Higher-ranked Exception Types [*]

Jane Doe
ACME University
jane.doe@acm.org

John Doe
ACME University
john.doe@acm.org

Abstract

We present a type-and-effect system that derives an exception-annotated type signature for any term in a simply typed non-strict functional language with general recursion and a list data type. This signature precisely (although not exactly) declares the set of exceptional values that may be present among the values of the term, using higher-ranked effect polymorphism and effect operators reminiscent of System F_ω.

By restricting the use of higher-ranked polymorphism and operators to the effects and not extending their use to the types we conjecture the inference problem to remain decidable. We give a type inference algorithm that builds on the techniques developed by Holdermans and Hage (2010).

The types in System F_ω form a simply typed λ-calculus. Similarly, the effects in our system form a simply typed λ-calculus embellished with the ACI1-structure of sets ($\lambda^\cup$). We briefly study this language in its own right.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Languages, Design, Theory

Keywords exceptions, higher-ranked polymorphism, polymorphic recursion, type-and-effect systems, type-based program analysis, type inference, type operators, unification theory

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 are extended with an *exceptional value* $\lightning$ (more commonly denoted $\perp$, 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 every expression an exception-annotated type as well.

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

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

For each type τ, we denote its exception-annotated type by $\tau\langle\xi\rangle$. For function types we occasionally write $\tau_1\langle\xi_1\rangle \xrightarrow{\xi} \tau_2\langle\xi_2\rangle$ instead of $(\tau_1\langle\xi_1\rangle \to \tau_2\langle\xi_2\rangle)\langle\xi\rangle$. If ξ is the empty exception set, then it may be omitted completely.

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

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

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 it into another exception set, for example by adding a number of new elements to it, or discarding it and returning the empty set. Furthermore, the function (closure) itself may be an exceptional value from the exception set e_3.

[*] This material may or may not be based upon work supported by some shady organization under a cryptic project title with a number no one cares about. We will categorically deny everything.

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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 elements in the lists may 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 constructors in the spine of this list may 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 exceptional value *const* $\lightning^{\mathbf{E}}$. These two functions can individually be given the exception types:

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

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 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.

1.1 Overview

In Section 2 we introduce the $\lambda^{\cup}$-calculus, a simply typed λ-calculus embellished with an associative, commutative, idempotent and unit (ACI1) structure. The $\lambda^{\cup}$-calculus forms the language of effects in the type-and-effect system. Section 3 describes the source language to which the analysis applies. In Section 4 we present the language of exception types and two type-and-effect systems for deriving exception types: a declarative type-and-effect system and a syntax-directed elaboration system that also produces an explicitly typed term. A type inference algorithm for this type-and-effect system is given in Section 5.

1.2 Contributions

- A *λ-calculus* extended with a union-operator that respect the associative, commutative, idempotent and unit structure of sets.
- A *type-and-effect system* with higher-ranked effect-polymorphic types and effect operators that precisely tracks exceptions.
- An *inference algorithm* for these higher-ranked exception types.

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

The $\lambda^{\cup}$-calculus is a simply typed λ-calculus extended with a set-union operator and singleton-set and empty-set constants at the term level. 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.

Types

$$\begin{aligned}
\tau \in \mathbf{Ty} &::= \star && \text{(base type)}\\
&\mid\ \tau_1 \to \tau_2 && \text{(function type)}
\end{aligned}$$

Terms

$$\begin{aligned}
t \in \mathbf{Tm} &::= x && \text{(variable)}\\
&\mid\ \lambda x : \tau.t && \text{(abstraction)}\\
&\mid\ t_1\ t_2 && \text{(application)}\\
&\mid\ \emptyset && \text{(empty)}\\
&\mid\ \{c\} && \text{(singleton)}\\
&\mid\ t_1 \cup t_2 && \text{(union)}
\end{aligned}$$

Environments

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

Figure 1. $\lambda^{\cup}$-calculus: syntax

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ [\text{T-VAR}] \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2}\ [\text{T-ABS}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ [\text{T-APP}]$$

$$\frac{}{\Gamma \vdash \emptyset : \star}\ [\text{T-EMPTY}] \qquad \frac{}{\Gamma \vdash \{c\} : \star}\ [\text{T-CON}]$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \cup t_2 : \tau}\ [\text{T-UNION}]$$

Figure 2. $\lambda^{\cup}$-calculus: type system

The empty-set and singleton-set constants are of base type and we can only take the set-union of two terms if they have the same type.

2.1 Semantics

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

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

Figure 3. $\lambda^{\cup}$-calculus: denotational semantics

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 $\Gamma \vdash t_1 \leqslant t_2$, 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 the terms t_1 and t_2 are *observationally equivalent*, denoted as $\Gamma \vdash t_1 \cong t_2$, if

1. $\Gamma \vdash t_1 \leqslant t_2$ and $\Gamma \vdash t_2 \leqslant t_1$, 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.

2.3.1 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 'forgotten' any empty set constants (unit elements), duplicate terms $f_i(t_{i1}, ..., t_{iq_i})$ (idempotent elements), and how the set union operator associates.

A *normal form* v of a term t—obtained by repeatedly applying the reduction rules from Figure 4—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).

$$\frac{}{(\lambda x . t_1)\ t_2 \longrightarrow t_1\,[t_2/x]}\ [\text{R-BETA}]$$

$$\frac{}{(t_1 \cup \cdots \cup t_n)\ t \longrightarrow t_1\ t \cup \cdots \cup t_n\ t}\ [\text{R-GAMMA}_1]$$

$$\frac{}{(\lambda x . t_1) \cup \cdots \cup (\lambda x . t_n) \longrightarrow \lambda x . t_1 \cup \cdots \cup t_n}\ [\text{R-GAMMA}_2]$$

Figure 4. $\lambda^{\cup}$-calculus: reduction

2.3.2 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 use it to order unions of terms.

First pick a 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 in the following manner:

1. Given two fully applied terms $k(v_1, ..., v_n)$ and $k'(v'_1, ..., v'_m)$ we define:
$$\begin{aligned}
k(v_1, ..., v_n) &\prec k'(v'_1, ..., v'_m) && \text{if } k \prec k' \\
k(v_1, ..., v_i, ..., v_n) &\prec k(v_1, ..., v_{i-1}, v'_i, ..., v'_n) && \text{if } v_i \prec v'_i
\end{aligned}$$
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 $\lfloor\!\lfloor t \rfloor\!\rfloor_\Gamma$ the $\beta\gamma$-normal η-long and canonically ordered derivation of the term t.

2.4 Pattern unification

Solving equations in λ-calculi is usually hard, but sometimes easy.

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 (Miller 1991; Dowek 2001).

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

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

can be uniquely solved by the unifier

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ [\textsc{U-Var}] \qquad \frac{}{\Gamma \vdash c_\tau : \tau}\ [\textsc{U-Con}] \qquad \frac{}{\Gamma \vdash \lightning^\ell_\tau : \tau}\ [\textsc{U-Crash}] \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t : \tau_1 \to \tau_2}\ [\textsc{U-Abs}]$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}\ [\textsc{U-App}] \qquad \frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \mathbf{fix}\ x : \tau.t : \tau}\ [\textsc{U-Fix}] \qquad \frac{\Gamma \vdash t_1 : \mathbf{int} \quad \Gamma \vdash t_2 : \mathbf{int}}{\Gamma \vdash t_1 \oplus t_2 : \mathbf{bool}}\ [\textsc{U-Op}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ \mathbf{seq}\ t_2 : \tau_2}\ [\textsc{U-Seq}] \qquad \frac{\Gamma \vdash t_1 : \mathbf{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \tau}\ [\textsc{U-If}]$$

$$\frac{}{\Gamma \vdash []_\tau : [\tau]}\ [\textsc{U-Nil}] \qquad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : [\tau]}{\Gamma \vdash t_1 :: t_2 : [\tau]}\ [\textsc{U-Cons}] \qquad \frac{\Gamma \vdash t_1 : [\tau_1] \quad \Gamma \vdash t_2 : \tau \quad \Gamma, x_1 : \tau_1, x_2 : [\tau_1] \vdash t_3 : \tau}{\Gamma \vdash \mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} : \tau}\ [\textsc{U-Case}]$$

Figure 5. Underlying type system ($\Gamma \vdash t : \tau$)

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ [\textsc{E-App}] \qquad \frac{}{(\lambda x : \widehat{\tau}\ \&\ \xi.t_1)\ t_2 \longrightarrow t_1[t_2/x]}\ [\textsc{E-AppAbs}] \qquad \frac{}{\lightning^\ell\ t_2 \longrightarrow \lightning^\ell}\ [\textsc{E-AppExn}]$$

$$\frac{t \longrightarrow t'}{t\ \langle\xi\rangle \longrightarrow t'\ \langle\xi\rangle}\ [\textsc{E-AnnApp}] \qquad \frac{}{(\Lambda e :: \kappa.t)\ \langle\xi\rangle \longrightarrow t[\xi/e]}\ [\textsc{E-AnnAppAbs}]$$

$$\frac{t \longrightarrow t'}{\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t \longrightarrow \mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t'}\ [\textsc{E-Fix}_1] \qquad \frac{}{\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t \longrightarrow t[\mathbf{fix}\ x : \widehat{\tau}\ \&\ \xi.t/x]}\ [\textsc{E-Fix}_2]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 \oplus t_2 \longrightarrow t_1' \oplus t_2}\ [\textsc{E-Op}_1] \qquad \frac{t_2 \longrightarrow t_2'}{t_1 \oplus t_2 \longrightarrow t_1 \oplus t_2'}\ [\textsc{E-Op}_2] \qquad \frac{}{v_1 \oplus v_2 \longrightarrow [\![v_1 \oplus v_2]\!]}\ [\textsc{E-Op}]$$

$$\frac{}{\lightning^\ell \oplus t_2 \longrightarrow \lightning^\ell}\ [\textsc{E-OpExn}_1] \qquad \frac{}{t_1 \oplus \lightning^\ell \longrightarrow \lightning^\ell}\ [\textsc{E-OpExn}_2]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ \mathbf{seq}\ t_2 \longrightarrow t_1'\ \mathbf{seq}\ t_2}\ [\textsc{E-Seq}_1] \qquad \frac{}{v_1\ \mathbf{seq}\ t_2 \longrightarrow t_2}\ [\textsc{E-Seq}_2] \qquad \frac{}{\lightning^\ell\ \mathbf{seq}\ t_2 \longrightarrow \lightning^\ell}\ [\textsc{E-SeqExn}]$$

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow \mathbf{if}\ t_1'\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3}\ [\textsc{E-If}] \qquad \frac{}{\mathbf{if\ true\ then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_2}\ [\textsc{E-IfTrue}]$$

$$\frac{}{\mathbf{if\ false\ then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow t_3}\ [\textsc{E-IfFalse}] \qquad \frac{}{\mathbf{if}\ \lightning^\ell\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \longrightarrow \lightning^\ell}\ [\textsc{E-IfExn}]$$

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

$$\frac{}{\mathbf{case}\ []\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow t_2}\ [\textsc{E-CaseNil}] \qquad \frac{}{\mathbf{case}\ t_1 :: t_1'\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \longrightarrow t_3[t_1; t_1'/x_1; x_2]}\ [\textsc{E-CaseCons}]$$

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

Figure 6. Operational semantics ($t_1 \longrightarrow t_2$)

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\mathbf{bool}}\ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_i \Rightarrow}\textsc{exn}}\ [\textsc{C-Bool}] \qquad \frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{int} : \widehat{\mathbf{int}}\ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_i \Rightarrow}\textsc{exn}}\ [\textsc{C-Int}]$$

$$\frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau}\ \&\ \xi \rhd \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau}\langle\xi\rangle]\ \&\ e\ \overline{e_i} \rhd e :: \overline{\kappa_i \Rightarrow}\textsc{exn}, \overline{e_j :: \kappa_j}}\ [\textsc{C-List}]$$

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

Figure 7. Type completion ($\Delta \vdash \tau : \widehat{\tau}\ \&\ \xi \rhd \Delta'$)

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

$$
\begin{array}{rcll}
t \in \mathbf{Tm} & ::= & x & \text{(term variable)} \\
& | & c_\tau & \text{(term constant)} \\
& | & \lightning^\ell_\tau & \text{(exceptional constant)} \\
& | & t_1\ t_2 & \text{(term application)} \\
& | & \lambda x : \tau . t & \text{(term abstraction)} \\
& | & \mathbf{fix}\ x : \tau . t & \text{(fixpoint)} \\
& | & t_1\ \mathbf{seq}\ t_2 & \text{(forcing)} \\
& | & t_1 \oplus t_2 & \text{(operator)} \\
& | & \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 & \text{(conditional)} \\
& | & []_\tau & \text{(nil constructor)} \\
& | & t_1 :: t_2 & \text{(cons constructor)} \\
& | & \mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} & \text{(list eliminator)}
\end{array}
$$

Values

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

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

Figure 8. Source language: syntax

Most constructs in Figure 8 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 for reference in Figure 5. 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.

3.2 Operational semantics

The operational semantics of the source language is given in Figure 6. Note that there is a small amount of non-determinism in the order of reduction. For example, in the derivation rules E-OPEXN$_1$ and E-OPEXN$_2$.[1]

The reduction rules E-ANNAPP and E-ANNABSAPP 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.

[1] We do not go so far as to have an *imprecise exception semantics* (Peyton Jones et al. 1999). For example, when the guard of a conditional evaluates to an exceptional value (E-IFEXN), we do not continue evaluation of the two branches in exception finding mode.

4. Exception types

The syntax of well-formed exception types is given in Figure 9 and Figure 10. An exception type $\widehat{\tau}$ is formed out of base types (booleans and integers), compound types (lists), function types, and quantifiers (ranging over exception variables).[2]

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 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.

$$
\begin{array}{rcll}
\kappa \in \mathbf{Ki} & ::= & \text{EXN} & \text{(exception set)} \\
& | & \kappa_1 \Rightarrow \kappa_2 & \text{(exception set operator)}
\end{array}
$$

$$
\begin{array}{rcll}
\xi, \zeta \in \mathbf{Exn} & ::= & e & \text{(exception set variables)} \\
& | & \lambda e : \kappa . \xi & \text{(exception set abstraction)} \\
& | & \xi_1\ \xi_2 & \text{(exception set application)} \\
& | & \emptyset & \text{(empty exception set)} \\
& | & \{\ell\} & \text{(singleton exception set)} \\
& | & \xi_1 \cup \xi_2 & \text{(exception set union)}
\end{array}
$$

$$
\begin{array}{rcll}
\widehat{\tau} \in \mathbf{ExnTy} & ::= & \forall e :: \kappa . \widehat{\tau} & \text{(exception set quantification)} \\
& | & \widehat{\mathbf{bool}} & \text{(boolean type)} \\
& | & \widehat{\mathbf{int}} & \text{(integer type)} \\
& | & [\widehat{\tau}\langle\xi\rangle] & \text{(list type)} \\
& | & \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle & \text{(function type)}
\end{array}
$$

Figure 9. Exception types: syntax

Type signatures are denoted as $f\ :\ \widehat{\tau}\ \&\ \xi$, where $\widehat{\tau}$ is the exception type of f and ξ is the (top-most) effect.

Example 1. The identity function

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

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.

[2] To avoid complicating the presentation we do *not* allow quantification over type variables, i.e. polymorphism in the underlying type system.

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

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

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

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

Figure 10. Exception types: well-formedness ($\Delta \vdash \widehat{\tau}$ wff)

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. 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 (Girard 1972) 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 resticted to the types, over effects, and not allowed 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 (Figure 11). 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 from the $\lambda^{\cup}$-calculus (Definition 2).

$$\frac{\Delta, e :: \kappa \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2}{\Delta \vdash \forall e :: \kappa.\widehat{\tau}_1 \leqslant \forall e :: \kappa.\widehat{\tau}_2}\ [\text{S-Forall}]$$

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

$$\frac{\Delta \vdash \widehat{\tau} \leqslant \widehat{\tau}' \quad \Delta \vdash \xi \leqslant \xi'}{\Delta \vdash [\widehat{\tau}\langle\xi\rangle] \leqslant [\widehat{\tau}'\langle\xi'\rangle]}\ [\text{S-List}]$$

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

Figure 11. Exception types: subtyping relation ($\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2$)

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.

These observations lead to the following definition of *conservative exception types*:[3]

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 is 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.

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 thus be exception set variables that have been universally quantified.

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

$$tail : \forall e_1\ e_2.\big[\widehat{\mathbf{bool}}\langle e_1\rangle\big]\langle e_2\rangle \to \big[\widehat{\mathbf{bool}}\langle e_1\rangle\big]\langle e_2 \cup \{\mathbf{E}\}\rangle\ \&\ \emptyset$$

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—being strict in both arguments—will propagate the exceptional values:

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

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.

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

The first (functional) argument of *apply* has exception type $\forall e_1 :: \text{EXN}.\widehat{\mathbf{bool}}\langle e_1\rangle \to \widehat{\mathbf{bool}}\langle e_3\ e_1\rangle$ 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

[3] Holdermans and Hage (2010) call pattern types *fully parametric* and conservative types *fully flexible*.

exception. All functions of underlying type **bool** $\to$ **bool** satisfy these constraints, so they do not really constrain us in any way.

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 the source language is not invariant under η-conversion in the presence of exceptional values—so neither are exception types. The term

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

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

$$\lightning^{\mathbf{E}}_{\mathbf{bool}\to\mathbf{bool}} \quad : \forall e :: \textsc{exn}.\widehat{\mathbf{bool}}\langle e\rangle \xrightarrow{\{\mathbf{E}\}} \widehat{\mathbf{bool}}\langle\emptyset\rangle$$

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 by forcing the closure

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

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 in Figure 7, defining a type completion relation $\Delta \vdash \tau : \widehat{\tau}\ \&\ \xi \triangleright \Delta'$. 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

$$\begin{array}{l} \forall e_2 :: \textsc{exn}.\forall e_2' :: \textsc{exn}.\forall e_3 :: \textsc{exn} \Rightarrow \textsc{exn}. \\ \quad [\forall e_1 :: \textsc{exn}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{e_2'} \widehat{\mathbf{bool}}\langle e_3\ e_1\rangle]\langle e_2\rangle \to \\ \qquad (\forall e_5 :: \textsc{exn}.\forall e_5' :: \textsc{exn}.[\widehat{\mathbf{bool}}\langle e_5'\rangle]\langle e_5\rangle \xrightarrow{e_6\ e_2\ e_2'\ e_3} \\ \qquad\qquad \big[\widehat{\mathbf{bool}}\langle e_7'\ e_2\ e_2'\ e_3\ e_5\ e_5'\rangle\big]\langle e_7\ e_2\ e_2'\ e_3\ e_5\ e_5'\rangle) \end{array}$$

with effect e_4, and while introducing free variables

$$\begin{array}{l} e_4 :: \textsc{exn}, \\ e_6 :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow (\textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow \textsc{exn}, \\ e_7, e_7' :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow (\textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn} \end{array}$$

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 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:

$$\begin{array}{l} f :: \forall e_1, e_2 :: \textsc{exn}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{\emptyset} \widehat{\mathbf{bool}}\langle e_2\rangle \xrightarrow{e_1} \widehat{\mathbf{bool}}\langle e_2\rangle \\ f = \lambda x : \mathbf{bool}.x\ \mathbf{seq}\ \lambda y : \mathbf{bool}.y \end{array}$$

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

4.4 Least exception types

Given an effect kind $\overline{\kappa_i \Rightarrow}\textsc{exn}$, denote by $\emptyset_{\overline{\kappa_i \Rightarrow}\textsc{exn}}$ the effect $\lambda\overline{e_i :: \kappa_i}.\emptyset$. Besides completing an underlying type τ to a most general exception type, we also want to compute a least exception type $\bot_\tau$. This can be accomplished 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

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

is the conservative type

$$\begin{array}{l} \forall e_2 :: \textsc{exn}.\forall e_2' :: \textsc{exn}.\forall e_3 :: \textsc{exn} \Rightarrow \textsc{exn}. \\ \quad [\forall e_1 :: \textsc{exn}.\widehat{\mathbf{bool}}\langle e_1\rangle \xrightarrow{e_2'} \widehat{\mathbf{bool}}\langle e_3\ e_1\rangle]\langle e_2\rangle \to \\ \qquad (\forall e_5 :: \textsc{exn}.\forall e_5' :: \textsc{exn}.[\widehat{\mathbf{bool}}\langle e_5'\rangle]\langle e_5\rangle \xrightarrow{\emptyset} \big[\widehat{\mathbf{bool}}\langle\emptyset\rangle\big]\langle\emptyset\rangle) \end{array}$$

4.5 Exception typing and elaboration

In Figure 13 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 constructs: term-level effect abstraction and effect application.

Terms

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

Figure 12. Source language: extended syntax

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 14).

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, so we can omit them from the type without introducing ambiguity. For example, the exception type of the *map* function from the introduction can be presented as:

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

$$\frac{}{\Gamma, x : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash x : \widehat{\tau} \mathbin{\&} \xi}\;[\text{T-Var}] \qquad \frac{}{\Gamma; \Delta \vdash c_\tau : \bot_\tau \mathbin{\&} \emptyset}\;[\text{T-Con}] \qquad \frac{}{\Gamma; \Delta \vdash \lightning^{\ell}_{\tau} : \bot_\tau \mathbin{\&} \{\ell\}}\;[\text{T-Crash}]$$

$$\frac{\Gamma, x : \widehat{\tau}_1 \mathbin{\&} \xi_1; \Delta \vdash t : \widehat{\tau}_2 \mathbin{\&} \xi_2}{\Gamma; \Delta \vdash \lambda x : \widehat{\tau}_1 \mathbin{\&} \xi_1 . t : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \emptyset}\;[\text{T-Abs}] \qquad \frac{\Gamma; \Delta, e :: \kappa \vdash t : \widehat{\tau} \mathbin{\&} \xi \quad e \notin \mathrm{fv}(\Gamma, \xi)}{\Gamma; \Delta \vdash \Lambda e :: \kappa . t : \forall e :: \kappa . \widehat{\tau} \mathbin{\&} \xi}\;[\text{T-AnnAbs}]$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \mathbin{\&} \xi \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau}_2 \mathbin{\&} \xi_2}{\Gamma; \Delta \vdash t_1\ t_2 : \widehat{\tau} \mathbin{\&} \xi}\;[\text{T-App}] \qquad \frac{\Gamma; \Delta \vdash t_1 : \forall e :: \kappa . \widehat{\tau} \mathbin{\&} \xi \quad \Delta \vdash \xi_2 : \kappa}{\Gamma; \Delta \vdash t_1\ \langle\xi_2\rangle : \widehat{\tau}[\xi_2/e] \mathbin{\&} \xi}\;[\text{T-AnnApp}]$$

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

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

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

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

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

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

Figure 13. Declarative type system ($\Gamma; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$)

$$\frac{}{\Gamma, x : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash x \hookrightarrow x : \widehat{\tau} \mathbin{\&} \xi}\;[\text{L-Var}] \qquad \frac{}{\Gamma; \Delta \vdash c_\tau \hookrightarrow c_\tau : \bot_\tau \mathbin{\&} \emptyset}\;[\text{L-Con}] \qquad \frac{}{\Gamma; \Delta \vdash \lightning^{\ell}_{\tau} \hookrightarrow \lightning^{\ell}_{\tau} : \bot_\tau \mathbin{\&} \{\ell\}}\;[\text{L-Crash}]$$

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

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

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

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

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

$$\frac{}{\Gamma; \Delta \vdash []_\tau \hookrightarrow []_\tau : [\bot_\tau\langle\emptyset\rangle] \mathbin{\&} \emptyset}\;[\text{L-Nil}] \qquad \frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\tau}_1 \mathbin{\&} \xi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \left[\widehat{\tau}_1'\langle\xi_1'\rangle\right] \mathbin{\&} \xi_2}{\Gamma; \Delta \vdash t_1 :: t_2 \hookrightarrow t_1' :: t_2' : \left[\widehat{\tau}_1 \sqcup \widehat{\tau}_1'\langle\xi_1 \cup \xi_1'\rangle\right] \mathbin{\&} \xi_2}\;[\text{L-Cons}]$$

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

Figure 14. Syntax-directed type elaboration system ($\Gamma; \Delta \vdash t \hookrightarrow t' : \widehat{\tau} \mathbin{\&} \xi$)

5. Type inference

In this section we give an inference algorithm for the exception types presented in the previous section. (Term elaboration is omitted, but is straightforward to implement.)

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

Figure 15. Type inference algorithm ($\mathcal{R}$)

5.1 Polymorphic abstraction

The cases for abstraction and application are handled similarly to the corresponding cases in Holdermans and Hage (2010).

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 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.

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

Figure 16. 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. This case is handled by a Kleene–Mycroft iteration—which we conjecture to always converge.[4]

Example 8 (Dussart, Henglein, Mossin)**.** Consider the term

$$
\begin{array}{l}
f\ :\ \mathbf{bool} \to \mathbf{bool} \to \mathbf{bool} \\
f = \mathbf{fix} f' : \mathbf{bool} \to \mathbf{bool} \to \mathbf{bool}. \\
\qquad\quad \lambda x : \mathbf{bool} . \lambda y : \mathbf{bool} .\ \mathbf{if}\ x\ \mathbf{then\ true\ else}\ f'\ y\ x
\end{array}
$$

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

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

First let's think about why the elaborated term is type-correct.

$$x : \widehat{\mathbf{bool}}\ \&\ e_1$$

$$\mathbf{true} : \widehat{\mathbf{bool}}\ \&\ \emptyset$$

$$f \langle e_2 \rangle\ y\ \langle e_1 \rangle\ x : \widehat{\mathbf{bool}}\ \&\ e_2 \cup e_1$$

Therefore,

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

[4] Holdermans and Hage (2010) 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.

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

$$\textbf{if}\ x\ \textbf{then true else}\ f\,\langle e_2\rangle\ y\ \langle e_1\rangle\ x : \widehat{\textbf{bool}}\ \&\ e_1 \cup e_2$$

Of course, checking type-correctness is easier than type-inference. To infer the type of the fixed-point f we have to "guess" a type for it. How do we guess this type? We first try the least exception type $\bot_{\textbf{bool}\to\textbf{bool}\to\textbf{bool}}$:

$$\forall e_1.\widehat{\textbf{bool}}\langle e_1\rangle \overset{\emptyset}{\to} \forall e_2.\widehat{\textbf{bool}}\langle e_2\rangle \overset{\emptyset}{\to} \widehat{\textbf{bool}}\langle \emptyset\rangle$$

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

$$\forall e_1.\widehat{\textbf{bool}}\langle e_1\rangle \overset{\emptyset}{\to} \forall e_2.\widehat{\textbf{bool}}\langle e_2\rangle \overset{\emptyset}{\to} \widehat{\textbf{bool}}\langle e_1\rangle$$

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:

$$\forall e_1.\widehat{\textbf{bool}}\langle e_1\rangle \overset{\emptyset}{\to} \forall e_2.\widehat{\textbf{bool}}\langle e_2\rangle \overset{\emptyset}{\to} \widehat{\textbf{bool}}\langle e_1 \cup e_2\rangle$$

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 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 annotations occurring in argument positions are always patterns makes this easy, as they must be equal up to α-renaming of bound variables (Holdermans and Hage 2010). This allows us to treat those arguments invariantly instead of contravariantly, obviating the need to also compute greatest lower bounds of types.

$$\begin{array}{lll}
\multicolumn{3}{l}{\cdot \sqcup \cdot\ : \textbf{ExnTy} \times \textbf{ExnTy} \to \textbf{ExnTy}} \\
\widehat{\textbf{bool}} & \sqcup\ \widehat{\textbf{bool}} & = \widehat{\textbf{bool}} \\
\widehat{\textbf{int}} & \sqcup\ \widehat{\textbf{int}} & = \widehat{\textbf{int}} \\
[\widehat{\tau}\langle\xi\rangle] & \sqcup\ \left[\widehat{\tau}'\langle\xi'\rangle\right] & = \left[(\widehat{\tau} \sqcup \widehat{\tau}')\langle\xi \cup \xi'\rangle\right] \\
\widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle\xi\rangle & \sqcup\ \widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2'\langle\xi'\rangle & = \widehat{\tau}_1\langle e\rangle \to (\widehat{\tau}_2 \sqcup \widehat{\tau}_2')\langle\xi \cup \xi'\rangle \\
(\forall e :: \kappa.\widehat{\tau}) & \sqcup\ (\forall e :: \kappa.\widehat{\tau}') & = \forall e :: \kappa.\widehat{\tau} \sqcup \widehat{\tau}'
\end{array}$$

Figure 17. 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 (Mairson 1990; Kfoury et al. 1990a). 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 (Statman 1979). 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.

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 (Hindley 1969; Milner 1978; Damas and Milner 1982). Type systems were extended with effects—including **let**-bound effect-polymorphism—by Lucassen and Gifford (1988); Jouvelot and Gifford (1991); and Talpin and Jouvelot (1992, 1994). 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 (Tofte and Talpin 1994), dimensions (Kennedy 1994; Rittri 1994, 1995), binding times (Dussart et al. 1995), and exceptions (Glynn et al. 2002; Koot and Hage 2015).

Inferring principal types in a type system with polymorphic recursion is equivalent to solving the undecidable semi-unification problem (Mycroft 1984; Kfoury et al. 1990b, 1993; Henglein 1993). When restricted to polymorphic recursion in the effects, the problem often becomes decidable again. In Tofte and Talpin (1994) this is a conjecture based on empirical observation. Rittri (1995) gives a semi-unification procedure based on the general semi-unification semi-algorithm by Baaz (1993) and proves it terminates in the special case of semi-unification in Abelian groups. Dussart et al. (1995) 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 Kennedy (1996a) 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 ($\Lambda_{\delta\omega}$). This language can type strictly more programs than the language without dimension operators:

$$\begin{array}{lll}
\textit{twice} & : & \forall F :: \text{DIM} \Rightarrow \text{DIM}. \\
 & & \quad (\forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle F\ d\rangle) \to \\
 & & \quad\quad (\forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle F\ (F\ d)\rangle) \\
\textit{twice} & = & \Lambda F :: \text{DIM} \Rightarrow \text{DIM}. \\
 & & \quad \lambda f : (\forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle F\ d\rangle). \\
 & & \quad\quad \Lambda d :: \text{DIM}.\lambda x : \textbf{real}\langle d\rangle.f\ \langle F\ d\rangle\ (f\ \langle d\rangle\ x) \\
\textit{square} & : & \forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle d^2\rangle \\
\textit{square} & = & \Lambda d :: \text{DIM}.\lambda x : \textbf{real}\langle d\rangle.x^2 \\
\textit{fourth} & : & \forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle d^4\rangle \\
\textit{fourth} & = & \textit{twice}\ \langle \Lambda d :: \text{DIM}.d^2\rangle\ \textit{square} \\
\textit{sixteenth} & : & \forall d :: \text{DIM}.\textbf{real}\langle d\rangle \to \textbf{real}\langle d^{16}\rangle \\
\textit{sixteenth} & = & \textit{twice}\ \langle \Lambda d :: \text{DIM}.d^4\rangle\ \textit{fourth}
\end{array}$$

Without dimension operators the type of the higher-order function *twice* would need to be specialized to a type that is either only applicable in *fourth*, or a type that is only applicable in *sixteenth*.

The language $\Lambda_{\delta\omega}$ bears a striking resemblance to the language in Figure 9: 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.

Faxén (1997) presents a type system for flow analysis that uses constrained type schemes in the style of Aiken and Wimmers (1993), 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. Smith and Wang (2000) present a similar framework, but one that can be instantiated with variants of either k-CFA (Shivers 1991) or CPA (Agesen 1995) to ensure termination.

Holdermans and Hage (2010) 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 extends upon 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 extended 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 $\lambda^{\cup}$-calculus

Tannen (1988), Okada (1989), and Tannen and Gallier (1991) 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.

Révész (1992) introduced an untyped λ-calculus with applicative lists. A model is given by Durfee (1997). 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 Yi (1994) is based on abstract interpretation. Guzmán and Suárez (1994) and Fähndrich et al. (1998) describe type-based exception analyses. Leroy and Pessaux (2000) presents a row-based type system for exception analysis that contains a data-flow analysis component targeted towards tracking value-carrying exceptions.

Glynn et al. (2002) developed the first exception analysis for a non-strict language. It is a type-based analysis using Boolean constraints. In (Koot and Hage 2015) we presented 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—while type correct in the underlying type system—they 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) unification problem, which is only semi-decidable. Snyder (1990) and Nipkow and Qian (1991) do give us semi-algorithms for solving such problems (although—at least in the latter approach—the equational theory is assumed to be regular, which the theory of Abelian groups is not).

Can we further improve the precision of exception types? It is argued by Koot and Hage (2015) 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 Mitchell and Runciman (2008):

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

The inference algorithm in Figure 15 assigns *risers* the type

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

where $\mathbf{E}$ is the label of the exception raised when the pattern-match in the **where**-clause fails.[5] 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 $\mathbf{E}$ can thus never be raised.

Koot and Hage (2015) 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. Koot and Hage (2015) use a slightly ad hoc form of conditional constraints to model this dependence. We now believe extending a λ-calculus with an equational theory of Boolean rings may form the basis of a more principled approach. Booleans rings have already been successfully used to design type systems for strictness analysis (Wright 1991), records (Kennedy 1996b) and exception tracking (Benton and Buchlovsky 2007).

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.

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

References

Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26. Springer-Verlag, 1995. ISBN 3-540-60160-0.

Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41. ACM, 1993. ISBN 0-89791-595-X.

Matthias Baaz. Note on the existence of most-general semi-unifiers. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 19–28. Oxford University Press, 1993.

Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 15–26. ACM, 2007. ISBN 1-59593-393-X.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982. ISBN 0-89791-065-6.

Gilles Dowek. Higher-order unification and matching. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier Science Publishers, 2001. ISBN 0-444-50812-0.

Glenn Durfee. A model for a list-oriented extension of the lambda calculus. MSc thesis, Carnegie Mellon University, 1997.

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

Manuel Fähndrich, Jeffrey Foster, Jason Cu, and Alexander Aiken. Tracking down exceptions in Standard ML programs. Technical report, 1998.

Karl-Filip Faxén. Polyvariance, polymorphism and flow analysis. In *Selected Papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, pages 260–278. Springer-Verlag, 1997. ISBN 3-540-62503-8.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

Kevin Glynn, Peter J. Stuckey, Martin Sulzmann, and Harald Søndergaard. Exception analysis for non-strict languages. ICFP '02, pages 98–109, 2002.

Juan Carlos Guzmán and Ascánder Suárez. An extended type system for exceptions. ML '94, pages 127–135, 1994.

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

J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969. ISSN 00029947.

Stefan Holdermans and Jurriaan Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 63–74. ACM, 2010. ISBN 978-1-60558-794-3.

Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 303–310. ACM, 1991. ISBN 0-89791-419-8.

Andrew J. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*, ESOP '94, pages 348–362. Springer, 1994. ISBN 3-540-57880-3.

Andrew J. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996a.

Andrew J. Kennedy. Type inference and equational theories. Technical Report LIX-RR-96-09, Laboratoire D'Informatique, École Polytechnique, 1996b.

Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. ML typability is DEXPTIME-complete. In *Proceedings of the Fifteenth Colloquium on CAAP'90*, CAAP '90, pages 206–220. Springer-Verlag, 1990a. ISBN 0-387-52590-4.

Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 468–476. ACM, 1990b. ISBN 0-89791-361-2.

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

Ruud Koot and Jurriaan Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 127–138. ACM, 2015. ISBN 978-1-4503-3297-2.

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

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57. ACM, 1988. ISBN 0-89791-252-7.

Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 382–401. ACM, 1990. ISBN 0-89791-343-4.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer, 1991. ISBN 978-3-540-53590-4.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984. ISBN 978-3-540-12925-7.

Tobias Nipkow and Zhenyu Qian. Modular higher-order E-unification. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1991. ISBN 978-3-540-53904-9.

Mitsuhiro Okada. Strong normalizability for the combined system of the types lambda calculus and an arbitrary convergent term rewrite system. In *Proc. ISSAC 89*, 1989.

Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. PLDI '99, pages 25–36, 1999.

György E. Révész. A list-oriented extension of the lambda-calculus satisfying the Church–Rosser theorem. *Theor. Comput. Sci.*, 93(1):75–89, February 1992. ISSN 0304-3975.

Mikael Rittri. Semi-unification of two terms in abelian groups. *Inf. Process. Lett.*, 52(2):61–68, October 1994. ISSN 0020-0190.

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

Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.

Scott F. Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67262-3.

Wayne Snyder. Higher order E-unification. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 573–587, London, UK, UK, 1990. Springer-Verlag. ISBN 3-540-52885-7.

Richard Statman. The typed lambda calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–82, 1979.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992. ISSN 1469-7653.

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

Val Tannen. Combining algebra and higher-order types. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 82–90, 1988.

Val Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3 – 28, 1991. ISSN 0304-3975.

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

David A. Wright. A new technique for strictness analysis. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT '91*, volume 494 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 1991. ISBN 978-3-540-53981-0.

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