Higher-ranked Exception Types

Ruud Koot
inbox@ruudkoot.nl

Utrecht University

Abstract. We present a type-and-effect system that derives an exception-annotated type signature for a given term of a simply typed non-strict functional language with general recursion and a list data type. This signature declares the set of exceptional values that may be present among the values of the term, or produced by terms of function type. Higher-ranked effect polymorphism and effect operators reminiscent of System F_ω help to achieve precision and clarity. By restricting the use of higher-ranked polymorphism and operators to the effects, we conjecture the inference problem to remain decidable (in contrast to the type inference problem for System F_ω). We give a type inference algorithm that builds on the techniques developed by [16]. The types in System F_ω form a simply typed λ-calculus. Similarly, the effects in our system form a simply typed algebraic λ-calculus embellished with the ACI1-structure of sets ($\lambda^\cup$). We briefly study this language in its own right.

1 Introduction

An often-heard selling point of non-strict functional languages is that they provide strong and expressive type systems that make side-effects explicit. This supposedly makes software more reliable by lessening the mental burden placed on programmers. Many programmers with a background in object-oriented languages are thus quite surprised, when making the transition to a functional language, that they lose a feature their type system formerly did provide: the tracking of uncaught exceptions.

There is an excuse for why this feature is missing from the type systems of contemporary non-strict functional languages: in a strict first-order language it is sufficient to annotate each function with a single set of uncaught exceptions the function may raise; in a non-strict higher-order language the situation becomes significantly more complicated. Let us first consider the two aspects 'higher-order' and 'non-strict' in isolation:

Higher-order functions The set of exceptions that may be raised by a higher-order function is not given by a fixed set of exceptions, but depends on the set of exceptions that may be raised by the function that is passed as its functional argument. Higher-order functions are thus *exception polymorphic*.

Non-strict evaluation In non-strictly evaluated languages, exceptions are not a form of control flow, but a kind of value. Typically the set of values of each

type is extended with an *exceptional value* $\lightning$ (more commonly denoted $\bot$, but we shall not do so to avoid ambiguity), or family of exceptional values $\lightning^\ell$. This means we do not only need to give all functions an exception-annotated function type, but give every other expression an exception-annotated type as well.

Now let us consider these two aspects in combination. Take as an example the *map* function:

$$\begin{aligned}
&\mathit{map} \;:\; \forall\alpha\beta.(\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\mathit{map} = \lambda f.\lambda xs.\ \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad [\,] \qquad\quad \mapsto [\,] \\
&\qquad\qquad (y :: ys) \mapsto f\ y :: \mathit{map}\ f\ ys
\end{aligned}$$

We denote the exception-annotated type of a term by $\widehat{\tau}\ \&\ \xi$ or $\widehat{\tau}\langle\xi\rangle$. For function types we occasionally write $\widehat{\tau}_1\langle\xi_1\rangle \xrightarrow{\xi} \widehat{\tau}_2\langle\xi_2\rangle$ instead of $(\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle)\langle\xi\rangle$. If ξ is the empty exception set, then we sometimes omit this annotation completely.

The fully exception-polymorphic and exception-annotated type, or *exception type*, of *map* is

$$\begin{aligned}
&\mathit{map} : \forall\alpha\ \beta\ e_2\ e_3.(\forall e_1.\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle) \\
&\qquad\qquad \xrightarrow{\emptyset} (\forall e_4\ e_5.[\alpha\langle e_4\rangle]\langle e_5\rangle \xrightarrow{\emptyset} [\beta\langle e_2\ e_4 \cup e_3\rangle]\langle e_5\rangle)
\end{aligned}$$

The exception type of the first argument $\forall e_1.\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle$ states that it can be instantiated with a function that accepts any exceptional value as its argument (as the exception set e_1 is universally quantified) and returns a possibly exceptional value. In case the return value is exceptional, then it is one from the exception set $e_2\ e_1$. Here e_2 is an *exception set operator*—a function that takes a number of exception sets and exception set operators, and transforms them into another exception set, for example by adding a number of new elements to them, or discarding them and returning the empty set. Furthermore, the function (closure) itself may be an exceptional value from the exception set e_3.

The exception type of the second argument $[\alpha\langle e_4\rangle]\langle e_5\rangle$ states that it should be a list. Any of the exceptional elements in the list must be exceptional values from the exception set e_4. Any exceptional values among the constructors that form the spine of the list must be exceptional values from the exception set e_5.

The result of *map* is a list with the exception type $[\beta\langle e_2\ e_4 \cup e_3\rangle]\langle e_5\rangle$. Any exceptional constructors in the spine of this list must be exceptional values from the exception set e_5, the same exception set as where exceptional values in the spine of the list argument *xs* come from. By looking at the definition of *map* we can see why this is the case: *map* only produces non-exceptional constructors, but the pattern-match on the list argument *xs* propagates any exceptional values encountered there. The elements of the list are produced by the function application $f\ y$. Recall that f has the exception type $\forall e_1.\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle$. Now, one of two things can happen:

1. If f is an exceptional function value, then it must be one from the exception set e_3. Applying the exceptional value to an argument causes the exceptional value to be propagated.
2. Otherwise, f is a non-exceptional value. The argument y has exception type $\alpha\langle e_4\rangle$—it is an element from the list argument xs—and so can only be applied to f if we instantiate e_1 to e_4 first. If $f\ y$ produces an exceptional value, then it is thus one from the exception set $e_2\ e_4$.

To account for both cases we need to take the union of the two exception sets, giving us a value with the exception type $\beta\langle e_2\ e_4 \cup e_3\rangle$.

To get a better intuition for the behavior of these exception types and exception set operators, let us see what happens when we apply *map* to two different functions: the identity function *id* and the constant exception-valued function $const\ \lightning^{\mathbf{E}}$. These two functions can individually be given the exception types:

$$\begin{aligned} id &= \lambda x.x \quad : \forall e_1.\alpha\langle e_1\rangle \xrightarrow{\emptyset} \alpha\langle e_1\rangle \\ const\ \lightning^{\mathbf{E}} &= \lambda x.\lightning^{\mathbf{E}} : \forall e_1.\alpha\langle e_1\rangle \xrightarrow{\emptyset} \beta\langle\{\mathbf{E}\}\rangle \end{aligned}$$

The term *id* merely propagates its argument to the result unchanged, so it also propagates any exceptional values unchanged. The term $const\ \lightning^{\mathbf{E}}$ discards its argument and always returns the exceptional value $\lightning^{\mathbf{E}}$. This behavior of *id* and $const\ \lightning^{\mathbf{E}}$ is also reflected in their exception types.

When we apply *map* to *id*, we need to unify the exception type of the formal parameter $\forall e_1.\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle$ with the exception type of the actual parameter $\forall e_1.\alpha\langle e_1\rangle \xrightarrow{\emptyset} \alpha\langle e_1\rangle$. This can be accomplished by instantiating e_3 to $\emptyset$ and e_2 to $\lambda x.x$—as $(\lambda x.x)\ e_1$ evaluates to e_1—giving us the resulting exception type

$$map\ id : \forall\alpha\ e_4\ e_5.[\alpha\langle e_4\rangle]\langle e_5\rangle \xrightarrow{\emptyset} [\alpha\langle e_4\rangle]\langle e_5\rangle$$

In other words, mapping the identity function over a list propagates all exceptional values already present in the list and introduces no new exceptional values.

When we apply *map* to $const\ \lightning^{\mathbf{E}}$ we unify the exception type of the formal parameter with $\forall e_1.\alpha\langle e_1\rangle \xrightarrow{\emptyset} \beta\langle\{\mathbf{E}\}\rangle$, which can be accomplished by instantiating e_3 to $\emptyset$ and e_2 to $\lambda x.\{\mathbf{E}\}$—as $(\lambda x.\{\mathbf{E}\})\ e_1$ evaluates to $\{\mathbf{E}\}$—giving us the exception type

$$map\ (const\ \lightning^{\mathbf{E}}) : \forall\alpha\ \beta\ e_4\ e_5.[\alpha\langle e_4\rangle]\langle e_5\rangle \xrightarrow{\emptyset} [\beta\langle\{\mathbf{E}\}\rangle]\langle e_5\rangle$$

In other words, mapping the constant function with the exceptional value $\lightning^{\mathbf{E}}$ as its range over a list discards all existing exceptional values from the list and produces only non-exceptional values or the exceptional value $\lightning^{\mathbf{E}}$ as elements of the list.

2 The $\lambda^{\cup}$-calculus

The $\lambda^{\cup}$-calculus is a simply typed λ-calculus extended at the term-level with empty set and singleton set constants, and a set union operator. The $\lambda^{\cup}$-calculus forms the language of effects in the type-and-effect system developed in Section 4.

2.1 Syntax, types and semantics

We let $x \in \mathbf{Var}$ range over an infinite set of variables and $c \in \mathbf{Con}$ over a non-empty set of constants. The syntax of types τ, terms t, and environments Γ is given by:

$$\tau \in \mathbf{Ty} \ ::= \ \star \mid \tau_1 \to \tau_2$$

$$t \in \mathbf{Tm} \ ::= \ x \mid \lambda x : \tau.t \mid t_1 \ t_2 \mid \emptyset \mid \{c\} \mid t_1 \cup t_2$$

$$\Gamma \in \mathbf{Env} \ ::= \ \emptyset \mid \Gamma, x : \tau$$

The typing relation of the $\lambda^{\cup}$-calculus is an extension of the typing relation of the simply typed λ-calculus.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2} \quad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \ t_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash \emptyset : \star} \quad \frac{}{\Gamma \vdash \{c\} : \star} \quad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \cup t_2 : \tau}$$

The empty set and singleton set constants are of base type and the set union of two terms can only be taken if the involved terms have the same type.

In the $\lambda^{\cup}$-calculus, terms are interpreted as sets and types as powersets.

Types and values

$$V_{\star} = \mathcal{P}(\mathbf{Con})$$
$$V_{\tau_1 \to \tau_2} = \mathcal{P}(V_{\tau_1} \to V_{\tau_2})$$

Environments

$$\rho : \mathbf{Var} \to \bigcup \{V_{\tau} \mid \tau \text{ type}\}$$

Terms

$$\begin{aligned}
[\![x]\!]_{\rho} &= \rho(x) \\
[\![\lambda x : \tau.t]\!]_{\rho} &= \left\{\lambda v \in V_{\tau}.[\![t]\!]_{\rho[x \mapsto v]}\right\} \\
[\![t_1 \ t_2]\!]_{\rho} &= \bigcup \{\varphi([\![t_2]\!]_{\rho}) \mid \varphi \in [\![t_1]\!]_{\rho}\} \\
[\![\emptyset]\!]_{\rho} &= \emptyset \\
[\![\{c\}]\!]_{\rho} &= \{c\} \\
[\![t_1 \cup t_2]\!]_{\rho} &= [\![t_1]\!]_{\rho} \cup [\![t_2]\!]_{\rho}
\end{aligned}$$

2.2 Subsumption and observational equivalence

The set-structure of the $\lambda^{\cup}$-calculus induces a partial order on the terms.

Definition 1. *Denote by $C[]$ a* context*—a $\lambda^{\cup}$-term with a single hole in it—and by $C[t]$ the term obtained by replacing the hole in $C[]$ with the term t.*

Definition 2. *Let t_1 and t_2 be terms such that $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$. We say the term t_2* subsumes *the term t_1, written $\Gamma \vdash t_1 \lesssim t_2$, if for any context $C[]$ such that $\vdash C[t_1] : \star$ and $\vdash C[t_2] : \star$ we have that $[\![C[t_1]]\!]_\emptyset \subseteq [\![C[t_2]]\!]_\emptyset$.*

Definition 3. *Let t_1 and t_2 be terms such that $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$. We say that terms t_1 and t_2 are* observationally equivalent*, denoted $\Gamma \vdash t_1 \cong t_2$, if*

1. *$\Gamma \vdash t_1 \lesssim t_2$ and $\Gamma \vdash t_2 \lesssim t_1$, or equivalently that*
2. *for any context $C[]$ such that $\vdash C[t_1] : \star$ and $\vdash C[t_2] : \star$ we have that $[\![C[t_1]]\!]_\emptyset = [\![C[t_2]]\!]_\emptyset$.*

2.3 Normalization

To reduce $\lambda^{\cup}$-terms to a canonical normal form we combine the β-reduction rule of the simply typed λ-calculus with rewrite rules that deal with the associativity, commutativity, idempotence and identity (ACI1) properties of the set union operator.

β- and γ-reduction If a term t is η-long—i.e., it cannot be η-expanded without introducing additional β-redexes—it can be written in the form

$$t = \lambda x_1 \cdots x_n . f_1(t_{11}, ..., t_{1q_1}) \cup \cdots \cup f_p(t_{p1}, ..., t_{pq_p})$$

where f_i can be a free or bound variable, a singleton-set constant, or another η-long term; and q_i is equal to the arity of f_i (for all $1 \leqslant i \leqslant p$). Here we have removed any empty set constants (unit elements), duplicate terms $f_i(t_{i1}, ..., t_{iq_i})$ (idempotent elements), and 'forgotten' how the set union operator associates.

A *normal form* v of a term t—obtained by repeatedly applying the reduction rules

$$\frac{}{(\lambda x.t_1)\ t_2 \longrightarrow t_1\,[t_2/x]} \qquad \frac{}{(t_1 \cup \cdots \cup t_n)\ t \longrightarrow t_1\ t \cup \cdots \cup t_n\ t}$$

$$\frac{}{(\lambda x.t_1) \cup \cdots \cup (\lambda x.t_n) \longrightarrow \lambda x.t_1 \cup \cdots \cup t_n}$$

and removing any empty set constants and duplicate terms—can be written as

$$v = \lambda x_1 \cdots x_n . k_1(v_{11}, ..., v_{1q_1}) \cup \cdots \cup k_p(v_{p1}, ..., v_{pq_p})$$

where k_i can be a free or bound variable, or a singleton-set constant, but not a λ-abstraction (as this would form a β-redex), nor a union (as this would form a γ_1-redex).

Canonical ordering To be able to efficiently check two normalized terms for definitional equality up to ACI1, we also need to deal with the commutativity of the union operator. We can bring normalized terms into a fully canonical form by defining a total order on terms and using it to order unions of terms.

First, pick a strict total order $\prec$ on variables and constants. The order must be fixed and be invariant under α-renaming of variables (for example, choose the De Bruijn index of a variable), but can otherwise be arbitrary. We extend this order to a total order on $\beta\gamma$-normal η-long terms lexicographically:

1. Given two fully applied terms $k(v_1, ..., v_n)$ and $k'(v'_1, ..., v'_m)$ we define:

$$\begin{aligned} k(v_1, ..., v_n) &\prec k'(v'_1, ..., v'_m) && \text{if } k \prec k' \\ k(v_1, ..., v_i, ..., v_n) &\prec k(v_1, ..., v_{i-1}, v'_i, ..., v'_m) && \text{if } v_i \prec v'_i \end{aligned}$$

2. Given two values $\lambda x_1 \cdots x_n.K_1 \cup \cdots \cup K_{i-1} \cup K_i \cup \cdots \cup K_p$ and $\lambda x_1 \cdots x_n.K_1 \cup \cdots \cup K_{i-1} \cup K'_i \cup \cdots \cup K'_q$ that have been ordered such that $K_1 \prec \cdots \prec K_{i-1} \prec K_i \prec \cdots \prec K_p$ and $K_1 \prec \cdots \prec K_{i-1} \prec K'_i \prec \cdots \prec K'_q$, we define:

$$\begin{aligned} &\lambda x_1 \cdots x_n.K_1 \cup \cdots \cup K_{i-1} \cup K_i \cup \cdots \cup K_p \\ &\qquad \prec \lambda x_1 \cdots x_n.K_1 \cup \cdots \cup K_{i-1} \cup K'_i \cup \cdots \cup K'_q \end{aligned}$$

   if $K_i \prec K'_i$.

Given a term t with the types of the free variables given by the environment Γ, we denote by $[\![t]\!]_\Gamma$ the $\beta\gamma$-normal η-long and canonically ordered reduction of the term t.

2.4 Pattern unification

Definition 4. *A $\lambda^\cup$-term is called a* pattern *if it is of the form $f(e_1, ..., e_n)$ where f is a free variable and $e_1, ..., e_n$ are distinct bound variables.*

Note that this definition is a special case of what is usually called a *pattern* in higher-order unification theory [28, 6].

If $f(e_1, ..., e_n)$ is a pattern and t a term, then the equation

$$f : \tau_1 \to \cdots \to \tau_n \to \tau \vdash \forall e_1 : \tau_1, ..., e_n : \tau_n.f(e_1, ..., e_n) = t$$

has a unique solution given by the unifier

$$\theta = [f \mapsto \lambda e_1 : \tau_1, ..., e_n : \tau_n.t]\,.$$

3 Source language

The type-and-effect system is applicable to a simple non-strict functional language that supports boolean, integer and list data types, as well as general recursion.

Terms

$$t \in \mathbf{Tm} \;::=\; x \mid c_\tau \mid \lightning^\ell_\tau \mid \lambda x : \tau.t \mid t_1\ t_2 \mid \mathbf{fix}\ x : \tau.t \\ t_1\ \mathbf{seq}\ t_2 \mid t_1 \oplus t_2 \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \\ []_\tau \mid t_1 :: t_2 \mid \mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\}$$

Values

$$v \in \mathbf{Val} \;::=\; c_\tau \mid \lambda x : \tau.t \mid \mathbf{fix}\ x : \tau.t \mid []_\tau \mid t_1 :: t_2$$

$$\widehat{v} \in \mathbf{ExnVal} \;::=\; \lightning^\ell_\tau \mid v$$

Most constructs in the source language should be familiar. The **seq**-construct evaluates the term on the left to a value and then continues evaluating the term on the right.

Missing from the language is a construct to 'catch' exceptional values. While this may be surprising to programmers familiar with strict languages, it is a common design decision to omit such a construct from the pure fragment of non-strict languages. The omission of such a construct allows for the introduction of a certain amount of non-determinism in the operational semantics of the language—giving more freedom to an optimizing compiler—without breaking referential transparency.

The values of the source language are stratified into non-exceptional values v and possibly exceptional values $\widehat{v}$.

3.1 Underlying type system

The type system of the source language is given here for reference. This is the *underlying type system* with respect to the type-and-effect system that is presented in Section 4. We assume that any term we type in the type-and-effect system is already well-typed in the underlying type system.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{}{\Gamma \vdash c_\tau : \tau} \quad \frac{}{\Gamma \vdash \lightning^\ell_\tau : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau} \quad \frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \mathbf{fix}\ x : \tau.t : \tau} \quad \frac{\Gamma \vdash t_1 : \mathbf{int} \quad \Gamma \vdash t_2 : \mathbf{int}}{\Gamma \vdash t_1 \oplus t_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ \mathbf{seq}\ t_2 : \tau_2} \quad \frac{\Gamma \vdash t_1 : \mathbf{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \tau}$$

$$\frac{}{\Gamma \vdash []_\tau : [\tau]} \quad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : [\tau]}{\Gamma \vdash t_1 :: t_2 : [\tau]}$$

$$\frac{\Gamma \vdash t_1 : [\tau_1] \quad \Gamma \vdash t_2 : \tau \quad \Gamma, x_1 : \tau_1, x_2 : [\tau_1] \vdash t_3 : \tau}{\Gamma \vdash \mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} : \tau}$$

3.2 Operational semantics

The operational semantics of the source language is given below. Note that there is a small amount of non-determinism in the order of reduction. For example, in the reduction rules for primitive operators.

We do not go so far as to have an *imprecise exception semantics* [34]. I.e., when the guard of a conditional evaluates to an exceptional value, we do not continue evaluation of the two branches in exception finding mode.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \frac{}{(\lambda x : \widehat{\tau}\ \&\ \xi.t_1)\ t_2 \longrightarrow t_1[t_2/x]} \quad \frac{}{\lightning^{\ell}\ t_2 \longrightarrow \lightning^{\ell}}$$

$$\frac{t \longrightarrow t'}{t\ \langle\xi\rangle \longrightarrow t'\ \langle\xi\rangle} \quad \frac{}{(\Lambda e :: \kappa.t)\ \langle\xi\rangle \longrightarrow t[\xi/e]}$$

$$\frac{t \longrightarrow t'}{\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t \longrightarrow \mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t'} \quad \frac{}{\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t \longrightarrow t[\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t/x]}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \oplus t_2 \longrightarrow t_1' \oplus t_2} \quad \frac{t_2 \longrightarrow t_2'}{t_1 \oplus t_2 \longrightarrow t_1 \oplus t_2'} \quad \frac{}{v_1 \oplus v_2 \longrightarrow [\![v_1 \oplus v_2]\!]}$$

$$\frac{}{\lightning^{\ell} \oplus t_2 \longrightarrow \lightning^{\ell}} \quad \frac{}{t_1 \oplus \lightning^{\ell} \longrightarrow \lightning^{\ell}}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \mathbf{seq}\ t_2 \longrightarrow t_1'\ \mathbf{seq}\ t_2} \quad \frac{}{v_1\ \mathbf{seq}\ t_2 \longrightarrow t_2} \quad \frac{}{\lightning^{\ell}\ \mathbf{seq}\ t_2 \longrightarrow \lightning^{\ell}}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3} \quad \frac{}{\mathbf{if\ true\ then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_2}$$

$$\frac{}{\mathbf{if\ false\ then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_3} \quad \frac{}{\mathbf{if}\ \lightning^{\ell}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow \lightning^{\ell}}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow \mathbf{case}\ t_1'\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\}}$$

$$\frac{}{\mathbf{case}\ []\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow t_2}$$

$$\frac{}{\mathbf{case}\ t_1 :: t_1'\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow t_3[t_1; t_1'/x_1; x_2]}$$

$$\frac{}{\mathbf{case}\ \lightning^{\ell}\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow \lightning^{\ell}}$$

The reduction rules on the second row apply to constructs that are introduced to the language in Section 4. This also holds for the additional annotations on the λ-abstraction and the **fix**-operator.

4 Exception types

The syntax of well-formed exception types is given in Figure 1. We let e range over an infinite set of exception set variables and ℓ over a finite set of exception labels. An exception type $\widehat{\tau}$ is formed out of base types (booleans and integers), compound types (lists), function types, and quantifiers (ranging over exception set variables—to avoid complicating the presentation we do *not* allow quantification over type variables, i.e. polymorphism in the underlying type system).

Kinds, types and exception annotations

$$
\begin{array}{rcll}
\kappa \in \mathbf{ExnKi} & ::= & \textsc{exn} & \text{(exception set)} \\
& | & \kappa_1 \Rightarrow \kappa_2 & \text{(exception set operator)} \\
\\
\xi, \zeta \in \mathbf{Exn} & ::= & e & \text{(exception set variables)} \\
& | & \lambda e : \kappa.\xi & \text{(exception set abstraction)} \\
& | & \xi_1\ \xi_2 & \text{(exception set application)} \\
& | & \emptyset & \text{(empty exception set)} \\
& | & \{\ell\} & \text{(singleton exception set)} \\
& | & \xi_1 \cup \xi_2 & \text{(exception set union)} \\
\\
\widehat{\tau} \in \mathbf{ExnTy} & ::= & \forall e :: \kappa.\widehat{\tau} & \text{(exception set quantification)} \\
& | & \widehat{\mathbf{bool}} & \text{(boolean type)} \\
& | & \widehat{\mathbf{int}} & \text{(integer type)} \\
& | & [\widehat{\tau}\langle\xi\rangle] & \text{(list type)} \\
& | & \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle & \text{(function type)}
\end{array}
$$

Well-formedness

$$\frac{\Delta, e :: \kappa \vdash \widehat{\tau}\ \text{wff}}{\Delta \vdash \forall e :: \kappa.\widehat{\tau}\ \text{wff}}\ [\text{W-FORALL}]$$

$$\frac{}{\Delta \vdash \widehat{\mathbf{bool}}\ \text{wff}}\ [\text{W-BOOL}] \qquad \frac{}{\Delta \vdash \widehat{\mathbf{int}}\ \text{wff}}\ [\text{W-INT}]$$

$$\frac{\Delta \vdash \widehat{\tau}\ \text{wff} \quad \Delta \vdash \xi :: \textsc{exn}}{\Delta \vdash [\widehat{\tau}\langle\xi\rangle]\ \text{wff}}\ [\text{W-LIST}]$$

$$\frac{\Delta \vdash \widehat{\tau}_1\ \text{wff}\ \Delta \vdash \xi_1 :: \textsc{exn}\ \Delta \vdash \widehat{\tau}_2\ \text{wff}\ \Delta \vdash \xi_2 :: \textsc{exn}}{\Delta \vdash \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle\ \text{wff}}\ [\text{W-ARR}]$$

Fig. 1. Exception types: syntax and well-formedness ($\Delta \vdash \widehat{\tau}$ wff)

For a list with exception type $[\widehat{\tau}\langle\xi\rangle]$ and effect ζ, the type $\widehat{\tau}$ of the elements in the list is *annotated* with an exception set expression ξ of kind EXN. This

expression gives a set of exceptions, from which any one may be raised when an element of the list is forced. The effect ζ gives a set of exceptions, from which any one may be raised when a constructor forming the spine of the list is forced.

For a function with exception type $\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$ and effect ζ, the argument of type $\widehat{\tau}_1$ is annotated with an exception set expression ξ_1 that gives a set of exceptions that may be raised if the argument is forced in the body of the function. The result of type $\widehat{\tau}_2$ is annotated with an exception set expression ξ_2 that gives the set of exceptions that may be raised when the result of the function is forced. The effect ζ gives the set of exceptions from which any one may be raised when the function closure is forced.

Example 1. The identity function

$$\begin{aligned} &id \;:\; \forall e.\widehat{\mathbf{bool}}\langle e\rangle \to \widehat{\mathbf{bool}}\langle e\rangle \;\&\; \emptyset \\ &id = \lambda x.x \end{aligned}$$

propagates any exceptional value passed to it as an argument to the result unchanged. As the identity function is constructed by a literal λ-abstraction, no exception is raised when the resulting closure is forced, hence the empty effect.

Example 2. The exceptional function value

$$\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}} : \forall e.\widehat{\mathbf{bool}}\langle e\rangle \to \widehat{\mathbf{bool}}\langle\emptyset\rangle \;\&\; \{\mathbf{E}\}$$

raises an exception when its closure is forced—as happens when it is applied to an argument, for example. As this function can never produce a result, it certainly cannot produce an exceptional value. So the result type is annotated with an empty exception set.

The exception set expressions ξ and their kinds κ are an instance of the $\lambda^{\cup}$-calculus, where exception set expressions are terms and kinds are the types. As the constants we take the set of exception labels present in the program. Two exception set expressions are considered equivalent if they are convertible as $\lambda^{\cup}$-terms, which is to say that they reduce to the same normal form.

The type system resembles System F_{ω} [11] in that we have quantification, abstraction and application at the type level. A key difference is that abstraction and application are restricted to the effects (**Exn**) and cannot be used in the types (**ExnTy**) directly. Quantification, on the other hand, is restricted to the types, where it ranges over effects, and is not allowed to appear in the effect itself. The types thus remain predicative.

4.1 Subtyping

Exception types are endowed with the usual subtyping relation for type-and-effect systems. The function type is contravariant in its first argument for both the type and the effect. The subeffecting relation $\Delta \vdash \xi_1 \leqslant \xi_2$ is the subsumption relation $\Gamma \vdash t_1 \lesssim t_2$ from the $\lambda^{\cup}$-calculus (Definition 2).

$$\frac{}{\Delta \vdash \widehat{\tau} \leqslant \widehat{\tau}}\ [\text{S-Refl}] \qquad \frac{\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \quad \Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}_3}{\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_3}\ [\text{S-Trans}]$$

$$\frac{\Delta, e :: \kappa \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2}{\Delta \vdash \forall e :: \kappa.\widehat{\tau}_1 \leqslant \forall e :: \kappa.\widehat{\tau}_2}\ [\text{S-Forall}] \qquad \frac{\Delta \vdash \widehat{\tau} \leqslant \widehat{\tau}' \quad \Delta \vdash \xi \leqslant \xi'}{\Delta \vdash [\widehat{\tau}\langle\xi\rangle] \leqslant [\widehat{\tau}'\langle\xi'\rangle]}\ [\text{S-List}]$$

$$\frac{\Delta \vdash \widehat{\tau}_1' \leqslant \widehat{\tau}_1 \quad \Delta \vdash \xi_1' \leqslant \xi_1 \quad \Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}_2' \quad \Delta \vdash \xi_2 \leqslant \xi_2'}{\Delta \vdash \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \leqslant \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_2'\langle\xi_2'\rangle}\ [\text{S-Arr}]$$

4.2 Conservative types

Any program that is typeable in the underlying type system should also have an exception type: the exception type system is a *conservative extension* of the underlying type system. Like type systems for strictness or control flow analysis—and unlike type systems for information flow security or dimensional analysis—we do not want to reject any program that is well-typed in the underlying type system, but merely provide more insight into its behavior.

If we furthermore want the type system to be modular—allowing type checking and inference to work on individual modules instead of whole programs—we cannot and need not make any assumptions about the exception types of the arguments that are applied to any function, as the function may be called from outside the module with an argument that also comes from outside the module and which we cannot know anything about.

For base and compound types that stand in an argument position their effect and any nested annotations must thus be able to be instantiated to any arbitrary exception set expression. They must therefore be exception set variables that have been universally quantified.

These observations lead to the following definition of *conservative exception types*:[1]

Definition 5. *An exception set expression ξ is* simple *if it is a single exception set variable e, an exception set expression is a* pattern *if it fits Definition 4, and any exception set expression is* conservative.

We lift these three judgments to exception types $\widehat{\tau}$ in the following manner:

- *If $\widehat{\tau} = \widehat{\mathbf{bool}}$ or $\widehat{\tau} = \widehat{\mathbf{int}}$, then $\widehat{\tau}$ is simple, a pattern and conservative.*
- *If $\widehat{\tau} = [\widehat{\tau}'\langle\xi\rangle]$, then $\widehat{\tau}$ is simple, a pattern or conservative if $\widehat{\tau}'$ and ξ are respectively simple, patterns or conservative.*
- *If $\widehat{\tau} = \forall \overline{e_i :: \kappa_i}.\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$, then $\widehat{\tau}$ is both simple and a pattern if $\widehat{\tau}_1$ and ξ_1 are simple and $\widehat{\tau}_2$ and ξ_2 are patterns; and $\widehat{\tau}$ is conservative if $\widehat{\tau}_1$ and ξ_1 are simple and $\widehat{\tau}_2$ and ξ_2 are conservative.*

Example 3. The function *tail* can be applied to any list, but may produce an additional exceptional value **E**, because it is partial:

[1] [16] call pattern types *fully parametric* and conservative types *fully flexible*.

$$\mathit{tail} : \forall e_1\ e_2.\, [\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle]\langle e_2\rangle \to [\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle]\langle e_2 \cup \{\mathbf{E}\}\rangle\ \&\ \emptyset$$

The type and effect of the argument are simple, while the type and effect of the result are conservative, making the whole type conservative.

The conjunction operator $\wedge$ can be applied to any two booleans, and—operators being strict in both arguments—will propagate any exceptional values:

$$\wedge\ : \forall e_1.\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle \to (\forall e_2.\widehat{\mathbf{boo}}\mathbf{l}\langle e_2\rangle \to \widehat{\mathbf{boo}}\mathbf{l}\langle e_1 \cup e_2\rangle)\langle\emptyset\rangle\ \&\ \emptyset$$

Here both arguments have simple types and effects.

For function types that stand in an argument position (the functional parameters of a higher-order function) the situation is slightly more complicated. For the argument of this function we can inductively assume that this is a universally quantified exception set variable. The result of this function, however, is some exception set expression that depends on the exception set variables that were quantified over in the argument. We cannot simply introduce a new exception set variable here, but must introduce a Skolem function that depends on each of the universally quantified exception set variables.

Example 4. Consider the higher-order function *apply* that applies its first argument to the second.

$$\begin{aligned}
\mathit{apply}\ :\ & \forall e_2 :: \textsc{exn}.\forall e_3 :: \textsc{exn} \Rightarrow \textsc{exn}.\\
& \quad (\forall e_1 :: \textsc{exn}.\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle \to \widehat{\mathbf{boo}}\mathbf{l}\langle e_3\ e_1\rangle)\langle e_2\rangle \to\\
& \qquad (\forall e_4 :: \textsc{exn}.\widehat{\mathbf{boo}}\mathbf{l}\langle e_4\rangle \to \widehat{\mathbf{boo}}\mathbf{l}\langle e_2 \cup e_3\ e_4\rangle)\langle\emptyset\rangle\\
& \&\ \emptyset\\
\mathit{apply} = {} & \lambda f.\lambda x.f\ x
\end{aligned}$$

The first (functional) argument of *apply* has exception type $\forall e_1 :: \textsc{exn}.\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle \to \widehat{\mathbf{boo}}\mathbf{l}\langle e_3\ e_1\rangle$ and effect e_2. It can be instantiated with any function that accepts an argument annotated with any exception set effect, and produces a result annotated with some exception set effect depending on the exception set effect of the argument; the function closure itself may raise any exception. All functions of underlying type $\mathbf{bool} \to \mathbf{bool}$ satisfy these constraints, so we are not really constrained at all.

As e_1 has been quantified over, only the exception set operator e_3 and the effect e_2 are left free. We quantify over them outside the outer function space constructor, allowing them to appear in the annotation $e_2 \cup e_3\ e_4$ on the result. The exception set operator e_3 is now applied to e_4, as the term-level application $f\ x$ instantiates the quantified exception set variable e_1 to e_4.

(Note that the exception annotation e_2 on the closure—unlike the exception set operator e_3 on the result—does not depend on the exception variable e_1, the annotation on the argument. As a closure is already a value, it being exceptional or not can never depend on the argument it is later applied to.)

Example 5. The semantics of terms in the source language is not invariant under η-conversion in the presence of exceptional values—thus neither are exception types. The term

$$\lambda x : \mathbf{bool}.\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}}\ x : \forall e :: \text{EXN}.\widehat{\mathbf{bool}}\langle e\rangle \xrightarrow{\emptyset} \widehat{\mathbf{bool}}\langle\{\mathbf{E}\}\rangle$$

does not have the same exception type as the η-equivalent term

$$\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}} \qquad : \forall e :: \text{EXN}.\widehat{\mathbf{bool}}\langle e\rangle \xrightarrow{\{\mathbf{E}\}} \widehat{\mathbf{bool}}\langle\emptyset\rangle$$

They cannot be distinguished by applying them to an argument

$$\begin{array}{ll}(\lambda x : \mathbf{bool}.\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}}\ x)\ \mathbf{true} & : \widehat{\mathbf{bool}}\ \&\ \{\mathbf{E}\}\\ \lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}}\ \mathbf{true} & : \widehat{\mathbf{bool}}\ \&\ \{\mathbf{E}\}\end{array}$$

but they can be distinguished by forcing the closure

$$\begin{array}{ll}(\lambda x : \mathbf{bool}.\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}}\ x)\ \mathbf{seq}\ \mathbf{true} & : \widehat{\mathbf{bool}}\ \&\ \emptyset\\ \lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}}\ \mathbf{seq}\ \mathbf{true} & : \widehat{\mathbf{bool}}\ \&\ \{\mathbf{E}\}\end{array}$$

4.3 Exception type completion

Given an underlying type τ we can compute the most general exception type $\widehat{\tau}$ that erases to τ. This is done using the type completion system below, that defines a type completion relation $\Delta \vdash \tau : \widehat{\tau}\ \&\ \xi \triangleright \Delta'$.

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\mathbf{bool}}\ \&\ e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \text{EXN}}\ [\text{C-BOOL}]$$

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{int} : \widehat{\mathbf{int}}\ \&\ e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \text{EXN}}\ [\text{C-INT}]$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau}\ \&\ \xi \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau}\langle\xi\rangle]\ \&\ e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \text{EXN}, \overline{e_j :: \kappa_j}}\ [\text{C-LIST}]$$

$$\frac{\vdash \tau_1 : \widehat{\tau}_1\ \&\ \xi_1 \triangleright \overline{e_j :: \kappa_j} \qquad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau}_2\ \&\ \xi_2 \triangleright \overline{e_k :: \kappa_k}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \to \tau_2 : \forall \overline{e_j :: \kappa_j}.\, (\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle)\ \&\ e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \text{EXN}, \overline{e_k :: \kappa_k}}\ [\text{C-ARR}]$$

A judgment $\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau}\ \&\ \xi \triangleright \overline{e_j :: \kappa_j}$ is read: if the kinded exception set variables $\overline{e_i :: \kappa_i}$ are in scope, then the underlying type τ is completed to the exception type $\widehat{\tau}$ and effect ξ, while introducing the kinded free exception set variables $\overline{e_j :: \kappa_j}$. A completed exception type is always a pattern type.

Example 6. The higher-order underlying type

$$[\mathbf{bool} \to \mathbf{bool}] \to [\mathbf{bool}] \to [\mathbf{bool}]$$

is completed to the pattern type

$$\begin{aligned}
&\forall e_2 :: \text{EXN}.\forall e_2' :: \text{EXN}.\forall e_3 :: \text{EXN} \Rightarrow \text{EXN}.\\
&\quad [\forall e_1 :: \text{EXN}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{e_2'} \widehat{\mathbf{bool}}\langle e_3\ e_1\rangle]\langle e_2\rangle \rightarrow\\
&\qquad (\forall e_5 :: \text{EXN}.\forall e_5' :: \text{EXN}.[\widehat{\mathbf{bool}}\langle e_5'\rangle]\langle e_5\rangle \xrightarrow{e_6\ e_2\ e_2'\ e_3}\\
&\qquad\qquad [\widehat{\mathbf{bool}}\langle e_7'\ e_2\ e_2'\ e_3\ e_5\ e_5'\rangle]\langle e_7\ e_2\ e_2'\ e_3\ e_5\ e_5'\rangle)
\end{aligned}$$

with effect e_4, and while introducing the free variables

$$\begin{aligned}
e_4 &:: \text{EXN},\\
e_6 &:: \text{EXN} \Rightarrow \text{EXN} \Rightarrow (\text{EXN} \Rightarrow \text{EXN}) \Rightarrow \text{EXN},\\
e_7, e_7' &:: \text{EXN} \Rightarrow \text{EXN} \Rightarrow (\text{EXN} \Rightarrow \text{EXN}) \Rightarrow \text{EXN} \Rightarrow \text{EXN} \Rightarrow \text{EXN}
\end{aligned}$$

Note that the types of both arguments are simple types with simple exception annotations. However, as the first argument is a functional argument, the result type of that function is still a pattern.

The exception annotation on the right-most function-space constructor is a pattern that depends on e_2, e_2' and e_3. While we previously noted that the annotation on a function-space constructor cannot depend on the annotation belonging to the argument of that function, it is possible for a set of exceptional values that the closure may come to depend on any previous arguments of the whole function. This is more concretely demonstrated by the following function:

$$\begin{aligned}
&f :: \forall e_1, e_2 :: \text{EXN}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{\emptyset} \widehat{\mathbf{bool}}\langle e_2\rangle \xrightarrow{e_1} \widehat{\mathbf{bool}}\langle e_2\rangle\\
&f = \lambda x : \mathbf{bool}.x\ \mathbf{seq}\ \lambda y : \mathbf{bool}.y
\end{aligned}$$

Whether the closure that is returned after partially applying f to one argument is an exceptional value or not, depends on that argument x being exceptional or not.

4.4 Least exception types

Besides completing an underlying type τ to a most general exception type, we also want to compute a least exception type $\bot_\tau$. Given an effect kind $\overline{\kappa_i \Rightarrow}\text{EXN}$, denote by $\emptyset_{\overline{\kappa_i \Rightarrow}\text{EXN}}$ the effect $\lambda\overline{e_i :: \kappa_i}.\emptyset$. We can construct a least exception type by first completing the type τ to the most general exception type, and then substituting $\emptyset_{\kappa_j}$ for all free freshly introduced exception set variables $\overline{e_j :: \kappa_j}$.

Example 7. The least exception type

$$\bot_{[\mathbf{bool}\rightarrow\mathbf{bool}]\rightarrow[\mathbf{bool}]\rightarrow[\mathbf{bool}]}$$

is the conservative type

$$\begin{aligned}
&\forall e_2 :: \text{EXN}.\forall e_2' :: \text{EXN}.\forall e_3 :: \text{EXN} \Rightarrow \text{EXN}.\\
&\quad [\forall e_1 :: \text{EXN}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{e_2'} \widehat{\mathbf{bool}}\langle e_3\ e_1\rangle]\langle e_2\rangle \rightarrow\\
&\qquad (\forall e_5 :: \text{EXN}.\forall e_5' :: \text{EXN}.[\widehat{\mathbf{bool}}\langle e_5'\rangle]\langle e_5\rangle \xrightarrow{\emptyset} [\widehat{\mathbf{bool}}\langle\emptyset\rangle]\langle\emptyset\rangle)
\end{aligned}$$

$$\frac{}{\Gamma, x : \widehat{\tau} \;\&\; \xi; \Delta \vdash x : \widehat{\tau} \;\&\; \xi} \text{ [T-Var]}$$

$$\frac{}{\Gamma; \Delta \vdash c_\tau : \bot_\tau \;\&\; \emptyset} \text{ [T-Con]} \qquad \frac{}{\Gamma; \Delta \vdash \lightning^\ell_\tau : \bot_\tau \;\&\; \{\ell\}} \text{ [T-Crash]}$$

$$\frac{\Gamma, x : \widehat{\tau}_1 \;\&\; \xi_1; \Delta \vdash t : \widehat{\tau}_2 \;\&\; \xi_2}{\Gamma; \Delta \vdash \lambda x : \widehat{\tau}_1 \;\&\; \xi_1 . t : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \;\&\; \emptyset} \text{ [T-Abs]}$$

$$\frac{\Gamma; \Delta, e :: \kappa \vdash t : \widehat{\tau} \;\&\; \xi \quad e \notin \mathrm{fv}(\Gamma, \xi)}{\Gamma; \Delta \vdash \Lambda e :: \kappa . t : \forall e :: \kappa . \widehat{\tau} \;\&\; \xi} \text{ [T-AnnAbs]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \;\&\; \xi \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau}_2 \;\&\; \xi_2}{\Gamma; \Delta \vdash t_1 \; t_2 : \widehat{\tau} \;\&\; \xi} \text{ [T-App]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \forall e :: \kappa . \widehat{\tau} \;\&\; \xi \quad \Delta \vdash \xi_2 : \kappa}{\Gamma; \Delta \vdash t_1 \; \langle\xi_2\rangle : \widehat{\tau}[\xi_2/e] \;\&\; \xi} \text{ [T-AnnApp]}$$

$$\frac{\Gamma, x : \widehat{\tau} \;\&\; \xi; \Delta \vdash t : \widehat{\tau} \;\&\; \xi}{\Gamma; \Delta \vdash \mathbf{fix}\; x : \widehat{\tau} \;\&\; \xi . t : \widehat{\tau} \;\&\; \xi} \text{ [T-Fix]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\mathbf{int}} \;\&\; \xi \quad \Gamma; \Delta \vdash t_2 : \widehat{\mathbf{int}} \;\&\; \xi}{\Gamma; \Delta \vdash t_1 \oplus t_2 : \widehat{\mathbf{bool}} \;\&\; \xi} \text{ [T-Op]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau}_1 \;\&\; \xi \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau}_2 \;\&\; \xi}{\Gamma; \Delta \vdash t_1 \; \mathbf{seq} \; t_2 : \widehat{\tau}_2 \;\&\; \xi} \text{ [T-Seq]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\mathbf{bool}} \;\&\; \xi \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau} \;\&\; \xi \quad \Gamma; \Delta \vdash t_3 : \widehat{\tau} \;\&\; \xi}{\Gamma; \Delta \vdash \mathbf{if} \; t_1 \; \mathbf{then} \; t_2 \; \mathbf{else} \; t_3 : \widehat{\tau} \;\&\; \xi} \text{ [T-If]}$$

$$\frac{}{\Gamma; \Delta \vdash []_\tau : [\bot_\tau\langle\emptyset\rangle] \;\&\; \emptyset} \text{ [T-Nil]}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau} \;\&\; \xi_1 \quad \Gamma; \Delta \vdash t_2 : [\widehat{\tau}\langle\xi_1\rangle] \;\&\; \xi_2}{\Gamma; \Delta \vdash t_1 :: t_2 : [\widehat{\tau}\langle\xi_1\rangle] \;\&\; \xi_2} \text{ [T-Cons]}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash t_1 : [\widehat{\tau}_1\langle\xi_1\rangle] \;\&\; \xi' \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau} \;\&\; \xi \\ \Gamma, x_1 : \widehat{\tau}_1 \;\&\; \xi_1, x_2 : [\widehat{\tau}_1\langle\xi_1\rangle] \;\&\; \xi'; \Delta \vdash t_3 : \widehat{\tau} \;\&\; \xi \quad \Delta \vdash \xi' \leqslant \xi\end{array}}{\Gamma; \Delta \vdash \mathbf{case} \; t_1 \; \mathbf{of} \; \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} : \widehat{\tau} \;\&\; \xi} \text{ [T-Case]}$$

$$\frac{\Gamma; \Delta \vdash t : \widehat{\tau}' \;\&\; \xi' \quad \Delta \vdash \widehat{\tau}' \leqslant \widehat{\tau} \quad \Delta \vdash \xi' \leqslant \xi}{\Gamma; \Delta \vdash t : \widehat{\tau} \;\&\; \xi} \text{ [T-Sub]}$$

Fig. 2. Declarative type system ($\Gamma; \Delta \vdash t : \widehat{\tau} \;\&\; \xi$)

$$\frac{}{\Gamma, x : \widehat{\tau} \;\&\; \xi; \Delta \vdash x \hookrightarrow x : \widehat{\tau} \;\&\; \xi} \text{ [L-Var]}$$

$$\frac{}{\Gamma; \Delta \vdash c_\tau \hookrightarrow c_\tau : \bot_\tau \;\&\; \emptyset} \text{ [L-Con]} \quad \frac{}{\Gamma; \Delta \vdash \lightning^\ell_\tau \hookrightarrow \lightning^\ell_\tau : \bot_\tau \;\&\; \{\ell\}} \text{ [L-Crash]}$$

$$\frac{\begin{array}{c}\Delta, \overline{e_i :: \kappa_i} \vdash \widehat{\tau}_1 \downarrow \tau_1 \quad \Delta, \overline{e_i :: \kappa_i} \vdash \xi_1 :: \textsc{exn} \\ \Gamma, x : \widehat{\tau}_1 \;\&\; \xi_1; \Delta, \overline{e_i :: \kappa_i} \vdash t \hookrightarrow t' : \widehat{\tau}_2 \;\&\; \xi_2\end{array}}{\Gamma; \Delta \vdash \lambda x : \tau_1 . t \hookrightarrow \Lambda \overline{e_i :: \kappa_i} . \lambda x : \widehat{\tau}_1 \;\&\; \xi_1 . t' : \forall \overline{e_i :: \kappa_i} . \widehat{\tau}_1 \langle \xi_1 \rangle \to \widehat{\tau}_2 \langle \xi_2 \rangle \;\&\; \emptyset} \text{ [L-Abs]}$$

$$\frac{\begin{array}{c}\Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}_1[\overline{\zeta_i}/\overline{e_i}] \quad \Delta \vdash \xi_2 \leqslant \xi_1[\overline{\zeta_i}/\overline{e_i}] \quad \overline{\Delta \vdash \zeta_i :: \kappa_i} \\ \Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \forall \overline{e_i :: \kappa_i} . \widehat{\tau}_1 \langle \xi_1 \rangle \to \widehat{\tau} \langle \xi \rangle \;\&\; \xi' \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \xi_2\end{array}}{\Gamma; \Delta \vdash t_1 \; t_2 \hookrightarrow t_1' \; \langle \overline{\zeta_i} \rangle \; t_2' : \widehat{\tau}[\overline{\zeta_i}/\overline{e_i}] \;\&\; \xi[\overline{\zeta_i}/\overline{e_i}] \cup \xi'} \text{ [L-App]}$$

$$\frac{\begin{array}{c}\Delta \vdash \widehat{\tau} \downarrow \tau \quad \Delta \vdash \xi :: \textsc{exn} \quad \Delta \vdash \widehat{\tau}' \leqslant \widehat{\tau} \quad \Delta \vdash \xi' \leqslant \xi \\ \Gamma, x : \widehat{\tau} \;\&\; \xi; \Delta \vdash t \hookrightarrow t' : \widehat{\tau}' \;\&\; \xi'\end{array}}{\Gamma; \Delta \vdash \mathbf{fix} \; x : \tau . t \hookrightarrow \mathbf{fix} \; x : \widehat{\tau} \;\&\; \xi . t' : \widehat{\tau} \;\&\; \xi} \text{ [L-Fix]}$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\mathbf{int}} \;\&\; \xi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\mathbf{int}} \;\&\; \xi_2}{\Gamma; \Delta \vdash t_1 \oplus t_2 \hookrightarrow t_1' \oplus t_2' : \widehat{\mathbf{bool}} \;\&\; \xi_1 \cup \xi_2} \text{ [L-Op]}$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\tau}_1 \;\&\; \xi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \xi_2}{\Gamma; \Delta \vdash t_1 \; \mathbf{seq} \; t_2 \hookrightarrow t_1' \; \mathbf{seq} \; t_2' : \widehat{\tau}_2 \;\&\; \xi_1 \cup \xi_2} \text{ [L-Seq]}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\mathbf{bool}} \;\&\; \xi_1 \\ \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \xi_2 \quad \Gamma; \Delta \vdash t_3 \hookrightarrow t_3' : \widehat{\tau}_3 \;\&\; \xi_3\end{array}}{\Gamma; \Delta \vdash \mathbf{if} \; t_1 \; \mathbf{then} \; t_2 \; \mathbf{else} \; t_3 \hookrightarrow \mathbf{if} \; t_1' \; \mathbf{then} \; t_2' \; \mathbf{else} \; t_3' : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \;\&\; \xi_1 \cup \xi_2 \cup \xi_3} \text{ [L-If]}$$

$$\frac{}{\Gamma; \Delta \vdash []_\tau \hookrightarrow []_\tau : [\bot_\tau \langle \emptyset \rangle] \;\&\; \emptyset} \text{ [L-Nil]}$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\tau}_1 \;\&\; \xi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : [\widehat{\tau}_1' \langle \xi_1' \rangle] \;\&\; \xi_2}{\Gamma; \Delta \vdash t_1 :: t_2 \hookrightarrow t_1' :: t_2' : [\widehat{\tau}_1 \sqcup \widehat{\tau}_1' \langle \xi_1 \cup \xi_1' \rangle] \;\&\; \xi_2} \text{ [L-Cons]}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : [\widehat{\tau}_1 \langle \xi_1 \rangle] \;\&\; \xi_1' \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \xi_2 \\ \Gamma, x_1 : \widehat{\tau}_1 \;\&\; \xi_1, x_2 : [\widehat{\tau}_1 \langle \xi_1 \rangle] \;\&\; \xi_1'; \Delta \vdash t_3 \hookrightarrow t_3' : \widehat{\tau}_3 \;\&\; \xi_3\end{array}}{\begin{array}{l}\Gamma; \Delta \vdash \mathbf{case} \; t_1 \; \mathbf{of} \; \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \hookrightarrow \\ \qquad \mathbf{case} \; t_1' \; \mathbf{of} \; \{[] \mapsto t_2'; x_1 :: x_2 \mapsto t_3'\} : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \;\&\; \xi_1' \cup \xi_2 \cup \xi_3\end{array}} \text{ [L-Case]}$$

Fig. 3. Syntax-directed type elaboration system ($\Gamma; \Delta \vdash t \hookrightarrow t' : \widehat{\tau} \;\&\; \xi$)

4.5 Exception typing and elaboration

In Figure 2 we give a declarative system for deriving exception typing judgments $\Gamma; \Delta \vdash t : \widehat{\tau} \,\&\, \xi$.

These judgments work on an explicitly typed language and for this purpose we extend the terms of the source language with two new term-level constructs: effect abstraction and effect application.

Terms

$$
\begin{array}{rcll}
t \in \mathbf{ExnTm} & ::= & \ldots & \\
& | & \lambda x : \widehat{\tau} \,\&\, \xi.t & \text{(term abstraction)} \\
& | & \mathbf{fix}\ x : \widehat{\tau} \,\&\, \xi.t & \text{(general recursion)} \\
& | & \ldots & \\
& | & \Lambda e :: \kappa.t & \text{(effect abstraction)} \\
& | & t\ \langle \xi \rangle & \text{(effect application)}
\end{array}
$$

As the source language is not explicitly typed, we also give a type elaboration system that given an implicitly typed term in the source language produces an explicitly typed term (Figure 3).

The auxiliary judgment $\Delta \vdash \widehat{\tau} \downarrow \tau$ holds for any exception type $\widehat{\tau}$ that erases to the underlying type τ. The type $\widehat{\tau}_1 \sqcup \widehat{\tau}_2$ is an exception type such that $\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$ and $\Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}_1 \sqcup \widehat{\tau}_2$.

4.6 Presentation of exception types

For most-general conservative exception types the location of the quantifiers is uniquely determined, we can therefore omit them from the type without introducing ambiguity. For example, the exception type of the *map* function from the introduction may be presented as:

$$(\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\ e_1\rangle) \to [\alpha\langle e_4\rangle]\langle e_5\rangle \to [\beta\langle e_2\ e_4 \cup e_3\rangle]\langle e_5\rangle$$

5 Type inference

A type inference algorithm is given in Figure 4.

5.1 Polymorphic abstraction

The cases for abstraction and application are handled similarly to the corresponding cases in [16].

In the case of abstractions, we first complete the type of the bound variable to a most general exception type using the procedure $\mathcal{C} : \mathbf{KiEnv} \times \mathbf{Ty} \to \mathbf{ExnTy} \times \mathbf{Exn} \times \mathbf{KiEnv}$. This procedure is a functional interpretation of the

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTm} \times \mathbf{ExnTy} \times \mathbf{Exn} \\
\mathcal{R}(\Gamma; \Delta; x) \qquad = x : \Gamma(x) \\
\mathcal{R}(\Gamma; \Delta; c_\tau) \qquad = c_\tau : \bot_\tau \ \&\ \emptyset \\
\mathcal{R}(\Gamma; \Delta; \lightning^\ell_\tau) \qquad = \lightning^\ell_\tau : \bot_\tau \ \&\ \{\ell\} \\
\mathcal{R}(\Gamma; \Delta; \lambda x : \tau.t) = \\
\quad \textbf{let } \widehat{\tau}_1 \ \&\ e \triangleright \overline{e_i :: \kappa_i} = \mathcal{C}(\emptyset; \tau) \\
\qquad\quad t' : \widehat{\tau}_2 \ \&\ \xi_2 \qquad = \mathcal{R}(\Gamma, x : \widehat{\tau}_1 \ \&\ e; \Delta, \overline{e_i :: \kappa_i}; t) \\
\quad \textbf{in } \ \Lambda \overline{e_i :: \kappa_i}.\lambda x : \widehat{\tau}_1 \ \&\ e.t' : \forall \overline{e_i :: \kappa_i}.\widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle \xi_2\rangle \ \&\ \emptyset \\
\mathcal{R}(\Gamma; \Delta; t_1\ t_2) \quad = \\
\quad \textbf{let } t_1' : \widehat{\tau}_1 \ \&\ \xi_1 \qquad\qquad = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad t_2' : \widehat{\tau}_2 \ \&\ \xi_2 \qquad\qquad = \mathcal{R}(\Gamma; \Delta; t_2) \\
\qquad\quad \widehat{\tau}_2'\langle e_2'\rangle \to \widehat{\tau}'\langle \xi'\rangle \triangleright \overline{e_i :: \kappa_i} = \mathcal{I}(\widehat{\tau}_1) \\
\qquad\quad \theta \qquad\qquad\qquad\quad = [e_2' \mapsto \xi_2] \circ \mathcal{M}(\emptyset; \widehat{\tau}_2'; \widehat{\tau}_2) \\
\quad \textbf{in } \ t_1'\ \langle \overline{\theta e_i}\rangle\ t_2' : [\![\theta\widehat{\tau}']\!]_\Delta \ \&\ [\![\theta\xi' \cup \xi_1]\!]_\Delta \\
\mathcal{R}(\Gamma; \Delta; \textbf{fix } x : \tau.t) = \\
\quad \textbf{do } i; \widehat{\tau}_0 \ \&\ \xi_0 \leftarrow 0; \bot_\tau \ \&\ \emptyset \\
\qquad \textbf{repeat } t'_{i+1} : \widehat{\tau}_{i+1} \ \&\ \xi_{i+1} \leftarrow \mathcal{R}(\Gamma, x : \widehat{\tau}_i \ \&\ \xi_i; \Delta; t) \\
\qquad\qquad\quad i \qquad\qquad\qquad\qquad\quad \leftarrow i + 1 \\
\qquad \textbf{until } \widehat{\tau}_i \ \&\ \xi_i \equiv \widehat{\tau}_{i-1} \ \&\ \xi_{i-1} \\
\qquad \textbf{return fix } x : \widehat{\tau}_i \ \&\ \xi_i.t_i' : \widehat{\tau}_i \ \&\ \xi_i \\
\mathcal{R}(\Gamma; \Delta; t_1 \ \textbf{seq}\ t_2) = \\
\quad \textbf{let } t_1' : \widehat{\tau}_1 \ \&\ \xi_1 = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad t_2' : \widehat{\tau}_2 \ \&\ \xi_2 = \mathcal{R}(\Gamma; \Delta; t_2) \\
\quad \textbf{in } \ t_1' \ \textbf{seq}\ t_2' : \widehat{\tau}_2 \ \&\ [\![\xi_1 \cup \xi_2]\!]_\Delta \\
\mathcal{R}(\Gamma; \Delta; t_1 \oplus t_2) = \\
\quad \textbf{let } t_1' : \widehat{\textbf{int}} \ \&\ \xi_1 = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad t_2' : \widehat{\textbf{int}} \ \&\ \xi_2 = \mathcal{R}(\Gamma; \Delta; t_2) \\
\quad \textbf{in } \ t_1' \oplus t_2' : \widehat{\textbf{bool}} \ \&\ [\![\xi_1 \cup \xi_2]\!]_\Delta \\
\mathcal{R}(\Gamma; \Delta; \textbf{if } t_1 \ \textbf{then } t_2 \ \textbf{else } t_3) = \\
\quad \textbf{let } t_1' : \widehat{\textbf{bool}} \ \&\ \xi_1 = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad t_2' : \widehat{\tau}_2 \ \&\ \xi_2 \qquad = \mathcal{R}(\Gamma; \Delta; t_2) \\
\qquad\quad t_3' : \widehat{\tau}_3 \ \&\ \xi_3 \qquad = \mathcal{R}(\Gamma; \Delta; t_3) \\
\quad \textbf{in } \ \textbf{if } t_1' \ \textbf{then } t_2' \ \textbf{else } t_3' : [\![\widehat{\tau}_2 \sqcup \widehat{\tau}_3]\!]_\Delta \ \&\ [\![\xi_1 \cup \xi_2 \cup \xi_3]\!]_\Delta \\
\mathcal{R}(\Gamma; \Delta; []_\tau) \qquad = []_\tau : [\bot_\tau\langle\emptyset\rangle] \ \&\ \emptyset \\
\mathcal{R}(\Gamma; \Delta; t_1 :: t_2) = \\
\quad \textbf{let } t_1' : \widehat{\tau}_1 \ \&\ \xi_1 \qquad = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad t_2' : [\widehat{\tau}_2\langle\xi_2'\rangle] \ \&\ \xi_2 = \mathcal{R}(\Gamma; \Delta; t_2) \\
\quad \textbf{in } \ t_1' :: t_2' : [\![(\widehat{\tau}_1 \sqcup \widehat{\tau}_2)\langle\xi_1 \cup \xi_2'\rangle]\!]_\Delta \ \&\ \xi_2 \\
\mathcal{R}(\Gamma; \Delta; \textbf{case } t_1 \ \textbf{of } \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\}) = \\
\quad \textbf{let } t_1' : [\widehat{\tau}_1\langle\xi_1'\rangle] \ \&\ \xi_1 = \mathcal{R}(\Gamma; \Delta; t_1) \\
\qquad\quad \Gamma' \qquad\qquad\quad = \Gamma, x_1 : \widehat{\tau}_1 \ \&\ \xi_1', x_2 : [\widehat{\tau}_1\langle\xi_1'\rangle] \ \&\ \xi_1 \\
\qquad\quad t_2' : \widehat{\tau}_2 \ \&\ \xi_2 \qquad = \mathcal{R}(\Gamma; \Delta; t_2) \\
\qquad\quad t_3' : \widehat{\tau}_3 \ \&\ \xi_3 \qquad = \mathcal{R}(\Gamma'; \Delta; t_3) \\
\quad \textbf{in } \ \textbf{case } t_1' \ \textbf{of } \{[] \mapsto t_2'; x_1 :: x_2 \mapsto t_3'\} : \\
\qquad\qquad\qquad [\![\widehat{\tau}_2 \sqcup \widehat{\tau}_3]\!]_\Delta \ \&\ [\![\xi_1 \cup \xi_2 \cup \xi_3]\!]_\Delta
\end{array}
$$

Fig. 4. Type inference algorithm ($\mathcal{R}$)

type completion relation $\Delta \vdash \tau : \widehat{\tau} \;\&\; \xi \triangleright \Delta'$, where the first two arguments Δ and τ are taken to be the domain and the last three arguments $\widehat{\tau}$, ξ and Δ' are taken to be the range. Next, we infer the exception type of the body of the abstraction under the assumption that the bound variable has the just completed exception type-and-effect $\widehat{\tau}_1 \;\&\; e_1$. Finally we quantify over all free variables $\overline{e_i :: \kappa_i}$ introduced by completion.

In the case of applications, we instantiate ($\mathcal{I}$) all quantified variables of the exception type of t_1 with fresh exception variables. Next we use the auxiliary procedure $\mathcal{M}$ to find a matching substitution between the exception types of the formal and the actual parameters.

$$
\begin{aligned}
&\mathcal{M} : \mathbf{KiEnv} \times \mathbf{ExnTy} \times \mathbf{ExnTy} \to \mathbf{Subst} \\
&\mathcal{M}(\Delta; \widehat{\mathbf{bool}}; \quad \widehat{\mathbf{bool}}) \quad = \emptyset \\
&\mathcal{M}(\Delta; \widehat{\mathbf{int}}; \quad \widehat{\mathbf{int}}) \quad = \emptyset \\
&\mathcal{M}(\Delta;\, [\widehat{\tau}'\langle e'\ \overline{e_i}\rangle];\, [\widehat{\tau}\langle\xi\rangle]) \\
&\qquad = \left[e' \mapsto \lambda\overline{e_i :: \Delta_{e_i}}.\xi\right] \circ \mathcal{M}(\Delta; \widehat{\tau}'; \widehat{\tau}) \\
&\mathcal{M}(\Delta; \widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2'\langle e'\ \overline{e_i}\rangle; \widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle\xi\rangle) \\
&\qquad = \left[e' \mapsto \lambda\overline{e_i :: \Delta_{e_i}}.\xi\right] \circ \mathcal{M}(\Delta; \widehat{\tau}_2'; \widehat{\tau}_2) \\
&\mathcal{M}(\Delta; \forall e :: \kappa.\widehat{\tau}';\ \forall e :: \kappa.\widehat{\tau}) = \mathcal{M}(\Delta, e :: \kappa; \widehat{\tau}'; \widehat{\tau})
\end{aligned}
$$

Fig. 5. Exception type matching ($\mathcal{M}$)

The interesting cases of exception type matching are the cases for list and function types, where we perform pattern unification on the exception annotations. The produced substitution θ covers all variables $\overline{e_i :: \kappa_i}$ freshly introduced by the instantiation procedure $\mathcal{I}$. Finally, we apply the substitution θ to the exception type $\widehat{\tau}'$ and effect ξ' of the result of t_1.

5.2 Polymorphic recursion

The **fix**-construct abstracts over a variable that is of an exception polymorphic type. The algorithm handles this case with a Kleene–Mycroft iteration—which we conjecture to always converge.[2]

Example 8 (Dussart–Henglein–Mossin). Consider the term

$$
\begin{aligned}
&f \;:\; \mathbf{bool} \to \mathbf{bool} \to \mathbf{bool} \\
&f = \mathbf{fix}\ f' : \mathbf{bool} \to \mathbf{bool} \to \mathbf{bool}. \\
&\qquad \lambda x : \mathbf{bool}.\lambda y : \mathbf{bool}.\ \mathbf{if}\ x\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ f'\ y\ x
\end{aligned}
$$

[2] [16] note that λ-bound polymorpism gives us **fix**-bound polymorphism "for free." We believe this statement to be overly optimistic. While the highly polymorphic nature of these types do effectively force us to also handle polymorphic recursion, the inference step is arguably as complicated as the case for polymorphic abstraction.

Algorithm $\mathcal{R}$ infers the exception type and elaborated term

$$f \;:\; \forall e_1.\mathbf{b\widehat{oo}l}\langle e_1\rangle \xrightarrow{\emptyset} \forall e_2.\mathbf{b\widehat{oo}l}\langle e_2\rangle \xrightarrow{\emptyset} \mathbf{b\widehat{oo}l}\langle e_1 \cup e_2\rangle$$
$$\begin{aligned} f = \mathbf{fix}\ f' : \forall e_1.\mathbf{b\widehat{oo}l}\langle e_1\rangle \xrightarrow{\emptyset} \forall e_2.\mathbf{b\widehat{oo}l}\langle e_2\rangle \xrightarrow{\emptyset} \mathbf{b\widehat{oo}l}\langle e_1 \cup e_2\rangle.\\ \Lambda e_1 :: \textsc{exn}.\lambda x : \mathbf{b\widehat{oo}l}\ \&\ e_1.\Lambda e_2 :: \textsc{exn}.\lambda y : \mathbf{b\widehat{oo}l}\ \&\ e_2.\\ \mathbf{if}\ x\ \mathbf{then\ true\ else}\ f'\ \langle e_2\rangle\ y\ \langle e_1\rangle\ x \end{aligned}$$

Let us convince ourselves that the elaborated term is type-correct.

$$\begin{aligned} x &: \mathbf{b\widehat{oo}l}\ \&\ e_1\\ y &: \mathbf{b\widehat{oo}l}\ \&\ e_2\\ \mathbf{true} &: \mathbf{b\widehat{oo}l}\ \&\ \emptyset\\ f'\langle e_2\rangle\ y\ \langle e_1\rangle\ x &: \mathbf{b\widehat{oo}l}\ \&\ e_2 \cup e_1 \end{aligned}$$

Therefore,

$$\mathbf{if}\ x\ \mathbf{then\ true\ else}\ f'\langle e_2\rangle\ y\ \langle e_1\rangle\ x : \mathbf{b\widehat{oo}l} \sqcup \mathbf{b\widehat{oo}l}\ \&\ e_1 \cup \emptyset \cup e_2 \cup e_1$$

By commutativity and idempotence of the union operator and the empty set being the unit, this reduces to

$$\mathbf{if}\ x\ \mathbf{then\ true\ else}\ f'\langle e_2\rangle\ y\ \langle e_1\rangle\ x : \mathbf{b\widehat{oo}l}\ \&\ e_1 \cup e_2$$

Type checking is easier than type inference, however. To infer the type of the recursive definition f we have to "guess" a type for it. How do we guess this type? We first try the least exception type $\bot_{\mathbf{bool}\to\mathbf{bool}\to\mathbf{bool}}$:

$$\forall e_1.\mathbf{b\widehat{oo}l}\langle e_1\rangle \xrightarrow{\emptyset} \forall e_2.\mathbf{b\widehat{oo}l}\langle e_2\rangle \xrightarrow{\emptyset} \mathbf{b\widehat{oo}l}\langle\emptyset\rangle$$

If we continue inferring the type with this guess, then we end up with a larger type than the guess:

$$\forall e_1.\mathbf{b\widehat{oo}l}\langle e_1\rangle \xrightarrow{\emptyset} \forall e_2.\mathbf{b\widehat{oo}l}\langle e_2\rangle \xrightarrow{\emptyset} \mathbf{b\widehat{oo}l}\langle e_1\rangle$$

We try inferring the type again, but now start with this type as our guess instead of the least type. We end up with an even larger type:

$$\forall e_1.\mathbf{b\widehat{oo}l}\langle e_1\rangle \xrightarrow{\emptyset} \forall e_2.\mathbf{b\widehat{oo}l}\langle e_2\rangle \xrightarrow{\emptyset} \mathbf{b\widehat{oo}l}\langle e_1 \cup e_2\rangle$$

Finally, if we take this type as our guess, we obtain the same type and conclude we have reached a fixed point.

5.3 Least upper bounds

The remaining cases of the algorithm are all relatively straightforward. Several of the cases (**if-then-else**, **case-of** and the list-consing constructor) require the least upper bound of two exception types to be computed. The fact that exception types and annotations occurring in argument positions of function types are always simple makes this easy, as they must be equal up to α-renaming [16]. This allows us to treat those arguments invariantly instead of contravariantly, obviating the need to also compute greatest lower bounds of exception types and annotations.

$$
\begin{array}{lll}
\sqcup : \mathbf{ExnTy} \times \mathbf{ExnTy} \to \mathbf{ExnTy} & & \\
\widehat{\mathbf{bool}} & \sqcup\, \widehat{\mathbf{bool}} & = \widehat{\mathbf{bool}} \\
\widehat{\mathbf{int}} & \sqcup\, \widehat{\mathbf{int}} & = \widehat{\mathbf{int}} \\
[\widehat{\tau}\langle\xi\rangle] & \sqcup\, [\widehat{\tau}'\langle\xi'\rangle] & = [(\widehat{\tau} \sqcup \widehat{\tau}')\langle\xi \cup \xi'\rangle] \\
\widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle\xi\rangle & \sqcup\, \widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2'\langle\xi'\rangle & = \widehat{\tau}_1\langle e\rangle \to (\widehat{\tau}_2 \sqcup \widehat{\tau}_2')\langle\xi \cup \xi'\rangle \\
\forall e :: \kappa.\widehat{\tau} & \sqcup\, \forall e :: \kappa.\widehat{\tau}' & = \forall e :: \kappa.(\widehat{\tau} \sqcup \widehat{\tau}')
\end{array}
$$

Fig. 6. Exception types: least upper bounds ($\sqcup$)

5.4 Complexity

There are three aspects that affect the run-time complexity of the algorithm: the complexity of the underlying type system, reduction of the effects, and the fixpoint-iteration in the inference step of the **fix**-construct. We have a simply typed underlying type system, but if we would extend this to full Hindley–Milner, then it is possible for types to become exponentially larger than terms [27, 21]. The effects are $\lambda^{\cup}$-terms, which contains the simply typed λ-calculus as a special case. Reduction of terms in the simply typed λ-calculus is non-elementary recursive [42]. It is also easy to find an artificial family of terms that requires at least a linear number of iterations to converge to a fixpoint. For these reasons we do not believe the algorithm to have an attractive theoretical bound on time-complexity.

Anecdotal evidence suggests that the practical time-complexity is acceptable, however. Hindley–Milner has almost linear complexity in non-pathological cases. Types do not grow larger than the terms. The same seems to hold for the effects. Reduction of effects takes a small number of steps, as does the convergence of the fixpoint-iteration. In cases where the exception annotation does become too large, a widening rule could be applied.

6 Related work

6.1 Higher-ranked polymorphism in type-and-effect systems

Effect polymorphism For plain type systems, Hindley–Milner's **let**-bound polymorphism generally provides a good compromise between expressiveness of the type system and complexity of the inference algorithm [15, 29, 5]. Type systems were extended with effects—including **let**-bound effect-polymorphism—by [26, 17]; and [43, 44]. In type-and-effect systems it has long been recognized that **fix**-bound polymorphism (polymorphic recursion) *in the effects* is often beneficial or even necessary for achieving precise analysis results. For example, in type-and-effect systems for regions [47], dimensions [18, 37, 38], binding times [8], and exceptions [12, 24].

Inferring principal types in a type system with polymorphic recursion is equivalent to solving the undecidable semi-unification problem [31, 23, 22, 14].

When restricted to polymorphic recursion in the effects, the problem often becomes decidable again. In [47] this is a conjecture based on empirical observation. [38] gives a semi-unification procedure based on the general semi-unification semi-algorithm by [3] and proves it terminates in the special case of semi-unification in Abelian groups. [8] use a constraint-based algorithm. They show that all variables that do not occur free in the context or type can be eliminated from the constraint set by a constraint reduction step during each Kleene–Mycroft iteration. As at most n^2 subeffecting constraints can be formed over n free variables, the whole procedure must terminate. By not restarting the Kleene–Mycroft iteration from bottom, their algorithm runs in polynomial time—even in the presence of nested fixpoints.

The extension to polymorphic effect-abstraction (λ-bound, higher-ranked effect polymorphism) remained less well-studied, possibly because it is of limited use without the simultaneous introduction of effect operators—in contrast to the situation of higher-ranked polymorphism in plain type systems.

Effect operators [19] presents a type system that ensures the dimensional consistency of an ML-like language extended with units of measure (ML_δ). This language has predicative prenex dimension polymorphism. Kennedy gives an Algorithm $\mathcal{W}$-like type inference procedure that uses equational unification to deal with the Abelian group (AG) structure of dimension expressions. Also described are two explicitly typed variants of the language: a System F-like language with higher-ranked dimension polymorphism (Λ_δ), and a System F_ω-like language that extends Λ_δ with dimension operators ($\Lambda_{\delta\omega}$). [19] notes that this language can type strictly more programs than the language without dimension operators:

$$
\begin{array}{l}
\mathit{twice} \;:\; \forall F :: \mathrm{DIM} \Rightarrow \mathrm{DIM}. \\
\qquad\qquad (\forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle F\ d\rangle) \to \\
\qquad\qquad\quad (\forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle F\ (F\ d)\rangle) \\
\mathit{twice} \;=\; \Lambda F :: \mathrm{DIM} \Rightarrow \mathrm{DIM}. \\
\qquad\qquad \lambda f : (\forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle F\ d\rangle). \\
\qquad\qquad\quad \Lambda d :: \mathrm{DIM}.\lambda x : \mathbf{real}\langle d\rangle.f\ \langle F\ d\rangle\ (f\ \langle d\rangle\ x) \\
\mathit{square} \;:\; \forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle d^2\rangle \\
\mathit{square} \;=\; \Lambda d :: \mathrm{DIM}.\lambda x : \mathbf{real}\langle d\rangle.x^2 \\
\mathit{fourth} \;:\; \forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle d^4\rangle \\
\mathit{fourth} \;=\; \mathit{twice}\ \langle \Lambda d :: \mathrm{DIM}.d^2\rangle\ \mathit{square}
\end{array}
$$

Without dimension operators the type of the higher-order function *twice* would not allow the application of the function *square* at the two distinct types $\forall d :: \mathrm{DIM}.\mathbf{real}\langle d\rangle \to \mathbf{real}\langle d^2\rangle$ and $\forall d :: \mathrm{DIM}.\mathbf{real}\langle d^2\rangle \to \mathbf{real}\langle d^4\rangle$ when invoked from the function *fourth*.

The language $\Lambda_{\delta\omega}$ bears a striking resemblance to the language in Section 4: the empty and singleton exception sets constants, and the exception set union operator have been replaced with a unit dimension, and dimension product and inverse operators—as dimensions have an AG structure, whereas exception sets have an ACI1 structure; in the dimension type system the annotation is placed

only on the real number base type instead of on the compound types, and there is no effect. No type inference algorithm is given for this language, however.

[10] presents a type system for flow analysis that uses constrained type schemes in the style of [2], and has λ-bound polymorphism (but no type operators) in the style of System F. To make the inference algorithm terminate for recursive programs the size of the name supply needs to be bounded, leading to imprecision. [40] present a similar framework, but one that can be instantiated with variants of either k-CFA [39] or CPA [1] to ensure termination.

[16] design a System F_ω-like type system for flow analysis for a strict language that has both polymorphic abstraction and effect operators. Our type inference algorithm builds on their techniques. A key difference is that they work with a constraint-based type system and a constraint solver, while we replace these with reduction of terms in an algebraic λ-calculus. This difference expresses itself particularly in how the case of (polymorphic) recursion is handled. We believe our approach will scale more easily to analyses that are either not conservative extensions of the underlying type system, or require more expressive effects (see Section 7).

6.2 $\boldsymbol{\lambda^{\cup}}$-calculus

[45], [33], and [46] prove that if a simply typed λ-calculus is extended with a many-sorted algebraic rewrite system R (by introducing the symbols of the algebraic theory as higher-order constants in the λ-calculus), then the combined rewrite system $\beta\eta R$ is confluent and strongly normalizing if R is confluent and strongly normalizing.

[36] introduced an untyped λ-calculus with applicative lists. A model is given by [7]. This calculus satisfies the equations

$$\langle t_1, ..., t_n \rangle\ t' = \langle t_1\ t', ..., t_n\ t' \rangle \qquad (\gamma_1)$$

$$\lambda x.\langle t_1, ...t_n \rangle = \langle \lambda x.t_1, ..., \lambda x.t_n \rangle \qquad (\gamma_2)$$

similar to our typed $\lambda^{\cup}$-calculus.

6.3 Exception analyses

Several exception analyses have been described in the literature; these primarily target the detection of uncaught exceptions in ML. The exception analysis by [49] is based on abstract interpretation. [13] and [9] describe type-based exception analyses. [25] presents a row-based type system for exception analysis that contains a data-flow analysis component targeted towards tracking value-carrying exceptions.

[12] developed the first exception analysis for a non-strict language; a type-based analysis using Boolean constraints. [24] present a constraint-based type system for exception analysis of a non-strict language, where the exception-flow could depend on the data-flow using conditional constraints. This increases the accuracy in the presence of exceptions raised by pattern-matching failures.

7 Further research

Can we infer types for Kennedy's higher-ranked $\Lambda_{\delta\omega}$? One problem that immediately presents itself is that this type system is not a conservative extension of the underlying type system: programs can be rejected because they, while being type correct in the underlying type system, may still be dimensionally inconsistent. Unlike the system in this paper, the annotations on function arguments will no longer be of the simple form (patterns) required for the straightforward matching step in the type inference algorithm. Instead, we suspect we have to solve a higher-order equational (pre)unification problem, which is only semi-decidable. [41], [32] and [35] do give us semi-algorithms for solving such problems.

Can we further improve the precision of exception types? [24] argue that an accurate exception typing system for non-strict languages should also take the data flow of the program into account, as many exceptions that can be raised in non-strict languages are caused by incomplete case-analyses during pattern-matching. The canonical example is the *risers* function—which splits a list into monotonically increasing subsegments; for example, *risers* $[1,3,5,1,2]$ evaluates to $[[1,3,5],[1,2]]$—by [30]:

$$
\begin{array}{l}
\mathit{risers}\ :[\mathbf{int}]\to[[\mathbf{int}]]\\
\mathit{risers}\ [\,] \qquad\qquad\quad = [\,]\\
\mathit{risers}\ [x] \qquad\qquad = [[x]]\\
\mathit{risers}\ (x_1 :: x_2 :: xs) =\\
\quad \mathbf{if}\ x_1 \leqslant x_2\ \mathbf{then}\ (x_1 :: y) :: ys\ \mathbf{else}\ [x_1] :: (y :: ys)\\
\qquad \mathbf{where}\ (y :: ys) = \mathit{risers}\ (x_2 :: xs)
\end{array}
$$

The inference algorithm in Figure 4 assigns *risers* the type

$$
\begin{array}{l}
\forall e_1 :: \textsc{exn}.\forall e_2 :: \textsc{exn}.\\
\quad [\widehat{\mathbf{int}}\langle e_2\rangle]\langle e_1\rangle \to [[\widehat{\mathbf{int}}\langle e_2\rangle]\langle\emptyset\rangle]\langle e_1 \cup e_2 \cup \{\mathbf{E}\}\rangle\ \&\ \emptyset
\end{array}
$$

where **E** is the label of the exception raised when the pattern-match in the **where**-clause fails.[3] However, the pattern-match happens on the result of the recursive call *risers* $(x_2 :: xs)$. When *risers* is given a non-empty list (such as $x_2 :: xs$) as an argument, it always returns a non-empty list as its result. The pattern-match can thus never fail, and the exception labelled **E** can thus never be raised.

[24] demonstrate how this exception can be elided by having the exception flow depend on the data flow. The $\lambda^{\cup}$-calculus terms that form the effect annotations cannot express this dependence, however. [24] use a slightly ad hoc form of conditional constraints to model this dependence. We believe that extending a λ-calculus with an equational theory of Boolean rings may form the basis of a more principled approach. Boolean rings have already been successfully used to design type systems for strictness analysis [48], records [20] and exception tracking [4].

[3] This exception is left implicit in the above program, but becomes explicit when the code is desugared into our core language.

Metatheory We have not yet worked out the metatheory of the type system presented in this paper. Of particular interest are the (syntactic) soundness, completeness and totality of the inference step for recursive definitions. We expect that soundness and completeness can be shown by a similar argument as in [31] and [8].

We conjectured the totality of our inference algorithm. We have a good reason to do so: we only expect the fixpoint iteration to diverge if no fixpoint exists—that is to say, the program is type incorrect. Assuming the program is well-typed in the underlying type system, there are no type incorrect programs in our exception typing system, however.

To show the fixpoint iteration is guaranteed to terminate in their binding-time analysis, [8] note that only a finite number of type constraints and therefore constrained type schemes can be formed over a finite number of variables (after constraints have been simplified). As it is still possible to form an infinite number of $\lambda^{\cup}$-normal forms over a finite number of variables, such an argument is not going to work directly.

8 Conclusion

We show that it is feasible to extend non-strict higher-order languages with exception-annotated types, as is already done in some strict first-order languages. We argue that higher-ranked exception polymorphic types with exception set operators *à la* System F_ω are not only more accurate, but are also more readable when presented to the programmer *vis-à-vis* constrained type schemes: the exception terms in the annotations more closely mirror what is happening at the term level than constraint sets do.

Acknowledgements

We would like to thank Jurriaan Hage and the members of the Software Technology reading club at Utrecht University for their comments on earlier drafts of this paper. Vincent van Oostrom and Femke van Raamsdonk provided some helpful pointers to related literature.

[1] Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. pp. 2–26. ECOOP '95 (1995)
[2] Aiken, A., Wimmers, E.L.: Type inclusion constraints and type inference. pp. 31–41. FPCA '93 (1993)
[3] Baaz, M.: Note on the existence of most-general semi-unifiers. In: Arithmetic, Proof Theory and Computational Complexity (1993)
[4] Benton, N., Buchlovsky, P.: Semantics of an effect analysis for exceptions. pp. 15–26. TLDI '07 (2007)
[5] Damas, L., Milner, R.: Principal type-schemes for functional programs. pp. 207–212. POPL '82 (1982)
[6] Dowek, G.: Higher-order unification and matching. In: Handbook of Automated Reasoning, pp. 1009–1062 (2001)
[7] Durfee, G.: A model for a list-oriented extension of the lambda calculus. MSc thesis, Carnegie Mellon University (1997)
[8] Dussart, D., Henglein, F., Mossin, C.: Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. pp. 118–135. SAS '95 (1995)
[9] Fähndrich, M., Foster, J., Cu, J., Aiken, A.: Tracking down exceptions in Standard ML programs. Tech. Rep. UCB/CSD-98-996, University of California at Berkeley (1998)
[10] Faxén, K.F.: Polyvariance, polymorphism and flow analysis. In: Selected Papers from the 5th LOMAPS Workshop. pp. 260–278 (1997)
[11] Girard, J.Y.: Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis (1972)
[12] Glynn, K., Stuckey, P.J., Sulzmann, M., Søndergaard, H.: Exception analysis for non-strict languages. ICFP '02 (2002)
[13] Guzmán, J.C., Suárez, A.: An extended type system for exceptions. pp. 127–135. ML '94 (1994)
[14] Henglein, F.: Type inference with polymorphic recursion. ACM TOPLAS 15(2), 253–289 (Apr 1993)
[15] Hindley, J.R.: The principal type-scheme of an object in combinatory logic. Trans. Amer. Math. Soc. 146, 29–60 (1969)
[16] Holdermans, S., Hage, J.: Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. pp. 63–74. ICFP '10 (2010)
[17] Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. pp. 303–310. POPL '91 (1991)
[18] Kennedy, A.J.: Dimension types. pp. 348–362. ESOP '94 (1994)
[19] Kennedy, A.J.: Programming Languages and Dimensions. Ph.D. thesis, University of Cambridge (1996)
[20] Kennedy, A.J.: Type inference and equational theories. Tech. Rep. LIX-RR-96-09 (1996)
[21] Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: ML typability is DEXPTIME-complete. pp. 206–220. CAAP '90 (1990)
[22] Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM TOPLAS 15(2), 290–311
[23] Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: The undecidability of the semi-unification problem. pp. 468–476. STOC '90 (1990)
[24] Koot, R., Hage, J.: Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. pp. 127–138. PEPM '15 (2015)

[25] Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. ACM TOPLAS 22(2), 340–377 (Mar 2000)
[26] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. pp. 47–57. POPL '88 (1988)
[27] Mairson, H.G.: Deciding ML typability is complete for deterministic exponential time. pp. 382–401. POPL '90 (1990)
[28] Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. LNCS, vol. 475 (1991)
[29] Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
[30] Mitchell, N., Runciman, C.: Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. pp. 49–60. Haskell '08 (2008)
[31] Mycroft, A.: Polymorphic type schemes and recursive definitions. LNCS, vol. 167, pp. 217–228 (1984)
[32] Nipkow, T., Qian, Z.: Modular higher-order E-unification. LNCS, vol. 488, pp. 200–214 (1991)
[33] Okada, M.: Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. pp. 357–363. ISSAC '89 (1989)
[34] Peyton Jones, S., Reid, A., Henderson, F., Hoare, T., Marlow, S.: A semantics for imprecise exceptions. PLDI '99 (1999)
[35] Qian, Z., Wang, K.: Modular higher-order equational preunification. Journal of Symbolic Computation 22(4), 401–424 (1996)
[36] Révész, G.E.: A list-oriented extension of the lambda-calculus satisfying the Church–Rosser theorem. TCS 93(1), 75–89 (Feb 1992)
[37] Rittri, M.: Semi-unification of two terms in abelian groups. Information Processing Letters 52(2), 61–68 (Oct 1994)
[38] Rittri, M.: Dimension inference under polymorphic recursion. pp. 147–159. FPCA '95 (1995)
[39] Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie Mellon University (1991)
[40] Smith, S.F., Wang, T.: Polyvariant flow analysis with constrained types. LNCS, vol. 1782, pp. 382–396 (2000)
[41] Snyder, W.: Higher order E-unification. CADE '90 (1990)
[42] Statman, R.: The typed λ-calculus is not elementary recursive. Theoretical Computer Science 9(1), 73–81 (1979)
[43] Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2(3), 245–271 (1992)
[44] Talpin, J.P., Jouvelot, P.: The type and effect discipline. Information and Computation 111(2), 245–296 (Jun 1994)
[45] Tannen, V.: Combining algebra and higher-order types. LICS '88 (1988)
[46] Tannen, V., Gallier, J.: Polymorphic rewriting conserves algebraic strong normalization. Theoretical Computer Science 83(1), 3 – 28 (1991)
[47] Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. POPL '94 (1994)
[48] Wright, D.A.: A new technique for strictness analysis. LNCS, vol. 494, pp. 235–258 (1991)
[49] Yi, K.: Compile-time detection of uncaught exceptions in Standard ML programs. LNCS, vol. 864, pp. 238–254 (1994)

A Prototype

A prototype implementation of the inference algorithm is available from `https://www.staff.science.uu.nl/~0422819/hret/supplementary-material.tar.gz`.