exceptions. Also, it is implied by the rules that any use expression will have a use variable involved in it. Therefore, any use expression u can be expressed as χ + u' where χ is a variable not appearing in u'. The least upper bound of two use expressions χ1 + u1 and χ2 + u2 is the union of sets u1 and u2, which can be obtained with the most general unifier (unique modulo equivalence of expressions on sets)

```
s = [χ1 ← χ + u1 + u2, χ2 ← χ + u1 + u2']
```

where χ is a use variable not appearing in u1 or in u2 and each u' is obtained by replacing in u1 variables χ and ε with ε.

Unification on types is the usual term unification extended with the unification on uses. For instance, the most general unifier of types

```
(τ u+χ, ε(τ → (α u β) → γ))
```

is the substitution

```
s = []
```

```
(τ u+χ, ε(τ → (α u β) → γ))
```

```
(τ u+χ, ε(τ → (α u β) → γ))
```

```
(τ u+χ, ε(τ → (α u β) → γ))
```

The exceptional behavior of programs is fully reconstrucatable based solely on the information about the exceptional behavior of the primitives used. For modules, the interface types need to be extended to also convey the information about the exceptions the typed function may raise. Other than this, any currently working ML program can be inferred its exceptional behavior. Further, the system can aid in synthesizing this information, given that a module module itself can be type-checked, and thus its types—including those belonging to its interface—reconstructed.

7 Related Work

The idea of decorating the arrows with operational properties of expressions was first exploited by Gifford, Lucasen et al in the context of detecting levels of imperative nature of expressions in a Scheme-like language. This work was first published in [4], and made into an experimental programming language called FX [3]. In [8] they provide an overview of the FX system, and offer a reconstruction algorithm for its type system. The FX system can detect side-effects to user-defined regions, which are arbitrary sets of locations where data is allocated. Data allocated in the same region share the same properties (allocation, mutation, etc.)

Guzmán and Hudak also worked on detecting side-effects, this time in the context of lazy functional languages [6, 5]. They extended Gifford’s work by adding the notion of liability—an environment that associated
non-local variables with operational properties. This extension widens the spectrum of possible applications, and makes it possible to control mutation to data structures—not regions—in single-threaded contexts. Also, in [7], they presented liveness analysis through a definition of an extended type system where not only arrows were all types—not only arrows were annotated.

Lecy and Weis have also worked on annotating types. Their work focuses on how to type ML references polymorphically [10]. Their system very successful in differentiating dangerous references (those whose types cannot be generalized) from safe ones.

8 Conclusions

We have presented a simple and intuitive yet powerful mechanism through which the set of potential exceptions that can happen in a program can be statically inferred.

The approach used was extending the type system. This provides a simple, clean, and seamless integration of this extension to the ML language. We have discussed several pragmatic issues concerning this fusion.

The type system is polymorphic on the annotations (exceptions that can be raised), as well as on types thus exploiting even further the powerful type system in the language. The type system is fully reconstructible.

A Correctness of the Type System

The type rules of ML has been adapted in this work to the handling of information about exceptions. This extension is conservative: Any proof in the original system can be decorated in order to obtain a proof in the extended system and thus, the soundness of the type system for exceptions is a direct consequence of the soundness of the original system. The existence of principal types for expressions is in the same case. For a proof of soundness of the type system of ML and of principality of the system see for instance [2, 14]. We prove in this section that if an expression evaluates to an exception, this exception appears in the use associated to its type.

Definition 9 (Values and Execution Environments)

Let \( C = C_1 \cup \ldots \cup C_n \) be a set of constants classified by their respective types \( \tau_1, \ldots, \tau_n \). An execution environment \( \phi : V \rightarrow Val \) is a partial mapping from identifiers to values. The set \( Val \) of execution values is defined as the disjoint union of constants and closures:

\[
Val = C \cup \{ \phi \} \quad \text{where} \quad \phi \text{ is an execution environment}
\]

The expressions of the language are extended to include values for which we introduce the typing rules of figure 6.

Figure 5 represents the evaluation rules for ML (variable \( \omega \) ranges over the values and exception expressions). These rules define the evaluation relation \( \equiv \phi \) between extended expressions. The normal forms of this relation are either values or exceptions of the form \( \text{raise} e \).

The following theorem states that the type of the result \( e' \) of the evaluation of an expression \( e \) is more particular than the type of \( e \). Additionally, the use of expression \( e \) contains the use of \( e' \).

\[
\frac{e \in C}{\Gamma \vdash e : (\tau, u)} \quad \text{(Ctt)}
\]

\[
\frac{y : \tau | \phi(y) : \tau, y \in \text{FV}(e)}{\Gamma \vdash \lambda x.e : (\tau_1 \rightarrow \tau_2, u') \quad \text{(Cls)}}
\]

Figure 6: Typing Rules for Values

Theorem 1 Let \( e, \tau, u, \Gamma, \phi \) be such that for each variable \( x \) free in \( e \), whenever \( \Gamma \vdash x : (\tau, u) \) we also have \( \phi(x) : \tau \). For each expression \( e' \), if there is a deduction for \( \Gamma \vdash e : (\tau, u) \) and a derivation \( e \equiv \phi \phi' \), there is a type \( \tau' \), an use \( u' \) and a derivation for \( \Gamma \vdash e' : (\tau', u') \) such that \( \tau \) is more general than \( \tau' \) and \( u' \subseteq u \).

Proof: The proof is made by induction on the number of steps of the derivation \( e \equiv \phi \phi' \). Values and exception expression are the normal forms of \( \equiv \phi \). The property vacuously hold for values. Expressions of the form \( \text{raise} e \) has any type and any use containing exception \( e \).

Var By definition of evaluation environments, \( \phi(x) \) is in normal form and by hypothesis, \( \phi(x) : \tau \). As exceptions are not valid elements of \( \phi \), we have \( \Gamma \vdash \phi(x) : (\tau, \emptyset) \).

Abs The valuation of an abstraction \( \lambda x.e \) is a closure, which is typed exactly as the abstraction it contains (with the typing rule (Cls)).

App If an application is evaluated using rules (EAApp) or (EApp), The property is verified by simple use of inductive hypothesis. If the reduction is made with rule (EAApp), we should explore again the type of a closure. By inductive hypothesis, there exist a typing environment \( \Gamma' \) verifying \( \Gamma' \vdash y : \tau \) for each \( y \) free in \( \lambda x.e \phi(y) \).\( \vdash \phi(y) : \tau \), such that the following diagram holds:

\[
\begin{array}{ll}
\Gamma' \vdash \{ x : \tau_1 \} \vdash e : (\tau_2, u_3) & \text{Abs} \\
\Gamma' \vdash \lambda x.e \phi(y) : (\tau_1 \rightarrow \tau_2, u_4) & \text{Cls} \\
\Gamma' \vdash \lambda x.e \phi(y) : (\tau_1 \rightarrow \tau_2, u_5) & \text{L} \\
\end{array}
\]

As \( \tau_1 \rightarrow \tau_2 \) is more general than \( \tau_1 \rightarrow \tau_2 \), we have \( u_4 \subseteq u_5 \). Again, by inductive hypothesis, \( u_4 \subseteq u_4 \) and \( u_5 \subseteq u_5 \) which completes the proof of the property for applications.

Let The polymorphism introduced by let expressions could also be introduced in a generic type system by replacing each occurrence of the variable introduced in the let by its associated expression (avoiding variable captures by an adequate renaming) [13]. The proof of the property for local declarations is very similar to the one for applications.
\[
\begin{align*}
    x \xrightarrow{\varphi} \varphi(x) \quad \text{(EVar)} \\
    e_1 \xrightarrow{\varphi} [\lambda x,e,\varphi] \quad e_2 \xrightarrow{\varphi} v_2 \quad e \xrightarrow{\varphi[v_2/x]} w \quad \text{(EApp)} \\
    (e_1 e_2) \xrightarrow{\varphi} w \quad \text{(EApp2)} \\
    e_1 \xrightarrow{\varphi} [\lambda x,e,\varphi] \quad e_2 \xrightarrow{\varphi} \text{raise } \varepsilon \quad \text{(EApp3)} \\
    (e_1 e_2) \xrightarrow{\varphi} \text{raise } \varepsilon \\
    e_1 \xrightarrow{\varphi} \text{true} \quad e_2 \xrightarrow{\varphi} w \quad \text{(ECond)} \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_2 \xrightarrow{\varphi} w \\
    e_1 \xrightarrow{\varphi} \text{raise } \varepsilon \quad \text{(ECond2)} \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_2 \xrightarrow{\varphi} w \\
    e_1 \xrightarrow{\varphi} \text{raise } \varepsilon \quad \text{(ECond3)} \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \xrightarrow{\varphi} \text{raise } \varepsilon \\
    e_1 \xrightarrow{\varphi} \text{raise } \varepsilon \quad e_2 \xrightarrow{\varphi} w \quad \text{(ETry)} \\
    \text{try } e_1 \text{ with } \varepsilon \rightarrow e_2 \xrightarrow{\varphi} v \\
    e_1 \xrightarrow{\varphi} \text{raise } \varepsilon \quad e_2 \xrightarrow{\varphi} w \quad \text{(ETry2)} \\
    \text{try } e_1 \text{ with } e_2 \rightarrow e_2 \xrightarrow{\varphi} \text{raise } e_2 \\
    e_1 \xrightarrow{\varphi} v_1 \quad e_2 \xrightarrow{\varphi[v_1/x]} w \quad \text{(ELet)} \\
    \text{let } x = e_1 \text{ in } e_2 \xrightarrow{\varphi} w \\
    e_1 \xrightarrow{\varphi} \text{raise } \varepsilon \quad \text{(ELet2)} \\
    \text{let } x = e_1 \text{ in } e_2 \xrightarrow{\varphi} \text{raise } \varepsilon
\end{align*}
\]

**Figure 5: Evaluation Rules**

as the polymorphism introduced by local declarations is just a particular case of the one introduced by evaluation.

Cond, Fix, Try

In these cases, the property is a direct consequence of the inductive hypothesis.

As a consequence of this theorem if there is a deduction for \( \Gamma \vdash e : (\forall a)u \) and a derivation \( e \xrightarrow{\varphi} \text{raise } \varepsilon \) then exception \( \varepsilon \) is an element of the use \( u \).

**References**


