Type-based Exception Analysis for Non-strict Higher-order Functional Languages with Imprecise Exception Semantics

Ruud Koot [*] Jurriaan Hage

Department of Computing and Information Sciences
Utrecht University
{r.koot,j.hage}@uu.nl

Abstract

Most statically typed functional programming languages allow programmers to write partial functions: functions that are not defined on all the elements of their domain as specified by their type. Applying a partial function to a value on which it is not defined will raise a run-time exception, thus in practice well-typed programs can and *do* still go wrong.

To warn programmers about such errors, contemporary compilers for functional languages employ a local and purely syntactic analysis to detect partial **case**-expressions—those that do not cover all possible patterns of constructors. As programs often maintain invariants on their data, restricting the potential values of the scrutinee to a subtype of its given or inferred type, many of these incomplete **case**-expressions are harmless. Such an analysis does not account for these invariants and will thus report many false positives, overwhelming the programmer.

We develop a constraint-based type system that detects harmful sources of partiality and prove it correct with respect to an imprecise exception semantics. The analysis accurately tracks the flow of both exceptions—the manifestation of partiality gone wrong—and ordinary data through the program, as well as the dependencies between them. The latter is crucial for usable precision, but has been omitted from previously published exception analyses.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Theory, Verification

Keywords type-based program analysis, exception analysis, imprecise exceptions, pattern-matching, polymorphic recursion, static contract checking

[*] This material is based upon work supported by the Netherlands Organisation for Scientific Research (NWO).

1. Introduction

Many modern programming languages come equipped with a type system. Type systems attempt to prevent bad behaviour at run-time by rejecting, at compile-time, programs that may cause such behaviour. As predicting the dynamic behaviour of an arbitrary program is undecidable, type systems—or at least those in languages relying on type inference—will always have to either reject some programs that can never cause bad behaviour, or accept some that do.

A major source of trouble are *partial functions*—functions that are not defined on all the elements of their domain, usually because they cannot be given any sensible definition on those elements. Canonical examples include the division operator, which is undefined when its right-hand argument is zero, and the *head* function (extracting the first element from a list) which is undefined on the empty list. Outright rejecting such functions does not seem like a reasonable course of action, so we are left only with the possibility of accepting them as well-typed at compile-time and having them raise an *exception* at run-time, when invoked on an element of their domain on which they were left undefined.

Still, we would like to warn the programmer when we accept a program as correctly typed that may potentially crash due to an exception at run-time. Most compilers for functional languages will already emit a warning when a function is defined by a non-exhaustive pattern-match, listing the missing patterns. These warnings are generated by a very local and syntactic analysis, however, and generate many false positives, distracting the programmer. For example, they will complain that the *head* function is missing a clause for the empty list, even if *head* is only ever invoked on a syntactically non-empty list, or not at all. Worse still are spurious warnings for non-escaping let-bound functions, for which one can no longer argue the warning is useful, as the function cannot be called from a different context at a future time.

Unlike other exception analyses that have appeared in the literature—which primarily attempt to track uncaught user-thrown or environment-induced exceptions, such as those that could be encountered when reading invalid data from disk—we are first and foremost concerned with accurately tracking the exceptions raised by failed pattern-matches. Therefore, our analysis might equally well be called a *pattern-match analysis*, although it is certainly not strictly limited to one as such. Getting results of usable accuracy—eliminating as many false positives as possible—requires carefully keeping track of the data-flow through the program. This additionally improves the accuracy of reporting potential exceptions not related to pattern matching and opens the way for employing the analysis to perform static contract checking.

1.1 Contributions

Our contributions include the following:

- We develop a type-driven—and thus *modular*—exception analysis that tracks data flow in order to give accurate warnings about exceptions raised due to pattern-match failures.
- Accuracy is achieved through the simultaneous use of *subtyping* (modelling data flow), *conditional constraints* (modelling control flow), *parametric polyvariance* (to achieve context-sensitivity) and *polyvariant recursion* (to avoid poisoning).
- The analysis works for *call-by-name* languages with an *imprecise exception semantics*. Such a semantics is necessary to justify several program transformations applied by optimizing compilers for call-by-name languages with distinguishable exceptions
- We give an operational semantics for imprecise exceptions and prove the analysis sound with respect to this semantics.
- The analysis presented in this paper is implemented as a prototype and, in addition to a pen-and-paper proof, the metatheory has been mostly mechanized in Coq. Both are available from: `http://www.staff.science.uu.nl/~0422819/tbea/`.

2. Motivation

Many algorithms maintain invariants on the data structures they use that cannot easily be encoded into their types. These invariants often ensure that certain incomplete **case**-expressions are guaranteed *not* to cause a pattern-match failure.

2.1 *risers*

An example of such an algorithm is the $risers$ function from Mitchell and Runciman (2008), which computes monotonically increasing subsegments of a list:

$$\begin{array}{l}
risers\ : Ord\ \alpha \Rightarrow [\alpha] \rightarrow [[\alpha]] \\
risers\ [\,] \qquad\qquad = [\,] \\
risers\ [x] \qquad\qquad = [[x]] \\
risers\ (x_1 :: x_2 :: xs) = \\
\quad \textbf{if}\ x_1 \leqslant x_2\ \textbf{then}\ (x_1 :: y) :: ys\ \textbf{else}\ [x_1] :: (y :: ys) \\
\qquad \textbf{where}\ (y :: ys) = risers\ (x_2 :: xs)
\end{array}$$

For example:

$$risers\ [1,3,5,1,2] \longrightarrow^{*} [[1,3,5],[1,2]].$$

The irrefutable pattern in the **where**-clause in the third alternative of $risers$ expects the recursive call to return a non-empty list. A naive analysis might raise a warning here. If we look a bit longer at the program, however, we see that we also pass the recursive call to $risers$ a non-empty list. This means we will end up in either the second or third alternative inside the recursive call. Both the second alternative and both branches of the **if**-expression in the third alternative produce a non-empty list, satisfying the assumption we made earlier and allowing us to conclude that this function is total and will never raise a pattern-match failure exception. (Raising or propagating an exception because we pattern-match on exceptional values present in the input is still possible, though.)

2.2 *bitstring*

Another example, from Freeman and Pfenning (1991), comprises a collection of mathematical operations working on bitstrings: integers encoded as lists of the binary digits 0 and 1, with the least significant bit first. We model bitstrings as the type

$$\textbf{type}\ Bitstring = [\mathbb{Z}]$$

This type is too lenient in that it does not restrict the elements of the lists to the digits 0 and 1. We can however maintain this property as an implicit invariant.

If we now define an addition operation on bitstrings:

$$\begin{array}{l}
add : Bitstring \rightarrow Bitstring \rightarrow Bitstring \\
add\ [\,] \qquad\quad y \qquad\quad = y \\
add\ x \qquad\quad [\,] \qquad\quad = x \\
add\ (0 :: x)\ (0 :: y) = 0 :: add\ x\ y \\
add\ (0 :: x)\ (1 :: y) = 1 :: add\ x\ y \\
add\ (1 :: x)\ (0 :: y) = 1 :: add\ x\ y \\
add\ (1 :: x)\ (1 :: y) = 0 :: add\ (add\ [1]\ x)\ y
\end{array}$$

we see that the patterns in add are far from complete. However, if only passed arguments that satisfy the invariant it will neither crash due to a pattern-match failure, nor invalidate the invariant.

2.3 *desugar*

Compilers work with large and complex data types to represent the abstract syntax tree. These data structures must be able to represent all syntactic constructs the parser is able to recognize. This results in an abstract syntax tree that is unnecessarily complex, and too cumbersome for the later stages of the compiler—such as the optimizer—to work with. This problem is resolved by *desugaring* the original abstract syntax tree into a simpler—but semantically equivalent—abstract syntax tree that does not use all of the constructors available in the original abstract syntax tree.

The compiler writer now has a choice between two different options: either write a desugaring stage $desugar : ComplexAST \rightarrow SimpleAST$—duplicating most of the data type representing and functions operating on the abstract syntax tree—or take the easy route $desugar : AST \rightarrow AST$ and assume certain constructors will no longer be present in the abstract syntax tree at stages of the compiler executed after the desugaring phase. The former has all the usual downsides of code duplication—such as having to manually keep multiple data types and the functions operating on them synchronized—while the latter forgoes many of the advantages of strong typing and type safety: if the compiler pipeline is restructured and one of the stages that was originally assumed to run only after desugaring suddenly runs before that point the error might only be detected at run-time by a pattern-match failure. A pattern-match analysis should be able to detect such errors statically.

3. Overview

We formulate our analysis in terms of a constraint-based type and effect system (Talpin and Jouvelot 1994; Nielson et al. 1997; Abadi et al. 1999).

3.1 Data flow

How would we capture the informal reasoning we used in Section 2.1 to convince ourselves that $risers$ does not cause a pattern-match failure using a type system? A reasonable first approach would be to annotate all list types with the kind of list it can be: **N** if it must be an empty list, a list that necessarily has a nil-constructor at the head of its spine; **C** if it must be a non-empty list having a cons-constructor at its head; **N** $\sqcup$ **C** if it can be either. We can then assign to each of the three individual branches of $risers$ the following types:

$$\begin{array}{ll}
risers_1 & : \forall\alpha. Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{N}} \rightarrow [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{N}} \\
risers_2, risers_3 & : \forall\alpha. Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{C}} \rightarrow [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{C}}
\end{array}$$

From the three individual branches we may infer:

$$risers : \forall\alpha. Ord\ \alpha \Rightarrow [\alpha]^{\mathbf{N}\sqcup\mathbf{C}} \rightarrow [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\mathbf{N}\sqcup\mathbf{C}}$$

Assigning this type to $risers$ will unfortunately still let us believe that a pattern-match failure may occur in the irrefutable pattern in the **where**-clause, as this type tells us any invocation of $risers$—including the recursive call in the **where**-clause—may evaluate to an empty list. The problem is that $risers_1$—the branch that can never be reached from the call in the **where**-clause—is *poisoning* the overall result. *Polyvariance* (or *property polymorphism*) can rescue us from this precarious situation, however. We can instead assign to each of the branches, and thereby the overall result, the type:

$$risers : \forall \alpha\beta. Ord\ \alpha \Rightarrow [\alpha]^{\beta} \to [[\alpha]^{\mathbf{N}\sqcup\mathbf{C}}]^{\beta}$$

In the recursive call to $risers$ we know the argument passed is a non-empty list, so we can instantiate β to **C**, informing us that the result of the recursive call will be a non-empty list as well and guaranteeing that the irrefutable pattern-match will succeed. There is one little subtlety here, though: in a conventional Hindley–Milner type system we are not allowed, or even able, to instantiate β to anything, as the type is kept monomorphic for recursive calls. We, therefore, have to extend our type system with *polyvariant recursion*. While inferring polymorphic recursive types is undecidable in general (Kfoury et al. 1993; Henglein 1993)—and, being an automatic analysis, we do not want to rely on any programmer-supplied annotations—earlier research (Tofte and Talpin 1994; Dussart et al. 1995; Rittri 1995; Leroy and Pessaux 2000) has shown that this special case of polyvariant recursion is often both crucial to obtain adequate precision and feasible to infer automatically.

3.2 Exception flow

The intention of our analysis is to track the exceptions that may be raised during the execution of a program. As with the data flow we express this set of exceptions as an annotation on the type of a program. For example, the program:

$$f\ x = x \div 0$$

should be given the exception type:

$$f : \forall \alpha. \mathbb{Z}^{\alpha} \xrightarrow{\emptyset} \mathbb{Z}^{\alpha \sqcup \textbf{division-by-zero}}$$

This type explains that f is a function accepting an integer as its first and only parameter. As we are working in a call-by-name language, this integer might actually still be a thunk that raises an exception from the set α when evaluated. The program then divides this argument by zero, returning the result. While the result will be of type integer, this operation is almost guaranteed to raise a division-by-zero exception. It is *almost* guaranteed and not completely guaranteed to raise a division-by-zero exception, as the division operator is strict in both of its arguments and might thus force the left-hand side argument to be evaluated before raising the **division-by-zero** exception. This evaluation might then in turn cause an exception from the set α to be raised first. The complete result type is thus an integer with an exception annotation consisting of the union (or *join*) of the exception set α on the argument together with an additional exception **division-by-zero**. Finally, we note that there is an empty exception set annotating the function space constructor, indicating that no exceptions will be raised when evaluating f to a closure.

While this approach seems promising at first, it is not immediately adequate for our purpose: detecting potential pattern-match failures that may occur at run time.

Consider the following program:

$$head\ (x :: xs) = x$$

After an initial desugaring step, a compiler will translate this program into:

$$\begin{aligned} head\ xs = \mathbf{case}\ xs\ \mathbf{of}\ & \\ [\,] &\mapsto \lightning^{\textbf{pattern-match-failure}} \\ y :: ys &\mapsto y \end{aligned}$$

which can be assigned the exception type:

$$head : \forall \tau\alpha\beta. [\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha\sqcup\beta\sqcup\textbf{pattern-match-failure}}$$

This type tells us that $head$ might always raise a **pattern-match-failure** exception, irrespective of what argument it is applied to. Clearly, we won't be able to outperform a simple syntactic analysis in this manner. What we need is a way to introduce a dependency of the exception flow on the data flow of the program, so we can express that $head$ will only raise a **pattern-match-failure** if it is possible for the argument passed to it to be an empty list. We can do so by introducing conditional constraints into our type system:

$$\begin{aligned} head : \forall \tau\alpha\beta\gamma. [\tau^{\alpha}]^{\beta} \xrightarrow{\emptyset} \tau^{\alpha\sqcup\beta\sqcup\gamma} & \\ \mathbf{with}\ \{\mathbf{N} \sqsubseteq \beta \Rightarrow \textbf{pattern-match-failure} \sqsubseteq \gamma\} & \end{aligned}$$

This type explains that $head$ will return an element of type τ that might—when inspected—raise any exception present in the elements of the list (α), the spine of the list (β) or from an additional set of exceptions (γ), with the constraint that if the list to which $head$ is applied is empty, then this exception set contains the **pattern-match-failure** exception, and otherwise is taken to be empty. (We apologize for the slight abuse of notation—using the annotation β to hold both data and exception-flow information—this will be remedied in the formal type system.)

4. Formalities

4.1 Language

As our language of discourse we take a call-by-name λ-calculus with booleans, integers, pairs, lists, exceptional values, general recursion, let-bindings, pattern-matching, and a set of primitive operators:

$$\begin{array}{rcl} v & ::= & b \mid n \mid \lightning^{\ell} \mid [\,] \mid \mathbf{close}\ e\ \mathbf{in}\ \rho \\ \\ e & ::= & x \mid v \mid \lambda x.e \mid \mathbf{fix}\ f.\ e \mid e_1\ e_2 \\ & \mid & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\ & \mid & e_1 \oplus e_2 \mid (e_1, e_2) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \\ & \mid & e_1 :: e_2 \mid \mathbf{case}\ e_1\ \mathbf{of}\ \{[\,] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \\ & \mid & \mathbf{bind}\ \rho\ \mathbf{in}\ e \end{array}$$

$$b \in \mathbb{B} \quad n \in \mathbb{Z} \quad f, x \in \mathbf{Var} \quad \ell \in \mathcal{P}(\mathbf{Lbl})$$

Syntax The values v of the language include an exceptional value $\lightning^{\ell}$ where the annotation ℓ denotes a set of *exception labels*. We leave the exception labels uninterpreted, but an actual implementation will use them to distinguish between distinct exceptions (for example, division-by-zero or a pattern-match failure), as well as to store any additional information necessary to produce informative error messages, such as source locations.

We adopt the following syntactic convention: we denote non-exceptional values by v and possibly exceptional values by v^{ℓ}, where ℓ corresponds to the set of exception labels in case v^{ℓ} is an exceptional value and corresponding to the empty set otherwise.

The **case**-construct in the language is always assumed to be complete. As our primary goal is to detect pattern-match failures produced by incomplete **case**-expressions, we assume any incomplete **case**-expressions written by the programmer has first been appropriately desugared into an equivalent complete **case**-expression by filling out any missing arms in the incomplete **case**-expression with an exceptional value $\lightning^{\textbf{pattern-match-failure}}$ before being passed to the analysis (Augustsson 1985; Maranget 2008).

The **close** and **bind** constructs are only necessary to formalize the small-step operational semantics (Section 4.4) and are assumed to be absent in the source program.

As call-by-name languages model exceptions as exceptional values, instead of exceptional control-flow, we do not need a **throw** or **raise**-construct. As most call-by-name languages omit a **catch**-construct from their pure fragment, we shall do so as well.

Static semantics We assume that the program is well-typed according to the canonical monomorphic type system and the underlying type of each subexpression is available to the analysis. In Section 6.1 we discuss how the analysis can be extended to a language with a polymorphic underlying type system.

4.2 Types

The types $\tau \in \mathbf{Ty}$ of the type system are given by:

$$\tau \quad ::= \quad \alpha \quad | \quad \tau_1 \xrightarrow{\alpha} \tau_2 \quad | \quad \tau_1 \times^\alpha \tau_2 \quad | \quad [\tau]^\alpha$$

Types are *simply annotated types*, comprised of a single base type consisting of an *annotation variable* $\alpha \in \mathbf{AnnVar}$ and the compound types for functions, pairs and lists, each having its constructor annotated with an annotation variable. Simply annotated types are given meaning in combination with a mapping or substitution from its free annotation variables to a lattice Λ, forming Λ-annotated types.

The auxiliary function $\lceil \cdot \rceil : \mathbf{Ty} \to \mathbf{AnnVar}$ extracts the outermost annotation from a simply annotated type τ:

$$\lceil \alpha \rceil = \alpha \qquad \lceil \tau_1 \times^\alpha \tau_2 \rceil = \alpha$$
$$\left\lceil \tau_1 \xrightarrow{\alpha} \tau_2 \right\rceil = \alpha \qquad \lceil [\tau]^\alpha \rceil = \alpha$$

A type τ can be combined with a constraint set C (Section 4.5) and have some of its free annotation variables quantified over into a type scheme $\sigma \in \mathbf{TySch}$:

$$\sigma \quad ::= \quad \forall \overline{\alpha}.\ \tau \ \textbf{with}\ C$$

4.3 Environments

An environment Γ binds variables to type schemes and an environment ρ binds variables to expressions:

$$\begin{aligned} \Gamma \quad &::= \quad \epsilon \quad | \quad \Gamma, x : \sigma \\ \rho \quad &::= \quad \epsilon \quad | \quad \rho, x : e \end{aligned}$$

As environments can be permuted, so long as shadowing of variables is not affected, we take the liberty of writing $\Gamma, x : \tau$ and $\rho, x : e$ to match the nearest (most recently bound) variable x in an environment.

4.4 Operational semantics

The operational semantics of the language is given in Figure 1 and models a call-by-name language with an *imprecise exception semantics* (Peyton Jones et al. 1999).

The small-step reduction relation $\rho \ \vdash \ e \longrightarrow e'$ has an explicit environment ρ, mapping variables to expressions (not necessarily values). Closures are represented by the operator **close** e **in** ρ, which closes the expression e in the environment ρ. The operator **bind** ρ **in** e binds the free variables in the expression e to expressions in the environment ρ. While similar there is one important distinction between the **close** and the **bind**-construct: a closure **close** e **in** ρ is a value, while a **bind**-expression can still be reduced further.

[E-VAR] reduces a variable by looking up the expression bound to it in the environment. As we do not allow for recursive definitions without a mediating **fix**-construct, the variable x is removed from the scope by binding e to the environment ρ, having the (nearest) binding of x removed. [E-ABS] reduces a lambda-abstraction to a closure, closing over its free variables in the current scope. [E-APP] first reduces e_1 until it has been evaluated to a function closure, after which [E-APPABS] performs the application by binding x to e_2 in the scope of e_1. The scope of e_2 is ρ_1, the scope that was closed over, while e_1 gets bound by the outer scope ρ, as it is not necessarily a value and may thus still have free variables that need to remain bound in the outer scope. In case e_1 does not evaluate to a function closure, but to an exceptional value, [E-APPEXN1] will first continue reducing the argument e_2 to a (possibly exceptional) value. Once this is done, [E-APPEXN2] will reduce the whole application to a single exceptional value with a set of exceptional labels consisting of the union of both the exception labels associated with the applicee as well as the applicant. This behaviour is part of the imprecise exception semantics of our language and will be further motivated in the reduction rules for the **if-then-else** construct. [E-LET] binds x to e_1 in the scope of e_2. [E-FIX] performs a one-step unfolding, binding any recursive occurrences of the binder to the original expression in its original scope. To reduce an **if-then-else** expression, [E-IF] will start by evaluating the conditional e_1. If it evaluates to either the value **true** or the value **false**, [E-IFTRUE] respectively [E-IFFALSE], will reduce the whole expression to the appropriate branch e_2 or e_3. In case the conditional e_1 reduces to an exceptional value $\lightning^\ell$, the rules [E-IFEXN1] and [E-IFEXN2] will continue evaluating both branches of the **if-then-else** expression, as dictated by the imprecise exception semantics. Finally, once both arms have been fully evaluated to possibly exceptional values v^{ℓ_2} and v^{ℓ_3}, [E-IFEXN3] will join all the exception labels from the conditional and both arms together—remembering that we by convention associate an empty set of exception labels with non-exceptional values—in an exceptional value $\lightning^{\ell_1 \sqcup \ell_2 \sqcup \ell_3}$. This "imprecision" is necessary to validate program transformations, such as **case**-switching, in the presence of distinguishable exceptions:

$$\begin{array}{l} \forall e_i.\textbf{if}\ e_1\ \textbf{then} \\ \qquad \textbf{if}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4 \\ \quad \textbf{else} \\ \qquad \textbf{if}\ e_2\ \textbf{then}\ e_5\ \textbf{else}\ e_6 = \textbf{if}\ e_2\ \textbf{then} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ e_1\ \textbf{then}\ e_3\ \textbf{else}\ e_5 \\ \qquad\qquad\qquad\qquad\qquad \textbf{else} \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ e_1\ \textbf{then}\ e_4\ \textbf{else}\ e_6 \end{array}$$

Note that [E-OP1] and [E-OP2] (as well as several other reduction rules) make the reduction relation non-deterministic, instead of enforcing a left-to-right evaluation order for operators by requiring its left-hand argument to already be fully evaluated. Not enforcing an evaluation order for operators will allow the compiler to apply optimizing transformations, such as making use of the associativity or commutativity of the operator. If both the left- and right-hand side of the operator reduce to a numeric value, [E-OPNUM] will reduce the expression to its interpretation $[\![n_1 \oplus n_2]\!]$. If either of, or both, the arguments reduce to an exceptional value, the rules [E-OPEXN1], [E-OPEXN2] and [E-OPEXN3] will propagate the exception labels of all the exceptional values in the expression. The rules [E-PAIR] and [E-CONS] wrap the syntactic pair and cons constructors in a closure to make them into values—in a call-by-name language constructors are only evaluated up to weak head normal form (*whnf*) and can still contain unevaluated subexpressions that

$$\frac{}{\rho, x : e \vdash x \longrightarrow \mathbf{bind}\ \rho\ \mathbf{in}\ e}\ [\text{E-VAR}] \qquad \frac{}{\rho \vdash \lambda x.e \longrightarrow \mathbf{close}\ \lambda x.e\ \mathbf{in}\ \rho}\ [\text{E-ABS}]$$

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash e_1\ e_2 \longrightarrow e_1'\ e_2}\ [\text{E-APP}] \qquad \frac{}{\rho \vdash (\mathbf{close}\ \lambda x.e_1\ \mathbf{in}\ \rho_1)\ e_2 \longrightarrow \mathbf{bind}\ (\rho_1, x : \mathbf{bind}\ \rho\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_1}\ [\text{E-APPABS}]$$

$$\frac{\rho \vdash e_2 \longrightarrow e_2'}{\rho \vdash \lightning^{\ell_1}\ e_2 \longrightarrow \lightning^{\ell_1}\ e_2'}\ [\text{E-APPEXN1}] \qquad \frac{}{\rho \vdash \lightning^{\ell_1}\ v_2^{\ell_2} \longrightarrow \lightning^{\ell_1 \sqcup \ell_2}}\ [\text{E-APPEXN2}]$$

$$\frac{}{\rho \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow \mathbf{bind}\ (\rho, x : \mathbf{bind}\ \rho\ \mathbf{in}\ e_1)\ \mathbf{in}\ e_2}\ [\text{E-LET}] \qquad \frac{}{\rho \vdash \mathbf{fix}\ f.\ e \longrightarrow \mathbf{bind}\ (\rho, f : \mathbf{bind}\ \rho\ \mathbf{in}\ \mathbf{fix}\ f.\ e)\ \mathbf{in}\ e}\ [\text{E-FIX}]$$

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3}\ [\text{E-IF}] \qquad \frac{}{\rho \vdash \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow e_2}\ [\text{E-IFTRUE}]$$

$$\frac{}{\rho \vdash \mathbf{if}\ \mathbf{false}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow e_3}\ [\text{E-IFFALSE}] \qquad \frac{\rho \vdash e_2 \longrightarrow e_2'}{\rho \vdash \mathbf{if}\ \lightning^{\ell_1}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow \mathbf{if}\ \lightning^{\ell_1}\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3}\ [\text{E-IFEXN1}]$$

$$\frac{\rho \vdash e_3 \longrightarrow e_3'}{\rho \vdash \mathbf{if}\ \lightning^{\ell_1}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \longrightarrow \mathbf{if}\ \lightning^{\ell_1}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3'}\ [\text{E-IFEXN2}] \qquad \frac{}{\rho \vdash \mathbf{if}\ \lightning^{\ell_1}\ \mathbf{then}\ v_2^{\ell_2}\ \mathbf{else}\ v_3^{\ell_3} \longrightarrow \lightning^{\ell_1 \sqcup \ell_2 \sqcup \ell_3}}\ [\text{E-IFEXN3}]$$

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash e_1 \oplus e_2 \longrightarrow e_1' \oplus e_2}\ [\text{E-OP1}] \qquad \frac{\rho \vdash e_2 \longrightarrow e_2'}{\rho \vdash e_1 \oplus e_2 \longrightarrow e_1 \oplus e_2'}\ [\text{E-OP2}] \qquad \frac{}{\rho \vdash n_1 \oplus n_2 \longrightarrow [\![n_1 \oplus n_2]\!]}\ [\text{E-OPNUM}]$$

$$\frac{}{\rho \vdash \lightning^{\ell_1} \oplus n_2 \longrightarrow \lightning^{\ell_1}}\ [\text{E-OPEXN1}] \qquad \frac{}{\rho \vdash n_1 \oplus \lightning^{\ell_2} \longrightarrow \lightning^{\ell_2}}\ [\text{E-OPEXN2}] \qquad \frac{}{\rho \vdash \lightning^{\ell_1} \oplus \lightning^{\ell_2} \longrightarrow \lightning^{\ell_1 \sqcup \ell_2}}\ [\text{E-OPEXN3}]$$

$$\frac{}{\rho \vdash (e_1, e_2) \longrightarrow \mathbf{close}\ (e_1, e_2)\ \mathbf{in}\ \rho}\ [\text{E-PAIR}] \qquad \frac{\rho \vdash e \longrightarrow e'}{\rho \vdash \mathbf{fst}\ e \longrightarrow \mathbf{fst}\ e'}\ [\text{E-FST}] \qquad \frac{\rho \vdash e \longrightarrow e'}{\rho \vdash \mathbf{snd}\ e \longrightarrow \mathbf{snd}\ e'}\ [\text{E-SND}]$$

$$\frac{}{\rho \vdash \mathbf{fst}\ (\mathbf{close}\ (e_1, e_2)\ \mathbf{in}\ \rho_1) \longrightarrow \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_1}\ [\text{E-FSTPAIR}] \qquad \frac{}{\rho \vdash \mathbf{snd}\ (\mathbf{close}\ (e_1, e_2)\ \mathbf{in}\ \rho_1) \longrightarrow \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_2}\ [\text{E-SNDPAIR}]$$

$$\frac{}{\rho \vdash \mathbf{fst}\ \lightning^{\ell} \longrightarrow \lightning^{\ell}}\ [\text{E-FSTEXN}] \qquad \frac{}{\rho \vdash \mathbf{snd}\ \lightning^{\ell} \longrightarrow \lightning^{\ell}}\ [\text{E-SNDEXN}] \qquad \frac{}{\rho \vdash e_1 :: e_2 \longrightarrow \mathbf{close}\ e_1 :: e_2\ \mathbf{in}\ \rho}\ [\text{E-CONS}]$$

$$\frac{\rho \vdash e_1 \longrightarrow e_1'}{\rho \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \longrightarrow \mathbf{case}\ e_1'\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\}}\ [\text{E-CASE}]$$

$$\frac{}{\rho \vdash \mathbf{case}\ []\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \longrightarrow e_2}\ [\text{E-CASENIL}]$$

$$\frac{}{\rho \vdash \mathbf{case}\ (\mathbf{close}\ e_1 :: e_1'\ \mathbf{in}\ \rho_1)\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \longrightarrow \mathbf{bind}\ (\rho, x_1 : \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_1, x_2 : \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_1')\ \mathbf{in}\ e_3}\ [\text{E-CASECONS}]$$

$$\frac{\rho \vdash e_2 \longrightarrow e_2'}{\rho \vdash \mathbf{case}\ \lightning^{\ell_1}\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \longrightarrow \mathbf{case}\ \lightning^{\ell_1}\ \mathbf{of}\ \{[] \mapsto e_2'; x_1 :: x_2 \mapsto e_3\}}\ [\text{E-CASEEXN1}]$$

$$\frac{\rho, x_1 : \lightning^{\emptyset}, x_2 : \lightning^{\emptyset} \vdash e_3 \longrightarrow e_3'}{\rho \vdash \mathbf{case}\ \lightning^{\ell_1}\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} \longrightarrow \mathbf{case}\ \lightning^{\ell_1}\ \mathbf{of}\ \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3'\}}\ [\text{E-CASEEXN2}]$$

$$\frac{}{\rho \vdash \mathbf{case}\ \lightning^{\ell_1}\ \mathbf{of}\ \left\{[] \mapsto v_2^{\ell_2}; x_1 :: x_2 \mapsto v_3^{\ell_3}\right\} \longrightarrow \lightning^{\ell_1 \sqcup \ell_2 \sqcup \ell_3}}\ [\text{E-CASEEXN3}]$$

$$\frac{\rho_1 \vdash e_1 \longrightarrow e_1'}{\rho \vdash \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_1 \longrightarrow \mathbf{bind}\ \rho_1\ \mathbf{in}\ e_1'}\ [\text{E-BIND1}] \qquad \frac{}{\rho \vdash \mathbf{bind}\ \rho_1\ \mathbf{in}\ v_1^{\ell_1} \longrightarrow v_1^{\ell_1}}\ [\text{E-BIND2}]$$

Figure 1. Operational semantics ($\rho \vdash e_1 \longrightarrow e_2$)

need to have their free variables closed over in their original scope. [E-FST] evaluates the argument passed to **fst** to a normal form. If it is an exceptional value, [E-FSTEXN] will propagate it; if it is a closed pair constructor, [E-FSTPAIR] will project the first argument and bind its free variables in the environment it has been closed over. Accordingly for [E-SND], [E-SNDEXN] and [E-SNDPAIR]. [E-CASE] will evaluate the scrutinee of a **case**-expression to a normal form. If it evaluates to a nil-constructor, [E-CASENIL] will select the first arm. If it evaluates to a closed cons-constructor, [E-CASECONS] will select the second arm, binding x_1 and x_2 to respectively the first and second component of the constructor in the environment the constructor was closed over. In case the scrutinee evaluates to an exceptional value [E-CASEEXN1], [E-CASEEXN2] and [E-CASEEXN3] will continue evaluating both arms and gather and propagate all exception labels encountered. In the reduction rule [E-CASEEXN2] we still need to bind x_1 and x_2 to some expression in ρ. As this expression we take $\lightning^{\emptyset}$ in both cases, as it is the least committing value in our system: it is associated with both an empty set of exceptional values, as well as with an empty set of non-exceptional values. [E-BIND1] and [E-BIND2] will continue evaluating any expression e in the given environment ρ_1 until it has been fully reduced to a value, which either contains no free variables, or has them explicitly closed over by a **close**-construct.

4.5 Constraints

A constraint c restricts the Λ-substitutions that may be applied to a simply annotated type to turn it into an Λ-annotated type. A constraint c is a conditional, consisting of a left-hand side g and a right-hand side r:

$$\begin{array}{rcl} c & ::= & g \Rightarrow r \\ g & ::= & \Lambda_\iota \sqsubseteq_\iota \alpha \quad | \quad \exists_\iota \alpha \quad | \quad g_1 \vee g_2 \quad | \quad \textbf{true} \\ r & ::= & \Lambda_\iota \sqsubseteq_\iota \alpha \quad | \quad \alpha_1 \sqsubseteq_\iota \alpha_2 \quad | \quad \tau_1 \leqslant_\iota \tau_2 \end{array}$$

The left-hand side g of a conditional constraint, its *guard*, consists of a disjunction of *atomic guards* $\Lambda_\iota \sqsubseteq_\iota \alpha$, relating an element of the lattice Λ_ι to an annotation variable α, and *non-emptiness guards* $\exists_\iota \alpha$, a predicate on the annotation variable α.

The right-hand side r of a conditional or unconditional constraint can either be an *atomic constraint* $\Lambda_\iota \sqsubseteq_\iota \alpha$, relating an element of the lattice Λ_ι to an annotation variable α, or an atomic constraint $\alpha_1 \sqsubseteq_\iota \alpha_2$, relating two annotation variables, or it can be a *structural constraint* $\tau_1 \leqslant_\iota \tau_2$, relating two simply annotated types.

The asymmetry between the allowed forms of the antecedent g and consequent r of the conditional constraints is intentional: they allow constraints to be formed that are expressive enough to build an accurate analysis, but limited enough so as to allow tractable constraint solving. Both allowing constraints of the form $g_1 \vee g_2$ on the right-hand side of a conditional, or allowing constraints of the form $\alpha_1 \sqsubseteq_\iota \alpha_2$ on the left-hand side of a conditional make constraint solving notoriously difficult: the former because it cannot be trivially decomposed into a set of simpler constraints and the latter because it does not behave monotonically under a fixed-point iteration.

The atomic and structural constraint relations are qualified by an index ι, which for the purpose of our analysis can be one of two constants:

$$\iota \quad ::= \quad \delta \quad | \quad \chi$$

Here δ is used to indicate data-flow, while χ indicates exception-flow.

To ease the syntactic burden we freely write constraint expressions indexed by the two constraint indices $\delta\chi$ simultaneously. Formally these should always be read as standing for two separate constraint expressions: one indexed by δ and the other indexed by χ. As none of the constraint expressions in this paper contain more than one such paired index, no ambiguities should arise. Furthermore,

$$\tilde{c} \quad ::= \quad c \quad | \quad g$$

and constraints **true** $\Rightarrow r$ shall be written simply as r.

Constraints are given meaning by the *constraint satisfaction* predicate $\theta \vDash \tilde{c}$, which relates a constraint c, g or r to the meaning of its free variables, which are in turn given by a pair of ground substitutions $\theta = \langle \theta_\delta, \theta_\chi \rangle$:

$$\frac{\theta_\iota \alpha_1 \sqsubseteq_\iota \theta_\iota \alpha_2}{\theta \vDash \alpha_1 \sqsubseteq_\iota \alpha_2}\ [\text{CM-Var}] \qquad \frac{\ell_\iota \sqsubseteq_\iota \theta_\iota \alpha}{\theta \vDash \ell_\iota \sqsubseteq_\iota \alpha}\ [\text{CM-Con}]$$

$$\frac{\ell_\iota \sqsubseteq_\iota \theta_\iota \alpha \quad \ell_\iota \neq \bot_\iota}{\theta \vDash \exists_\iota \alpha}\ [\text{CM-Exists}] \qquad \frac{}{\theta \vDash \textbf{true}}\ [\text{CM-True}]$$

$$\frac{\theta \vDash g_1}{\theta \vDash g_1 \vee g_2}\ [\text{CM-Left}] \qquad \frac{\theta \vDash g_2}{\theta \vDash g_1 \vee g_2}\ [\text{CM-Right}]$$

$$\frac{\theta \vDash g \Rightarrow \theta \vDash r}{\theta \vDash g \Rightarrow r}\ [\text{CM-Impl}] \qquad \frac{\theta_\iota \tau_1 \leqslant_\iota \theta_\iota \tau_2}{\theta \vDash \tau_1 \leqslant_\iota \tau_2}\ [\text{CM-Sub}]$$

Working with constraints in terms of the constraint satisfaction predicate is rather tedious, so we prefer to work with a *constraint entailment* relation $C \Vdash \tilde{c}$ and an associated constraint logic:

$$\frac{}{C \Vdash \bot_\iota \sqsubseteq_\iota \alpha}\ [\text{CL-}\bot] \qquad \frac{}{C \Vdash \alpha \sqsubseteq_\iota \top_\iota}\ [\text{CL-}\top]$$

$$\frac{C, g \Vdash r}{C \Vdash g \Rightarrow r}\ [\text{CL-}{\Rightarrow}\text{I}] \qquad \frac{C \Vdash g \Rightarrow r}{C, g \Vdash r}\ [\text{CL-}{\Rightarrow}\text{E}]$$

$$\frac{C \Vdash g_1}{C \Vdash g_1 \vee g_2}\ [\text{CL-}{\vee}\text{I}] \qquad \frac{C \Vdash g_1 \vee g_2}{C \Vdash g_2 \vee g_1}\ [\text{CL-}{\vee}\text{C}]$$

$$\frac{C \Vdash \Lambda_\iota \sqsubseteq_\iota \alpha}{C \Vdash \exists_\iota \alpha}\ [\text{CL-}\exists\text{I}] \qquad \frac{C_1 \Vdash \tilde{c}}{C_1, C_2 \Vdash \tilde{c}}\ [\text{CL-Weak}]$$

$$\frac{C \Vdash \tilde{c}_1 \quad C, \tilde{c}_1 \Vdash \tilde{c}_2}{C \Vdash \tilde{c}_2}\ [\text{CL-MP}]$$

We lift the constraint entailment relation to work on constraint sets $C_1 \Vdash C_2$ in the obvious way.

Theorem 1 (Soundness of constraint logic)**.** *If $\theta \vDash C$ and $C \Vdash D$ then $\theta \vDash D$.*

The subtyping relation is as usual for a type and effect system, thus note the subeffecting of the annotations:

$$\frac{C \Vdash \tau_1 \leqslant_\iota \tau_2 \quad C \Vdash \tau_2 \leqslant_\iota \tau_3}{C \Vdash \tau_1 \leqslant_\iota \tau_3}\ [\text{S-Trans}]$$

$$\frac{}{C \Vdash \tau \leqslant_\iota \tau}\ [\text{S-Refl}] \qquad \frac{C \Vdash \alpha_1 \sqsubseteq_\iota \alpha_2}{C \Vdash \alpha_1 \leqslant_\iota \alpha_2}\ [\text{SA-Base}]$$

$$\frac{C \Vdash \tau_3 \leqslant_\iota \tau_1 \quad C \Vdash \tau_2 \leqslant_\iota \tau_4 \quad C \Vdash \alpha_1 \sqsubseteq_\iota \alpha_2}{C \Vdash \tau_1 \xrightarrow{\alpha_1} \tau_2 \leqslant_\iota \tau_3 \xrightarrow{\alpha_2} \tau_4}\ [\text{SA-Fun}]$$

$$\frac{C \Vdash \tau_1 \leqslant_\iota \tau_3 \quad C \Vdash \tau_2 \leqslant_\iota \tau_4 \quad C \Vdash \alpha_1 \sqsubseteq_\iota \alpha_2}{C \Vdash \tau_1 \times^{\alpha_1} \tau_2 \leqslant_\iota \tau_3 \times^{\alpha_2} \tau_4}\ [\text{SA-Pair}]$$

$$\frac{C \Vdash \tau_1 \leqslant_\iota \tau_2 \quad C \Vdash \alpha_1 \sqsubseteq_\iota \alpha_2}{C \Vdash [\tau_1]^{\alpha_1} \leqslant_\iota [\tau_2]^{\alpha_2}}\ [\text{SA-List}]$$

4.6 Type system

A syntax-directed type system for exception analysis is given in Figure 2.

[T-Var] combines a lookup in the type environment with instantiation of variables quantified over in the type scheme. [T-Con] and [T-Exn] make sure that any constants and exception literals flow into the top-level annotations of their type. Constants will have to be abstracted into an element of the lattice Λ_δ, using the auxiliary function i, first. [T-App] incorporates a subtyping check between the formal and the actual parameter and flows all exceptions than can be caused by evaluating the function abstraction—as represented by the annotation on the function-space constructor—into the top-level annotation of the resulting type. The final premise flows any exceptions that can be raised by evaluating the argument to weak head normal form to the top-level annotation on the result type if it is possible for the function that the argument is applied to, to evaluate to an exceptional value. This is necessary to soundly model the imprecise exception semantics. [T-Abs] is standard, with only an additional annotation present on the function-space constructor. [T-Fix] and [T-Let] are the conventional rules for polymorphic recursion and polymorphic let-bindings with constrained types and are the only rules with a non-trivial algorithmic interpretation (see Section 5). [T-If] uses subtyping to ensure that the exceptional and non-exceptional values of both branches, as well as any exceptions that can occur while evaluating the conditional are propagated to the resulting type. Additionally, conditional constraints are used to ensure that only reachable branches

$$\frac{C \Vdash D\llbracket\overline{\beta}/\overline{\alpha}\rrbracket}{C;\Gamma, x : \forall\overline{\alpha}.\,\tau \textbf{ with } D \vdash x : \tau\llbracket\overline{\beta}/\overline{\alpha}\rrbracket}\;[\text{T-V}\textsc{ar}] \quad \frac{C \Vdash i(c) \sqsubseteq_\delta \alpha}{C;\Gamma \vdash c : \alpha}\;[\text{T-C}\textsc{on}] \quad \frac{C \Vdash \ell \sqsubseteq_\chi \lceil\tau\rceil}{C;\Gamma \vdash \lightning^\ell : \tau}\;[\text{T-E}\textsc{xn}]$$

$$\frac{C;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\alpha} \tau_2 \quad C;\Gamma \vdash e_2 : \tau_3 \quad C \Vdash \tau_3 \leqslant_{\delta\chi} \tau_1 \quad C \Vdash \tau_2 \leqslant_{\delta\chi} \tau_4 \quad C \Vdash \alpha \sqsubseteq_\chi \lceil\tau_4\rceil \quad C \Vdash \exists_\chi\alpha \Rightarrow \lceil\tau_3\rceil \sqsubseteq_\chi \lceil\tau_4\rceil}{C;\Gamma \vdash e_1\ e_2 : \tau_4}\;[\text{T-A}\textsc{pp}]$$

$$\frac{C;\Gamma, x : \tau_1 \vdash e : \tau_2}{C;\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\alpha} \tau_2}\;[\text{T-A}\textsc{bs}] \quad \frac{D;\Gamma \vdash e_1 : \tau_1 \quad C;\Gamma, x : \forall\overline{\alpha}.\,\tau_1 \textbf{ with } D \vdash e_2 : \tau_2 \quad \overline{\alpha} \cap fv(\Gamma) = \emptyset}{C;\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}\;[\text{T-L}\textsc{et}]$$

$$\frac{C \Vdash D\llbracket\overline{\beta}/\overline{\alpha}\rrbracket \quad D;\Gamma, f : \forall\overline{\alpha}.\,\tau_1 \textbf{ with } D \vdash e : \tau_2 \quad D \Vdash \tau_2 \leqslant_{\delta\chi} \tau_1 \quad \overline{\alpha} \cap fv(\Gamma) = \emptyset}{C;\Gamma \vdash \textbf{fix } f.\ e : \tau_1\llbracket\overline{\beta}/\overline{\alpha}\rrbracket}\;[\text{T-F}\textsc{ix}]$$

$$\frac{\begin{array}{c} C;\Gamma \vdash e_1 : \alpha_1 \quad C;\Gamma \vdash e_2 : \tau_2 \quad C;\Gamma \vdash e_3 : \tau_3 \\ C \Vdash \mathbf{T} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi\alpha_1 \Rightarrow \tau_2 \leqslant_{\delta\chi} \tau \quad C \Vdash \mathbf{F} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi\alpha_1 \Rightarrow \tau_3 \leqslant_{\delta\chi} \tau \quad C \Vdash \alpha_1 \sqsubseteq_\chi \lceil\tau\rceil \end{array}}{C;\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\;[\text{T-I}\textsc{f}]$$

$$\frac{C;\Gamma \vdash e_1 : \alpha_1 \quad C;\Gamma \vdash e_2 : \alpha_2 \quad C \Vdash \alpha_1 \sqsubseteq_\chi \alpha \quad C \Vdash \alpha_2 \sqsubseteq_\chi \alpha \quad C \Vdash \omega_\oplus(\alpha_1, \alpha_2, \alpha)}{C;\Gamma \vdash e_1 \oplus e_2 : \alpha}\;[\text{T-O}\textsc{p}]$$

$$\frac{C;\Gamma \vdash e_1 : \tau_1 \quad C;\Gamma \vdash e_2 : \tau_2}{C;\Gamma \vdash (e_1, e_2) : \tau_1 \times^\alpha \tau_2}\;[\text{T-P}\textsc{air}]$$

$$\frac{C;\Gamma \vdash e : \tau_1 \times^\alpha \tau_2 \quad C \Vdash \tau_1 \leqslant_{\delta\chi} \tau \quad C \Vdash \alpha \sqsubseteq_\chi \lceil\tau\rceil}{C;\Gamma \vdash \textbf{fst } e : \tau}\;[\text{T-F}\textsc{st}] \quad \frac{C;\Gamma \vdash e : \tau_1 \times^\alpha \tau_2 \quad C \Vdash \tau_2 \leqslant_{\delta\chi} \tau \quad C \Vdash \alpha \sqsubseteq_\chi \lceil\tau\rceil}{C;\Gamma \vdash \textbf{snd } e : \tau}\;[\text{T-S}\textsc{nd}]$$

$$\frac{C \Vdash \mathbf{N} \sqsubseteq_\delta \alpha}{C;\Gamma \vdash [] : [\tau]^\alpha}\;[\text{T-N}\textsc{il}] \quad \frac{\begin{array}{c} C;\Gamma \vdash e_1 : \tau_1 \quad C;\Gamma \vdash e_2 : [\tau_2]^{\alpha_2} \\ C \Vdash \tau_1 \leqslant_{\delta\chi} \tau \quad C \Vdash \tau_2 \leqslant_{\delta\chi} \tau \quad C \Vdash \mathbf{C} \sqsubseteq_\delta \alpha \quad C \Vdash \alpha_2 \sqsubseteq_\chi \alpha \end{array}}{C;\Gamma \vdash e_1 :: e_2 : [\tau]^\alpha}\;[\text{T-C}\textsc{ons}]$$

$$\frac{\begin{array}{c} C;\Gamma \vdash e_1 : [\tau_1]^{\alpha_1} \quad C;\Gamma \vdash e_2 : \tau_2 \quad C;\Gamma, x_1 : \tau_1, x_2 : [\tau_1]^\beta \vdash e_3 : \tau_3 \\ C \Vdash \mathbf{N} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi\alpha_1 \Rightarrow \tau_2 \leqslant_{\delta\chi} \tau \quad C \Vdash \mathbf{C} \sqsubseteq_\delta \alpha_1 \vee \exists_\chi\alpha_1 \Rightarrow \tau_3 \leqslant_{\delta\chi} \tau \quad C \Vdash \alpha_1 \sqsubseteq_\chi \lceil\tau\rceil \\ C \Vdash \mathbf{N} \sqcup \mathbf{C} \sqsubseteq_\delta \beta \quad C \Vdash \alpha_1 \sqsubseteq_\chi \beta \end{array}}{C;\Gamma \vdash \textbf{case } e_1 \textbf{ of } \{[] \mapsto e_2; x_1 :: x_2 \mapsto e_3\} : \tau}\;[\text{T-C}\textsc{ase}]$$

$$\frac{C;\Delta \vdash e : \sigma \quad C \vdash \Delta \bowtie \rho}{C;\Gamma \vdash \textbf{close } e \textbf{ in } \rho : \sigma}\;[\text{T-C}\textsc{lose}] \quad \frac{C;\Delta \vdash e : \sigma \quad C \vdash \Delta \bowtie \rho}{C;\Gamma \vdash \textbf{bind } \rho \textbf{ in } e : \sigma}\;[\text{T-B}\textsc{ind}]$$

Figure 2. Syntax-directed type system ($C;\Gamma \vdash e : \sigma$)

will contribute to the resulting type. A branch is considered reachable if either the conditional can evaluate to **true** respectively **false**, or because evaluating the conditional can cause an exception and we have to consider both branches as being reachable to validate the **case**-switching transformation. The typing rule [T-OP] for primitive operators will be discussed in Section 4.7. [T-PAIR] is the standard type rule for pairs, except that we add an unconstrained annotation to the pair constructor. [T-FST] and [T-SND] are standard type rules for projections from a pair. As they implicitly perform a pattern-match, they have to propagate any exceptions that can occur when evaluating the pair-constructor to the resulting type. [T-NIL] gives the nil-constructor the type list of τ, with τ unconstrained, and an annotation α indicating the head of the spine of the list can at least contain a nil-constructor. [T-CONS] merges data-flow and exception-flow of the head and elements in the tail of the list, annotates the resulting type with an α indicating the head of the spine of the list can at least contain a cons-constructor, and propagates the exceptions that can occur in the tail of the spine of the list to the resulting type. [T-CASE] is similar to [T-IF]. The additional complication lies in the fact that the pattern for a cons-constructor must also bring its two fields into scope and give them an appropriate type. Note how in [T-CONS] and [T-CASE] exceptional and non-exceptional values are treated asymmetrically. In the rule [T-CONS] exceptional values in the spine of the tail of the list flow into the result, while we only remember **C** as the non-exceptional value than can occur at the head of the spine of the resulting list. This choice means that in [T-CASE], while we have more precise information about the head of the list, we have to be *pessimistic* about the shape of the list that gets bound to x_2. Conversely, for exceptional values we have less precise information about the head of the spine of the list. We can, however, be *optimistic* about the exceptional values occuring in the spine of the list that gets bound to x_2. Not being able to do so would be disasterous for the precision of the analysis. Any further pattern-matching on the spine of x_2, whether directly or through applying operations such as *map* or *reverse* to it, would cause the—likely spurious—exceptions to propagate. Finally, [T-CLOSE] and [T-BIND] type the expression e they close over or bind in an expression environment ρ, under a type environment Δ that types the expression environment ρ. These rules relate the type environments Γ with the expression environments ρ using an *environmental consistency* relation $C \vdash \Gamma \bowtie \rho$:

$$\frac{}{C \vdash \epsilon \bowtie \epsilon}\ [\text{EC-EMPTY}]$$

$$\frac{C \vdash \Gamma \bowtie \rho \quad C; \Gamma \vdash e : \sigma}{C \vdash \Gamma, x : \sigma \bowtie \rho, x : e}\ [\text{EC-EXTEND}]$$

4.7 Primitive operators

The typing rule [T-OP] for primitive operators relies on an auxiliary function ω that assigns a constraint set for each operator, giving its abstract interpretation. For an addition operator $+$ one can take the constraint set:

$$\omega_{+}(\alpha_1, \alpha_2, \alpha) \overset{\text{def}}{=} \{\top_{\mathbb{Z}} \sqsubseteq_\delta \alpha\}$$

or the more precise:

$$\omega_{+}(\alpha_1, \alpha_2, \alpha) \overset{\text{def}}{=} \left\{\begin{array}{c} \textbf{-} \sqsubseteq_\delta \alpha_1 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \mathbf{0} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \end{array}\right\}$$

If we would extend our constraints to allow conjunctions on the left-hand side of conditionals we can even get rid of the spurious **0**'s:

$$\omega_{+}(\alpha_1, \alpha_2, \alpha) \overset{\text{def}}{=} \left\{\begin{array}{c} \textbf{-} \sqsubseteq_\delta \alpha_1 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_1 \wedge \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \mathbf{0} \sqsubseteq_\delta \alpha \\ \mathbf{0} \sqsubseteq_\delta \alpha_1 \wedge \mathbf{0} \sqsubseteq_\delta \alpha_2 \Rightarrow \mathbf{0} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \wedge \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \mathbf{0} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \end{array}\right\}$$

Similarly, we are able to detect division-by-zero exceptions caused by an integer division operator $\div$:

$$\omega_{\div}(\alpha_1, \alpha_2, \alpha) \overset{\text{def}}{=} \left\{\begin{array}{c} \mathbf{0} \sqsubseteq_\delta \alpha_2 \Rightarrow \{\textbf{div-by-0}\} \sqsubseteq_\chi \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_1 \wedge \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \textbf{-} \sqsubseteq_\delta \alpha_1 \wedge \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \wedge \textbf{-} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{-} \sqsubseteq_\delta \alpha \\ \textbf{+} \sqsubseteq_\delta \alpha_1 \wedge \textbf{+} \sqsubseteq_\delta \alpha_2 \Rightarrow \textbf{+} \sqsubseteq_\delta \alpha \\ \mathbf{0} \sqsubseteq_\delta \alpha \end{array}\right\}$$

We do impose two restrictions on the operator constraint sets; they need to be *consistent* and *monotonic*:

Definition 1. An operator constraint set $\omega_\oplus$ is said to be *consistent* with respect to an operator interpretation $[\![\cdot \oplus \cdot]\!]$ if, whenever $C; \Gamma \vdash n_1 : \alpha_1$, $C; \Gamma \vdash n_2 : \alpha_2$, and $C \Vdash \omega_\oplus(\alpha_1, \alpha_2, \alpha)$ then $C; \Gamma \vdash [\![n_1 \oplus n_2]\!] : \alpha'$ with $C \Vdash \alpha' \leqslant_{\delta\chi} \alpha$ for some α'.

This restriction states that the interpretation of an operator and its abstract interpretation by the operator constraint set should coincide. We need one slightly more technical restriction to be able to prove our system sound:

Definition 2. An operator constraint set $\omega_\oplus$ is *monotonic* if, whenever $C \Vdash \omega_\oplus(\alpha_1, \alpha_2, \alpha)$ and $C \Vdash \alpha_1' \sqsubseteq_{\delta\chi} \alpha_1$, $C \Vdash \alpha_2' \sqsubseteq_{\delta\chi} \alpha_2$, $C \Vdash \alpha \sqsubseteq_{\delta\chi} \alpha'$ then $C \Vdash \omega_\oplus(\alpha_1', \alpha_2', \alpha')$.

Informally this means that operator constraint sets can only let the result of evaluating an operator and its operands depend on the values of those operands and not the other way around: the operator constraint sets must respect the fact that we are defining a *forwards* analysis.

4.8 Declarative rules

While we have formulated the analysis directly as a syntax-directed type system, we do need to appeal to the three logical rules that have been folded into the non-logical ones—in order to make the system syntax-directed—in the metatheoretic proofs. Their formulation should hold no surprises:

$$\frac{C; \Gamma \vdash e : \forall \overline{\alpha}.\ \tau\ \textbf{with}\ D \quad C \Vdash D\left[\overline{\beta}/\overline{\alpha}\right]}{C; \Gamma \vdash e : \tau \left[\overline{\beta}/\overline{\alpha}\right]}\ [\text{T-INST}]$$

$$\frac{C, D; \Gamma \vdash e : \tau \quad \overline{\alpha} \cap fv(\Gamma; C) = \emptyset}{C; \Gamma \vdash e : \forall \overline{\alpha}.\ \tau\ \textbf{with}\ D}\ [\text{T-GEN}]$$

$$\frac{C; \Gamma \vdash e : \tau \quad C \Vdash \tau \leqslant_{\delta\chi} \tau'}{C; \Gamma \vdash e : \tau'}\ [\text{T-SUB}]$$

4.9 Metatheory

Three theorems imply the correctness of the analysis:

Theorem 2 (Conservative extension)**.** *If e is well-typed in the underlying type system, then it can be given a type in the annotated type system.*

Theorem 3 (Progress). *If $C; \Gamma \vdash e : \sigma$ then either e is a value or there exist an e', such that for any ρ with $C \vdash \Gamma \bowtie \rho$ we have $\rho \vdash e \longrightarrow e'$.*

Theorem 4 (Preservation). *If $C; \Gamma \vdash e : \sigma_1$, $\rho \vdash e \longrightarrow e'$ and $C \vdash \Gamma \bowtie \rho$ then $C; \Gamma \vdash e' : \sigma_2$ with $C \Vdash \sigma_2 \leqslant_{\delta\chi} \sigma_1$.*

5. Algorithm

The analysis can be implemented as a three-stage type inference process. In the first stage we invoke a standard Hindley–Milner type inference algorithm to make sure the input program is well-typed and to give the second stage access to the underlying types of all subexpressions. The second stage generates a set of constraints and the third stage solves those constraints.

5.1 Constraint generation

The constraint inference algorithm $\mathcal{W}$ is given in Figure 3. The case for **fix** is discussed in Section 5.3. As the **close** and **bind**-constructs are included for metatheoretic purposes only and assumed not to be present in the initial unevaluated program text, we do not need to include any cases for them in the algorithm.

The auxiliary function *freshFrom* creates a type with the same type-constructor shape as the given underlying type v, with all annotation variables fresh; *gen* quantifies over all annotation variables free in τ but not free in Γ; *inst* instantiates all quantified annotation variables in τ and C with fresh ones.

5.2 Constraint solving

The constraint solver $\mathcal{S}$ takes a constraint set C and produces a substitution θ that solves it. The solver assumes that all structural constaints ($\leqslant_\iota$) have been decomposed into atomic constraints ($\sqsubseteq_\iota$) using the syntax-directed part of the subtyping relation (SA-) as a decomposition algorithm.

The constraint solver $\mathcal{S}$ relies on the function dv that determines the *dependent variables* of a constraint c: the variables that if updated may require the constraint to be reevaluated during the fixpoint iteration.

```
$\mathcal{W} : \mathbf{Env} \times \mathbf{Expr} \to \mathbf{Ty} \times \mathcal{P}\,\mathbf{Constr}$
$\mathcal{W}\ \Gamma\ x = \mathit{inst}\ \Gamma_x$
$\mathcal{W}\ \Gamma\ c = \mathit{typeOf}\ c$
$\mathcal{W}\ \Gamma\ (\lightning^{\ell} : \upsilon) =$ do $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    return $\langle \tau, \{\, \ell \sqsubseteq_{\chi} \lceil \tau \rceil \} \rangle$
$\mathcal{W}\ \Gamma\ (\lambda x : \upsilon . e) =$ do $\tau_1 \leftarrow \mathit{freshFrom}\ \upsilon$
    $\langle \tau_2, C_2 \rangle \leftarrow \mathcal{W}\ (\Gamma, x : \tau_1)\ e$
    $\alpha \leftarrow \mathit{fresh}$
    return $\left\langle \tau_1 \xrightarrow{\alpha} \tau_2, C_2 \right\rangle$
$\mathcal{W}\ \Gamma\ (e_1\ e_2 : \upsilon) =$ do $\left\langle \tau_1 \xrightarrow{\alpha} \tau_2, C_1 \right\rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle \tau_3, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\tau_4 \leftarrow \mathit{freshFrom}\ \upsilon$
    $C_3 \leftarrow \left\{ \begin{array}{c} \tau_3 \leqslant_{\delta\chi} \tau_1, \tau_2 \leqslant_{\delta\chi} \tau_4, \alpha \sqsubseteq_{\chi} \lceil \tau_4 \rceil \\ \exists_{\chi}\ \alpha \Rightarrow \lceil \tau_3 \rceil \sqsubseteq_{\chi} \lceil \tau_4 \rceil \end{array} \right\}$
    return $\langle \tau_4, C_1 \cup C_2 \cup C_3 \rangle$
$\mathcal{W}\ \Gamma\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$
  $=$ do $\langle \tau_1, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\sigma_1 \leftarrow \mathit{gen}\ \Gamma\ \tau_1\ C_1$
    $\langle \tau_2, C_2 \rangle \leftarrow \mathcal{W}\ (\Gamma, x : \sigma_1)\ e_2$
    return $\langle \tau_2, C_1 \cup C_2 \rangle$
$\mathcal{W}\ \Gamma\ (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \upsilon)$
  $=$ do $\langle \alpha_1, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle \tau_2, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\langle \tau_3, C_3 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_3$
    $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    $C_4 \leftarrow \left\{ \begin{array}{c} \alpha_1 \sqsubseteq_{\chi} \lceil \tau \rceil \\ \mathbf{T} \sqsubseteq_{\delta} \alpha_1 \vee \exists_{\chi}\ \alpha_1 \Rightarrow \tau_2 \leqslant_{\delta\chi} \tau \\ \mathbf{F} \sqsubseteq_{\delta} \alpha_1 \vee \exists_{\chi}\ \alpha_1 \Rightarrow \tau_3 \leqslant_{\delta\chi} \tau \end{array} \right\}$
    return $\langle \tau, C_1 \cup C_2 \cup C_3 \cup C_4 \rangle$
$\mathcal{W}\ \Gamma\ (e_1 \oplus e_2) =$ do $\langle \alpha_1, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle \alpha_2, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\alpha \leftarrow \mathit{fresh}$
    $C_3 \leftarrow \{\, \alpha_1 \sqsubseteq_{\chi} \alpha, \alpha_2 \sqsubseteq_{\chi} \alpha \}$
    $C_4 \leftarrow \omega_{\oplus}(\alpha_1, \alpha_2, \alpha)$
    return $\langle \alpha, C_1 \cup C_2 \cup C_3 \cup C_4 \rangle$

$\mathcal{W}\ \Gamma\ (e_1, e_2) =$ do $\langle \tau_1, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle \tau_2, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\alpha \leftarrow \mathit{fresh}$
    return $\langle \tau_1 \times^{\alpha} \tau_2, C_1 \cup C_2 \rangle$
$\mathcal{W}\ \Gamma\ (\mathbf{fst}\ e : \upsilon) =$ do $\langle \tau_1 \times^{\alpha} \tau_2, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e$
    $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    $C_2 \leftarrow \{\, \tau_1 \leqslant_{\delta\chi} \tau, \alpha \sqsubseteq_{\chi} \lceil \tau \rceil \}$
    return $\langle \tau, C_1 \cup C_2 \rangle$
$\mathcal{W}\ \Gamma\ (\mathbf{snd}\ e : \upsilon) =$ do $\langle \tau_1 \times^{\alpha} \tau_2, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e$
    $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    $C_2 \leftarrow \{\, \tau_2 \leqslant_{\delta\chi} \tau, \alpha \sqsubseteq_{\chi} \lceil \tau \rceil \}$
    return $\langle \tau, C_1 \cup C_2 \rangle$
$\mathcal{W}\ \Gamma\ ([] : \upsilon) =$ do $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    $\alpha \leftarrow \mathit{fresh}$
    return $\langle [\tau]^{\alpha}, \{\, \mathbf{N} \sqsubseteq_{\delta} \alpha \} \rangle$
$\mathcal{W}\ \Gamma\ (e_1 :: e_2) =$ do $\langle \tau_1, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle [\tau_2]^{\alpha_2}, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\tau \leftarrow \mathit{fresh}$
    $\alpha \leftarrow \mathit{fresh}$
    $C_3 \leftarrow \left\{ \begin{array}{c} \tau_1 \leqslant_{\delta\chi} \tau, \tau_2 \leqslant_{\delta\chi} \tau \\ \mathbf{C} \sqsubseteq_{\delta} \alpha, \alpha_2 \sqsubseteq_{\chi} \alpha \end{array} \right\}$
    return $\langle [\tau]^{\alpha}, C_1 \cup C_2 \cup C_3 \rangle$
$\mathcal{W}\ \Gamma\ (\mathbf{case}\ e_1\ \mathbf{of}\ \{ [] \mapsto e_2; x_1 :: x_2 \mapsto e_3 \} : \upsilon)$
  $=$ do $\langle [\tau_1]^{\alpha_1}, C_1 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_1$
    $\langle \tau_2, C_2 \rangle \leftarrow \mathcal{W}\ \Gamma\ e_2$
    $\beta \leftarrow \mathit{fresh}$
    $\langle \tau_3, C_3 \rangle \leftarrow \mathcal{W}\ (\Gamma, x_1 : \tau_1, x_2 : [\tau_1]^{\beta})\ e_3$
    $\tau \leftarrow \mathit{freshFrom}\ \upsilon$
    $C_4 \leftarrow \left\{ \begin{array}{c} \alpha_1 \sqsubseteq_{\chi} \lceil \tau \rceil \\ \mathbf{N} \sqsubseteq_{\delta} \alpha_1 \vee \exists_{\chi}\ \alpha_1 \Rightarrow \tau_2 \leqslant_{\delta\chi} \tau \\ \mathbf{C} \sqsubseteq_{\delta} \alpha_1 \vee \exists_{\chi}\ \alpha_1 \Rightarrow \tau_3 \leqslant_{\delta\chi} \tau \\ \mathbf{N} \sqcup \mathbf{C} \sqsubseteq_{\delta} \beta, \alpha_1 \sqsubseteq_{\chi} \beta \end{array} \right\}$
    return $\langle \tau, C_1 \cup C_2 \cup C_3 \cup C_4 \rangle$
```

Figure 3. Constraint inference algorithm

5.3 Knowing when to stop

So far we have omitted the algorithmic rule for **fix**. It performs a Kleene–Mycroft fixed-point iteration:

```
$\mathcal{W}\ \Gamma\ (\mathbf{fix}\, f : \upsilon .\ e) =$ do $\sigma \leftarrow \bot_{\upsilon}$
    repeat
        $\sigma' \leftarrow \sigma$
        $\langle \tau, C \rangle \leftarrow \mathcal{W}\ (\Gamma, f : \sigma)\ e$
        $\sigma \leftarrow \mathit{gen}\ \Gamma\ \tau\ C$
        until $\sigma \preceq \sigma'$
        $\langle \tau_2, C_2 \rangle \leftarrow \mathit{inst}\ \sigma$
        return $\langle \tau_2, C_2 \rangle$
```

Problematically, it is not guaranteed that σ' will ever become a generic instance of σ. Dussart et al. (1995) show that this is the case for a simpler type of constraints (subtyping only) by noting that any variable that does not occur free in σ or Γ can be eliminated (e.g., the constraint set $\{\alpha_1 \sqsubseteq \alpha_2, \alpha_2 \sqsubseteq \alpha_3\}$ with α_2 not free in σ and Γ can be reduced to $\{\alpha_1 \sqsubseteq \alpha_3\}$). The result then follows from the fact that only a finite (quadratic) number of subtype constraints can be formed over the finite set of remaining variables.

However, we also have conditional constraints and variables occurring in their left-hand side cannot easily be eliminated. If we want to maintain soundness and termination of the analysis we will have to introduce an additional layer of approximation.

The constraint set

$$\{\Lambda_{\iota} \sqsubseteq_{\iota} \alpha \Rightarrow r, g_1 \Rightarrow \beta_1 \sqsubseteq_{\iota} \alpha, g_2 \Rightarrow \beta_2 \sqsubseteq_{\iota} \alpha\}$$

with α not free in σ or Γ, can—after having suitably extended the allowed syntax of constraints to allow for nested implications—be rewritten into:

$$\{\Lambda_{\iota} \sqsubseteq_{\iota} (g_1 \Rightarrow \beta_1 \sqcup g_2 \Rightarrow \beta_2) \Rightarrow r\}$$

As g_1 and g_2 may again contain variables that need to be eliminated, and we want to keep the size of the individual constraints bounded, we may eventually need to approximate constraints $g \Rightarrow \alpha$ by setting their guard g to **true**.

Alternatively, during the first k Kleene–Mycroft iterations we can neglect to eliminate variables that occur on the left-hand side of a constraint, building an additional finite set I of ineliminable variables. During the later iterations we intentionally introduce poisoning by instead of generating a fresh variable that will end up in the left-hand side of constraint, reusing a variable from I. The trick would then be in picking variables in such a way that would not disturb a fixed-point that may already have been reached after k iterations.

```
$\mathcal{S}$     : $\mathcal{P}$ Constr → Subst
$\mathcal{S}\ C =$ do — initialization
        for each $\alpha \in fv\ C$ do
            $\theta_\delta[\alpha] \leftarrow \emptyset$
            $\theta_\chi[\alpha] \leftarrow \emptyset$
            $D[\alpha] \leftarrow \emptyset$
        — dependency analysis of constraints
        for each $c \in C$ do
            for each $\alpha \in dv\ c$ do
                $D[tv\ c] \leftarrow D[\alpha] \cup \{c\}$
        — fix-point iteration using worklist
        $W \leftarrow C$
        while $W \not\equiv \emptyset$ do
            $\{g \Rightarrow r\} \cup W \leftarrow W$
            if $\langle\theta_\delta, \theta_\chi\rangle \vDash g$ then
                case $r$ of
                    $\Lambda_\iota \sqsubseteq_\iota \alpha$   $\mapsto$ if $\theta_\iota[\alpha] \not\equiv \theta_\iota[\alpha] \sqcup \Lambda_\iota$ then
                                      $\theta_\iota[\alpha] \leftarrow \theta_\iota[\alpha] \sqcup \Lambda_\iota$
                                      $W \leftarrow W \cup D[\alpha]$
                    $\alpha_1 \sqsubseteq_\iota \alpha_2$ $\mapsto$ if $\theta_\iota[\alpha_2] \not\equiv \theta_\iota[\alpha_1] \sqcup \theta_\iota[\alpha_2]$ then
                                      $\theta_\iota[\alpha_2] \leftarrow \theta_\iota[\alpha_1] \sqcup \theta_\iota[\alpha_2]$
                                      $W \leftarrow W \cup D[\alpha_2]$
        return $\langle\theta_\delta, \theta_\chi\rangle$

$dv$    : Constr → $\mathcal{P}$ Var
$dv\ c =$ do $V \leftarrow \emptyset$
          case $c$ of
              $g \Rightarrow \alpha_1 \sqsubseteq_\iota \alpha_2 \mapsto V \leftarrow V \cup \{\alpha_1\}$
          case $c$ of
              $\Lambda_\iota \sqsubseteq_\iota \alpha \Rightarrow r$   $\mapsto V \leftarrow V \cup \{\alpha\}$
              $\exists_\iota \alpha \Rightarrow r$         $\mapsto V \leftarrow V \cup \{\alpha\}$
              $g_1 \vee g_2 \Rightarrow r$    $\mapsto V \leftarrow V \cup dv\ g_1 \cup dv\ g_2$
          return $V$

$tv$    : Constr → Var
$tv\ c$ = do case $c$ of
              $r \Rightarrow \Lambda_\iota \sqsubseteq_\iota \alpha$   $\mapsto$ return $\alpha$
              $r \Rightarrow \alpha_1 \sqsubseteq_\iota \alpha_2 \mapsto$ return $\alpha_2$
```

Figure 4. Constraint solver

6. Extensions

6.1 Polymorphism

Consider the polymorphic function *apply*:

$$apply : \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta$$
$$apply\ f\ x = f\ x$$

As we cannot inspect an expression with a polymorphic type other than by the exceptions that it may raise when forced to weak head normal form, it is sufficient to treat them as any other base type.

Thus, the function *apply* will be given the following type by our analysis:

$$\forall\alpha\beta\gamma\delta\epsilon\zeta.\ \left(\alpha \xrightarrow{\delta} \beta\right) \xrightarrow{\epsilon} \gamma \xrightarrow{\zeta} \beta$$
$$\textbf{with } \{\gamma \leqslant_{\delta\chi} \alpha, \delta \sqsubseteq_\chi \beta, \exists_\chi\delta \Rightarrow \gamma \sqsubseteq_\chi \beta\}$$

Care needs to be taken when we instantiate the polymorphic variables of a polymorphic type in the underlying type system. When doing so we must also simultaneously update the corresponding polyvariant type used by the analysis.

For example, instantiating the polymorphic underlying type of *apply* with $[\alpha \mapsto \mathbb{Z} \times \mathbb{Z}, \beta \mapsto \mathbb{B} \times \mathbb{B}]$ will give rise to the instantiation $\left[\alpha \mapsto \eta \times^\alpha \theta, \beta \mapsto \iota \times^\beta \kappa, \gamma \mapsto \lambda \times^\gamma \mu\right]$ of the polyvariant type used in the analysis:

$$\forall\alpha\beta\gamma\delta\epsilon\zeta\eta\theta\iota\kappa\lambda\mu.\ \left(\eta \times^\alpha \theta \xrightarrow{\delta} \iota \times^\beta \kappa\right) \xrightarrow{\epsilon} \lambda \times^\gamma \mu \xrightarrow{\zeta} \iota \times^\beta \kappa$$
$$\textbf{with } \{\lambda \times^\gamma \mu \leqslant_{\delta\chi} \eta \times^\alpha \theta, \delta \sqsubseteq_\chi \beta, \exists_\chi\delta \Rightarrow \gamma \sqsubseteq_\chi \beta\}$$

In the general case we need to:

1. Generate an *almost* fresh linear type for each of the polyvariant variables positionally corresponding to a polymorphic variable that has been instantiated in the underlying type. This type is almost, but not entirely, fresh as we do need to preserve the original variable as the top-level annotation on the new fresh type. Thus, for a type substitution $\alpha \mapsto \tau$, we need $\lceil\tau\rceil = \alpha$.

   Note that multiple polyvariant type variables in the type inferred by the analysis might correspond to a single polymorphic type variable that is being instantiated in the underlying type. In the example given above both the polyvariant variables α and γ correspond positionally to the polymorphic type variable α.
2. We need to quantify over all the fresh variables introduced.
3. We need to apply the type substitution to the constraint set, but only to the structural constraints ($\leqslant_\iota$) and not the atomic constraints ($\sqsubseteq_\iota$), as the latter were generated solely to relate top-level annotations to each other.

Due to structural constraints being treated differently from atomic constraints, the analysis now must also be careful not to decompose the structural constraints into atomic constraints too early. While an implementation will already want to postpone this until right before sending all the gathered constraints to the constraint solver for performance reasons, this now also becomes important for correctness.

6.2 Algebraic data types

We admit that "non-strict higher-order functional languages with imprecise exception semantics" is something of a euphemism for Haskell. The biggest remaining piece of the puzzle to scaling this analysis to work on the full Haskell language is the support for pattern-matching on arbitrary user-defined algebraic data types.

While the construction and destruction rules we defined for lists—keeping track of the constructors that can occur at the head of the spine, assuming the constructors occurring in the tail can always be both a nil and a cons-constructor—work adequately for functions operating on nil-terminated lists, this approach will not give useful results when extended to other algebraic data types and applied to the desugaring example from Section 2.3, even if extended to keep track of the constructors that can occur in the first *k* positions of the spine. We critically rely on knowing which of the constructors can occur throughout the *whole* spine of the abstract syntax tree.

We would need to combine both approaches for an accurate analysis: we need to know which constructor can occur at the head of a data structure and which constructors can occur throughout the rest of the spine or, formulated differently, at the recursive positions of the data type. This approach is reminiscent of Catch's *multipatterns* (Mitchell and Runciman 2008).

The most straightforward implementation splits the data flow into two separate flows δ_1 and δ_2: one to track the constructors occurring at the head of a spine and one to keep track of the constructors occuring in the tail of the spine.

$$\iota, \kappa \quad ::= \quad \delta_1 \quad | \quad \delta_2 \quad | \quad \chi$$

While technically simple, the required modifications to the type system are notationally heavy as—unlike between the data flow and the exception flow—values can flow directly between δ_1 and

δ_2 and it therefore is no longer sufficient to attach a single flow index ι to atomic constraints between variables:

$$r \quad ::= \quad \ldots \quad | \quad \alpha_1 \ {}_{\iota}{\sqsubseteq}_{\kappa} \ \alpha_2 \quad | \quad \ldots$$

The updated typing rules involving lists would read:

$$\frac{C \Vdash \mathbf{N} \sqsubseteq_{\delta_{12}} \alpha}{C; \Gamma \vdash [] : [\tau]^{\alpha}} \text{ [T-NIL]}$$

$$\frac{\begin{array}{c} C; \Gamma \vdash e_1 : \tau_1 \quad C; \Gamma \vdash e_2 : [\tau_2]^{\alpha_2} \\ C \Vdash \tau_1 \leqslant_{\delta_{12}\chi} \tau \quad C \Vdash \tau_2 \leqslant_{\delta_{12}\chi} \tau \\ C \Vdash \mathbf{C} \sqsubseteq_{\delta_1} \alpha \quad C \Vdash \alpha_2 \ {}_{\chi}{\sqsubseteq}_{\chi} \ \alpha \quad C \Vdash \alpha_2 \ {}_{\delta_{12}}{\sqsubseteq}_{\delta_2} \ \alpha \end{array}}{C; \Gamma \vdash e_1 :: e_2 : [\tau]^{\alpha}} \text{ [T-CONS]}$$

$$\frac{\begin{array}{c} C; \Gamma \vdash e_1 : [\tau_1]^{\alpha_1} \quad C; \Gamma \vdash e_2 : \tau_2 \\ C; \Gamma, x_1 : \tau_1, x_2 : [\tau_1]^{\beta} \vdash e_3 : \tau_3 \\ C \Vdash \mathbf{N} \sqsubseteq_{\delta_1} \alpha_1 \vee \exists_{\chi} \alpha_1 \Rightarrow \tau_2 \leqslant_{\delta_{12}\chi} \tau \\ C \Vdash \mathbf{C} \sqsubseteq_{\delta_1} \alpha_1 \vee \exists_{\chi} \alpha_1 \Rightarrow \tau_3 \leqslant_{\delta_{12}\chi} \tau \\ C \Vdash \alpha_1 \ {}_{\delta_2}{\sqsubseteq}_{\delta_{12}} \ \beta \quad C \Vdash \alpha_1 \ {}_{\chi}{\sqsubseteq}_{\chi} \ \beta \quad C \Vdash \alpha_1 \ {}_{\chi}{\sqsubseteq}_{\chi} \ \lceil \tau \rceil \end{array}}{C; \Gamma \vdash \textbf{case } e_1 \textbf{ of } \{ [] \mapsto e_2; x_1 :: x_2 \mapsto e_3 \} : \tau} \text{ [T-CASE]}$$

Some additional issues need to be resolved to generalize to arbitrary algebraic data types. In general polyvariance in the analysis should not be directly related to polymorphism in the underlying type system, so all fields in a data type can or should be polyvariantly parameterized, irrespective of whether the field is polymorphically parameterized in the underlying system (Wansbrough 2002). Since we are making use of subtyping, the variance (co-, contra-, in- or non-) of fields should be propagated to the polyvariant parameters.

6.3 Static contract checking

Our analysis can also be used for static contract checking. A contract can be desugared into a Findler–Felleisen wrapper (Findler and Felleisen 2002), raising a contract violation exception (together with some additional information on who to blame for the violation) when a contract violation is detected. The data flow-dependence of the analysis should be able to statically determine some contract can never be violated and prevent the contract violation exception from being propagated.

6.4 Code optimization

While designed as a validating analysis, the analysis can also be used to improve the performance of compiled code. Currently, the Glasgow Haskell Compiler generates inefficient code for *risers*: it will emit code that still contains a **case**-expression with a branch testing the result in the recursive call as being a nil-constructor and raising an exception if so. Based on the results of our analysis this test can be elided. A small subtlety is that the polyvariance of our analysis should be reduced, as only a single instance of the code will be generated for all polyvariant instantiations; more ambitiously we might want to let the compiler generate specialized instances of the code for different polyvariant instances.

7. Related Work

Catch and Dialyzer The Catch case totality checker (Mitchell and Runciman 2008) employs a first-order backwards analysis, inferring preconditions on functions under which it is guaranteed that no exceptions will be raised or pattern-match failures will occur. To analyze Haskell programs, a specially crafted and incomplete defunctionalization step is required. In contrast, our forwards analysis is type-driven and will naturally work on higher-order programs. Furthermore, Catch assumes functions are strict in all their arguments, while our analysis tries to model the call-by-name semantics more accurately.

The Dialyzer discrepancy analyzer for Erlang (Lindahl and Sagonas 2006) works in a similar spirit to our analysis, except that it has a dual notion of soundness: Dialyzer will only warn about function applications that are guaranteed to generate an exception.

Exception analyses Several exception analyses have been described in the literature, primarily targeting the detection of uncaught exceptions in ML. The exception analysis in Yi (1994) is based on abstract interpretation. Guzmán and Suárez (1994) and Fahndrich et al. (1998) describe type-based exception analyses, neither are very precise. The row-based type system for exception analysis described in Leroy and Pessaux (2000) does containing a data-flow analysis component, although one that is specialized towards tracking value-carrying exceptions instead of value-dependent exceptions and thus employs a less precise unification method instead of subtyping. Glynn et al. (2002) developed the first exception analysis for non-strict languages. It is a type-based analysis using Boolean constraints and, although it does not take data flow into account, has a similar flavour to our system.

Conditional constraints The use of conditional constraints in program analysis can be traced back to Reynolds (1968). Heintze (1994) uses conditional constraints to model branches in **case**-expressions for a dead-code analysis. Constraint-based k-CFA analyses (Shivers 1988), although traditionally not formulated as a type system, can use conditional constraints to let the control flow depend on the data flow. Aiken et al. (1994) uses conditional constraints to formulate a soft-typing system for dynamic languages. Pottier (2000) developed an expressive constraint-based type system incorporating subtyping, conditional constraints and rows and applied it to several inference problems, including accurate pattern-matchings.

Refinement types (*à la *dependent types) and contract checking There has been a long line of work exploring various approaches of *refinement types* in the sense of using *dependent types* to specify contracts (also termed *contract types* or *refinement predicates*). A refinement type expressing all natural numbers greater than or equal to five would be written as:

$$\{x : \mathbb{N} \mid x \geqslant 5\}$$

Dependent ML (Xi 2007) purposely limits the expressiveness of contracts, so contract checking—although not inference—remains decidable. Xu et al. (2009) uses symbolic evaluation to check contracts on Haskell programs. Knowles and Flanagan (2010) developed a framework for *hybrid type checking*, where the checking of contracts that could not be proven to either always hold or be violated at compile-time are deferred until run-time. The work on *liquid types* by Rondon et al. (2008) attempts to automatically infer such refinements using the technique of predicate abstraction. MoChi (Kobayashi et al. 2011) employs higher-order model checking. HALO (Vytiniotis et al. 2013) is a static contract checker that works by translating a Haskell program and its contracts into first-order logic, which can then be proven using an SMT solver.

Refinement types (*à la *intersection types) The refinement type system in Freeman and Pfenning (1991) attempts to assign more accurate, refined types to already well-typed ML programs using *union* and *intersection types* (Pierce 1991) with the detection of potential pattern-match failures and reduction of warnings about incomplete patterns as one of its goals. In addition to allowing the programmer to define algebraic data types, e.g.:

$$\textbf{data } [\alpha] = [] \mid \alpha :: [\alpha]$$

it also allows the programmer to specify a finite number of "interesting" recursive types that refine those algebraic types:

$$\begin{aligned} &\textbf{rectype } [\alpha]^{\mathbf{0}} = \quad [] \\ &\textbf{rectype } [\alpha]^{\mathbf{1}} = \alpha :: [] \end{aligned}$$

The refinements $[\alpha]^{\mathbf{0}}$ and $[\alpha]^{\mathbf{1}}$ respectively select the subtypes of empty and singleton lists from the complete list type. From these recursive types, and using a type union operator, a finite type lattice can be computed automatically:

$$\begin{tikzcd}
& {[\alpha]} \arrow[d, dash] & \\
& {[\alpha]^{\mathbf{0}} \vee [\alpha]^{\mathbf{1}}} \arrow[dl, dash] \arrow[dr, dash] & \\
{[\alpha]^{\mathbf{0}}} \arrow[dr, dash] & & {[\alpha]^{\mathbf{1}}} \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

The constructors of the algebraic data type in question can then be more accurately typed in terms of intersection types over this lattice:

$$\begin{aligned} [] \quad &: \forall\alpha.\, [\alpha]^{\mathbf{0}} \\ \_ :: \_ &: \forall\alpha.\alpha \to [\alpha]^{\mathbf{0}} \to [\alpha]^{\mathbf{1}} \\ &\quad \wedge\ \alpha \to [\alpha]^{\mathbf{1}} \to [\alpha] \\ &\quad \wedge\ \alpha \to [\alpha] \quad \to [\alpha] \end{aligned}$$

Finally, the **case**-construct, interpreted as a higher-order function and specialized to lists, can be given the intersection type:

$$\begin{aligned} \forall\alpha\ \beta_1\ \beta_2.\ &[\alpha]^{\mathbf{0}} \to \beta_1 \to (\alpha \to [\alpha] \quad \to \bot\ ) \to \beta_1 \\ \wedge \quad &[\alpha]^{\mathbf{1}} \to \beta_1 \to (\alpha \to [\alpha]^{\mathbf{0}} \to \beta_2) \to \beta_1 \vee \beta_2 \\ \wedge \quad &[\alpha] \quad \to \beta_1 \to (\alpha \to [\alpha] \quad \to \beta_2) \to \beta_1 \vee \beta_2 \end{aligned}$$

Compared to our analysis the inference of intersection types would make this a *relational analysis*, while our use of subtyping and conditional constraints only define a *functional* relation between input and output variables. This would seem to imply our analysis is less precise. As the system by Freeman put some restrictions on recursive definitions of recursive types, it is unclear to us if $risers$ could be given a suitable refinement type that precludes the occurrence of pattern-match failures. Additionally, the use of intersection types leads to a superexponential blowup of the size of types in the number of **rectype**-definitions.

Acknowledgments

We would like to thank Stefan Holdermans for his contributions in the earlier stages of this research and Neil Mitchell for answering several questions related to Catch.

References

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. POPL '99, pages 147–160, 1999.

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. POPL '94, pages 163–173, 1994.

L. Augustsson. Compiling pattern matching. FPCA '85, pages 368–381, 1985.

D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. SAS '95, pages 118–135, 1995.

M. Fahndrich, J. Foster, J. Cu, and A. Aiken. Tracking down exceptions in Standard ML programs. Technical report, 1998.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. ICFP '02, pages 48–59, 2002.

T. Freeman and F. Pfenning. Refinement types for ML. PLDI '91, pages 268–277, 1991.

K. Glynn, P. J. Stuckey, M. Sulzmann, and H. Søndergaard. Exception analysis for non-strict languages. ICFP '02, pages 98–109, 2002.

J. C. Guzmán and A. Suárez. An extended type system for exceptions. ML '94, pages 127–135, 1994.

N. Heintze. Set-based analysis of ML programs. LFP '94, pages 306–317, 1994.

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

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

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, Feb. 2010.

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. PLDI '11, pages 222–233, 2011.

X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, Mar. 2000.

T. Lindahl and K. Sagonas. Practical type inference based on success typings. PPDP '06, pages 167–178, 2006.

L. Maranget. Compiling pattern matching to good decision trees. ML '08, pages 35–46, 2008.

N. Mitchell and C. Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. Haskell '08, pages 49–60, 2008.

H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *LNCS*, pages 141–171. 1997.

S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. PLDI '99, pages 25–36, 1999.

B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical report, 1991.

F. Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, Dec. 2000.

J. C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

M. Rittri. Dimension inference under polymorphic recursion. FPCA '95, pages 147–159, 1995.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. PLDI '08, pages 159–169, 2008.

O. Shivers. Control flow analysis in Scheme. PLDI '88, pages 164–174, 1988.

J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, June 1994.

M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. POPL '94, pages 188–201, 1994.

D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. POPL '13, pages 431–442, 2013.

K. Wansbrough. Simple polymorphic usage analysis. Technical report, 2002.

H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, Mar. 2007.

D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. POPL '09, pages 41–52, 2009.

K. Yi. Compile-time detection of uncaught exceptions in Standard ML programs. In *Static Analysis*, volume 864 of *LNCS*, pages 238–254. 1994.