Higher-ranked Exception Types *

Ruud Koot
Utrecht University
inbox@ruudkoot.nl

Jurriaan Hage
Utrecht University
j.hage@uu.nl

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 extends 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 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:

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

For each type τ, we denote its exception-annotated type by $\tau\langle\xi\rangle$. For function types we sometimes 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 is based upon work supported by the Netherlands Organisation for Scientific Research (NWO) under the project *Higher-ranked Polyvariance Explored* (612.001.120).

CONF 'yy, Month d–d, 20yy, City, ST, Country.

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 of 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 an argument to an exceptional value 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 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 two different functions to *map*: 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 introduce 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 produce only non-exceptional values or the exceptional value $\lightning^{\mathbf{E}}$ as elements of the lists.

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 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.

Types

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

Terms

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

Environments

$$\Gamma \in \mathbf{Env} ::= \cdot \quad | \quad \Gamma, x : \tau$$

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

2.1 Typing relation

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

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

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

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

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

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.

2.2 Semantics

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

Types and values

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

Environments

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

Terms

$$\llbracket x \rrbracket_\rho = \rho(x)$$
$$\llbracket \lambda x : \tau . t \rrbracket_\rho = \left\{ \lambda v \in V_\tau . \llbracket t \rrbracket_{\rho[x \mapsto v]} \right\}$$
$$\llbracket t_1\ t_2 \rrbracket_\rho = \bigcup \{ \varphi(\llbracket t_2 \rrbracket_\rho) \mid \varphi \in \llbracket t_1 \rrbracket_\rho \}$$
$$\llbracket \emptyset \rrbracket_\rho = \emptyset$$
$$\llbracket \{c\} \rrbracket_\rho = \{c\}$$
$$\llbracket t_1 \cup t_2 \rrbracket_\rho = \llbracket t_1 \rrbracket_\rho \cup \llbracket t_2 \rrbracket_\rho$$

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

Lemma 1. *The terms* $(t_1 \cup t_2)\ t$ *and* $t_1\ t \cup t_2\ t$ *are equivalent.*

Lemma 2. *The terms* $(\lambda x : \tau . t_1) \cup (\lambda x : \tau . t_2)$ *and* $\lambda x : \tau . t_1 \cup t_2$ *are extensionally equivalent.*

Proof. We show that

$$\llbracket ((\lambda x : \tau . t_1) \cup (\lambda x : \tau . t_2))\ t_3 \rrbracket_\rho = \llbracket (\lambda x : \tau . t_1 \cup t_2)\ t_3 \rrbracket_\rho$$

for all suitable ρ and t_3. □

2.3 TO DO.Subset relation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

2.4 TO DO.Normalization

- TO DO: We can make union only work on base types (as we not longer *need* to distribute unions over applications)? Then the denotation of the function space would be simpler and might generalize to other structures..

To reduce $\lambda^{\cup}$-terms to a 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 set-union operator.

If a term t is η-long it can be written in the form

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

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 $1 \leqslant i \leqslant p$). The notation $\{ f_1(t_{11}, ..., t_{1q_1}), ..., f_p(t_{p1}, ..., t_{pq_p}) \}$ is a shorthand for $f_1(t_{11}, ..., t_{1q_1}) \cup \cdots \cup f_p(t_{p1}, ..., t_{pq_p}) \}$, where we forget the associativity of the set-union operator and any empty-set constants. Note that despite the suggestive notation, this is not a true set, as there may still be duplicate elements $f_i(t_{i1}, ..., t_{iq_i})$.

A normal form v of a term t can be written as

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

where k_i can be a free or bound variable, or a singleton-set constant, but not a term as this would form a β-redex.[1] For each k_i, k_j with $i < j$ we must also have that $k_i < k_j$ for some total order on $\mathbf{Var} \cup \mathbf{Con}$. Not only does this imply that each term $k_i(v_{i1}, ..., v_{iq_i})$ occurs only once in $k_1(v_{11}, ..., v_{1q_1}), ..., k_p(v_{p1}, ..., v_{pq_p})$, but also the stronger condition that $k_i \neq k_j$ for all $i \neq j$.

-- normalization of terms
$\llbracket \cdot \rrbracket :: \mathbf{Tm} \to \mathbf{Nf}$
$\llbracket \lambda x_1 \cdots x_n . T \rrbracket =$
$\quad \lambda x_1 \cdots x_n . \wr \{ \lfloor f_i(\llbracket t_{i1} \rrbracket, ..., \llbracket t_{iq_i} \rrbracket) \rfloor \mid f_i(t_{i1}, ..., t_{iq_i}) \in T \} \wr$
-- β-reduction
$\lfloor k(v_1, ..., v_q) \rfloor$
$\quad = k(v_1, ..., v_q)$
$\lfloor (\lambda y_1 \cdots y_q . T)\ (v_1, \cdots, v_q) \rfloor$
$\quad = SUBST\ x\ y\ z$
-- set-rewriting
$\wr \{ \cdots, k_i\ (\cdots), \cdots, k_j\ (\cdots), \cdots \} \wr$
$\quad \mid k_j < k_i = \wr \{ \cdots, k_i\ (\cdots), \cdots, k_j\ (\cdots), \cdots \} \wr$
$\wr \{ \cdots, k\ (\cdots), k\ (\cdots), \cdots \} \wr$
$\quad = \wr \{ \cdots, k\ (\cdots), \cdots \} \wr$
$\wr T \wr$
$\quad = T$

Figure 4. TO DO.Normalization algorithm of $\lambda^{\cup}$-terms.

2.5 Pattern unification

Definition 1. 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 normally 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

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

[1] Technically, terms that bind at least one variable would form a β-redex. Terms that do not bind any variables do not occur either as they merely form a subsequence of $k_1(v_{11}, ..., v_{1q_1}), ..., k_p(v_{p1}, ..., v_{pq_p})$ in this notation.

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

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

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

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

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ [\text{E-App}] \qquad \frac{}{(\lambda x : \widehat{\tau}\ \&\ \xi . t_1)\ t_2 \longrightarrow t_1[t_2/x]}\ [\text{E-AppAbs}] \qquad \frac{t \longrightarrow t'}{t\ \langle \xi \rangle \longrightarrow t'\ \langle \xi \rangle}\ [\text{E-AnnApp}]$$

$$\frac{}{(\Lambda e :: \kappa . t)\ \langle \xi \rangle \longrightarrow t[\xi/e]}\ [\text{E-AnnAppAbs}] \qquad \frac{t \longrightarrow t'}{\mathbf{fix}\ t \longrightarrow \mathbf{fix}\ t'}\ [\text{E-Fix}] \qquad \frac{}{\mathbf{fix}\ (\lambda x : \widehat{\tau}\ \&\ \xi . t) \longrightarrow t[\mathbf{fix}\ (\lambda x : \widehat{\tau}\ \&\ \xi . t)/x]}\ [\text{E-FixAbs}]$$

$$\frac{}{\lightning^\ell\ t_2 \longrightarrow \lightning^\ell}\ [\text{E-AppExn}] \qquad \frac{}{\mathbf{fix}\ \lightning^\ell \longrightarrow \lightning^\ell}\ [\text{E-FixExn}] \qquad \frac{t_1 \longrightarrow t_1'}{t_1 \oplus t_2 \longrightarrow t_1' \oplus t_2}\ [\text{E-Op}_1] \qquad \frac{t_2 \longrightarrow t_2'}{t_1 \oplus t_2 \longrightarrow t_1 \oplus t_2'}\ [\text{E-Op}_2]$$

$$\frac{}{v_1 \oplus v_2 \longrightarrow [\![v_1 \oplus v_2]\!]}\ [\text{E-Op}] \qquad \frac{}{\lightning^\ell \oplus t_2 \longrightarrow \lightning^\ell}\ [\text{E-OpExn}_1] \qquad \frac{}{t_1 \oplus \lightning^\ell \longrightarrow \lightning^\ell}\ [\text{E-OpExn}_2]$$

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

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

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

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

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

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

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

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

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

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

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

3. Source language

Our analysis is applicable to a simple non-strict functional language that supports Boolean, integer and list data types. In this section we'll briefly state its syntax and semantics.

3.1 Syntax

The terms and values are given in Figure 8.

Terms

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

Values

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

$$\widehat{v} \in \mathbf{ExnVal} \ ::= \ \lightning_\tau^\ell \mid v$$

Figure 8. Source language: syntax

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 $\widehat{v}$.

3.2 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 and we assume that any term we type in the type-and-effect system is already well-typed in the underlying type system.

3.3 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$_2$.

We do not go as far as having an imprecise exception semantics (Peyton Jones et al. 1999). For example, if 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.

The reduction rules E-ANNAPP, 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.

4. Exception types

4.1 Exception types

The syntax of well-formed exception types are given in Figure 9 and Figure 10. An exception type $\widehat{\tau}$ is formed out of base types (booleans), compound types (lists), function types and quantification 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 which may be raised when an element of the list is forced. The effect ζ gives a set of exceptions that 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 by 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 that may be raised if the function closure is forced.

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

Figure 9. Exception types: syntax

Example 1. The identity function

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

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, for example as happens when it is applied to an argument. 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. polymorpism in the underlying type system.

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

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

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

$$\frac{\begin{array}{c}\Delta \vdash \widehat{\tau}_1\ \mathrm{wff} \quad \Delta \vdash \xi_1 :: \textsc{exn} \\ \Delta \vdash \widehat{\tau}_2\ \mathrm{wff} \quad \Delta \vdash \xi_2 :: \textsc{exn}\end{array}}{\Delta \vdash \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle\ \mathrm{wff}}\ [\textsc{W-Arr}]$$

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.

4.2 To do.Conservativeness

To do: Atomicity: $e_1 \cup e_2 \to \mathit{ExnTyList}\ e_1\ e_2$ is not useful, because no introspection

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 our type system to be modular—allowing type checking and inference to work on individual modules instead of whole programs—we cannot 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.[3]

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

- To do.check all examples types against prototype
- To do.properly typeset example types
- To do.Skolemization and expliciy existential quantication over unification variables?

Example 3 (To do.).

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

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

[3] Holdermans and Hage (2010) call such types *fully flexible*.

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.

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

The first (functional) argument of *apply* has exception type $\forall e_1 : \textsc{exn}.\widehat{\mathbf{bool}}\langle e_1\rangle \to \widehat{\mathbf{bool}}\langle e_3\ \ e_1\rangle$ 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 exception. All functions of underlying type $\mathbf{bool} \to \mathbf{bool}$ 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 annotation 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. Exception types are not invariant under η-conversion. 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

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

We cannot observe a difference between these terms by applying an argument to them

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

but we can by forcing the closure

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

4.3 To do.Subtyping

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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

$$\frac{}{\Delta \vdash \widehat{\mathbf{bool}} \leqslant \widehat{\mathbf{bool}}}\ [\textsc{S-Bool}] \qquad \frac{}{\Delta \vdash \widehat{\mathbf{int}} \leqslant \widehat{\mathbf{int}}}\ [\textsc{S-Int}]$$

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

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

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

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

4.4 To do.Exception type completion

Give an underlying type τ we can compute a 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 judgement $\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau} \,\&\, \xi \triangleright \overline{e_j :: \kappa_j}$ is read as: 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 $\overline{e_j :: \kappa_j}$.

Example 6 (To do.)**.** First-order, multiple arguments:

$$\begin{array}{ll}
\vdash & [\mathbf{bool}] \to [\mathbf{bool}] \to [\mathbf{bool}] \\
: & \forall e_1 :: \textsc{exn}.\forall e_1' :: \textsc{exn}.\left[\widehat{\mathbf{bool}}\langle e_1\rangle\right]\langle e_1'\rangle \to \\
& \quad (\forall e_2 :: \textsc{exn}.\forall e_2' :: \textsc{exn}.\left[\widehat{\mathbf{bool}}\langle e_2\rangle\right]\langle e_2'\rangle \to \\
& \qquad \left[\widehat{\mathbf{bool}}\langle e_3\ e_1\ e_1'\ e_2\ e_2'\rangle\right]\langle e_3'\ e_1\ e_1'\ e_2\ e_2'\rangle)\langle e_4\rangle \\
\& & e_5 \\
\triangleright & e_3, e_3' :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}, e_4, e_5 :: \textsc{exn}
\end{array}$$

Higher-order:

$$\begin{array}{ll}
\vdash & ([\mathbf{bool}] \to [\mathbf{bool}]) \to [\mathbf{bool}] \\
: & \forall e_2 :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}.\forall e_2' :: \textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}.\forall e_3 :: \textsc{exn}. \\
& \quad (\forall e_1 :: \textsc{exn}.\forall e_1' :: \textsc{exn}.\left[\widehat{\mathbf{bool}}\langle e_1\rangle\right]\langle e_1'\rangle \to \left[\widehat{\mathbf{bool}}\langle e_2\ e_1\ e_1'\rangle\right]\langle e_2'\ e_1\ e_1'\rangle)\langle e_3\rangle \to \\
& \qquad \left[\widehat{\mathbf{bool}}\langle e_4\ e_2\ e_2'\ e_3\rangle\right]\langle e_4'\ e_2\ e_2'\ e_3\rangle \\
\& & e_5 \\
\triangleright & e_4, e_4' :: (\textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow (\textsc{exn} \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}) \Rightarrow \textsc{exn} \Rightarrow \textsc{exn}, e_5 :: \textsc{exn}
\end{array}$$

4.5 To do.Least exception types

To do.

4.6 To do.Declarative exception type system

In Figure 13 we give a declarative system for deriving exception typing judgements $\Gamma; \Delta \vdash t : \widehat{\tau} \,\&\, \xi$.

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

4.7 To do.Type elaboration system

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 and explitly typed term (Figure 14).

Terms

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

Figure 12. Source language: extended syntax

4.8 Type simplification

In their fully flexible form 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 to the programmer as:

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

- To do: Type erasure relation
- To do: leat upper bound (declarativly in terms of subtyping

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

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

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

$$\frac{\Delta \vdash \xi' \leqslant \xi \quad \Delta \vdash \xi'' \leqslant \xi \quad \Gamma; \Delta \vdash t : \widehat{\tau}\langle \xi' \rangle \to \widehat{\tau}\langle \xi' \rangle \,\&\, \xi''}{\Gamma; \Delta \vdash \mathbf{fix}\ t : \widehat{\tau} \,\&\, \xi}\ [\text{T-Fix}]$$

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma; \Delta \vdash t \hookrightarrow t' : \forall \overline{e_i :: \kappa_i} . \widehat{\tau}\langle \xi \rangle \to \widehat{\tau}'\langle \xi' \rangle \,\&\, \xi'' \quad \Delta \vdash \widehat{\tau}'[\overline{\xi_i}/\overline{e_i}] \leqslant \widehat{\tau}[\overline{\xi_i}/\overline{e_i}] \quad \Delta \vdash \xi'[\overline{\xi_i}/\overline{e_i}] \leqslant \xi[\overline{\xi_i}/\overline{e_i}] \quad \overline{\Delta \vdash \xi_i :: \kappa_i}}{\Gamma; \Delta \vdash \mathbf{fix}\ t \hookrightarrow \mathbf{fix}\ t'\ \langle \overline{\xi_i} \rangle : \widehat{\tau}[\overline{\xi_i}/\overline{e_i}] \,\&\, \xi[\overline{\xi_i}/\overline{e_i}] \cup \xi''}\ [\text{L-Fix}]$$

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

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

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

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

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

5. Type inference

We now give an algorithm that infers the exception types presented in the previous section.

$$
\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\quad = \Gamma_x\\
\mathcal{R}\ \Gamma\ \Delta\ c_\tau \qquad\quad = \langle \bot_\tau ; \emptyset \rangle\\
\mathcal{R}\ \Gamma\ \Delta\ \lightning^{\ell}_\tau \qquad\quad = \langle \bot_\tau ; \{\ell\} \rangle\\
\mathcal{R}\ \Gamma\ \Delta\ (\lambda x : \tau . t) =\\
\quad \mathbf{let}\ \langle \widehat{\tau}_1 ; e_1 ; \overline{e_i :: \kappa_i} \rangle = \mathcal{C}\ \emptyset\ \tau\\
\qquad \langle \widehat{\tau}_2 ; \xi_2 \rangle \qquad = \mathcal{R}\ (\Gamma, x : \widehat{\tau}_1\ \&\ e_1)\ (\Delta, \overline{e_i :: \kappa_i})\ t\\
\quad \mathbf{in}\ \langle \forall \overline{e_i :: \kappa_i} . \widehat{\tau}_1 \langle e_1 \rangle \to \widehat{\tau}_2 \langle \xi_2 \rangle ; \emptyset \rangle\\
\mathcal{R}\ \Gamma\ \Delta\ (t_1\ t_2) \quad =\\
\quad \mathbf{let}\ \langle \widehat{\tau}_1 ; \xi_1 \rangle \qquad\qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t_1\\
\qquad \langle \widehat{\tau}_2 ; \xi_2 \rangle \qquad\qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t_2\\
\qquad \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 \theta \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}\ t) =\\
\quad \mathbf{let}\ \langle \widehat{\tau} ; \xi \rangle \qquad\qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t\\
\qquad \langle \widehat{\tau}' \langle e' \rangle \to \widehat{\tau}'' \langle \xi'' \rangle ; \overline{e_i :: \kappa_i} \rangle = \mathcal{I}\ \widehat{\tau}\\
\quad \mathbf{in}\ \langle \widehat{\tau}_0 ; \xi_0 ; i \rangle \qquad \leftarrow \langle \bot_{\lfloor \widehat{\tau}' \rfloor} ; \emptyset ; 0 \rangle\\
\qquad \mathbf{do}\ \theta \qquad\qquad \leftarrow [e' \mapsto \xi_i] \circ \mathcal{M}\ \emptyset\ \widehat{\tau}'\ \widehat{\tau}_i\\
\qquad\quad \langle \widehat{\tau}_{i+1} ; \xi_{i+1} ; i \rangle \leftarrow \langle \llbracket \theta \widehat{\tau}'' \rrbracket_\Delta ; \llbracket \theta \xi'' \rrbracket_\Delta ; i + 1 \rangle\\
\qquad \mathbf{until}\ \langle \widehat{\tau}_i ; \xi_i \rangle \equiv \langle \widehat{\tau}_{i-1} ; \xi_{i-1} \rangle\\
\qquad \mathbf{return}\ \langle \widehat{\tau}_i ; \llbracket \xi \cup \xi_i \rrbracket_\Delta \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 \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 \quad = \mathcal{R}\ \Gamma\ \Delta\ t_1\\
\qquad \langle \widehat{\tau}_2 ; \xi_2 \rangle \quad = \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 \langle \widehat{\tau}_2 ; \xi_2 \rangle \quad = \mathcal{R}\ \Gamma\ \Delta\ t_2\\
\qquad \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 = \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 = \mathcal{R}\ \Gamma\ \Delta\ t_1\\
\qquad \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 \qquad = \mathcal{R}\ \Gamma\ \Delta\ t_1\\
\qquad \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 \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

5.1 Polymorphic abstraction

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

In the case of abstractions, we first complete the type of the bound variable to an exception type using the auxiliary 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.

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

Figure 16. Exception type matching

The interesting cases of exception type matching are the cases for list and function types, where we perform pattern unification of the exception annotations. The produced substitution θ covers all variables 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 fixpoint construct abstracts over a variable that is of an exception polymorphic type. This case is handled in the inference algorithm by a Kleene–Mycroft iteration.[4]

Example 7 (Dussart, Henglein, Mossin)**.** Consider the term

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

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

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

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 more complicated than 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 $\bot_{\textbf{bool}\to\textbf{bool}\to\textbf{bool}}$:

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

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

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

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:

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

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.

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

Figure 17. Exception types: least upper bound

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 on 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. TO DO: Widening (Lemma 2)

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). These systems where 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 an 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-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 and 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 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 much less well studied, possibly because it is of limited use without the simultaneous introduction of effect operators, in contrast to higher-ranked polymorphism in plain type systems.

Effect operators Kennedy (1996a) presents a type system that ensures the dimensional consistency of a 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 arbitrary rank 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:

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

Without dimension operators we would have to specialize the type of the higher-order function *twice* to one that is either only application in *fourth*, or one that is only applicable in *sixteenth*.

The language $\Lambda_{\delta\omega}$ bears a striking resemblance to our 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, while 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 presented 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-bounded 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 very 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 scales 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) 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, primarily targeting the detection of uncaught exceptions in ML. The exception analysis in Yi (1994) is based on abstract interpretation. Guzmán and Suárez (1994) and 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 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—the program may still be dimensionally inconsistent. Unlike in this system, the annotations on function arguments no longer are of the simple form (patterns) required for the straightforward matching step in our 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? In previous work (Koot and Hage 2015) we argued 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 pattern-matches. The canonical example is the *risers* function—which splits a list into monotonically increasing subsegments; e.g., *risers* $[1, 3, 5, 1, 2] \longrightarrow [[1, 3, 5], [1, 2]]$—by Mitchell and Runciman (2008):

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

The inference algorithm in Figure 5 assigns *risers* the type

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

where **E** is the label of the exception raised when the pattern-match in the **where**-clause fails.[5] However, we pattern-match 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.

In our previous work we demonstrated how this exception can be elided by having the exception flow depend on the data flow. The $\lambda^\cup$-calculus terms that form our effect annotations cannot express this dependence, however. In the earlier work we used 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 type with exception set operators *à la* System F_ω, are not only more accurate, but are also more readable for the programmer *vis-à-vis* constrained type-schemes: the exception terms in the annotations more closely mirror what is happening at the term level 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.

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.

A. Metatheory

A.1 $\lambda^{\cup}$-calculus

Lemma 3. *The terms $(t_1 \cup t_2)\ t$ and $t_1\ t \cup t_2\ t$ are equivalent.*

Proof.

$$
\begin{aligned}
&[\![(t_1 \cup t_2)\ t]\!]_\rho\\
&= \bigcup \{\varphi([\![t]\!]_\rho) \mid \varphi \in [\![t_1 \cup t_2]\!]_\rho\}\\
&= \bigcup \{\varphi([\![t]\!]_\rho) \mid \varphi \in [\![t_1]\!]_\rho \cup [\![t_2]\!]_\rho\}\\
&= \bigcup \{\varphi([\![t]\!]_\rho) \mid \varphi \in [\![t_1]\!]_\rho\} \cup \bigcup \{\varphi([\![t]\!]_\rho) \mid \varphi \in [\![t_2]\!]_\rho\}\\
&= [\![t_1\ t]\!]_\rho \cup [\![t_2\ t]\!]_\rho\\
&= [\![(t_1\ t) \cup (t_2\ t)]\!]_\rho
\end{aligned}
$$

□

Lemma 4. *The terms $(\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2)$ and $\lambda x : \tau.t_1 \cup t_2$ are extensionally equivalent.*

Proof. We show that

$$[\![((\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2))\ t_3]\!]_\rho = [\![(\lambda x : \tau.t_1 \cup t_2)\ t_3]\!]_\rho$$

for all suitable ρ and t_3.

$$
\begin{aligned}
&[\![((\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2))\ t_3]\!]_\rho\\
&= \bigcup \{\varphi([\![t_3]\!]_\rho) \mid \varphi \in [\![(\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2)]\!]_\rho\}\\
&= \bigcup \{\varphi([\![t_3]\!]_\rho) \mid \varphi \in [\![\lambda x : \tau.t_1]\!]_\rho \cup [\![\lambda x : \tau.t_2]\!]_\rho\}\\
&= \bigcup \{\varphi([\![t_3]\!]_\rho) \mid \varphi \in \{\lambda v \in V_\tau.[\![t_i]\!]_{\rho[x\mapsto v]} \mid i \in \{1,2\}\}\}\\
&= \bigcup \left\{[\![t_1]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]}, [\![t_2]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]}\right\}\\
&= [\![t_1]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]} \cup [\![t_2]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]}\\
&= \bigcup \left\{[\![t_1]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]} \cup [\![t_2]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]}\right\}\\
&= \bigcup \left\{[\![t_1 \cup t_2]\!]_{\rho[x\mapsto[\![t_3]\!]_\rho]}\right\}\\
&= \bigcup \{\varphi([\![t_3]\!]_\rho) \mid \varphi \in \{\lambda v \in V_\tau.[\![t_1 \cup t_2]\!]_{\rho[x\mapsto v]}\}\}\\
&= \bigcup \{\varphi([\![t_3]\!]_\rho) \mid \varphi \in [\![\lambda x : \tau.t_1 \cup t_2]\!]_\rho\}\\
&= [\![(\lambda x : \tau.t_1 \cup t_2)\ t_3]\!]_\rho
\end{aligned}
$$

□

A.2 Declarative type system

Lemma 5 (Canonical forms)**.**

1. *If $\widehat{v}$ is a possibly exceptional value of type* $\widehat{\mathbf{bool}}$, *then $\widehat{v}$ is either* **true**, **false**, *or $\lightning^\ell$.*
2. *If $\widehat{v}$ is a possibly exceptional value of type* $\widehat{\mathbf{int}}$, *then $\widehat{v}$ is either some integer n, or an exceptional value $\lightning^\ell$.*
3. *If $\widehat{v}$ is a possibly exceptional value of type $[\widehat{\tau}\langle\xi\rangle]$ then $\widehat{v}$ is either $[]$, $t :: t'$, or $\lightning^\ell$.*
4. *If $\widehat{v}$ is a possibly exceptional value of type $\widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle$, then $\widehat{v}$ is either $\lambda x : \widehat{\tau}_1\ \&\ \xi_1.t'$ or $\lightning^\ell$.*
5. *If $\widehat{v}$ is a possibly exceptional value of type $\forall e :: \kappa.\widehat{\tau}$, then $\widehat{v}$ is $\Lambda e :: \kappa.t$*

Proof. For each part, inspect all forms of $\widehat{v}$ and discard the unwanted cases by inversion of the typing relation. Note that $\perp_\tau$ cannot give us a type of the form $\forall e :: \kappa.\widehat{\tau}$. □

TO DO.: Say something about T-SUB?

Theorem 1 (Progress)**.** *If $\Gamma; \Delta \vdash t : \widehat{\tau}\ \&\ \xi$ with t a closed term, then t is either a possibly exceptional value $\widehat{v}$ or there is a closed term t' such that $t \longrightarrow t'$.*

Proof. By induction on the typing derivation $\Gamma; \Delta \vdash t : \widehat{\tau}\ \&\ \xi$.

The case T-VAR can be discarded, as a variable is not a closed term. The cases T-CON, T-CRASH, T-ABS, T-ANNABS, T-NIL and T-CONS are immediate as they are values.

Case T-APP: We can immediately apply the induction hypothesis to $\Gamma; \Delta \vdash t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle\ \&\ \xi$, giving us either a t_1' such that $t_1 \longrightarrow t_1'$ or that $t_1 = \widehat{v}$. In the former case we can make progress using E-APP. In the latter case the canonical forms lemma tells us that either $t_1 = \lambda x : \widehat{\tau}_2\ \&\ \xi_2.t_1'$ or $t_1 = \lightning^\ell$, in which case we can make progress using E-APPABS or E-APPEXN, respectively.

The remaining cases follow by analogous reasoning. □

$$
\begin{aligned}
e[\xi/e] &\equiv \xi\\
e'[\xi/e] &\equiv e' && \text{if } e \neq e'\\
\{\ell\}[\xi/e] &\equiv \{\ell\}\\
\emptyset[\xi/e] &\equiv \emptyset\\
(\lambda e' : \kappa.\xi')\,[\xi/e] &\equiv \lambda e' : \kappa.\xi'[\xi/e] && \text{if } e \neq e' \text{ and } e' \notin \mathrm{fv}(\xi)\\
(e_1\ e_2)\,[\xi/e] &\equiv (e_1[\xi/e])\ \ (e_2[\xi/e])\\
(e_1 \cup e_2)\,[\xi/e] &\equiv e_1[\xi/e] \cup e_2[\xi/e]
\end{aligned}
$$

Figure 18. Annotation substitution

$$
\begin{aligned}
x[t/x] &\equiv t\\
x'[t/x] &\equiv x' && \text{if } x \neq x'\\
c_\tau[t/x] &\equiv c_\tau\\
(\lambda x' : \widehat{\tau}.t')\,[t/x] &\equiv \lambda x' : \widehat{\tau}.t'[t/x] && \text{if } x \neq x' \text{ and } x' \notin \mathrm{fv}(t)\\
&\ \ \dots
\end{aligned}
$$

Figure 19. Term substitution

Lemma 6 (Annotation substitution)**.**

1. *If $\Delta, e : \kappa' \vdash \xi : \kappa$ and $\Delta \vdash \xi' : \kappa'$ then $\Delta \vdash \xi[\xi'/e] : \kappa$.*
2. *If $\Delta, e : \kappa' \vdash \xi_1 \leqslant \xi_2$ and $\Delta \vdash \xi' :: \kappa'$ then $\Delta \vdash \xi_1[\xi'/e] \leqslant \xi_2[\xi'/e]$.*
3. *If $\Delta, e : \kappa' \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ and $\Delta \vdash \xi' :: \kappa'$ then $\Delta \vdash \widehat{\tau}_1[\xi'/e] \leqslant \widehat{\tau}_2[\xi'/e]$.*
4. *If $\Gamma; \Delta, e : \kappa' \vdash t : \widehat{\tau}\ \&\ \xi$ and $\Delta \vdash \xi' : \kappa'$ then $\Gamma; \Delta \vdash t[\xi'/e] : \widehat{\tau}[\xi'/e]\ \&\ \xi$.*

TO DO.: In part 4, either we need the assumption $e \notin \mathrm{fv}(\xi)$ (which seems to be satisfied everywhere we want to apply this lemma), or we also need to apply the substitution to ξ (is this expected or not in a type-and-effect system)? T-FIX seems to be to only rule where an exception variable can flow from $\widehat{\tau}$ to ξ....

Proof. 1. By induction on the derivation of $\Delta, e : \kappa' \vdash \xi : \kappa$. The cases A-VAR, A-ABS and A-APP are analogous to the respective cases in the proof of term substitution below. In the case A-CON one can strengthen the assumption $\Delta, e : \kappa' \vdash \{\ell\} : \mathrm{EXN}$ to $\Delta \vdash \{\ell\} : \mathrm{EXN}$ as $e \notin \mathrm{fv}(\{\ell\})$, the result is then immediate; similarly for A-EMPTY. The case A-UNION goes analogous to A-APP.

2. TO DO.

3. TO DO.

4. By induction on the derivation of $\Gamma; \Delta, e : \kappa' \vdash t : \widehat{\tau}\ \&\ \xi$. Most cases can be discarded by a straightforward application of the induction hypothesis; we show only the interesting case.

Case T-ANNAPP: TO DO.

TO DO.

□

Lemma 7 (Term substitution). *If* $\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi'; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$ *and* $\Gamma; \Delta \vdash t' : \widehat{\tau}' \mathbin{\&} \xi'$ *then* $\Gamma; \Delta \vdash t[t'/x] : \widehat{\tau} \mathbin{\&} \xi$.

Proof. By induction on the derivation of $\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$.

Case T-VAR: We either have $t = x$ or $t = x'$ with $x \neq x'$. In the first case we need to show that $\Gamma; \Delta \vdash x[t'/x] : \widehat{\tau} \mathbin{\&} \xi$, which by definition of substitution is equal to $\Gamma; \Delta \vdash x : \widehat{\tau} \mathbin{\&} \xi$, but this is one of our assumptions. In the second case we need to show that $\Gamma, x' : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash x'[t/x] : \widehat{\tau} \mathbin{\&} \xi$, which by definition of substitution is equal to $\Gamma, x' : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash x' : \widehat{\tau} \mathbin{\&} \xi$. This follows immediately from T-VAR.

Case T-ABS: Our assumptions are

$$\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi', y : \widehat{\tau}_1 \mathbin{\&} \xi_1; \Delta \vdash t : \widehat{\tau}_2 \mathbin{\&} \xi_2 \tag{1}$$
$$\Gamma; \Delta \vdash t' : \widehat{\tau}' \mathbin{\&} \xi'. \tag{2}$$

By the Barendregt convention we may assume that $y \neq x$ and $y \notin \mathrm{fv}(t')$. We need to show that $\Gamma; \Delta \vdash (\lambda y : \widehat{\tau}_1 \mathbin{\&} \xi_1.t)[t'/x] : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \emptyset$, which by definition of substitution is equal to

$$\Gamma; \Delta \vdash \lambda y : \widehat{\tau}_1 \mathbin{\&} \xi_1.t[t'/x] : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \emptyset. \tag{3}$$

We weaken (2) to $\Gamma, y : \widehat{\tau}_1 \mathbin{\&} \xi_1; \Delta \vdash t' : \widehat{\tau}' \mathbin{\&} \xi'$ and apply the induction hypothesis on this and (1) to obtain

$$\Gamma, y : \widehat{\tau}_1 \mathbin{\&} \xi_1; \Delta \vdash t[t'/x] : \widehat{\tau}_2 \mathbin{\&} \xi_2. \tag{4}$$

The desired result (3) can be constructed from (4) using T-ABS.

Case T-ANNABS: Our assumptions are $\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi'; \Delta, e : \kappa \vdash t : \widehat{\tau} \mathbin{\&} \xi$ and $\Gamma; \Delta \vdash t' : \widehat{\tau}' \mathbin{\&} \xi'$. By the Barendregt convention we may assume that $e \notin \mathrm{fv}(t')$. We need to show that $\Gamma; \Delta \vdash (\Lambda e :: \kappa.t)\,[t'/x] : \widehat{\tau} \mathbin{\&} \xi$, which is equal to $\Gamma; \Delta \vdash \Lambda e :: \kappa.t[t'/\kappa] : \widehat{\tau} \mathbin{\&} \xi$ by definition of substitution. By applying the induction hypothesis we obtain $\Gamma; \Delta, e : \kappa \vdash t[t'/x] : \widehat{\tau} \mathbin{\&} \xi$. The desired result can be constructed using T-ANNABS.

Case T-APP: Our assumptions are

$$\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi'; \Delta \vdash t_1 : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \mathbin{\&} \xi \tag{5}$$
$$\Gamma, x : \widehat{\tau}' \mathbin{\&} \xi'; \Delta \vdash t_2 : \widehat{\tau}_2 \mathbin{\&} \xi_2. \tag{6}$$

We need to show that $\Gamma; \Delta \vdash (t_1\ t_2)[t'/x] : \widehat{\tau} \mathbin{\&} \xi$, which by definition of substitution is equal to

$$\Gamma; \Delta \vdash \left(t_1[t'/x]\right)\ \left(t_2[t'/x]\right) : \widehat{\tau} \mathbin{\&} \xi. \tag{7}$$

By applying the induction hypothesis to (5) respectively (6) we obtain

$$\Gamma; \Delta \vdash t_1[t'/x] : \widehat{\tau}_2\langle\xi_2\rangle \to \widehat{\tau}\langle\xi\rangle \mathbin{\&} \xi \tag{8}$$
$$\Gamma; \Delta \vdash t_2[t'/x] : \widehat{\tau}_2 \mathbin{\&} \xi_2. \tag{9}$$

The desired result (7) can be constructed by applying T-APP to (8) and (9).

All other cases are either immediate or analogous to the case of T-APP. □

Lemma 8 (Inversion).

1. *If* $\Gamma; \Delta \vdash \lambda x : \widehat{\tau} \mathbin{\&} \xi.t : \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle \mathbin{\&} \xi_3$*, then*
   - $\Gamma, x : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash t : \widehat{\tau}' \mathbin{\&} \xi'$,
   - $\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}$ *and* $\Delta \vdash \xi_1 \leqslant \xi$,
   - $\Delta \vdash \widehat{\tau}' \leqslant \widehat{\tau}_2$ *and* $\Delta \vdash \xi' \leqslant \xi_2$.
2. *If* $\Gamma; \Delta \vdash \Lambda e :: \kappa.t : \forall e :: \kappa.\widehat{\tau} \mathbin{\&} \xi$*, then*
   - $\Gamma; \Delta, e : \kappa \vdash t : \widehat{\tau}' \mathbin{\&} \xi'$,
   - $\Delta, e : \kappa \vdash \widehat{\tau}' \leqslant \widehat{\tau}$,
   - $\Delta \vdash \xi' \leqslant \xi$.
   - TO DO. $e \notin \mathrm{fv}(\xi)$ *and/or* $e \notin \mathrm{fv}(\xi')$.

Proof. 1. By induction on the typing derivation.

Case T-ABS: We have $\widehat{\tau} = \widehat{\tau}_1$, $\xi = \xi_1$ and take $\widehat{\tau}' = \widehat{\tau}_2$, $\xi' = \xi_2$, the result then follows immediately from the assumption $\Gamma, x : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash t : \widehat{\tau}_2 \mathbin{\&} \xi_2$ and reflexivity of the subtyping and subeffecting relations.

Case T-SUB: We are given the additional assumptions

$$\Gamma; \Delta \vdash \lambda x : \widehat{\tau} \mathbin{\&} \xi.t : \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_2'\langle\xi_2'\rangle \mathbin{\&} \xi_3', \tag{10}$$
$$\Delta \vdash \widehat{\tau}_1'\langle\xi_1'\rangle \to \widehat{\tau}_2'\langle\xi_2'\rangle \leqslant \widehat{\tau}_1\langle\xi_1\rangle \to \widehat{\tau}_2\langle\xi_2\rangle, \tag{11}$$
$$\Delta \vdash \xi_3' \leqslant \xi_3. \tag{12}$$

Applying the induction hypothesis to (10) gives us

$$\Gamma, x : \widehat{\tau} \mathbin{\&} \xi; \Delta \vdash t : \widehat{\tau}_2'' \mathbin{\&} \xi_2'', \tag{13}$$
$$\Delta \vdash \widehat{\tau}_1' \leqslant \widehat{\tau}, \quad \Delta \vdash \xi_1' \leqslant \xi, \tag{14}$$
$$\Delta \vdash \widehat{\tau}_2'' \leqslant \widehat{\tau}_2', \quad \Delta \vdash \xi_2'' \leqslant \xi_2'. \tag{15}$$

Inversion of the subtyping relation on (11) gives us

$$\Delta \vdash \widehat{\tau}_1' \leqslant \widehat{\tau}, \quad \Delta \vdash \xi_1' \leqslant \xi, \tag{16}$$
$$\Delta \vdash \widehat{\tau}_2'' \leqslant \widehat{\tau}_2', \quad \Delta \vdash \xi_2'' \leqslant \xi_2'. \tag{17}$$

The result follows from (13) and combining (16) with (14) and (15) with (17) using the transitivity of the subtyping and subeffecting relations.

2. By induction on the typing derivation.

Case T-ANNABS: We need to show that $\Gamma; \Delta, e : \kappa \vdash t : \widehat{\tau} \mathbin{\&} \xi$, which is one of our assumptions, and that $\Delta, e : \kappa \vdash \widehat{\tau} \leqslant \widehat{\tau}$ and $\Delta \vdash \xi \leqslant \xi$; this follows from the reflexivity of the subtyping, respectively subeffecting, relation (noting that $e \notin \mathrm{fv}(\xi)$).

Case T-SUB: Similar to the case T-SUB in part 1. □

Theorem 2 (Preservation). *If* $\Gamma; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$ *and* $t \longrightarrow t'$*, then* $\Gamma; \Delta \vdash t' : \widehat{\tau} \mathbin{\&} \xi$.

Proof. By induction on the typing derivation $\Gamma; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$.

The cases for T-VAR, T-CON, T-CRASH, T-ABS, T-ANNABS, T-NIL, and T-CONS can be discarded immediately, as they have no applicable evaluation rules.

TO DO. □

A.3 Syntax-directed type elaboration

A.4 Type inference algorithm

Theorem 3 (Syntactic soundness). *If* $\mathcal{R}\ \Gamma\ \Delta\ t = \langle\widehat{\tau};\xi\rangle$*, then* $\Gamma; \Delta \vdash t : \widehat{\tau} \mathbin{\&} \xi$.

Proof. By induction on the term t.

TO DO. □

Theorem 4 (Termination). $\mathcal{R}\ \Gamma\ \Delta\ t$ *terminates.*

Proof. By induction on the term t.

TO DO. □

B. TODO

- standard polyrec examples (DHM + GSM)
- polyrec does not “come for free”
- type inference: we have a fixpoint a la DHM
- widening
- “algebraic” effects?
- unexpected decidablity
- ack: Vincent + Femke; Andrew + Jeremy + Stephanie + Andres; ST-RC
- check wiki and folder for notes
- exception type of twice (and other h-o funs)
- no slanted-greek for lambda and Lambda
- typeset System F_ω correctly
- re-enable more fields in bibliography, full first names
- Stefan’s terminology: fully parametric vs. fully flexible
- roll Metatheory into earlier sections, add new section Analysis (also add to Overview)
- Untracked exceptions can break information flow security.
- Elaborte in the subsection “Contributions”. Mention prototype?
- : vs ::

Abstract

- Decidability is only conjectured, so far.
- title: Higher-order effect types

B.1 Introduction

- Why not $\alpha\langle e_1\rangle \xrightarrow{e_3} \beta\langle e_2\rangle$?! Give some examples why higher-rankedness is needed. The example on the poster/*map* isn’t sufficient. Postpone to a later section?

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

- TO DO: Prove more Lemmas about reduction rules (esp. γ_1)
- Counterexample for the widening rule.
- TO DO: Rewrite subsection about Normalization
- Prove semantics is ACI1. We have a different unit for each type!
- $\mathcal{P}(V_{\tau_1} \to V_{\tau_2}) \simeq V_{\tau_1} \to \mathcal{P}(V_{\tau_2})$? Cardinallity suggests not: $2^{(\beta^\alpha)} \neq (2^\beta)^\alpha$.
- If we don’t distribute unions over applications, can we ever get them deep inside terms?
- If we don’t *and* the outermost lambdas are not there because is always of kind star, can we get non-trivial terms? I.e. something other than $e_1(e_{11},...,e_{1n_1})\cup\cdots\cup e_k(e_{k1},...,e_{kn_k}$ (note: e and not t as arguments).

B.3 Source language

- We either need to omit the type annotations on $\lightning^\ell_\tau$, or add them to **if then else** and **case of** $\{[]\mapsto;::\mapsto\}$.
- We do not have a rule E-ANNAPPEXN. Check that the canonical forms lemma gives us that terms of universally quantified type cannot be exceptional values.

B.4 Exception types

- Elaborate on well-formedness
- TO DO: Rename stuff in T-APP in the elaboration system (now subtype/-effect of the result instead of the argument and clashes with the indices enumerated over by i!
- TO DO: T-ANNABS: $e \notin \mathrm{fv}(\Gamma)$
- $e \in$ **ExnVar**
- Well-formedness of exception types: embed conservativity / full-flexibility?
- Can we roll UNIV and ARR into a single construct: $\forall e :: \kappa.\widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle\xi(e)\rangle$? Still need to deal with the well-formedness of $\widehat{\tau}_1$... Also may need to quantify over more than one variable simultaneously...

B.4.1 Subtyping

- Is S-REFL an admissable/derivable rule, or should we drop S-BOOL and S-INT?
- Possibly useful lemma: $\widehat{\tau}_1 = \widehat{\tau}_2 \iff \widehat{\tau}_1 \leqslant \widehat{\tau}_2 \wedge \widehat{\tau}_2 \leqslant \widehat{\tau}_1$.

B.4.2 Declarative type system

- In T-Abs and T-AnnAbs, should the term-level term-abstraction also have an explicit effect annotation?
- In T-AnnAbs, might need a side condition stating that e is not free in Δ.
- In T-App, note the double occurence of ξ when typing t_1. Is subeffecting sufficient here? Also note that we do *not* expect an exception variable in the left-hand side annotation of the function space constructor.
- In T-AnnApp, note the substitution. We need a substitution lemma for annotations.
- In T-Fix, the might be some universal quantifiers in our way. Do annotation applications in t take care of this, already? Perhaps we do need to change **fix** t into a binding construct to resolve this? Also, there is some implicit subeffecting going on between the annotations and effect.
- In T-Case, note the use of explicit subeffecting. Can this be done using implicit subeffecting?
- For T-Sub, should we introduce a term-level coercion, as in Dussart–Henglein–Mossin? We now do shape-conformant subtyping, is subeffecting sufficient?
- Do we need additional kinding judgements in some of the rules? Can we merge the kinding judgement with the subtyping and/or -effecting judgement? Kind-preserving substitutions.

B.4.3 Type elaboration system

- In T-APP and T-Fix, note that there are substitutions in the premises of the rules. Are these inductive? (Probably, as these premises are not “recursive” ones.)

B.5 Type inference

- Complexity: reduction corresponds to agressive constraint simplification
- alternative (faster?) version of Kleene-Mycroft
- In R-App and R-Fix: check that the fresh variables generated by $\mathcal{I}$ are subsituted away by the substitution θ created by $\mathcal{M}$. Also, we don’t need those variables in the algorithm if we don’t generate the elaborated term.
- In R-Fix we could get rid of the auxillary underlying type function if the fixpoint construct was replaced with a binding variant with an explicit type annotation.

- For R-Fix, make sure the way we handle fixpoints of exceptional value in a manner that is sound w.r.t. to the operational semantics we are going to give to this.
- Note that we do not construct the elaborated term, as it is not useful other than for metatheoretic purposes.
- Lemma: The algorithm maintains the invariant that exception types and exceptions are in normal form.

B.6 Related work

- More differences between (Holdermans and Hage 2010) (e.g. data types)?
- Christian Mossin. "Exact flow types" (intersection types, also non-elementary recursive by Statman)

B.7 Future research

- higher-ranked algebraic effect types, Koka