Type-based analysis of uncaught exceptions

XAVIER LEROY and FRANÇOIS PESSAUX
INRIA Rocquencourt

This paper presents a program analysis to estimate uncaught exceptions in ML programs. This analysis relies on unification-based type inference in a non-standard type system, using rows to approximate both the flow of escaping exceptions (a la effect systems) and the flow of result values (a la control-flow analyses). The resulting analysis is efficient and precise; in particular, arguments carried by exceptions are accurately handled.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; *Operational semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives*; *Type structure*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; *Symbolic execution*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; *ML*

General Terms: Languages, Reliability, Theory

Additional Key Words and Phrases: Caml, exceptions, polymorphism, rows, static debugging, SML, type and effect systems, type inference

1. INTRODUCTION

Many modern programming languages such as Ada, Modula-3, ML and Java provide built-in support for exceptions: raising an exception at some program point transfers control to the nearest handler for that exception found in the dynamic call stack. Exceptions provide safe and flexible error handling in applications: if an exception is not explicitly handled in a function by the programmer, it is automatically propagated upwards in the call graph until a function that "knows" how to deal with the exception is found. If no handler is provided for the exception, program execution is immediately aborted, thus pinpointing the unexpected condition during testing. This stands in sharp contrast with the traditional C-style reporting of error conditions as "impossible" return values (such as null pointers or the integer -1): in this approach, the programmer must write significant amount of code to propagate error conditions upwards; moreover, it is very easy to ignore an error condition altogether, often causing the program to crash much later, or even complete but produce incorrect results.

The downside of using exceptions for error reporting and as a general non-local

Authors' addresses: Xavier Leroy, INRIA Rocquencourt, Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France, `Xavier.Leroy@inria.fr`; François Pessaux, Dept. of Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, U.S.A., `fpessaux@cs.stevens-tech.edu`.
This work has been partially supported by CNET, France Télécom.

control structure is that it is very easy to forget to catch an exception at the right place, i.e. to handle an error condition. ML compilers generate no errors or warnings in this case, and the programming mistake will only show up during testing. Exhaustive testing of applications is difficult, and even more so in the case of error conditions that are infrequent or hard to reproduce. Our experience with large ML applications is that uncaught exceptions are the most frequent mode of failure.

To address this issue, languages such as Modula-3 and Java require the programmer to declare, for each function or method, the set of exceptions that may escape out of it. Those declarations are then checked statically during type-checking by a simple intraprocedural analysis. This forces programmers to be conscious of the flow of exceptions through their programs.

Declaring escaping exceptions in functions and method signatures works well in first-order, monomorphic programs, but is not adequate for the kind of higher-order, polymorphic programming that ML promotes. Consider the `map` iterator on lists, which applies a given function to every element of a list. In Modula-3 or Java, the programmer must declare a set E of exceptions that the function argument to `map` may raise; `map`, then, may raise the same exceptions E. But E is fixed arbitrarily, thus preventing `map` from being applied to functions that raise exceptions not in E. The genericity of `map` can be restored by taking for E the set of all possible exceptions, but then the precision of the exception analysis is dramatically decreased: all invocations of `map` are then considered as potentially raising any exception. (Similar problems arise in highly object-oriented Java programs using container classes and iterators intensively.) To deal properly with higher-order functions, a very rich language for exception declarations is required, including at least exception polymorphism (variables ranging over sets of exceptions) and unions of exception sets. (See section 2 for a more detailed discussion.) We believe that such a complex language for declaring escaping exceptions is beyond what programmers are willing to tolerate.

The alternative that we follow in this paper is to infer escaping exceptions from unannotated ML source code. In other terms, we view the problem of detecting potentially uncaught exceptions as a static debugging problem, where static analyses are applied to the programs not to make them faster via better code generation, but to make them safer by pinpointing possible run-time failures. This approach has several advantages with respect to the Modula-3/Java approach: it blends better with ML type inference; it does not change the language and supports the static debugging of "legacy" applications; it allows the use of complex approximations of exception sets, as those need not be written by the programmer (within reason – the results of the analysis must still be understandable to the programmer). Finally, the exception inference needs not be fully compatible with the ML module system: a whole program analysis can be considered (again within reason – analysis time should remain practical).

Several exception analyses for ML have been proposed [Guzmán and Suárez 1994; Yi 1998; Yi and Ryu 1997; Fähndrich and Aiken 1997; Fähndrich et al. 1998], some based on effect systems, some on control-flow analyses, some on combinations of both (see section 6 for a detailed discussion). The analysis presented in this paper attempts to combine the efficiency of effect systems with the precision of flow analyses. It is based on unification and non-standard type inference algorithms that have excellent running time and should scale well to large applications. At the same time, our analysis is still fairly precise; in particular, it approximates not only the names of the escaping exceptions, but also the arguments they carry – a feature that is essential to analyze precisely many existing ML programs. This constitutes the main technical contribution of this paper: integrate in the same unification-based framework both approximation of exception effects in the style of effect systems [Talpin and Jouvelot 1994], and approximation of sets of values computed at each program point in the style of flow analyses and soft typing [Shivers 1991; Wright and Cartwright 1997]. Finally, our analysis has been implemented to cover the whole Objective Caml language – not only core ML, but also datatypes, objects, and the module system. We present some preliminary experimental results obtained with our implementation.

The remainder of this paper is organized as follows. Section 2 lists the main requirements for an ML exception analysis. Section 3 presents the non-standard type system we use for exception analysis. Extension to the full Objective Caml language is discussed in section 4; experimental results obtained with our implementation, in section 5; and related work, in section 6. Concluding remarks can be found in section 7. Algorithms and proofs are shown in appendices.

2. DESIGN REQUIREMENTS

In this section, we list the main requirements for an effective exception analysis for ML, and show that they go much beyond what can be expressed by exception declarations in Modula-3 or Java. Existing exception analyses address some of these requirements, but none addresses all.

2.1 Handling higher-order functions precisely

The exception behavior of higher-order functions depends on the exceptions that can be raised by their functional arguments. A form of polymorphism over escaping exceptions is thus needed to analyze higher-order functions precisely. Consider the `map` iterator over lists mentioned in introduction. An application `map f l` may raise whatever exception the `f` argument may raise. Writing $\tau \xrightarrow{\varphi} \tau'$ for the annotated type of functions from type τ to type τ' whose set of potentially escaping exception is φ, the behavior of `map` is captured by the following annotated type scheme:

$$\mathtt{map} : \forall \alpha, \beta, \varphi.\ (\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} (\alpha\ \mathtt{list} \xrightarrow{\varphi} \beta\ \mathtt{list})$$

where α, β range over types and φ ranges over sets of exceptions. In general, the escaping exceptions for a higher-order function are combinations $\varphi_1 \cup \ldots \cup \varphi_n \cup \{C_1; \ldots; C_n\}$ where the φ_i are variables representing the escaping exceptions for functional arguments and the C_j are exception constants. For instance, we have the following annotated type for function composition $\lambda f.\lambda g.\lambda x.f(g(x))$:

$$\forall \alpha, \beta, \gamma, \varphi, \psi.\ (\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} (\gamma \xrightarrow{\psi} \alpha) \xrightarrow{\emptyset} \gamma \xrightarrow{\varphi \cup \psi} \beta$$

Given the frequent use of higher-order functions in ML programs, an exception analysis for ML must handle them with precision similar to what the annotated types above suggest.

Similar issues arise when functions are stored into data structures such as lists or hash tables (as in callback tables for instance). The exception analysis should keep track of the union of the exceptions that can be raised by functions contained in the structure. It is not acceptable to say that any exception can be raised by applying a function retrieved from the structure.

2.2 Handling exceptions as first-class values

In ML and Java, exceptions are first-class values: exception values can be built in advance and passed through functions before being raised. Consider for instance the following contrived example:

```
let test = λexn. try raise(exn) with E → 0
```

The exception behavior of this function is that `test exn` raises the exception contained in the argument `exn`, except when `exn` is actually the exception `E`, in which case no exception escapes out of `test`. We seek exception analyses precise enough to capture this behavior.

It is true that the first-class character of exception values is rarely, if ever, used in actual ML programs. However, there is one important idiom where an exception value appears: finalization. Consider:

```
let f = λx. try g(x)
            with E → 0
               | exn → finalization code; raise(exn)
```

Assuming `g` can raise exceptions `E` and `E'`, the exception analyzer should recognize that the `exn` exception variable can only take the value `E'`, thus the `raise(exn)` that re-raises the exception after finalization can only raise `E'`, and so does the function `f` itself.

2.3 Keeping track of exception arguments

ML exceptions can optionally carry arguments, just like all other data type constructors. This argument can be tested in the `with` part of an exception handler, using pattern-matching on the exception value, so that only certain exceptions with certain arguments are caught. Consider the following example:

```
exception Failure of string
let f = λx. if ... then ... else raise(Failure "f")
let g = λx. try f(x) with Failure "f" → 0
```

An exception analysis that only keeps track of the exception head constructors (i.e. `Failure` above) but not of their arguments (i.e. the string `"f"` above) fails to analyze this example with sufficient precision: the analysis records that function `f` may raise the `Failure` exception, hence it considers that the application `f(x)` in `g` may raise `Failure` with any argument. Since the exception handler traps only `Failure "f"`, the analyzer concludes that `g` may raise `Failure`, while in reality no exception can escape `g`.

This lack of precision can be brushed aside as "unimportant" and "bad programming style anyway". Indeed, the programmer should have declared a specific constant exception `Failure_f` to report the error in `f`, rather than rely on the general-purpose `Failure` exception. However, code fragments similar to the example above appear in legacy Caml applications that we would like to analyze. More importantly, there are also legitimate uses of exceptions with parameters. For instance, the Caml interface to Unix system calls uses the following scheme to report Unix error conditions:

```
type unix_error = EACCES | ENOENT | ENOSPC | ...
(* enumerated type with 67 constructors representing Unix error codes *)
exception Unix_error of unix_error
```

This allows user code to trap all Unix errors at once (`try` ... `with Unix_error(_) ->` ...), and also to trap particular errors (`try` ... `with Unix_error(ENOENT) ->` ...). Replacing `Unix_error` by 67 distinct exceptions, one for each error code, would make the former very painful. It is desirable that the exception analysis be able to show that certain `Unix_error` exceptions with arguments representing common errors (e.g. `Unix_error(ENOENT)`, "no such file") are handled in the program and thus do not escape, while we can accept that other `Unix_error` exceptions representing rare errors are not handled in the program and may escape.

The problem with exception arguments is made worse by the availability (in the Caml standard library at least) of predefined functions to raise general-purpose exceptions such as `Failure` above. Indeed, the example with `Failure` above is more likely to appear under the following form:

```
exception Failure of string
let failwith = λmsg. raise(Failure msg)
let f = λx. if ... then ... else failwith("f")
let g = λx. try f(x) with Failure "f" → 0
```

Precise exception analysis in this example requires tracking the string constant `"f"` not only when it appears as immediate argument to the `Failure` exception constructor, but also when it is passed to the function `failwith`. Hence the exception analysis must also include some amount of data flow analysis, not limited to exception values.

2.4 Running faster than control-flow analyses

All the requirements we have listed so far point towards control-flow analyses for functional languages in the style of k-CFA [Shivers 1991] or set-based analysis [Heintze 1994]. In order to determine the flow of control at function applications, these analyses need to track the flow of functional values throughout the program; to do this, they build an approximation of the set of values that can flow to each program point. It is entirely straightforward to extend them to approximate also the set of escaping exceptions at each program point at the same time as they approximate the set of result values. Alternatively, the exception analysis can be run as a second pass of dataflow analysis exploiting the results of control-flow analysis [Yi and Ryu 1997], although this results in some loss of precision, as the control flow can be determined more accurately if exception information is available. This exception analysis benefits from the relatively precise approximation of values provided by the control-flow analysis, especially as far as exception arguments are concerned.

Our first implementation of an exception analyzer for Objective Caml was indeed based on control-flow analysis: 0-CFA initially, then polymorphic splitting [Jagannathan and Wright 1998]. Our practical experience with this approach was mixed: the precision of the exception analysis was satisfactory (at least with polymorphic splitting), but the speed of the analysis left a lot to be desired. For instance, analyzing a 600-line program (a simplified version of the Knuth-Bendix benchmark) took 18 seconds on a 150 Mhz Pentium Pro. Those figures should be taken with a grain of salt: our implementation of CFA was semi-naive and did not implement all of the optimizations described or alluded to in the literature on CFA and other analyses based on set inclusion constraints [Fähndrich and Aiken 1996; Flanagan and Felleisen 1997; Fähndrich et al. 1998; Pottier 1996]. Still, we observed quadratic behavior on several examples, indicating that the analysis would not scale easily to large programs[1].

For these reasons, we decided to abandon analyses based on CFA or more generally set inclusion constraints, and settled for less precise but faster analyses based on equality constraints and unification.

3. A TYPE SYSTEM FOR EXCEPTION ANALYSIS

In the style of effect systems [Lucassen and Gifford 1988; Talpin and Jouvelot 1994], our exception analysis is presented as a type inference algorithm for a non-standard type system. The type system uses unified mechanisms based on row variables both to keep track of the effects (sets of escaping exceptions) of expressions and to refine the usual ML types by more precise information about the possible values of expressions. In this section, we present first the typing rules for our type system (that is, the specifications for the exception analysis), then type inference issues (the actual analysis).

3.1 The source language

The source language we consider in this paper is a simple subset of ML with integers and exceptions as the only data types, the ability to raise and handle exceptions, and simplified pattern-matching.

Terms:	$a ::= x$	identifier
	$\mid i$	integer constant
	$\mid \lambda x.\, a$	abstraction
	$\mid a_1(a_2)$	application
	$\mid \texttt{let}\ x = a_1\ \texttt{in}\ a_2$	the `let` binding
	$\mid \texttt{match}\ a_1\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3$	pattern-matching
	$\mid C \mid D(a)$	exception constructors
	$\mid \texttt{try}\ a_1\ \texttt{with}\ x \rightarrow a_2$	exception handler

[1]The complexity of 0-CFA alone is $O(n^3)$, where n is the size of the whole program. We did not observe cubic behavior on our tests, however. Quadratic behavior arises in the following not uncommon case: assume that a group of functions of size $k = O(n)$ recurses over a list of $m = O(n)$ elements given in extension in the program source. At least m iteration of the analysis is required before fixpoint is reached on the parameters and results of the functions. Since each iteration takes time proportional to k, the time of the analysis is $O(n^2)$.

Patterns:	$p ::= x$	variable pattern
	$\mid i \mid C$	constant patterns
	$\mid D(p)$	constructed pattern

The construct $\texttt{match}\ a_1\ \texttt{with}\ p \to a_2 \mid x \to a_3$ performs pattern-matching on the value of a_1; if it matches the pattern p, the branch a_2 is evaluated; otherwise, a_3 is evaluated. Multi-case pattern matchings can be expressed by cascading `match` expressions. The construct $\texttt{try}\ a_1\ \texttt{with}\ x \to a_2$ evaluates a_1; if an exception is raised, its value is bound to x and a_2 is evaluated. There is no syntactic form for raising an exception; instead, we assume predefined a `raise` function in the environment. The `try` construct catches all exceptions; catching only a given exception C is performed by:

$$\texttt{try}\ a_1\ \texttt{with}\ x \to \texttt{match}\ x\ \texttt{with}\ C \to a_2 \mid y \to \texttt{raise}(y)$$

The dynamic semantics for this language is given by the reduction rules in figure 1, in the style of [Wright and Felleisen 1994]. Values, evaluation contexts, and evaluation results are defined as:

Values:	$v ::= i \mid C \mid D(v) \mid \lambda x.a \mid \texttt{raise}$
Evaluation contexts:	$\Gamma ::= [\,] \mid \Gamma(a) \mid v(\Gamma) \mid D(\Gamma)$
	$\mid \texttt{let}\ x = \Gamma\ \texttt{in}\ a$
	$\mid \texttt{match}\ \Gamma\ \texttt{with}\ p \to a_2 \mid x \to a_3$
	$\mid \texttt{try}\ \Gamma\ \texttt{with}\ x \to a$
Evaluation results:	$r ::= v \mid \texttt{raise}\ v$

A result of v indicates normal termination with return value v; a result of $\texttt{raise}\ v$ indicates an uncaught exception v.

3.2 The type algebra

The type system uses the following type algebra:

Type expressions:	$\tau ::= \alpha$	type variable
	$\mid \texttt{int}[\varphi]$	integer type
	$\mid \texttt{exn}[\varphi]$	exception type
	$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	function type
Type schemes:	$\sigma ::= \forall \alpha_i, \rho_j, \delta_k.\tau$	
Rows:	$\varphi ::= \rho$	row variable
	$\mid \top$	all possible elements
	$\mid \varepsilon; \varphi$	the element ε plus whatever is in φ
Row elements:	$\varepsilon ::= i : \pi$	integer constant
	$\mid C : \pi$	constant exception
	$\mid D(\tau)$	parameterized exception
Presence annotations:	$\pi ::= \texttt{Pre}$	element is present
	$\mid \delta$	presence variable

$$
\begin{aligned}
(\lambda x.a)(v) &\Rightarrow a\{x \leftarrow v\} && (1)\\
\texttt{let}\ x = v\ \texttt{in}\ a &\Rightarrow a\{x \leftarrow v\} && (2)\\
\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow \sigma(a_2) \text{ if } \sigma = M(v,p) \text{ is defined} && (3)\\
\texttt{match}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow a_3\{x \leftarrow v\} \text{ if } M(v,p) \text{ is undefined} && (4)\\
\texttt{try}\ v\ \texttt{with}\ x \rightarrow a_2 &\Rightarrow v && (5)\\
(\texttt{raise}\ v)(a) &\Rightarrow \texttt{raise}\ v && (6)\\
(\lambda x.a)(\texttt{raise}\ v) &\Rightarrow \texttt{raise}\ v && (7)\\
D(\texttt{raise}\ v) &\Rightarrow \texttt{raise}\ v && (8)\\
\texttt{let}\ x = \texttt{raise}\ v\ \texttt{in}\ a &\Rightarrow \texttt{raise}\ v && (9)\\
\texttt{match}\ \texttt{raise}\ v\ \texttt{with}\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow \texttt{raise}\ v && (10)\\
\texttt{try}\ \texttt{raise}\ v\ \texttt{with}\ x \rightarrow a_2 &\Rightarrow a_2\{x \leftarrow v\} && (11)\\
\Gamma[a] &\Rightarrow \Gamma[a'] \text{ if } a \Rightarrow a' && (12)
\end{aligned}
$$

The pattern-matching function $M(v,p)$:

$$
M(v,x) = \{x \leftarrow v\} \qquad M(i,i) = id \qquad M(C,C) = id
$$
$$
M(D(v), D(p)) = M(v,p)
$$

Fig. 1. Reduction rules

As in effect systems, our function types $\tau_1 \xrightarrow{\varphi} \tau_2$ are annotated by the latent effect φ of the function, that is, the set of exceptions that may be raised during application of the function. In addition, the base types $\texttt{exn}[\varphi]$ and $\texttt{int}[\varphi]$ are also annotated by sets of exceptions and integers respectively. Those sets refine the ML types `exn` and `int` by restricting the values that an expression of type $\texttt{exn}[\varphi]$ or $\texttt{int}[\varphi]$ can have.

Sets of exceptions or integers are represented by *rows* similar to those used for typing extensible records [Wand 1987; Rémy 1989; 1993b]. A row is either $\top$, meaning that all values of the type are possible (we do not have any more precise information), or a sequence of row elements $\varepsilon_1 \ldots \varepsilon_n$ terminated by a *row variable* ρ. We impose the following equational theory on rows to express that the order of elements in a row does not matter (equation 1), and that $\top$ is absorbing (equation 2):

$$
\begin{aligned}
\varepsilon_1;\, \varepsilon_2;\, \varphi &= \varepsilon_2;\, \varepsilon_1;\, \varphi && (1)\\
i : \texttt{Pre};\, \top &= \top && (2)
\end{aligned}
$$

The absorption equation 2 applies only to integer row elements because we intend $\top$ to be used only in rows annotating the `int` type. (The kinding rules in section 3.3 enforce this invariant.) A $\top$ symbol is required for base types such as `int`, which have an infinite (or at least very large) signature. It is not required for datatypes such as `exn`, which have a finite signature: a row enumerating all possible constructors can be used instead, as discussed in section 4.1.4 below. Moreover, combining $\top$ and rows containing parameterized constructors raises technical problems[2]; we prefer to avoid the difficulty by restricting $\top$ to rows containing only integer elements.

Rows and row variables support both polymorphism over sets and a form of set union in a unification framework. For instance, the two rows $\varepsilon_1; \rho_1$ and $\varepsilon_2; \rho_2$, which informally represent the sets $\{\varepsilon_1\}$ and $\{\varepsilon_2\}$ respectively, unify into the row $\varepsilon_1; \varepsilon_2; \rho$ representing the set $\{\varepsilon_1; \varepsilon_2\}$ via the substitution $\{\rho_1 \leftarrow (\varepsilon_2; \rho);\ \rho_2 \leftarrow (\varepsilon_1; \rho)\}$.

A row element ε is either an integer constant i, a constant exception constructor C, or a parameterized exception constructor $D(\tau)$ carrying the annotated type τ of its argument. To maintain crucial kinding invariants (see below), the constant row elements (i and C) also carry a *presence annotation*, written π. A presence annotation can be either `Pre`, meaning that the element is present in the set denoted by the row expression; or a *presence variable* δ meaning that the element is actually not present in the set denoted by the row expression, but may be considered as present in order to satisfy unification constraints.

At this point, the reader may wonder about the lack of a row constant $\emptyset$ to denote the empty row, and of a presence annotation `Abs` denoting the absence of a row element. How are we going to express that a function has no effect, or that an integer expression cannot take a particular value? The answer is: by using universally quantified row variables and presence variables that occur only positively in type schemes[3]. For instance, a function of type $\forall\rho.\ \mathtt{int} \xrightarrow{\rho} \mathtt{int}$ cannot raise any exception, and an integer expression of type $\forall\delta.\ \mathtt{int}[0\!:\!\delta;\varphi]$ cannot evaluate to 0, for the same reasons that an expression of type $\forall\alpha.\alpha$ cannot evaluate to a value. This can easily be proved by considering a standard ideal model [MacQueen et al. 1986] for our type algebra.

Here are some examples of type expressions in this algebra. The type $\mathtt{int}[\top]$ denotes all integer values. The type of integer addition is

$$\forall\rho_1,\rho_2,\rho_3,\rho_4.\ \mathtt{int}[\rho_1] \xrightarrow{\rho_2} \mathtt{int}[\rho_3] \xrightarrow{\rho_4} \mathtt{int}[\top]$$

(no effects, no information known on the return value).

The type scheme $\forall\rho.\ \mathtt{int}[1\!:\!\mathtt{Pre};\ 2\!:\!\mathtt{Pre};\ \rho]$ stands for the set $\{1;2\}$ and is the type of integer expressions that can only evaluate to 1 or to 2. As previously mentioned, the universally quantified row variable ρ should be read as denoting the empty set of row elements, since it occurs only positively in the type scheme.

The type scheme $\forall\rho,\delta.\,\mathtt{int}[1\!:\!\delta;\ 2\!:\!\mathtt{Pre};\ \rho]$ stands for the set $\{2\}$. Although 1 is mentioned in the row, it should not be considered present in the set, since its

[2]The obvious absorption equation $D(\tau);\top = \top$ is unsound, as it allows deductions such as $D(\alpha);\top = \top = D(\beta);\top$, which lead to inconsistent typings. If ML had subtyping and a supertype $\top$ of all types, a correct equation would be $D(\top);\ \top = \top$. This equation allows $\top$ to absorb any $D(\tau)$ (because $D(\tau);\top <: D(\top);\top = \top$), but only allows expansion of $\top$ into $D(\top);\ \top$, meaning correctly that no information is available on the argument of D.

[3]The notion of positive and negative occurrences of a variable that we use here is the standard notion from type theory [Girard et al. 1990]. Briefly, if types and type schemes are viewed as trees, a type variable is said to occur negatively in a type scheme if there exists a path from the root of the type scheme to the variable that crosses an arrow type constructor to the left an odd number of times. A variable is said to occur only positively in a type scheme if it does not occur negatively in that scheme.

presence annotation δ is universally quantified and occurs only positively.

The type scheme $\forall\rho,\rho'.\,\mathtt{exn}[D(\mathtt{int}[3:\mathtt{Pre};\ 4:\mathtt{Pre};\ \rho]);\ \rho']$ stands for the set of exceptions $\{D(3); D(4)\}$.

The `raise` predefined function has the following type scheme: $\forall\alpha,\rho.\,\mathtt{exn}[\rho] \xrightarrow{\rho} \alpha$. This scheme captures the fact that an application of `raise` never returns and raises exactly the exceptions that it receives as argument.

3.3 Kinding of rows

To simplify the formulation of the typing rules and to ensure the existence of principal unifiers and principal typings, we require the following four structural invariants on rows:

(1) A given integer constant or exception constructor should occur at most once in a row. For instance, $(D(\tau);\ D(\tau');\ \varphi)$ is not well-formed.

(2) A row variable ρ is preceded by the same set of integer constants and exception constructors in all row expressions where it occurs. For instance, we cannot have both $(1:\mathtt{Pre};\ \rho)$ and $(2:\mathtt{Pre};\ \rho)$ in the same derivation.

(3) A row φ annotating an integer type $\mathtt{int}[\varphi]$ can only contain integer elements i.

(4) A row φ annotating an exception type $\mathtt{exn}[\varphi]$ or a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ can only contain constant or parameterized constructors C, D and must not end with $\top$.

Invariants (1) and (2) are well known from earlier work on record types [Rémy 1993b]. Invariants (3) and (4) are more unusual. They ensure a clear separation between annotations of `int` types (composed of integer elements and possibly $\top$) and annotations of the `exn` types (composed of constructors and no $\top$). Since $\top$ absorbs only integer elements (equation 2), we do not want it to occur in rows containing exception constructors C, D.

Following [Rémy 1993b; Ohori 1995], we use *kinds* to enforce the invariants above. Our kinds κ are composed of a tag (either `INT` or `EXN`) and a set of constants and constructors:

Kinds: $\kappa ::= \mathtt{INT}(\{i_1,\ldots,i_n\}) \mid \mathtt{EXN}(\{C_1,\ldots,C_p,D_1,\ldots,D_q\})$

The constants and constructors appearing in the set part of a kind are those constants and constructors that must not appear in rows of that kind, because they already appear in elements concatenated before these rows. We assume given a global mapping K assigning kinds to row variables, and such that for each κ there are infinitely many variables of that kind (i.e. $K^{-1}(\kappa)$ is infinite). The kinding rules are shown in figure 2. They define the two judgements $\vdash \varphi :: \kappa$ (row φ has kind κ) and $\vdash \tau\ \mathtt{wf}$ (type τ is well-formed).

3.4 The typing rules

Figure 3 shows the typing rules for our system. They define the judgement $E \vdash a : \tau/\varphi$, where E is the typing environment, a the term to type, τ the type of values that a may evaluate to, and φ the set of exceptions that may escape during the evaluation of a. We assume that typing starts in the initial environment $E_0 = \{\mathtt{raise} : \forall\alpha,\rho.\,\mathtt{exn}[\rho] \xrightarrow{\rho} \alpha\}$. We write $E_1 \oplus E_2$ for the asymmetric concatenation of

$$\vdash \rho :: K(\rho) \qquad \vdash \top :: \mathtt{INT}(S) \qquad \frac{i \notin S \qquad \vdash \varphi :: \mathtt{INT}(S \cup \{i\})}{\vdash (i : \pi;\ \varphi) :: \mathtt{INT}(S)}$$

$$\frac{C \notin S \qquad \vdash \varphi :: \mathtt{EXN}(S \cup \{C\})}{\vdash (C : \pi;\ \varphi) :: \mathtt{EXN}(S)} \qquad \frac{D \notin S \qquad \vdash \varphi :: \mathtt{EXN}(S \cup \{D\}) \qquad \vdash \tau\ \mathtt{wf}}{\vdash (D(\tau);\ \varphi) :: \mathtt{EXN}(S)}$$

$$\vdash \alpha\ \mathtt{wf} \qquad \frac{\vdash \varphi :: \mathtt{INT}(\emptyset)}{\vdash \mathtt{int}[\varphi]\ \mathtt{wf}} \qquad \frac{\vdash \varphi :: \mathtt{EXN}(\emptyset)}{\vdash \mathtt{exn}[\varphi]\ \mathtt{wf}}$$

$$\frac{\vdash \tau_1\ \mathtt{wf} \qquad \vdash \varphi :: \mathtt{EXN}(\emptyset) \qquad \vdash \tau_2\ \mathtt{wf}}{\vdash \tau_1 \xrightarrow{\varphi} \tau_2\ \mathtt{wf}}$$

Fig. 2. Kinding rules

E_1 and E_2; that is, $(E_1 \oplus E_2)(x) = E_2(x)$ if $x \in \mathrm{Dom}(E_2)$, and $(E_1 \oplus E_2)(x) = E_1(x)$ if $x \in \mathrm{Dom}(E_1) \setminus \mathrm{Dom}(E_2)$.

The rules for variables and `let` bindings (rules 1 and 5) are standard, except that we generalize over all three kinds of type variables. (The instantiation and generalization predicates are defined in figure 3.) For variables as well as other language constructs that never raise exceptions (rules 1, 2, 3, 7), the φ component of the result is unconstrained and can be chosen as needed to satisfy equality constraints in the remainder of the typing derivation.

The rule for function abstraction (rule 3) is the usual rule for effect systems: the effect of the function body becomes the latent effect of the function type. For applications $a_1(a_2)$ (rule 4), the usual approach is to take as effect of the application the union of the effect of a_1, latent effect of the function denoted by a_1, and effect of a_2. Since our algebra of effects lacks an union constructor, we approximate the union by requiring that those three effects (effect of a_1, latent effect of a_1, effect of a_2) are equal to the same set φ of exception. In our unification-based type inference algorithm, this corresponds simply to unifying these three effects.

For integer constants and exception constructors (rules 2, 7 and 8), we record the actual value of the expression in the approximation part of the type `int` or `exn`. For instance, the type of i must be of the form $\mathtt{int}[i : \mathtt{Pre}; \varphi]$, forcing $i : \mathtt{Pre}$ to appear in the type of the expression. In rules 8 and 13, we write $\mathit{TypeArg}(D)$ for the type scheme of the argument of constructor D, e.g. $\mathit{TypeArg}(D) = \forall \rho.\ \mathtt{int}[\rho]$ for an integer-valued exception D.

For an exception handler $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$ (rule 9), the effect φ_1 of a_1 is injected in the type $\mathtt{exn}[\varphi_1]$ assumed for x in a_2.

The most interesting rule is rule 6 for the `match` construct. This rule is crucial to the precision of our exception analysis. When typing $\mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3$, we want to reflect the fact that the second alternative ($x \to a_3$) is selected only when the first alternative ($p \to a_2$) does not match the value of a_1. In other terms, the type of values that can "flow" to x in the second alternative is not the type of the matched value a_1, but the type of a_1 from which we have excluded all values matching the pattern p in the first alternative.

Typing of expressions:

$$\frac{\tau \le E(x)}{E \vdash x : \tau}\ (1) \qquad \frac{\vdash \varphi' :: \mathtt{INT}(\{i\}) \quad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash i : \mathtt{int}[i : \mathtt{Pre};\ \varphi']/\varphi}\ (2)$$

$$\frac{\vdash \tau_1\ \mathtt{wf} \quad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \quad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash \lambda x.\, a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi}\ (3)$$

$$\frac{E \vdash a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \quad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi}\ (4)$$

$$\frac{E \vdash a_1 : \tau_1/\varphi \quad E \oplus \{x : \mathtt{Gen}(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau/\varphi}\ (5)$$

$$\frac{\begin{array}{c} E \vdash a_1 : \tau_1/\varphi \quad \vdash p : \tau_1 \Rightarrow E' \quad \vdash \tau_1 - p \leadsto \tau_2 \\ E \oplus E' \vdash a_2 : \tau/\varphi \quad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash \mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3 : \tau/\varphi}\ (6)$$

$$\frac{\vdash \varphi' :: \mathtt{EXN}(\{C\}) \quad \vdash \varphi :: \mathtt{EXN}(\emptyset)}{E \vdash C : \mathtt{exn}[C : \mathtt{Pre};\ \varphi']/\varphi}\ (7)$$

$$\frac{\tau \le \mathit{TypeArg}(D) \quad E \vdash a : \tau/\varphi \quad \vdash \varphi' :: \mathtt{EXN}(\{D\})}{E \vdash D(a) : \mathtt{exn}[D(\tau);\ \varphi']/\varphi}\ (8)$$

$$\frac{E \vdash a_1 : \tau/\varphi_1 \quad E \oplus \{x : \mathtt{exn}[\varphi_1]\} \vdash a_2 : \tau/\varphi}{E \vdash \mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2 : \tau/\varphi}\ (9)$$

Typing of patterns:

$$\vdash x : \tau \Rightarrow \{x : \tau\}\ (10) \qquad \vdash i : \mathtt{int}[i : \pi; \varphi] \Rightarrow \{\}\ (11)$$

$$\vdash C : \mathtt{exn}[C : \pi; \varphi] \Rightarrow \{\}\ (12) \qquad \frac{\tau \le \mathit{TypeArg}(D) \quad \vdash p : \tau \Rightarrow E}{\vdash D(p) : \mathtt{exn}[D(\tau); \varphi] \Rightarrow E}\ (13)$$

Pattern subtraction:

$$\vdash \mathtt{int}[i : \pi; \varphi] - i \leadsto \mathtt{int}[i : \pi'; \varphi]\ (14)$$

$$\vdash \mathtt{exn}[C : \pi; \varphi] - C \leadsto \mathtt{exn}[C : \pi'; \varphi]\ (15)$$

$$\frac{\vdash \tau'\ \mathtt{wf}}{\vdash \tau - x \leadsto \tau'}\ (16) \qquad \frac{\vdash \tau - p \leadsto \tau'}{\vdash \mathtt{exn}[D(\tau); \varphi] - D(p) \leadsto \mathtt{exn}[D(\tau'); \varphi]}\ (17)$$

Instantiation and generalization:

$\tau' \le \forall \alpha_i \rho_j \delta_k.\, \tau$ if and only if there exists τ_i, φ_j, π_k such that $\vdash \tau_i$ $\mathtt{wf}$ and $\vdash \varphi_j :: K(\rho_j)$ and $\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$

$\mathtt{Gen}(\tau, E, \varphi)$ is $\forall \alpha_i \rho_j \delta_k.\, \tau$ where $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$.

Fig. 3. The typing rules

To achieve this, rules 14–17 define the pattern subtraction predicate $\vdash \tau - p \rightsquigarrow \tau'$, meaning that τ' is a correct type for the values of type τ that do not match pattern p. For a variable pattern $p = x$ (rule 16), all values match the pattern, so it is correct to assume any τ' for the type of the non-matched values. For an integer pattern $p = i$ (rule 14), we force τ to unify with $\mathtt{int}[i : \pi; \varphi]$, thus exposing in φ the set of all possible values of type τ that are different from i. Then, we take $\tau' = \mathtt{int}[i : \pi'; \varphi]$ for a suitable π'. In particular, if that π' is unconstrained in the remainder of the derivation, we can take π' to be a fresh presence variable δ, thus reflecting that i is not among the possible values of type τ'. The rules for exception patterns (rules 15 and 17) are similar. If the exception has an argument, instead of changing a presence annotation, we recursively subtract the type of the argument of the exception.

3.5 Examples of typings

We now show some typings derivable in our system. These are principal typings identical to those found by our exception analyzer. Consider first a simple handler for one exception `C`.

```
try raise(C)
with x → match x with C → 1 | y → raise y
```

The effect of `raise(C)` is $\mathtt{C} : \mathtt{Pre};\ \rho$. Hence, the type of `x` is $\mathtt{exn}[\mathtt{C} : \mathtt{Pre};\ \rho]$. Subtracting the pattern `C` from this type, we obtain the type $\mathtt{exn}[\mathtt{C} : \delta;\ \rho]$ for `y`. Hence the effect of the whole `match` expression, and also of the whole `try` expression, is $\mathtt{C} : \delta;\ \rho$. The type is $\mathtt{int}[1 : \mathtt{Pre};\ \rho']$. Since δ, ρ and ρ' are generalizable and occur only positively, we have established that no exception escapes the expression, and that it can only evaluate to the integer 1.

We now extend the previous example along the lines of the `failwith` example of section 2.3.

```
let failwith = λn. raise(D(n)) in
try failwith(42)
with x → match x with D(42) → 0 | y → raise y
```

We obtain the following intermediate typings:

$$\begin{aligned} \mathtt{failwith} &: \forall \alpha, \rho_1, \rho_2.\, \mathtt{int}[\rho_1] \xrightarrow{D(\mathtt{int}[\rho_1]);\rho_2} \alpha \\ \mathtt{x} &: \mathtt{exn}[D(\mathtt{int}[42 : \mathtt{Pre};\ \rho_3]); \rho_4] \\ \mathtt{y} &: \mathtt{exn}[D(\mathtt{int}[42 : \delta;\ \rho_3]); \rho_4] \end{aligned}$$

Thus we conclude as before that no exception escapes this expression.

For a representative example of higher-order functions, consider function composition:

```
let compose = λf. λg. λx. f(g(x)) in
compose (λy. 0) (λz. raise(C)) 1
```

The type scheme for `compose` is $\forall \alpha, \beta, \gamma, \rho, \rho', \rho''.\ (\alpha \xrightarrow{\rho} \beta) \xrightarrow{\rho'} (\gamma \xrightarrow{\rho} \alpha) \xrightarrow{\rho''} \gamma \xrightarrow{\rho} \beta$. The three occurrences of ρ express the union of the effects of `f` and `g`. The application of `compose` above has effect $\mathtt{C} : \mathtt{Pre};\ \rho_3$.

Concerning exceptions as first-class values, the first example from section 2.2 becomes:

```
let test =
  λexn. try raise(exn)
        with x → match x with C → 1 | y → raise(y)
in test(C)
```

The type scheme for `test` is $\forall \rho, \rho', \delta.\ \mathtt{exn}[\mathtt{C}:\mathtt{Pre};\ \rho] \xrightarrow{\mathtt{C}:\delta;\ \rho} \mathtt{int}[1:\mathtt{Pre};\ \rho']$, expressing that the function raises whatever exception it receives as argument, except `C`. The application `test(C)` has thus type $\mathtt{int}[1:\mathtt{Pre};\ \rho_1]$ and effect $\mathtt{C}:\delta_2;\ \rho_2$. Hence no exception escapes. The application `test(E)` where `E` is another exception distinct from `C` would have effect $\mathtt{C}:\delta_3;\ \mathtt{E}:\mathtt{Pre};\ \rho_3$, thus showing that `E` may escape.

Finally, here is an (anecdotal) example that is ill-typed in ML, but well-typed in our type system due to the refined typing of pattern-matching.

```
match 1 with x -> x | e -> raise e
```

Since the first case of the matching is a catch-all, rule 6 lets us assign the type $\mathtt{exn}[\rho']$ for a fresh ρ' to the variable `e` bound by the second case, even though the matched value is an integer. Hence the expression is well-typed, and moreover we obtain that it has type $\mathtt{int}[1:\mathtt{Pre}; \rho]$ and raises no exceptions (its effect is $\forall \rho'.\rho'$).

3.6 Type soundness and correctness of the exception analysis

We now establish the correctness of our exception analysis: all uncaught exceptions are predicted by our effect system. This property is closely connected to the type soundness of our system.

THEOREM 1. *(Subject reduction.) If $E_0 \vdash a : \tau/\varphi$ and $a \Rightarrow a'$, then $E_0 \vdash a' : \tau/\varphi$*

The proof of this theorem, as well as all other theorems in this section, is given in appendix C. A key lemma is the following property of pattern subtraction.

LEMMA 2. *(Correctness of subtraction.) If $E_0 \vdash v : \tau/\varphi$ and $M(v,p)$ is undefined (v does not match pattern p, as defined in figure 1) and $\vdash \tau - p \rightsquigarrow \tau'$, then $E_0 \vdash v : \tau'/\varphi$.*

The correctness of our exception analysis (all uncaught exceptions are detected) is a simple corollary of subject reduction.

THEOREM 3. *(Correctness of exception analysis.) Let a be a complete program. Assume $E_0 \vdash a : \tau/\varphi$ and $a \stackrel{*}{\Rightarrow}$ `raise` v. Then, either $v = C$ and $\varphi = C:\mathtt{Pre};\ \varphi'$ for some C and φ', or $v = D(v')$ and $\varphi = D(\tau'); \varphi'$ and $E_0 \vdash v' : \tau'/\varphi$ for some D, v', τ', φ'. In either case, the uncaught exception v is correctly predicted in the effect φ.*

Type soundness for our non-standard type system follows from the subject reduction property and the following lemma showing that a well-typed expression either reduces to a value or to an uncaught exception, or loops, but never gets "stuck".

LEMMA 4. *(Progress.) If $E_0 \vdash a : \tau/\varphi$, then either a is a value v, or a is an uncaught exception* `raise` *v, or there exists a' such that $a \Rightarrow a'$.*

3.7 Principal types and inference of types and exceptions

Just like the ML type system, our type system admits principal types, which can be computed by a simple extension of the Damas-Milner algorithm, thus implementing the exception analysis. The inference algorithm is shown in appendix B, along with the associated unification algorithm in appendix A. The existence of principal unifiers follows from the fact that our equational theory is syntactic and regular [Rémy 1993a].

THEOREM 5. *(Principal types.) There exists a type inference algorithm I operating on closed terms a that satisfies the following conditions:*

—*(Correctness) If $(\tau, \varphi) = I(a)$ is defined, then $\emptyset \vdash a : \tau/\varphi$.*

—*(Completeness) If there exists a type τ' and a row φ' such that $\emptyset \vdash a : \tau'/\varphi'$, then $(\tau, \varphi) = I(a)$ is defined and there exists a substitution ψ such that $\tau' = \psi(\tau)$ and $\varphi' = \psi(\varphi)$.*

4. EXTENSION TO THE FULL OBJECTIVE CAML LANGUAGE

In this section, we discuss the main issues in extending the analysis presented in section 3 to deal with the whole Objective Caml language [Leroy et al. 1996].

4.1 Datatypes

User-defined datatypes (sum types) can be approximated in several different ways, depending on the desired trade-off between precision and speed of the analysis. We have considered the four approaches listed below (from most precise to least precise) and illustrated in figure 4.

4.1.1 *Full approximation of datatypes.* The first approach applies to datatypes the same treatments as for exceptions: we annotate the type by a row φ approximating the possible values of that type, as constant constructors with presence annotations, and unary constructors with types of arguments. Consider the source-level datatype definition

$$\texttt{type}\ \vec{\alpha}\ t = C_1 \mid \ldots \mid C_n \mid D_1\ \texttt{of}\ \nu_1 \mid \ldots \mid D_m\ \texttt{of}\ \nu_m$$

where the ν_i are unannotated ML types. The propagation of approximations is captured by the following type schemes assigned to the constructors C_i and D_i:

$$C_i\ :\ \forall \vec{\alpha}, \rho'.\ \vec{\alpha}\ t\,[C_i : \texttt{Pre};\ \rho']$$

$$D_i\ :\ \forall \vec{\alpha}, \vec{\rho}, \rho', \rho''.\ \tau_i \xrightarrow{\rho''} \vec{\alpha}\ t\,[D_i(\tau_i);\ \rho']$$

where τ_i is the annotated type obtained from ν_i by adding distinct fresh row variables taken from $\vec{\rho}$ on every type constructor that carries a row annotation. For instance, given the declaration

```
type intlist = Nil | Cons of int * intlist
```

we assign `Nil` and `Cons` the type schemes

$$\texttt{Nil}\ :\ \forall \rho.\ \texttt{intlist}[\texttt{Nil}:\texttt{Pre};\ \rho]$$

$$\texttt{Cons}\ :\ \forall \rho_1, \rho_2, \rho_3, \rho_4.\ \texttt{int}[\rho_1] \times \texttt{intlist}[\rho_2] \xrightarrow{\rho_3} \texttt{intlist}[\texttt{Cons}(\texttt{int}[\rho_1] \times \texttt{intlist}[\rho_2]); \rho_4]$$

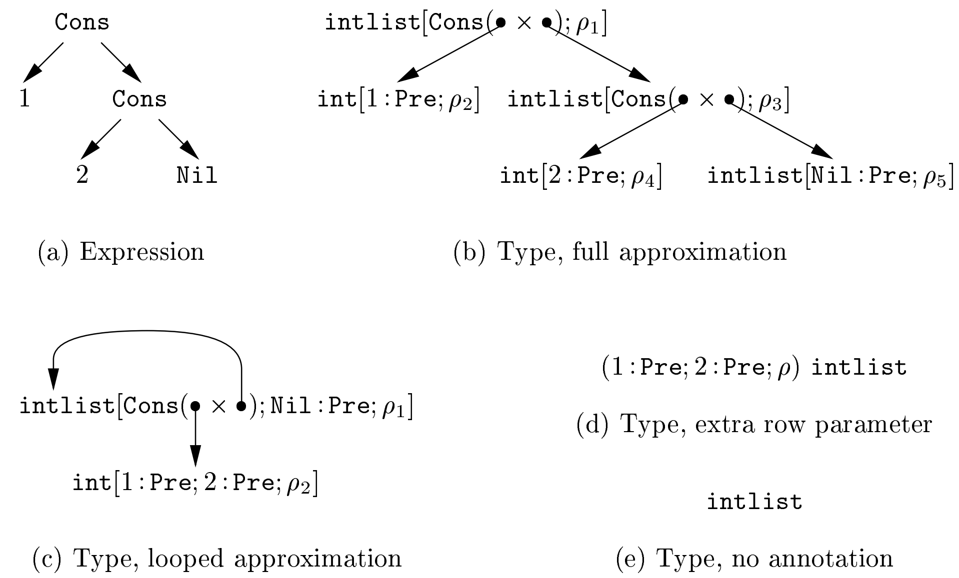

Fig. 4. Examples of data type approximations for the datatype `intlist = Nil | Cons of int * intlist`

Recursive datatypes such as `intlist` above naturally lead to recursive type expressions. Consider:

```
let tail = λx. match x with Cons(hd,tl) → tl | l → l
```

During inference, `tl` and `l` receive types $\mathtt{intlist}[\rho_1]$ and $\mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[\rho_2] \times \mathtt{intlist}[\rho_1]); \rho_3]$ respectively. If only finite type expressions are allowed, those two types have no unifier and the program is rejected by the analysis. This is not acceptable, so we extend our type system with recursive type expressions, that is, type expressions that are infinite but regular. On the example above, we obtain the recursive type $\mu\alpha.\, \mathtt{intlist}[\mathtt{Cons}(\mathtt{int}[\rho_2] \times \alpha); \rho_3]$ for the result of `tail`. The extension of our type system with recursive type expressions involves replacing term unification by graph unification in the type inference algorithm. This causes no algorithmic difficulties, but we have not extended our proofs to the case of recursive type expressions.

4.1.2 *"Looped" approximations for recursive datatypes.* The approximation scheme described above has the undesirable side-effect of recording in the type approximation the whole structure of a data structure given in extension. If the data types involved are recursive, we may end up with very large type approximations. Continuing the `intlist` example above, consider the expression

$$\ell_n = \mathtt{Cons}(i_1, \mathtt{Cons}(i_2, \ldots, \mathtt{Cons}(i_n, \mathtt{Nil}) \ldots)).$$

With the type of `Cons` given in section 4.1.1, this expression is given an annotated type that is of depth n and records not only the fact that the list contains the integers $i_1 \ldots i_n$ (an information that might be useful to analyze exceptions), but also the fact that the list has length n and that its first element is i_1, the second i_2, etc. (See figure 4b.) The latter piece of information is, on practical examples, useless for analyzing exceptions. Moreover, such large approximations slow down the analysis.

A solution to this problem comes from the following remark: as soon as one of those big data structures given in extension is passed to a sufficiently complex function, its big, unfolded annotated type is going to be unified with a recursive type, forcing all the information in the big type to be folded back into a smaller recursive type. For instance, if we pass the list ℓ_n to the `tail` function shown above, the type of the list will be unified into

$$\tau_n = \mu\alpha.\ \texttt{intlist}[\texttt{Cons}(\texttt{int}[i_1:\texttt{Pre};\ldots;i_n:\texttt{Pre};\rho_1]\times\alpha);\texttt{Nil}:\texttt{Pre};\rho_2].$$

The idea, then, is to force this folding into a recursive type when the data structure is created, by giving recursive, pre-folded types to the data type constructors. This is easily achieved by unifying, in the type of the constructors, all occurrences of the recursively-defined type in argument position with the occurrence of the recursively-defined type in result position. For instance, in the case of the `Cons` constructor of type `intlist`, we start with the type

$$\texttt{int}[\rho_1]\times\underline{\texttt{intlist}[\rho_2]}\xrightarrow{\rho_3}\underline{\texttt{intlist}[\texttt{Cons}(\texttt{int}[\rho_1]\times\texttt{intlist}[\rho_2]);\rho_4]}$$

as in section 4.1.1, then unify the two underlined `intlist` types, then generalize the free variables, obtaining `Cons` : $\forall\rho_1,\rho_3,\rho_4.\ \texttt{int}[\rho_1]\times\tau\xrightarrow{\rho_3}\tau$ where τ is $\mu\alpha.\texttt{intlist}[\texttt{Cons}(\texttt{int}[\rho_1]\times\alpha);\rho_4]$. With this type for `Cons`, the list ℓ_n is given the reasonably compact type τ_n shown above.

This technique of "looping" the types of constructors also works for parameterized datatypes, as long as they are regular (the data type constructor is used with the same parameters in the argument types of the constructors). For non-regular datatypes such as

```
type 'a nonreg = Leaf of 'a | Node of 'a list nonreg
```

the unification of the occurrences of `nonreg` in the type of `Node` would render that constructor essentially useless. Fortunately, such non-regular data types are extremely rare in actual programs, so we can use full approximations for them without impacting performance.

4.1.3 *Adding row parameters to datatypes.* An alternative to annotating datatype constructors with rows is to add row parameters to the type constructor reflecting the row annotations on `exn`, `int` and function types contained within the datatype. This technique is used by Fähndrich *et al* [1998]. For instance, the ML datatype definition

```
type t = A of int | B of exn | C of t
```

is turned into

$$\texttt{type}\ (\rho_1,\rho_2)\ \texttt{t = A of int}[\rho_1]\ \texttt{| B of exn}[\rho_2]\ \texttt{| C of}\ (\rho_1,\rho_2)\ \texttt{t}$$

Two parameters ρ_1 and ρ_2 were added in order to reflect in the type `t` the possible values of types `int` and `exn` contained in that type. The type `t` itself is not annotated by a row recording which constructors `A`, `B` or `C` are present in values of that type. The net effect is to forget the structure of terms of type `t`, while correctly remembering the integers and exception values contained in the structure.

In practice, this solution appears to be slightly less precise and slightly more efficient than full approximations of non-recursive datatypes and looped approximations of recursive datatypes: type expressions are smaller, but in the case of `t` above, looped approximations can express the fact that a value of type `t` lacks the constructor `C`, while this is not captured in the solution based on extra row parameters.

On datatypes that are not annotated by a row, we can no longer perform type subtraction during pattern-matching, since we have no approximation on the structure of values of that type. Hence, we simply consider that subtraction is the identity relation on those datatypes.

4.1.4 *Datatypes without any approximations.* For maximal speed and minimal precision, we can put no annotations at all on a datatype: neither a row approximation nor extra row parameters. This way, we forget not only the structure of values of that type, but also the exceptions, functions and base values contained in that type. Of course, this forces us to make very pessimistic assumptions on values extracted from a datatype without approximation. For instance, if we extract an integer by pattern-matching on such a datatype, we must give it type $\mathtt{int}[\top]$ since it can really be any integer. This is reflected in the types of constructors by putting $\top$ annotations on all annotated types in the constructor argument. In the `intlist` example above, if we choose not to annotate `intlist` at all, we must give its constructors the following types:

$$\begin{aligned} \mathtt{Nil} &: \mathtt{intlist} \\ \mathtt{Cons} &: \forall\rho.\ \mathtt{int}[\top] \times \mathtt{intlist} \xrightarrow{\rho} \mathtt{intlist} \end{aligned}$$

This approach assumes that we have $\top$ annotations for all types, while the type system from section 3 only has $\top$ for type `int`. However, we can allow $\top$ to annotate other base types such as `float` and `string`. For exceptions and other datatypes, since there are finitely many constructors, we can use a (potentially recursive) row enumerating all constructors of the datatype instead of a built-in constant $\top$. In the case of lists, for instance, we can use the following "top row" $\top_{\mathtt{list}}(\alpha, \rho)$:

$$\top_{\mathtt{list}}(\alpha, \rho) = \mu\rho'.\ \mathtt{Nil}:\mathtt{Pre}; \mathtt{Cons}(\alpha \times \alpha\ \mathtt{list}[\rho']); \rho$$

The annotated type $\tau\ \mathtt{list}[\top_{\mathtt{list}}(\tau, \rho)]$ correctly represents any list of elements of type τ.

The "no approximation" approach described in this paragraph may look excessively coarse, but is actually quite effective for datatypes that introduce no base types, nor exception types, nor function types. Prominent examples are the built-in ML types α `list` and α `array`, where the α parameter already records all the information we need about list and array elements. For instance, a list of functions from integers to booleans has type $(\mathtt{int}[\varphi_1] \xrightarrow{\varphi_2} \mathtt{bool}[\varphi_3])\ \mathtt{list}$, where φ_2 denotes the union of the effects of all functions present in the list. A function extracted from that list and applied has effect φ_2, and not any exception as one might naively expect.

4.1.5 *Choosing a datatype approximation.* The choice between the four datatype analysis strategies described above can be done on a per-datatype basis, depending on the shape of the datatype definition. We have considered several simple heuristics to perform this choice. Our first prototype used full approximations for non-parameterized datatypes, and no approximations for parameterized datatypes. Our current prototype uses full approximations for non-recursive or non-regular datatypes, looped approximations for recursive datatypes, and no approximations for built-in types without interesting structure (arrays and floating-point numbers, for instance). Another factor that we plan to integrate in the heuristic is whether the datatype introduces any exception type, function type, or base type likely to be an exception argument (`string` and `int`, essentially); if not, we could favor the "no approximation" approach.

4.2 Tuples and records

Tuple types are not approximated specially: each component of the tuple type carries its own annotation. For instance, $\texttt{int}[1:\texttt{Pre};\ 2:\texttt{Pre};\ \rho] \times \texttt{int}[3:\texttt{Pre};\ 4:\texttt{Pre};\ \rho']$ stands for the set of four pairs $\{1;2\} \times \{3;4\}$. Pattern subtraction on tuple types is not pointwise subtraction, which would lead to incorrect results. Consider the type $\texttt{int}[1:\texttt{Pre};\ \rho] \times \texttt{int}[2:\texttt{Pre};\ 3:\texttt{Pre};\ \rho']$. Subtracting pointwise the pattern $(1,2)$ from this type would lead to type $\texttt{int}[1:\delta;\ \rho] \times \texttt{int}[2:\delta';\ 3:\texttt{Pre};\ \rho']$, which is incorrect since the value $(1,3)$ is no longer in the set. Therefore, the current implementation perform no subtraction on tuples: we take $\vdash (\tau_1 \times \tau_2) - (p_1, p_2) \rightsquigarrow \tau_1 \times \tau_2$. For a more refined behavior, we could perform subtraction on one of the components if all other components are matched against catch-all patterns. For instance, we could take $\vdash (\tau_1 \times \tau_2) - (p_1, x_2) \rightsquigarrow \tau_1' \times \tau_2$ if $\vdash \tau_1 - p_1 \rightsquigarrow \tau_1'$.

Unlike in SML, records in Caml are declared and matched by name. We analyze them like datatypes, by annotating the name of the record type by a row of a particular form. The row contains exactly one element recording the annotated type of every field. Pattern subtraction for record types behaves as in the case of tuples.

To summarize, the extended type algebra for datatypes, tuples and records is as follows:

Type expressions:	$\tau ::= \ldots$	
	$\mid \vec{\tau}\ \texttt{t}[\varphi]$	approximated type constructor
	$\mid \vec{\tau}\ \texttt{t}$	non-approximated type constructor
	$\mid \tau_1 \times \ldots \times \tau_n$	tuple type
Row elements:	$\varepsilon ::= \ldots \mid \{lbl_1 : \tau_1; \ldots; lbl_n : \tau_n\}$	

4.3 Mutable data structures

Mutable data structures (references, arrays, records with mutable fields) are trivially handled: it suffices to introduce the standard value restriction on `let`-generalization [Wright 1995]. This results in a precise approximation of mutable data. For instance, an array of functions has type $(\tau_1 \xrightarrow{\varphi} \tau_2)$ `array`, where φ is the union of the latent effects of all functions stored in the array. In contrast, control-flow analyses would lose track of which functions are stored in the array, and thus also of the exceptions they may raise, unless supplemented by a region analysis (aliasing analysis).

4.4 Objects and classes

Because our system already uses recursive types, OCaml-style objects do not add significant complexity to our framework. We just need to extend the type algebra with object types, that is, polymorphic records of methods [Rémy and Vouillon 1998]. The type of each method is annotated by its latent effect. No extension to rows and row elements are needed. Since there are no object patterns in pattern-matching, pattern subtraction needs not be modified.

The OCaml class language interferes very little with the exception analysis. No significant modifications to the class type-checker are needed.

4.5 Modules and functors

Structures are assigned annotated signatures containing annotated types for the value components. Type abbreviations are currently handled by systematic expansion of their definitions[4].

For matching a structure S against a signature Σ, there are two possible semantics. The opaque semantics says that the only things known about the restriction $(S : \Sigma)$ is what Σ publicizes. In our case, since user-provided signatures Σ contain no annotations, this amounts to forgetting the result of the analysis of S and assume $\top$ annotation on all value components of the restricted structure. The transparent semantics simply check that S matches Σ, but the restriction $(S : \Sigma)$ retains all information known about S. We implemented the transparent semantics, as the opaque semantics results in too much information loss. (The opaque semantics also precludes choosing datatype annotations based on the definition of the datatype.)

Similar problems arise with functors. All is known about the parameter of a functor is its syntactic signature. Hence, a naive analysis would assume $\top$ annotation on all components of the functor argument. For better precision, one could use techniques based on conjunctive types such as [Shao and Appel 1993]. Other issues with functors are still unclear, such as the generativity of exception declaration in functor bodies, and the impact of the "exception polymorphism" offered by functors (a functor can take one or several exceptions as arguments, and have a different exception behavior depending on whether those arguments are instantiated later with identical or different exceptions).

For simplicity, we chose not to analyze functors when they are defined, but instead expand the functor body at each application and re-analyze the β-reduced body. Although this transformation increases the size of the analyzed source, the Caml programs we are interested in use only small functors and this simple approach to analyzing functors works well in practice.

[4]This might cause performance problems in conjunction with OCaml objects, which relies intensively on type abbreviations to make type expressions more manageable [Rémy and Vouillon 1998]. If this turns out to be a problem, we could also handle abbreviations by adding extra row parameters to the type constructors, as described in [Fähndrich et al. 1998] and in section 4.1.3.

4.6 Separate analysis

Transparent signature matching precludes "true" separate analysis, where any module can be analyzed separately knowing only the syntactic signatures of the modules it imports. We can still do "bottom-up" separate analysis, however: a module can be analyzed separately provided the implementations of its imports have been analyzed already, and their annotated signatures inferred.

Since an annotated signature for a module may contain free row variables (e.g. if the module defines mutable structures), separately analyzing several clients of that module may result in independent instantiations of those free variables. Those instantiations are recorded in the result of the analysis of each module, and reconciled in a final "linking" pass before displaying the results of the analysis.

4.7 Polymorphic recursion

Polymorphic recursion as introduced by Mycroft [1984] is not needed to type-check the source OCaml language, but is desirable to enhance the precision of our exception analyzer. With ML-style monomorphic recursion, we obtain false positives on functions that recursively call themselves inside a `try...with`. Consider:

```
let rec f =
  λx. try if ... then raise(C) else f(x)
      with C → () | y → raise y
```

The latent effect inferred for `f` is $\mathtt{C};\rho$ because the effect of `f(x)` is unified with that of `raise(C)` at a time where the type of `f` is not yet generalized. With polymorphic recursion, we can assign `f` the type scheme $\forall\alpha,\rho.\,\alpha \xrightarrow{\rho} \mathtt{unit}$ both outside and inside the recursion; it is a fresh instance of that type scheme that gets unified with the effect of `raise(C)`, thus not polluting the type scheme of `f`.

Although type inference with polymorphic recursion is undecidable [Kfoury et al. 1993], there exists semi-algorithms that work very well in practice, such as Henglein's semi-algorithm [Henglein 1993]. We experimented with a home-grown incomplete algorithm based on restricted fixpoint iteration, which always terminates but may return non-principal types, and obtained good results. This algorithm is described in the second author's PhD thesis [Pessaux 1999].

5. EXPERIMENTAL RESULTS

In this section, we present some experimental results obtained with our implementation. Currently, our analyzer implements all extensions described in section 4 except objects[5]. The analyzer is compiled with the OCaml 2.00 native-code compiler and runs on a Pentium II 333 Mhz workstation under Linux.

5.1 Analysis speed

Figure 5 gives timings for the analysis of various small to medium-sized OCaml programs. We give timings both without and with polymorphic recursion. For comparison, we also give the time OCaml takes to parse and type-check those

[5]The analysis of objects and classes was prototyped separately and remains to be merged in our main implementation.

Test program	Size (lines)	Analysis time	Analysis speed (lines/sec.)	Typing time
1. Huffman compression	233	0.07/0.08 s	3300/2900 l/s	0.08 s
2. Knuth-Bendix	441	0.14/0.16 s	3200/2800 l/s	0.14 s
3. Docteur (Eliza clone)	556	0.81/0.83 s	680/670 l/s	0.10 s
4. Lexer generator	1169	0.27/0.32 s	4300/3700 l/s	0.20 s
5. Nucleic	2919	1.90/1.88 s	1530/1550 l/s	0.62 s
6. OCaml standard library	3082	2.52/2.52 s	1200/1200 l/s	1.89 s
7. Analyzer of `.h` files	3088	0.54/0.58 s	5700/5300 l/s	0.27 s
8. Our exception analyzer	12235	10.3/16.1 s	1200/760 l/s	3.86 s
9. OCaml bytecode compiler	17439	12.6/22.9 s	1400/760 l/s	4.00 s

Fig. 5. Experimental results (without polymorphic recursion/with polymorphic recursion)

programs. (The analysis times given include parsing and pre-processing as well as analysis.)

The overall performances are quite good, in the order of 1000–2000 lines of source per second. Programs that contain large data structures given in extension (Nucleic, Docteur) take longer to analyze due to the large size of the rows annotating the types of those data structures. On average, the exception analysis takes twice as much time as OCaml type inference; the ratio ranges between 1 (on simple programs) and 8 (on Docteur, because of the large constant data structures). Polymorphic recursion increases the analysis time by a factor of 1.5 on benchmark 8 and 1.8 on benchmark 9, but has negligible impact on the other benchmarks. The slowdown remains acceptable compared with the increase in precision.

5.2 Precision of the analysis

Generally speaking, exceptions reported as escaping by our analyzer fall in four classes:

—True positives: these are exceptions that can actually escape during an execution of the program. These indicate potential errors in the program, and require programmer intervention.

—True negatives: a consequence of using presence annotations in rows is that the analysis can also display exceptions that are raised in the program, but provably always handled. The programmer can be confident that those exceptions are correctly treated in the program.

—Unavoidable false positives: these exceptions cannot actually escape during any execution of the program, but discovering this fact is beyond the aims of our analysis. Typical examples are the exceptions raised on a division by zero or an out-of-bounds array access: our analysis assumes that those exceptions can always be raised by a division or an array access, although the structure of the program may be such that the divisor is never null and the array index is always within bounds. Removing those false positives requires either extra analyses, programmer-supplied invariants, or even general program proof.

—Avoidable false positives: these false positives result from a lack of precision in our analysis, and could conceivably be avoided with a more precise tracking of the flow of values and exceptions. Some of those false positives are caused by the bidirectional flow of information inherent in our unification-based analysis; others correspond to insufficiently polymorphic typing of recursive definitions.

We have manually inspected the output of the analyzer on our benchmark programs. Programs 1, 3, 4, 5 and 7 have a relatively simple exception behavior, and our analysis reports no avoidable false positives for those programs, but only true positives and "division by zero" and "array bound error" exceptions.

For Knuth-Bendix, which has a quite complicated exception structure, 8 exceptions (`Failure` with 8 different string arguments) appearing in the source are correctly reported as non-escaping; 7 exceptions (one `Invalid_argument` and 6 `Failure`) are reported as potentially escaping, and can actually occur in some circumstances. Without polymorphic recursion, the analysis reports two false positives (one `Not_found` and one `Failure`), which correspond to recursive functions containing `try` ... `with` around recursive calls. Adding polymorphic recursion as discussed in section 4.7 removes one of those false positives. The other one is still there, because our incomplete inference algorithm for polymorphic recursion fails to give a type polymorphic enough to one of the functions. We believe the inference algorithm could be strengthened to eliminate the other false positive as well.

The larger examples 8 and 9 exhibit another source of avoidable false positives: mutable data structures (references and arrays) containing functions. As mentioned in section 4.3, the row variables appearing in approximations of mutable data structures are not generalized, hence "collect" all exceptions at their use sites. For instance:

```
let r = ref(λx. ...) in
let f = λy. if cond then !r y else raise(C)
in !r 0
```

The body of `let r` is typed under the initial assumption that `r` has type $\texttt{int} \xrightarrow{\rho} \texttt{int}$ where ρ is not generalized. When typing `f`, the effect of `raise C` is unified with that of `!r y`, hence ρ becomes $\texttt{C} : \texttt{Pre}; \rho'$ and the application `!r 0` appears to raise `C`.

6. RELATED WORK

6.1 Exception analyses for ML

Several exception analyses for ML are described in the literature. Guzmán and Suárez [1994] develop a simple type and effect system to keep track of escaping exceptions. Their system does not handle exceptions as first-class values, nor exceptions carrying arguments. An effect system with the same characteristics is presented in section 5.4.2 of [Nielson et al. 1999]. The first exception analysis proposed by Yi [1998] is based on general abstract interpretation techniques, and runs too slowly to be usable in practice. Later, Yi and Ryu [1997] developed a more efficient analysis roughly equivalent to a conventional control-flow analysis to approximate the call graph and the values of exceptions, followed by a data-flow analysis to estimate uncaught exceptions.

Fähndrich and Aiken [1997; 1998] have applied their BANE toolkit for constraint-based program analyses to the problem of analyzing uncaught exceptions in SML. Their system uses a combination of inclusion constraints (as in control-flow analyses) to approximate the control flow, and equality constraints (unification) between annotated types to keep track of exception values.

To compare performances between [Yi and Ryu 1997], [Fähndrich and Aiken 1997] and our analyzer, we used two of our benchmarks for which we have a faithful SML translation: Knuth-Bendix and Nucleic. The times reported below are of the form t_1/t_2, where t_1 is the time spent in exception analysis only, and t_2 is the total program analysis time, including parsing and type-checking in addition to exception analysis.

Test program	Yi-Ryu (version 0.98)	BANE (version 1.5)	Our system
Knuth-Bendix	0.7/1.0 s	1.6/2.2 s	0.06/0.14 s
Nucleic	1.8/5.2 s	3.3/7.6 s	1.4/1.9 s

From these figures, our exception analysis appears to be the fastest of the three. However, there are many external factors that influence the total running times of the analyses (such as the Yi-Ryu and BANE analyses being compiled by SML/NJ while ours is compiled by Objective Caml), so the figures above are not fully conclusive.

The main difference between the analyses of [Yi and Ryu 1997], [Fähndrich and Aiken 1997], and ours is the approximation of arguments carried by exceptions: they approximate only exception and function values carried by exceptions, but our analysis is the only one that also approximates exception arguments that are strings, integers, or datatypes. As explained in section 2.3, approximating all arguments of exceptions is crucial to obtain precise analysis of many real applications.

In theory, our unification-based analysis should be less precise than analyses based on inclusion constraints such as [Yi and Ryu 1997; Fähndrich and Aiken 1997]: the bidirectional propagation of information performed by unification causes exception effects to "leak" in types where those exceptions cannot actually occur. It is easy to construct artificial examples of such leaks, e.g. by replacing `let`-bound identifiers by λ-bound identifiers. However, those examples do not seem to occur in actual programs. The only leaks we observed in actual programs were related either to deficiencies of our incomplete algorithm for typing polymorphic recursion, or to functions contained inside mutable data structures. On those two cases, the analysis of [Fähndrich and Aiken 1997] obtains more precise results than our analysis.

6.2 Other related work

Our use of rows with row variables and presence annotations to approximate values of base types and sum types is essentially identical to Rémy's typing of extensible variants [Rémy 1989]. Another application of Rémy's encoding is the soft typing system for Scheme of Wright and Cartwright [1997]. Like our analysis, this soft typing system uses presence flags to keep track of whether a value can be a cons, an integer, an atom, etc. Being intended for Scheme, their analysis is specialized to a fixed algebra of S-expressions, while ours also handles extensible and user-defined data types. Cartwright and Felleisen [1996] briefly compare the unification-based approach to soft typing with another approach using set-based analysis.

There is a natural connection between exception analysis and type inference for extensible variants: using the well-known functional encoding of exceptions (where each subexpression is transformed to return a value of a variant type, either an exception tag or `NormalResult`(v) where v is the value of the subexpression), estimating uncaught exceptions is equivalent to inferring precise variant types. Pottier [1998] outlines an exception analysis thus derived from a type inferencer for ML with subtyping.

Refinement types [Freeman and Pfenning 1991] and the dependent types of Xi and Pfenning [1999] also introduce annotations on types to characterize subsets of ML's data types. Our approach is less ambitious than refinement types, in that it does not try to capture "deep" structural invariants of recursive data structures; on the other hand, type inference is much easier.

The principles of effect systems were studied extensively circa 1990 [Lucassen and Gifford 1988; Talpin and Jouvelot 1994], but few practical applications have been developed since. An impressive application is the region analysis of [Tofte and Talpin 1997; Tofte and Birkedal 1998]. Like ours, its precision is improved by typing recursion polymorphically.

Several program analyses based on unification and running in quasi-linear time have been proposed as faster alternatives to more conventional dataflow analyses. Two well-known examples are Henglein's tagging analysis [Henglein 1992] and Steensgaard's aliasing analysis [Steensgaard 1996]. Unification-based analyses have also been applied to the detection of year 2000 problems in Cobol programs [Eidorf et al. 1999; Ramalingam et al. 1999]. Baker [1990] suggests other examples of unification-based analyses.

The Church project has investigated the use of intersection types for program analyses [Dimock et al. 1997]. It can be argued that intersection types are a more natural way to analyze recursive functions than polymorphic recursion. However, type inference for intersection types is also undecidable, and inference algorithms for finite-rank fragments have only recently been proposed [Kfoury and Wells 1999].

The extended static checking project [Leino and Nelson 1998] develops static debugging tools for Modula-3 and Java that keep track of uncaught exceptions. Extended static checking is more ambitious than our analysis, in that it also detects dereferencing of null pointers, out-of-bound array accesses, and mutex locking errors in multi-threaded programs. Consequently, it relies on programmer-supplied annotations (e.g. preconditions to functions and methods).

7. CONCLUSIONS AND FUTURE WORK

It is often said that unification-based program analyses are faster, but less precise than more general constraint-based analyses such as CFA or SBA. For exception analysis, our experience indicates that a combination of unification, `let`-polymorphism, and polymorphic recursion is in practice almost as precise as analyses based on inclusion constraints. (The only case where our analysis is noticeably less precise than inclusion constraints is when references to functions are used intensively.) The running times of our algorithm seem excellent (although its theoretical complexity is at least as high as that of ML type inference). In turn, this good efficiency of our analysis allows us to keep more information on exception arguments than the other exception analyses, increasing greatly the precision of the analysis on certain ML programs. Thus, we see an interesting case of "less is more", where an *a priori* imprecise technology (unification) allows to improve eventually the precision of the analysis.

Some engineering issues remain to be solved before our analysis can be applied to large ML applications. The main practical issue is displaying the results of the analysis in a readable way. The volume of information contained in annotated type expressions can be overwhelming. The implementation of our analysis developed by the second author provides a graphical browser for annotated types that allows the programmer to select different levels of display for each annotated type, abstracting some of the information. It remains to see the effectiveness of this tool on large programs.

Another direction for future work is to combine our analysis with array bound analyses and integer interval analyses, in order to eliminate some of the "unavoidable false positives" currently reported.

APPENDIX

A. THE UNIFICATION ALGORITHM

In this appendix, we give the unification algorithm for our type algebra modulo the two equations (1) and (2). We define the head constructor $H(\varepsilon)$ of a row element ε as follows:

$$H(i : \pi) = i \qquad H(C : \pi) = C \qquad H(D(\tau)) = D$$

The algorithm is in the style of Robinson's unification algorithm, and handles the left commutativity axiom (equation (1)) like in [Rémy 1993b]. Namely, to unify two constructed rows $\varepsilon_1; \varphi_1$ and $\varepsilon_2; \varphi_2$ when the head constructors of ε_1 and ε_2 differ, we pick a fresh row variable ρ of the appropriate kind and solve the two equations $\varphi_1 = \varepsilon_2; \rho$ and $\varphi_2 = \varepsilon_1; \rho$. As shown in appendix C, theorem 21, any solution of those equations also solves $\varepsilon_1; \varphi_1 = \varepsilon_2; \varphi_2$.

To make precise the generation of "fresh" row variables during unification, we add an extra parameter V and an extra result V' to the unification algorithm, which becomes $\mathtt{mgu}_V(Q) = (\psi, V')$. The parameter V is a set of variables that must not be used as fresh variables during unification. We always assume V finite, so that it is always possible to choose a row variable not in V and of any given kind. The second result V' is the union of V and of the set of variables that have been used as fresh variables during unification. We write $\mathtt{mgu}_V(Q) \circ \theta$ to stand for $(\psi \circ \theta, V')$ where $(\psi, V') = \mathtt{mgu}_V(Q)$.

$$\mathtt{mgu}_V(\emptyset) \; = \; (id, V)$$

Unification between types:

$$\begin{aligned}
\mathtt{mgu}_V(\{\alpha = \alpha\} \cup Q) \; &= \; \mathtt{mgu}_V(Q) \\
\mathtt{mgu}_V(\{\alpha = \tau\} \cup Q) \; &= \; \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\} \text{ if } \alpha \notin FV(\tau) \\
\mathtt{mgu}_V(\{\tau = \alpha\} \cup Q) \; &= \; \mathtt{mgu}_V(Q\{\alpha \leftarrow \tau\}) \circ \{\alpha \leftarrow \tau\} \text{ if } \alpha \notin FV(\tau) \\
\mathtt{mgu}_V(\{\mathtt{int}[\varphi_1] = \mathtt{int}[\varphi_2]\} \cup Q) \; &= \; \mathtt{mgu}_V(\{\varphi_1 = \varphi_2\} \cup Q) \\
\mathtt{mgu}_V(\{\mathtt{exn}[\varphi_1] = \mathtt{exn}[\varphi_2]\} \cup Q) \; &= \; \mathtt{mgu}_V(\{\varphi_1 = \varphi_2\} \cup Q)
\end{aligned}$$

$$\mathtt{mgu}_V(\{\tau_1 \xrightarrow{\varphi_1} \tau_1' = \tau_2 \xrightarrow{\varphi_2} \tau_2'\} \cup Q) \;=\; \mathtt{mgu}_V(\{\tau_1 = \tau_2; \varphi_1 = \varphi_2; \tau_1' = \tau_2'\} \cup Q)$$

Unification between rows:

$$\begin{aligned}
\mathtt{mgu}_V(\{\rho = \rho\} \cup Q) &= \mathtt{mgu}_V(Q)\\
\mathtt{mgu}_V(\{\rho = \varphi\} \cup Q) &= \mathtt{mgu}_V(Q\{\rho \leftarrow \varphi\}) \circ \{\rho \leftarrow \varphi\} \text{ if } \rho \notin FV(\varphi)\\
\mathtt{mgu}_V(\{\varphi = \rho\} \cup Q) &= \mathtt{mgu}_V(Q\{\rho \leftarrow \varphi\}) \circ \{\rho \leftarrow \varphi\} \text{ if } \rho \notin FV(\varphi)\\
\mathtt{mgu}_V(\{\top = \top\} \cup Q) &= \mathtt{mgu}_V(Q)\\
\mathtt{mgu}_V(\{(i:\pi;\varphi) = \top\} \cup Q) &= \mathtt{mgu}_V(\{\pi = \mathtt{Pre}; \varphi = \top\} \cup Q)\\
\mathtt{mgu}_V(\{\top = (i:\pi;\varphi)\} \cup Q) &= \mathtt{mgu}_V(\{\pi = \mathtt{Pre}; \varphi = \top\} \cup Q)\\
\mathtt{mgu}_V(\{(\varepsilon_1;\varphi_1) = (\varepsilon_2;\varphi_2)\} \cup Q) &= \mathtt{mgu}_V(\{\varepsilon_1 = \varepsilon_2; \varphi_1 = \varphi_2\} \cup Q)\\
&\quad \text{if } H(\varepsilon_1) = H(\varepsilon_2)\\
\mathtt{mgu}_V(\{(\varepsilon_1;\varphi_1) = (\varepsilon_2;\varphi_2)\} \cup Q) &= \mathtt{mgu}_{V\cup\{\rho\}}(\{\varphi_1 = (\varepsilon_2;\rho); \varphi_2 = (\varepsilon_1;\rho)\} \cup Q)\\
&\quad \text{if } H(\varepsilon_1) \neq H(\varepsilon_2)\\
&\quad \text{and } \rho \notin V \text{ and } \rho \text{ not free in the left-hand side}\\
&\quad \text{and } \rho \text{ has kind } T(S \cup \{H(\varepsilon_1), H(\varepsilon_2)\})\\
&\quad \text{where } T(S) \text{ is the kind of } \varepsilon_1;\varphi_1 \text{ and } \varepsilon_2;\varphi_2\\
&\quad \text{and } T \text{ stands for either } \mathtt{EXN} \text{ or } \mathtt{INT}
\end{aligned}$$

Unification between row elements:

$$\begin{aligned}
\mathtt{mgu}_V(\{(i:\pi_1) = (i:\pi_2)\} \cup Q) &= \mathtt{mgu}_V(\{\pi_1 = \pi_2\} \cup Q)\\
\mathtt{mgu}_V(\{(C:\pi_1) = (C:\pi_2)\} \cup Q) &= \mathtt{mgu}_V(\{\pi_1 = \pi_2\} \cup Q)\\
\mathtt{mgu}_V(\{D(\tau_1) = D(\tau_2)\} \cup Q) &= \mathtt{mgu}_V(\{\tau_1 = \tau_2\} \cup Q)
\end{aligned}$$

Unification between presence annotations:

$$\begin{aligned}
\mathtt{mgu}_V(\{\delta = \pi\} \cup Q) &= \mathtt{mgu}_V(Q\{\delta \leftarrow \pi\}) \circ \{\delta \leftarrow \pi\}\\
\mathtt{mgu}_V(\{\pi = \delta\} \cup Q) &= \mathtt{mgu}_V(Q\{\delta \leftarrow \pi\}) \circ \{\delta \leftarrow \pi\}\\
\mathtt{mgu}_V(\{\mathtt{Pre} = \mathtt{Pre}\} \cup Q) &= \mathtt{mgu}_V(Q)
\end{aligned}$$

If none of the cases above is applicable, $\mathtt{mgu}_V(Q)$ is undefined.

B. THE TYPE INFERENCE ALGORITHM

The type inference algorithm defined below is similar to Damas and Milner's W algorithm. One difference is that it infers not only the type for the given expression, but also its effect. Another difference is that we make explicit the notion of "fresh" variable, so that the claim of completeness of W can be made precise. Hence, we add an extra parameter V and an extra result V' to the algorithm W, which becomes $W(E, a, V) = (\tau, \varphi, \theta, V')$. As in the case of the unification algorithm, the parameter V is a set of type variables which cannot be used as fresh variable by this execution of the W algorithm. The result V' is V plus all type variables that have been used as fresh variables by this execution of W, and therefore must not be used again as fresh variables later.

The result of the algorithm $W(E, a, V)$ is the quadruple $(\tau, \varphi, \theta, V')$ defined by induction on a as follows:

—If a is x (with $x \in \mathrm{Dom}(E)$):
let $\rho \notin V$ be a fresh row variable of kind $\mathtt{EXN}(\emptyset)$
take $(\tau, V') = \mathtt{Inst}(E(x), V \cup \{\rho\})$ and $\varphi = \rho$ and $\theta = id$.

—If a is i:
let $\rho \notin V$ be a fresh row variable of kind $\mathtt{INT}(\{i\})$
let $\rho' \notin V$ be a fresh row variable of kind $\mathtt{EXN}(\emptyset)$
take $\tau = \mathtt{int}[i : \mathtt{Pre}; \rho]$ and $\varphi = \rho'$ and $\theta = id$ and $V' = V \cup \{\rho, \rho'\}$.

—If a is $\lambda x.\, a_1$:
let $\alpha \notin V$ be a fresh type variable
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E \oplus \{x : \alpha\}, a_1, V \cup \{\alpha\})$
let $\rho \notin V_1$ be a fresh row variable of kind $\mathtt{EXN}(\emptyset)$
take $\tau = \theta_1(\alpha) \xrightarrow{\varphi_1} \tau_1$ and $\varphi = \rho$ and $\theta = \theta_1$ and $V' = V_1 \cup \{\rho\}$.

—If a is $a_1(a_2)$:
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
let $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1)$
let $\alpha \notin V_2$ be a fresh type variable
let $(\mu, V_3) = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha,\ \theta_2(\varphi_1) = \varphi_2\}$
take $\tau = \mu(\alpha)$ and $\varphi = \mu(\varphi_2)$ and $\theta = \mu \circ \theta_2 \circ \theta_1$ and $V' = V_3$.

—If a is $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$:
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
let $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \mathtt{Gen}(\tau_1, \theta_1(E), \varphi_1)\}, a_2, V_1)$
let $(\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2(\varphi_1) = \varphi_2\}$
take $\tau = \mu(\tau_2)$ and $\varphi = \mu(\varphi_2)$ and $\theta = \mu \circ \theta_2 \circ \theta_1$ and $V' = V_3$.

—If a is $\mathtt{match}\ a_1\ \mathtt{with}\ p \to a_2 \mid x \to a_3$:
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
let $(E', \tau', \psi, V_1') = \mathtt{Patsubtr}(p, \tau_1, V_1)$
let $(\tau_2, \varphi_2, \theta_2, V_2) = W(\psi(\theta_1(E)) \oplus E', a_2, V_1')$
let $(\tau_3, \varphi_3, \theta_3, V_3) = W(\theta_2(\psi(\theta_1(E))) \oplus \{x : \theta_2(\tau')\}, a_3, V_2)$
let $(\mu, V_4) = \mathtt{mgu}_{V_3}\{\theta_3(\tau_2) = \tau_3,\ \theta_3(\varphi_2) = \varphi_3,\ \theta_3(\theta_2(\psi(\varphi_1))) = \varphi_3\}$
take $\tau = \mu(\tau_3)$ and $\varphi = \mu(\varphi_3)$ and $\theta = \mu \circ \theta_3 \circ \theta_2 \circ \psi \circ \theta_1$ and $V' = V_4$.

—If a is C:
let $\rho \notin V$ be a fresh row variable of kind $\mathtt{EXN}(\{C\})$
let $\rho' \notin V$ be a fresh row variable of kind $\mathtt{EXN}(\emptyset)$
take $\tau = \mathtt{exn}[C : \mathtt{Pre}; \rho]$ and $\varphi = \rho'$ and $\theta = id$ and $V' = V \cup \{\rho, \rho'\}$.

—If a is $D(a_1)$:
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
let $(\tau_2, V_2) = \mathtt{Inst}(\mathit{TypeArg}(D), V_1)$
let $(\mu, V_3) = \mathtt{mgu}_{V_2}\{\tau_2 = \tau_1\}$
let $\rho \notin V_3$ be a fresh row variable of kind $\mathtt{EXN}(\{D\})$
let $\rho' \notin V_3 \cup \{\rho\}$ be a fresh row variable of kind $\mathtt{EXN}(\emptyset)$
take $\tau = \mathtt{exn}[D(\mu(\tau_1)); \rho]$ and $\varphi = \rho'$ and $\theta = \mu \circ \theta_1$ and $V' = V_3 \cup \{\rho, \rho'\}$.

—If a is $\mathtt{try}\ a_1\ \mathtt{with}\ x \to a_2$:
let $(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V)$
let $(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E) \oplus \{x : \mathtt{exn}[\varphi_1]\}, a_2, V_1)$
let $(\mu, V_3) = \mathtt{mgu}_{V_2}\{\theta_2(\tau_1) = \tau_2\}$

take $\tau = \mu(\tau_2)$ and $\varphi = \mu(\varphi_2)$ and $\theta = \mu \circ \theta_2 \circ \theta_1$ and $V' = V_3$.

If none of the cases above applies, or if one of the unification steps fails, then type inference fails and $W(E, a, V)$ is undefined.

The type inference algorithm I for closed terms mentioned in section 3.7, theorem 5, is defined in terms of W as follows: if $W(\emptyset, a, \emptyset) = (\tau, \varphi, \theta, V')$, then $I(a) = (\tau, \varphi)$; if $W(\emptyset, a, \emptyset)$ is undefined, then so is $I(a)$.

The auxiliary function $\mathtt{Inst}(\sigma, V)$ (trivial instance)

$\mathtt{Inst}(\forall \alpha_i, \rho_j, \delta_k.\, \tau,\ V)$ is $(\tau\{\alpha_i \leftarrow \alpha'_i, \rho_j \leftarrow \rho'_j, \delta_k \leftarrow \delta'_k\},\ V \cup \{\alpha'_i, \rho'_j, \delta'_k\})$ where $\alpha'_i, \rho'_j, \delta'_k$ are fresh variables not in V such that ρ'_j and ρ_j have the same kind for all j.

The auxiliary function $\mathtt{Patsubtr}$ (typing of patterns and pattern subtraction)

$\mathtt{Patsubtr}(p, \tau, V)$ is the quadruple (E, τ', θ, V') defined by induction on p as follows:

—If p is x:
let $\alpha \notin V$ be a fresh type variable
take $E = \{x : \tau\}$ and $\tau' = \alpha$ and $\theta = id$ and $V' = V \cup \{\alpha\}$.

—If p is i:
let $\rho \notin V$ be a fresh row variable of kind $\mathtt{INT}(\{i\})$
and $\delta \notin V \cup \{\rho\}$ be a fresh presence variable
let $(\mu, V_1) = \mathtt{mgu}_{V \cup \{\rho,\delta\}}\{\tau = \mathtt{int}[i : \delta; \rho]\}$
let $\delta' \notin V_1$ be a fresh presence variable
take $E = \emptyset$ and $\tau' = \mathtt{int}[i : \delta'; \mu(\rho)]$ and $\theta = \mu$ and $V' = V_1 \cup \{\delta\}$.

—If p is C:
let $\rho \notin V$ be a fresh row variable of kind $\mathtt{EXN}(\{C\})$
and $\delta \notin V \cup \{\rho\}$ be a fresh presence variable
let $(\mu, V_1) = \mathtt{mgu}_{V \cup \{\rho,\delta\}}\{\tau = \mathtt{exn}[C : \delta; \rho]\}$
let $\delta' \notin V_1$ be a fresh presence variable
take $E = \emptyset$ and $\tau' = \mathtt{exn}[C : \delta'; \mu(\rho)]$ and $\theta = \mu$ and $V' = V_1 \cup \{\delta\}$.

—If p is $D(p_1)$:
let $\tau_1 = \mathtt{Inst}(\mathit{TypeArg}(D))$
let $(E_1, \tau'_1, \theta_1, V_1) = \mathtt{Patsubtr}(p_1, \tau_1, V)$
let $\rho \notin V_1$ be a fresh row variable of kind $\mathtt{EXN}(\{D\})$
let $(\mu, V_2) = \mathtt{mgu}_{V_1 \cup \{\rho\}}\{\tau = \mathtt{exn}[D(\theta_1(\tau_1)); \rho]\}$
take $E = \mu(E_1)$ and $\tau' = \mathtt{exn}[D(\mu(\tau'_1)); \mu(\rho)]$ and $\theta = \mu \circ \theta_1$ and $V' = V_2$.

C. PROOFS

In this appendix, we prove the claims made in sections 3.6 and 3.7. More detailed proofs can be found in the second author's thesis [Pessaux 1999].

C.1 Properties of the typing judgement

We say that an environment E is well-formed if for all $x \in \mathrm{Dom}(E)$, $E(x)$ is $\forall \vec{\alpha}, \vec{\rho}, \vec{\delta}.\, \tau$ with $\vdash \tau \ \mathtt{wf}$.

LEMMA 6. *(Typings are well-kinded.) Let E be a well-formed environment. Then, $E \vdash a : \tau/\varphi$ implies $\vdash \tau \ \mathtt{wf}$ and $\vdash \varphi :: \mathtt{EXN}(\emptyset)$.*

PROOF. We first show (by induction on p) that if $\vdash \tau$ wf and $\vdash p : \tau \Rightarrow E$, then E is well formed, and if $\vdash \tau - p \leadsto \tau'$, then $\vdash \tau'$ wf. The result then follows by an easy induction on the derivation of $E \vdash a : \tau/\varphi$. □

LEMMA 7. *(Commutation between instantiation and substitution.) If $\tau \leq \sigma$, then $\theta(\tau) \leq \theta(\sigma)$ for all substitutions θ.*

PROOF. Straightforward by definition of $\leq$. □

In the following lemma, we write $\mathrm{Rng}(\theta)$ for the range of the substitution θ, that is, $\bigcup\{FV(\theta(v)) \mid v \in \mathrm{Dom}(\theta)\}$. We say that a type variable v is out of reach of a substitution θ if $v \notin \mathrm{Dom}(v) \cup \mathrm{Rng}(v)$. In other terms, v is out of reach of θ if and only if $\theta(v) = v$, and for all variables $v' \neq v$, v is not free in $\theta(v')$.

LEMMA 8. *(Commutation between generalization and substitution.) If all variables in $FV(\tau) \setminus (FV(E) \cup FV(\varphi))$ are out of reach of the substitution θ, then $\mathtt{Gen}(\theta(\tau), \theta(E), \theta(\varphi)) = \theta(\mathtt{Gen}(\tau, E, \varphi))$.*

PROOF. It is easy to see that a variable α out of reach of θ is free in a type τ if and only if it is free in $\theta(\tau)$. Hence, $FV(\theta(\tau)) \setminus (FV(\theta(E)) \cup FV(\theta(\varphi))) = FV(\tau) \setminus (FV(E) \cup FV(\varphi))$, and the result follows. □

LEMMA 9. *(Typing is stable by substitution.) Let θ be a substitution.*

(1) If $\vdash p : \tau \Rightarrow E$ then $\vdash p : \theta(\tau) \Rightarrow \theta(E)$.
(2) If $\vdash \tau - p \leadsto \tau'$ then $\vdash \theta(\tau) - p \leadsto \theta(\tau')$.
(3) If $E \vdash a : \tau/\varphi$ then $\theta(E) \vdash a : \theta(\tau)/\theta(\varphi)$.

PROOF. The proof of 1 and 2 is by structural induction on p. The proof of 3 is by structural induction on a and uses 1 and 2. For the base case $a = x$, we apply lemma 7. For the case $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$, we first rename the generalized variables in the typing of a_1 so that they are out of reach of θ, then apply lemma 8 to the typing of a_2. □

We say that a schema σ is more general than a schema σ', and write $\sigma \geq \sigma'$, if all instances of σ' are also instances of σ.

LEMMA 10. *(Typing is stable under more general hypotheses.) Assume $\mathrm{Dom}(E_1) = \mathrm{Dom}(E_2)$ and $E_2(x) \geq E_1(x)$ for all $x \in \mathrm{Dom}(E_1)$. Then, $E_1 \vdash a : \tau/\varphi$ implies $E_2 \vdash a : \tau/\varphi$.*

PROOF. The proof is by structural induction on a. The base case $a = x$ is straightforward by hypothesis $E_2(x) \geq E_1(x)$. For the case $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$, notice that $\sigma \geq \sigma'$ implies $FV(\sigma) \subseteq FV(\sigma')$; therefore, $FV(E_2) \subseteq FV(E_1)$, and it follows that $\mathtt{Gen}(\tau, E_2, \varphi) \geq \mathtt{Gen}(\tau, E_1, \varphi)$. □

C.2 Type soundness

LEMMA 11. *(Values have no effects.) Let v be a value. Assume $E_0 \vdash v : \tau/\varphi$. Then, for all rows φ' of kind $\mathtt{EXN}(\emptyset)$, we have $E_0 \vdash v : \tau/\varphi'$ as well.*

PROOF. The result follows by examination of the typing rules that can apply to a value (rules 2, 3, 7 and 8). □

In the sequel, we write $\vdash v : \tau$ as an abbreviation for "there exists some φ such that $E_0 \vdash v : \tau/\varphi$". By lemma 11, if there exists one such φ, then $E_0 \vdash v : \tau/\varphi$ holds for all φ of kind $\mathtt{EXN}(\emptyset)$.

LEMMA 12. *(Substitution lemma.) Assume* $\vdash v : \tau'$ *and* $E \oplus \{x : \forall\alpha_1 \ldots \alpha_n.\tau'\} \vdash a : \tau/\varphi$ *where the variables* $\alpha_1 \ldots \alpha_n$ *are not free in* E. *Then,* $E \vdash a\{x \leftarrow v\} : \tau/\varphi$.

PROOF. The proof is by structural induction on a. We write $E_x = E \oplus \{x : \forall\alpha_1 \ldots \alpha_n.\tau'\}$. The base case $a = x$ follows from lemmas 9 and 11. For the case $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$, notice that $\mathtt{Gen}(\tau, E, \varphi) \geq \mathtt{Gen}(\tau, E_x, \varphi)$ since $FV(E) \subseteq FV(E_x)$, and use lemma 10. □

LEMMA 13. *(Substitution lemma for pattern-matching.) Assume* $\vdash v : \tau'$ *and* $\vdash p : \tau' \Rightarrow E'$ *and* $E \oplus E' \vdash a : \tau/\varphi$. *If* $\xi = M(v,p)$ *is defined, then* $E \vdash \xi(a) : \tau/\varphi$.

PROOF. The proof is an easy inductive argument on p, using lemma 12 for the base case $p = x$. □

We say that a value v belongs to a row φ if one of the following holds:

—v is an integer i and $\varphi = i : \mathtt{Pre}; \varphi'$ for some φ';

—v is a constant exception C and $\varphi = C : \mathtt{Pre}; \varphi'$ for some φ';

—v is a parameterized exception $D(v')$ and $\varphi = D(\tau'); \varphi'$ for some φ' and τ' such that $\vdash v' : \tau'$.

LEMMA 14. *(Shape of values by type.) Let* v *be a value. Assume* $\vdash v : \tau$.

—*If* τ *is* $\mathtt{int}[\varphi]$, *then* v *is an integer* i, *and this integer* i *belongs to* φ.

—*If* τ *is* $\mathtt{exn}[\varphi]$, *then* v *is either* C *or* $D(v')$, *and in both cases* v *belongs to* φ.

—*If* τ *is* $\tau_1 \xrightarrow{\varphi} \tau_2$, *then* v *is a function* $\lambda x.a$, *and* $E_0 \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi$.

PROOF. The result holds for each of the typing rules that can apply to a value (rules 2, 3, 7 and 8). □

LEMMA 2. *(Correctness of subtraction.) If* $\vdash v : \tau$ *and* $M(v,p)$ *is undefined (value* v *does not match pattern* p*) and* $\vdash \tau - p \rightsquigarrow \tau'$, *then* $\vdash v : \tau'$.

PROOF. The proof proceeds by induction and case analysis on the pattern p.

If $p = x$, the result holds vacuously, as $M(v,p)$ is defined regardless of v.

If $p = i$, by rule 14, we have $\tau = \mathtt{int}[i : \pi; \varphi]$ and $\tau' = \mathtt{int}[i : \pi'; \varphi]$ for some φ, π, and π'. Since v has type τ, lemma 14 shows that v is an integer j and moreover j belongs to the row $i : \pi; \varphi$. Since $M(v,p)$ is undefined, we have $i \neq j$. Hence j belongs to the row φ. In other terms, $\varphi = j : \mathtt{Pre}; \varphi'$ for some φ', and $\tau' = \mathtt{int}[j : \mathtt{Pre}; i : \pi'; \varphi']$. Thus we can derive $\vdash j : \tau'$ by rule 2.

The case $p = C$ is similar to the previous case.

Finally, if $p = D(p_1)$, by rule 17, we have $\tau = \mathtt{exn}[D(\tau_1); \varphi]$ and $\tau' = \mathtt{exn}[D(\tau_1'); \varphi]$ with $\vdash \tau_1 - p_1 \rightsquigarrow \tau_1'$. From the hypothesis that v has type τ, lemma 14 shows that v is either a constructed term $D(v_1)$ with $\vdash v_1 : \tau_1$, or a constant constructor C or constructed term $D'(v_1)$, $D' \neq D$, that belongs to φ. In the latter case, v also belongs to $D(\tau_1'); \varphi$ and the result follows by rule 7 or 8. In the former case, $M(v_1, p_1)$ is undefined, otherwise $M(v,p)$ would be defined. Applying the induction hypothesis to v_1 and p_1, we obtain $\vdash v_1 : \tau_1'$. The expected result $\vdash v : \tau'$ follows by rule 8. □

LEMMA 16. *(Effects of exceptions.)* $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$ *if and only if* v *belongs to* φ.

PROOF. A typing derivation for $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$ has the following shape:

$$\cfrac{\cfrac{\texttt{exn}[\varphi] \xrightarrow{\varphi} \tau \leq E_0(\texttt{raise})}{E_0 \vdash \texttt{raise} : \texttt{exn}[\varphi] \xrightarrow{\varphi} \tau} \qquad E_0 \vdash v : \texttt{exn}[\varphi]/\varphi}{E_0 \vdash \texttt{raise}\ v : \tau/\varphi}$$

By lemma 14, if $E_0 \vdash v : \texttt{exn}[\varphi]/\varphi$, then v belongs to φ. Conversely, if v belongs to φ, we can derive $E_0 \vdash v : \texttt{exn}[\varphi]/\varphi$ using rule 7 or 8. $\square$

THEOREM 1. *(Subject reduction.) Reduction preserves typing: if* $E_0 \vdash a : \tau/\varphi$ *and* $a \Rightarrow a'$*, then* $E_0 \vdash a' : \tau/\varphi$

PROOF. We show the result first for head reductions (reduction rules 1–11), by case on the reduction rule used. For rules 1 and 2 (β-reduction), use lemma 12. For rule 3, the result follows from lemma 13. For rule 4, lemma 2 shows that $\vdash v : \tau'$ where τ' is the type obtained by subtracting p from τ. Then, the result follows from lemma 12. The case of rule 5 is straightforward. For rules 6 to 10, notice that in all those rules, a' is $\texttt{raise}\ v$ and a is an expression containing $\texttt{raise}\ v$ as a subexpression outside of a `try` construct. Thus, in a derivation of $E_0 \vdash a : \tau/\varphi$, a subexpression $\texttt{raise}\ v$ is assigned the effect φ. By lemma 16, it follows that v belongs to φ, and that $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$. Finally, the result in the case of rule 11 follows from lemma 12.

The result then extends to reductions under a context Γ (rule 12) by a straightforward structural induction over Γ. $\square$

THEOREM 3. *(Correctness of exception analysis.) Let* a *be a complete program. Assume* $E_0 \vdash a : \tau/\varphi$ *and* $a \overset{*}{\Rightarrow} \texttt{raise}\ v$*. Then,* v *belongs to* φ.

PROOF. By the subject reduction theorem 1, it follows that $E_0 \vdash \texttt{raise}\ v : \tau/\varphi$. Lemma 16 then shows that v belongs to φ. $\square$

LEMMA 4. *(Progress.) If* $E_0 \vdash a : \tau/\varphi$*, then either* a *is a value* v*, or* a *is an uncaught exception* $\texttt{raise}\ v$*, or there exists* a' *such that* $a \Rightarrow a'$.

PROOF. The proof is by structural induction and case analysis on a.

If a is an identifier x, since it is well-typed in E_0, we must have $a = \texttt{raise}$ and this is a value. If a is a constant i, C or a λ-abstraction, a is a value.

If a is $D(a_1)$, we have that a_1 is well-typed in E_0, hence by induction hypothesis, either a_1 is a value, or it reduces, or it is $\texttt{raise}\ v$. In the first case, a is a value; in the second case, a reduces by the context rule 12; in the third case, a reduces by rule 8.

If a is $a_1(a_2)$, applying the induction hypothesis to a_1 and a_2, we have the following cases to consider. If a_1 and a_2 are values, since a_1 has an arrow type, it must be a λ-abstraction (lemma 14), hence a reduces by rule 1. If a_1 is an uncaught exception $\texttt{raise}\ v$, a reduces by rule 6. If a_1 is a value and a_2 an uncaught exception, a_1 must be a λ-abstraction and a reduces by rule 7. Otherwise, either a_1 reduces or a_1 is a λ-abstraction and a_2 reduces; in both cases, a reduces by the context rule 12.

If a is `let` $x = a_1$ `in` a_2: by induction hypothesis, either a_1 is a value and a can β-reduce (rule 2), or a_1 is an uncaught exception and a reduces by rule 9, or a_1 reduces and a reduces also by the context rule.

If a is `match` a_1 `with` $p \to a_2 \mid x \to a_3$, either a_1 is a value and a can reduce by rule 3 or 4, or a_1 is an uncaught exception and a reduces by rule 10, or a_1 reduces and a reduces also by the context rule.

Finally, if a is `try` a_1 `with` $x \to a_2$, either a_1 is a value and a can reduce by rule 5, or a_1 is an uncaught exception and a reduces by rule 11, or a_1 reduces and a reduces also by the context rule. $\square$

THEOREM 20. *(Type soundness.) Let a be a complete program. Assume $E_0 \vdash a : \tau/\varphi$. If $a \overset{*}{\Rightarrow} a'$ and a' is in normal form with respect to the reduction rules, then a' is either a value v or an uncaught exception* `raise` v.

PROOF. The result is a corollary of theorem 1 and lemma 4. $\square$

C.3 Properties of the type inference algorithm

In the following theorem, we write $\theta =_V \theta'$ to mean $\theta(v) = \theta'(v)$ for all variables $v \in V$. We say that a system of equations $Q = \{\tau_i = \tau'_i; \varphi_j = \varphi'_j; \pi_l = \pi'_l\}$ is well-kinded if for all i, $\vdash \tau_i$ `wf` and $\vdash \tau'_i$ `wf`, and for all j, there exists a kind κ_j such that $\varphi_j :: \kappa_j$ and $\varphi'_j :: \kappa_j$.

THEOREM 21. *(Principal unifiers.) Let Q be a set of well-kinded equations and V a finite set of variables such that $FV(Q) \subseteq V$.*

—*Correctness: if $(\theta, V') = \mathtt{mgu}_V(Q)$ is defined, then θ is a unifier of Q, and θ preserves kinds. Moreover, V' is finite, $V \subseteq V'$, and $\mathrm{Dom}(\theta) \cup \mathrm{Rng}(\theta) \subseteq V'$.*

—*Completeness and principality: if ψ preserves kinds and is a solution of Q, then $(\theta, V') = \mathtt{mgu}_V(Q)$ is defined and there exists a kind-preserving substitution ξ such that $\psi =_V \xi \circ \theta$.*

PROOF. The proof is a standard inductive argument on the execution of `mgu` and case analysis on the shape of Q. We show the only case that differs from the usual proof of Robinson's algorithm: $Q = \{(\varepsilon_1; \varphi_1) = (\varepsilon_2; \varphi_2)\} \cup Q_1$ where $H(\varepsilon_1) \neq H(\varepsilon_2)$. Let $t(S)$ be the kind of $\varepsilon_1; \varphi_1$ and $\varepsilon_2; \varphi_2$. Let ρ be a variable of kind $T(S \cup \{H(\varepsilon_1); H(\varepsilon_2)\})$ such that $\rho \notin V$. Consider $Q' = \{\varphi_1 = (\varepsilon_2; \rho); \varphi_2 = (\varepsilon_1; \rho)\} \cup Q_1$. The equations in Q' are well-kinded: the first one has kind $T(S \cup \{H(\varepsilon_1)\})$; the second one has kind $T(S \cup \{H(\varepsilon_2)\})$. Moreover, $FV(Q') = FV(Q) \cup \{\rho\} \subseteq V \cup \{\rho\}$.

For the correctness part, assume $(\theta, V') = \mathtt{mgu}_{V \cup \{\rho\}}(Q')$ is defined. By induction hypothesis, θ is a solution of Q', θ preserves kinds, V' is finite, $V \cup \{\rho\} \subseteq V'$, and $\mathrm{Dom}(\theta) \cup \mathrm{Rng}(\theta) \subseteq V'$. Since θ is a solution of Q', it is a solution of Q_1. Moreover,

$$\begin{aligned} \theta(\varepsilon_1; \varphi_1) &= \theta(\varepsilon_1); \theta(\varphi_1) = \theta(\varepsilon_1); \theta(\varepsilon_2); \theta(\rho) \\ \theta(\varepsilon_2; \varphi_2) &= \theta(\varepsilon_2); \theta(\varphi_2) = \theta(\varepsilon_2); \theta(\varepsilon_1); \theta(\rho) \end{aligned}$$

Those two rows are equal modulo the left commutativity equation. Hence, θ is a solution of Q.

For the completeness part, assume that ψ is a solution of Q. Since $\psi(\varepsilon_1; \varphi_1)$ and $\psi(\varepsilon_2; \varphi_2)$ are equal, they are both equal to a row of the form $\psi(\varepsilon_1); \psi(\varepsilon_2); \varphi$ for some φ. Take $\psi' = \psi \oplus \{\rho \leftarrow \varphi\}$. By construction of φ, the substitution ψ'

is a solution of Q'. Applying the induction hypothesis, $(\theta, V') = \mathtt{mgu}_{V \cup \{\rho\}}(Q')$ is defined, and $\psi' =_{V \cup \{\rho\}} \xi \circ \theta$ for some ξ. Hence $\mathtt{mgu}_V(Q) = (\theta, V')$ is defined, and since $\psi' =_V \psi$, we have $\psi =_V \xi \circ \theta$ as expected. □

The following lemma summarizes some simple properties of the W algorithm that are useful to show the correctness and completeness of W. The finiteness of the V parameter and of the V' result ensures that fresh variables can always be found outside of V, and thus that W and mgu do not fail when picking fresh variables.

LEMMA 22. *(Structural properties of W.) Assume $(\tau, \varphi, \theta, V') = W(E, a, V)$ is defined, V is finite, and $FV(E) \subseteq V$. Then:*

—$V \subseteq V'$;

—V' is finite;

—$FV(\tau) \subseteq V'$ and $FV(\varphi) \subseteq V'$;

—all variables not in V' are out of reach of θ;

—$FV(\theta(E)) \subseteq V'$.

PROOF. The result follows by examination of the cases of W and by the properties of mgu shown in theorem 21. □

THEOREM 23. *(Correctness of algorithm W.) If $FV(E) \subseteq V$ and $(\tau, \varphi, \theta, V') = W(E, a, V)$ is defined, then $\theta(E) \vdash a : \tau/\varphi$. Moreover, we have $\vdash \tau \ \mathtt{wf}$ and $\vdash \varphi :: \mathtt{EXN}(\emptyset)$, and θ preserves kinds.*

PROOF. The proof is a standard inductive argument on a, using the stability of typing judgements by substitution (lemma 9) and the correctness of mgu (theorem 21, first part). Notice that $\vdash \tau \ \mathtt{wf}$ and $\vdash \varphi :: \mathtt{EXN}(\emptyset)$ follow from $\theta(E) \vdash a : \tau/\varphi$ by lemma 6. We show one case to illustrate the proof. Assume $a = a_1(a_2)$. By hypothesis, $W(E, a, V)$ is defined, hence we have

$$\begin{gathered}(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \qquad (\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1) \\ \alpha \notin V_2 \qquad (\mu, V_3) = \mathtt{mgu}_{V_2 \cup \{\alpha\}}\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha,\ \theta_2(\varphi_1) = \varphi_2\} \\ \tau = \mu(\alpha) \qquad \varphi = \mu(\varphi_2) \qquad \theta = \mu \circ \theta_2 \circ \theta_1\end{gathered}$$

Applying the induction hypothesis to the recursive calls of W, we obtain derivations of $\theta_1(E) \vdash a_1 : \tau_1/\varphi_1$ and $\theta_2(\theta_1(E)) \vdash a_2 : \tau_2/\varphi_2$. Applying lemma 9 to those derivations (with substitution $\mu \circ \theta_2$ for the left derivation and μ for the right derivation), we obtain derivations of $\theta(E) \vdash a_1 : \mu(\theta_2(\tau_1))/\mu(\theta_2(\varphi_1))$ and $\theta(E) \vdash a_2 : \mu(\tau_2)/\mu(\varphi_2)$.

The set of equations $\{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha,\ \theta_2(\varphi_1) = \varphi_2\}$ is well-kinded since θ_2 is kind-preserving, τ_1 and τ_2 are well-formed, and φ_1 and φ_2 have kind $\mathtt{EXN}(\emptyset)$. Hence, μ is a unifier of this set of equations, and we have

$$\theta(E) \vdash a_1 : \mu(\tau_2) \xrightarrow{\varphi} \tau/\varphi \quad \text{and} \quad \theta(E) \vdash a_2 : \mu(\tau_2)/\varphi$$

from which we can derive $\theta(E) \vdash a_1(a_2) : \tau/\varphi$ by rule 4 as desired. □

THEOREM 24. *(Completeness of algorithm W.) Let E be a well-kinded environment, a be an expression, and V be a finite set of type variables such that*

$FV(E) \subseteq V$. If there exists a kind-preserving substitution θ' and types τ', φ' such that $\theta'(E) \vdash a : \tau'/\varphi'$, then $(\tau, \varphi, \theta, V') = W(E, a, V)$ is defined and there exists a kind-preserving substitution ψ such that $\tau' = \psi(\tau)$ and $\varphi' = \psi(\varphi)$ and $\theta' =_V \psi \circ \theta$.

PROOF. The proof proceeds by structural induction on a. We show one case to illustrate the proof. Assume $a = a_1(a_2)$. By hypothesis, we have a derivation of

$$\frac{\theta'(E) \vdash a_1 : \tau'' \xrightarrow{\varphi'} \tau'/\varphi' \qquad \theta'(E) \vdash a_2 : \tau''/\varphi'}{\theta'(E) \vdash a_1(a_2) : \tau'/\varphi'}$$

We apply the induction hypothesis to the left premise, obtaining:

$$(\tau_1, \varphi_1, \theta_1, V_1) = W(E, a_1, V) \quad \tau'' \xrightarrow{\varphi'} \tau' = \psi_1(\tau_1) \quad \varphi' = \psi_1(\varphi_1) \quad \theta' =_V \psi_1 \circ \theta_1$$

By lemma 22, the conditions are met to apply the induction hypothesis to $W(\varphi_1(E), a_2, V_1)$ and to the substitution ψ_1. We obtain:

$$(\tau_2, \varphi_2, \theta_2, V_2) = W(\theta_1(E), a_2, V_1) \quad \tau'' = \psi_2(\tau_2) \quad \varphi' = \psi_2(\varphi_2) \quad \psi_1 =_{V_1} \psi_2 \circ \theta_2$$

Take $\alpha \notin V_2$ as in the algorithm, and consider the substitution $\psi_3 = \psi_2 \oplus \{\alpha \leftarrow \tau'\}$. It is easy to see that ψ_3 is a unifier for the set of equations

$$Q = \{\theta_2(\tau_1) = \tau_2 \xrightarrow{\varphi_2} \alpha,\ \theta_2(\varphi_1) = \varphi_2\}.$$

Moreover, this set of equations is well-sorted because θ_2 is kind-preserving, and ψ_3 is kind-preserving because ψ_2 is (by induction hypothesis) and τ' is well-formed by lemma 6. In addition, the variables free in Q all belong to $V_2 \cup \{\alpha\}$ by construction of Q and by lemma 22. By theorem 21, the principal unifier $(\mu, V_3) = \mathtt{mgu}_{V_2 \cup \{\alpha\}}(Q)$ is therefore defined, and $W(E, a, V)$ does not fail. Moreover, since μ is principal, we have $\psi_3 =_{V_2 \cup \{\alpha\}} \psi_4 \circ \mu$ for some kind-preserving substitution ψ_4. We then take $\psi = \psi_4$ and show that this ψ satisfies the conclusions of the theorem. We do have

$$\psi(\tau) = \psi(\mu(\alpha)) = \psi_3(\alpha) = \tau'$$
$$\psi(\varphi) = \psi(\mu(\varphi_2)) = \psi_3(\varphi_2) = \psi_2(\varphi_2) = \varphi'.$$

Let v be a variable in V. Since $V \subseteq V_1$ and all variables not in V_1 are out of reach for θ_1, we have $FV(\theta_1(v)) \subseteq V_1$. Since $V_1 \subseteq V_2$ and all variables not in V_2 are out of reach for θ_2, we have $FV(\theta_2(\theta_1(v))) \subseteq V_2$. It follows that

$$\begin{array}{rcll}
\psi(\theta(v)) & = & \psi(\mu(\theta_2(\theta_1(v)))) & \text{by definition of } \theta \text{ in the algorithm} \\
& = & \psi_3(\theta_2(\theta_1(v))) & \text{because } FV(\theta_2(\theta_1(v))) \subseteq V_2 \cup \{\alpha\} \\
& = & \psi_2(\theta_2(\theta_1(v))) & \text{because } \alpha \notin FV(\theta_2(\theta_1(v))) \\
& = & \psi_1(\theta_1(v)) & \text{because } FV(\theta_1(v)) \subseteq V_1 \\
& = & \theta'(v) & \text{because } v \in V
\end{array}$$

This is the expected result. □

ACKNOWLEDGEMENTS

The inference algorithm for polymorphic recursion used in our implementation was designed in collaboration with Pierre Weis. We thank François Pottier and Didier Rémy for interesting discussions. Pierre Crégut, an early user of our analyzer, provided valuable feedback. The detailed comments of the anonymous referees helped improve the exposition of this paper.

REFERENCES

Baker, H. G. 1990. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Lisp and Functional Programming 1990*. ACM Press.

Cartwright, R. and Felleisen, M. 1996. Program verification through soft typing. *Computing surveys 28,* 2, 349–351.

Dimock, A., Muller, R., Turbak, F., and Wells, J. B. 1997. Strongly typed flow-directed representation transformations. In *International Conference on Functional Programming 1997*. ACM Press, 11–24.

Eidorf, P. H., Henglein, F., Mossin, C., Niss, H., Sørensen, M. H., and Tofte, M. 1999. Anno Domini: from type theory to year 2000 conversion tool. In *26th symposium Principles of Programming Languages*. ACM Press, 1–14.

Fähndrich, M. and Aiken, A. 1996. Making set-constraint based program analyses scale. Tech. Rep. 96-917, University of California at Berkeley, Computer Science Division.

Fähndrich, M. and Aiken, A. 1997. Program analysis using mixed term and set constraints. In *Static Analysis Symposium '97*. Number 1302 in Lecture Notes in Computer Science. Springer-Verlag, 114–126.

Fähndrich, M., Foster, J. S., Aiken, A., and Cu, J. 1998. Tracking down exceptions in Standard ML programs. Tech. Rep. 98-996, University of California at Berkeley, Computer Science Division.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Programming Language Design and Implementation 1998*. ACM Press, 85–96.

Flanagan, C. and Felleisen, M. 1997. Componential set-based analysis. In *Programming Language Design and Implementation 1997*. ACM Press.

Freeman, T. and Pfenning, F. 1991. Refinement types for ML. In *Programming Language Design and Implementation 1991*. ACM Press, 268–277.

Girard, J.-Y., Taylor, P., and Lafont, Y. 1990. *Proofs and Types*. Cambridge University Press.

Guzmán, J. C. and Suárez, A. 1994. A type system for exceptions. In *Proc. 1994 workshop on ML and its applications*. Research report 2265, INRIA, 127–135.

Heintze, N. 1994. Set-based analysis of ML programs. In *Lisp and Functional Programming '1994*. ACM Press, 306–317.

Henglein, F. 1992. Global tagging optimization by type inference. In *Lisp and Functional Programming 1992*. ACM Press.

Henglein, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15,* 2, 253–289.

Jagannathan, S. and Wright, A. 1998. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Trans. Program. Lang. Syst. 20,* 1, 166–207.

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. 1993. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst. 15,* 2, 290–311.

Kfoury, A. J. and Wells, J. B. 1999. Principality and decidable type inference for finite-rank intersection types. In *26th symposium Principles of Programming Languages*. ACM Press, 161–174.

Leino, K. R. M. and Nelson, G. 1998. An extended static checker for Modula-3. In *Compiler Construction '98*, K. Koskimies, Ed. Number 1383 in Lecture Notes in Computer Science. Springer-Verlag, 302–305.

Leroy, X., Vouillon, J., and Doligez, D. 1996. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/ocaml/`.

Lucassen, J. M. and Gifford, D. K. 1988. Polymorphic effect systems. In *15th symposium Principles of Programming Languages*. ACM Press, 47–57.

MacQueen, D. B., Plotkin, G., and Sethi, R. 1986. An ideal model for recursive polymorphic types. *Information and Control 71*, 95–130.

MYCROFT, A. 1984. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*. Number 167 in Lecture Notes in Computer Science. Springer-Verlag, 217–228.

NIELSON, F., NIELSON, H. R., AND HANKIN, C. 1999. *Principles of program analysis*. Springer-Verlag.

OHORI, A. 1995. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst. 17,* 6, 844–895.

PESSAUX, F. 1999. Détection statique d'exceptions non rattrapées en Objective Caml. Ph.D. thesis, Université Paris 6.

POTTIER, F. 1996. A framework for type inference with subtyping. In *International Conference on Functional Programming 1998*. ACM Press, 228–238.

POTTIER, F. 1998. Type inference in the presence of subtyping: from theory to practice. Research report 3483, INRIA. Sept.

RAMALINGAM, G., FIELD, J., AND TIP, F. 1999. Aggregate structure identification and its application to program analysis. In *26th symposium Principles of Programming Languages*. ACM Press, 119–132.

RÉMY, D. 1989. Records and variants as a natural extension of ML. In *16th symposium Principles of Programming Languages*. ACM Press, 77–88.

RÉMY, D. 1993a. Syntactic theories and the algebra of record terms. Research report 1869, INRIA.

RÉMY, D. 1993b. Type inference for records in a natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming*, C. A. Gunter and J. C. Mitchell, Eds. MIT Press.

RÉMY, D. AND VOUILLON, J. 1998. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems 4,* 1, 27–50.

SHAO, Z. AND APPEL, A. 1993. Smartest recompilation. In *20th symposium Principles of Programming Languages*. ACM Press, 439–450.

SHIVERS, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *23rd symposium Principles of Programming Languages*. ACM Press, 32–41.

TALPIN, J.-P. AND JOUVELOT, P. 1994. The type and effect discipline. *Information and Computation 111,* 2, 245–296.

TOFTE, M. AND BIRKEDAL, L. 1998. A region inference algorithm. *ACM Trans. Program. Lang. Syst. 20,* 4, 724–767.

TOFTE, M. AND TALPIN, J.-P. 1997. Region-based memory management. *Information and Computation 132,* 2, 109–176.

WAND, M. 1987. Complete type inference for simple objects. In *Logic in Computer Science 1987*. IEEE Computer Society Press, 37–44.

WRIGHT, A. K. 1995. Simple imperative polymorphism. *Lisp and Symbolic Computation 8,* 4, 343–356.

WRIGHT, A. K. AND CARTWRIGHT, R. 1997. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst. 19,* 1, 87–152.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1, 38–94.

XI, H. AND PFENNING, F. 1999. Dependent types in practical programming. In *26th symposium Principles of Programming Languages*. ACM Press, 214–227.

YI, K. 1998. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Sci. Comput. Program. 31,* 1, 147–173.

YI, K. AND RYU, S. 1997. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium '97*. Number 1302 in Lecture Notes in Computer Science. Springer-Verlag, 98–113.