# Higher-ranked Exception Types

Ruud Koot

October 5, 2015

# Contents

# List of Figures

# Chapter 1

# Higher-ranked Exception Types

## 1.1 Motivation

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 of programmers. Many object-oriented programmers are quite surprised, then, that when they make the transition to a functional language, that they lose a feature their type system formerly did provide: tracking of uncaught exceptions.

There is a good 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 throw, 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 are 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 will thus end up being

*exception polymorphic*.

To do.concrete example?

**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 $\bot$, but we shall not do so for reasons of 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.

To do.concrete example?

Take as an example the *map* function:

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

For each type $\tau$, we denote its exception-annotated type by $\tau\langle\xi\rangle$. For function types we will 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 $\xi$ is the empty exception set, then we will omit it completely.

The fully exception-polymorphic and exception-annotated type, or *exception type*, of *map* isTo do.cramped formatting

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

To do.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?

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 will be one from the exception set $e_2\ e_1$. Here $e_2$ is an *exception operator*—a function that takes a number of exception sets and exception 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 itself may be an exceptional value from the exception set $e_3$.

The exception type of the second argument $[\alpha\langle e_4\rangle]\langle e_5\rangle$ states it should be a list. Any of the elements in the lists may be exception 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* will be 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 input list could come from. By looking at the definition of *map* we can see why this is the case: *map* will only produce non-exceptional constructors, but the pattern-match on the input list will propagate 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 will cause 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 input list—and so can only be applied to $f$ if we instantiate $e_1$ to $e_4$ first. If $f\ y$ will produce an exceptional value it will thus be on 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 $\beta\langle e_2\ e_4 \cup e_3\rangle$.

The get a better feeling of how these exception type and exception operators behave let us see what happens when we apply two different functions to *map*: the identity function *id* and the constant exceptional values $const\ \bot^{\mathbf{E}}$. These two functions can be given the exception types:

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

The term *id* simply propagates its input, so it will also propagate any exceptional values. The term $const\ \bot^{\mathbf{E}}$ discards it input and will always return the exceptional value $\bot^{\mathbf{E}}$. This behavior is also reflected in their exception types.

If 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 $(\lambda x.x)\ e_1 \rightsquigarrow e_1$. This gives us the resulting exception type

$$map\ id : \forall \alpha\ e_4\ e_5.[\alpha\langle e_4\rangle]\langle e_5\rangle \to [\alpha\langle e_4\rangle]\langle e_5\rangle$$

I.e., mapping the identity function over a list will propagate all existing exceptional values in the list and add no new ones.

If we apply *map* to *const* $\bot^{\mathbf{E}}$ we need to 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 $(\lambda x.\{\mathbf{E}\})\ e_1 \rightsquigarrow \{\mathbf{E}\}$. This gives us the resulting exception type

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

I.e., mapping the constant exceptional value over a list will discard all existing exceptional values from the list and only output non-exceptional values or the exceptional value $\bot^{\mathbf{E}}$ as elements of the lists.

### 1.1.1 Overview

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.

### 1.1.2 Contributions

– A *type system* than precisely tracks the uncaught exceptions using higher-ranked types.

– An *inference algorithm* that automatically infers such higher-ranked exception types.

**To do.**

– To do.Untracked exceptions can break information flow security.

## 1.2 The $\lambda^{\cup}$-calculus

– To do: If we don't distribute unions over applications, can we ever get them deep inside terms?

– To do: 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 arugments).

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

### 1.2.1 Typing relation

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

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

### 1.2.2 Semantics

– To do: $\mathcal{P}(V_{\tau_1} \to V_{\tau_2}) \simeq V_{\tau_1} \to \mathcal{P}(V_{\tau_2})$? Cardinallity suggests not: $2^{(\beta^{\alpha})} \neq (2^{\beta})^{\alpha}$.

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

**Lemma 1.** *The terms* $(\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2)$ *and* $\lambda x : \tau.t_1 \cup t_2$ *are extensionally equal.*

**Types**

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

**Terms**

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

**Environments**

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

Figure 1.1: $\lambda^{\cup}$-calculus: syntax

*Proof.* We prove 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 $\rho$ and $t_3$.

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

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

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

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

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

Figure 1.2: $\lambda^{\cup}$-calculus: type system

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

□

### 1.2.3 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 $\beta$-reduction rule of the simply typed $\lambda$-calculus with rewrite rules that deal with the associativity, commutativity, idempotence and identity (ACI1) properties of set-union operator.

**Types and values**

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

**Environments**

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

**Terms**

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

Figure 1.3: $\lambda^\cup$-calculus: denotational semantics

If a term $t$ is $\eta$-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 $\eta$-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 $\beta$-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 ter m $k_i(v_{i1}, ..., v_{iq_i})$ 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 $i \neq j$.

$$
\begin{array}{l}
\quad \text{-- normalization of terms} \\
[\![\cdot]\!] : \mathbf{Tm} \to \mathbf{Nf} \\
[\![\lambda x_1 \cdots x_n.T]\!] = \\
\quad \lambda x_1 \cdots x_n.\wr\{\, \lfloor f_i([\![t_{i1}]\!], ..., [\![t_{iq_i}]\!]) \rfloor \mid f_i(t_{i1}, ..., t_{iq_i}) \in T \,\}\wr \\
\quad \text{-- } \beta\text{-reduction} \\
\lfloor k(v_1, ..., v_q) \rfloor \\
\quad = k(v_1, ..., v_q) \\
\lfloor (\lambda y_1 \cdots y_q.T)\ (v\_1, \cdots, v\_q) \rfloor \\
\quad = SUBST\ x\ y\ z \\
\quad \text{-- set-rewriting} \\
\wr\{\cdots, k\_i\ (\cdots), \cdots, k\_j\ (\cdots), \cdots\}\wr \\
\quad \mid k\_j < k\_i = \wr\{\cdots, k\_i\ (\cdots), \cdots, k\_j\ (\cdots), \cdots\}\wr \\
\wr\{\cdots, k\ (\cdots), k\ (\cdots), \cdots\}\wr \\
\quad = \wr\{\cdots, k\ (\cdots), \cdots\}\wr \\
\wr T\wr \\
\quad = T
\end{array}
$$

Figure 1.4: Normalization algorithm for $\lambda^{\cup}$-terms.

## 1.3 Source language

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

[1] Technically, terms that bind at least one variable would form a $\beta$-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.

**Terms**

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

Figure 1.5: Source language: syntax

– To DO: Remove type annotation from $c_\tau$, make the operator a boolean one, and drop support for integers?

### 1.3.1 Underlying type system

### 1.3.2 Operational semantics

– The reduction relation is non-deterministic.

– We do not have a Haskell-style imprecise exception semantics (e.g. E-IF).

– 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.

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

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

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

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

Figure 1.6: Underlying type system ($\Gamma \vdash t : \tau$)

## 1.4 Exception types

### 1.4.1 Declarative exception 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 $\Delta$.
- In T-App, note the double occurence of $\varphi$ 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 will 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.

### 1.4.2 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.)
- For T-Fix: how would a binding fixpoint construct work?

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

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

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

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

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

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

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

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

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

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

Figure 1.7: Operational semantics ($t_1 \longrightarrow t_2$)

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

$$\frac{\Gamma, x : \widehat{\tau}_1 \;\&\; \varphi_1; \Delta \vdash t : \widehat{\tau}_2 \;\&\; \varphi_2}{\Gamma; \Delta \vdash \lambda x : \widehat{\tau}_1 \;\&\; \varphi_1 . t : \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \;\&\; \emptyset}\ [\text{T-Abs}] \qquad \frac{\Gamma; \Delta, e : \kappa \vdash t : \widehat{\tau} \;\&\; \varphi \quad e \notin \mathrm{fv}(\varphi)}{\Gamma; \Delta \vdash \Lambda e : \kappa . t : \forall e : \kappa . \widehat{\tau} \;\&\; \varphi}\ [\text{T-A}$$

$$\frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau}_2\langle\varphi_2\rangle \to \widehat{\tau}\langle\varphi\rangle \;\&\; \varphi \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau}_2 \;\&\; \varphi_2}{\Gamma; \Delta \vdash t_1\ t_2 : \widehat{\tau} \;\&\; \varphi}\ [\text{T-App}] \qquad \frac{\Gamma; \Delta \vdash t_1 : \forall e : \kappa . \widehat{\tau} \;\&\; \varphi \quad \Delta \vdash \varphi_2}{\Gamma; \Delta \vdash t_1\ \langle\varphi_2\rangle : \widehat{\tau}[\varphi_2/e] \;\&\; \varphi}$$

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

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

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

$$\frac{}{\Gamma; \Delta \vdash []_\tau : [\bot_\tau\langle\emptyset\rangle] \;\&\; \emptyset}\ [\text{T-Nil}] \qquad \frac{\Gamma; \Delta \vdash t_1 : \widehat{\tau} \;\&\; \varphi_1 \quad \Gamma; \Delta \vdash t_2 : [\widehat{\tau}\langle\varphi_1\rangle] \;\&\; \varphi_2}{\Gamma; \Delta \vdash t_1 :: t_2 : [\widehat{\tau}\langle\varphi_1\rangle] \;\&\; \varphi_2}\ [\text{T-Cons}]$$

$$\frac{\Delta \vdash \varphi' \leqslant \varphi \quad \Gamma; \Delta \vdash t_1 : [\widehat{\tau}_1\langle\varphi_1\rangle] \;\&\; \varphi' \quad \Gamma; \Delta \vdash t_2 : \widehat{\tau} \;\&\; \varphi \quad \Gamma, x_1 : \widehat{\tau}_1 \;\&\; \varphi_1, x_2 : [\widehat{\tau}_1\langle\varphi_1\rangle] \;\&\; \varphi'; \Delta}{\Gamma; \Delta \vdash \mathbf{case}\ t_1\ \mathbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} : \widehat{\tau} \;\&\; \varphi}$$

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

Figure 1.8: Declarative type system ($\Gamma; \Delta \vdash t : \widehat{\tau} \;\&\; \varphi$)

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

$$\frac{\Delta, \overline{e_i : \kappa_i} \vdash \widehat{\tau}_1 \triangleright \tau_1 \quad \Delta, \overline{e_i : \kappa_i} \vdash \varphi_1 : \textsc{exn} \quad \Gamma, x : \widehat{\tau}_1 \;\&\; \varphi_1; \Delta, \overline{e_i : \kappa_i} \vdash t \hookrightarrow t' : \widehat{\tau}_2 \;\&\; \varphi_2}{\Gamma; \Delta \vdash \lambda x : \tau_1.t \hookrightarrow \Lambda \overline{e_i : \kappa_i}.\lambda x : \widehat{\tau}_1 \;\&\; \varphi_1.t' : \forall \overline{e_i : \kappa_i}.\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \;\&\; \emptyset}\ [\text{T-Abs}]$$

$$\frac{\begin{array}{c}\Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}[\overline{\varphi_i}/\overline{e_i}] \quad \Delta \vdash \varphi_2 \leqslant \varphi[\overline{\varphi_i}/\overline{e_i}] \quad \overline{\Delta \vdash \varphi_i : \kappa_i} \\ \Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \forall \overline{e_i : \kappa_i}.\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}\langle\varphi\rangle \;\&\; \varphi' \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \varphi_2\end{array}}{\Gamma; \Delta \vdash t_1\ t_2 \hookrightarrow t_1'\ \langle\overline{\varphi_i}\rangle\ t_2' : \widehat{\tau}[\overline{\varphi_i}/\overline{e_i}] \;\&\; \varphi[\overline{\varphi_i}/\overline{e_i}] \cup \varphi'}\ [\text{T-App}]$$

$$\frac{\Gamma; \Delta \vdash t \hookrightarrow t' : \forall \overline{e_i : \kappa_i}.\widehat{\tau}\langle\varphi\rangle \to \widehat{\tau}'\langle\varphi'\rangle \;\&\; \varphi'' \quad \Delta \vdash \widehat{\tau}'[\overline{\varphi_i}/\overline{e_i}] \leqslant \widehat{\tau}[\overline{\varphi_i}/\overline{e_i}] \quad \Delta \vdash \varphi'[\overline{\varphi_i}/\overline{e_i}] \leqslant \varphi[\overline{\varphi_i}/\overline{e_i}] \quad \overline{\Delta}}{\Gamma; \Delta \vdash \textbf{fix}\ t \hookrightarrow \textbf{fix}\ t'\ \langle\overline{\varphi_i}\rangle : \widehat{\tau}[\overline{\varphi_i}/\overline{e_i}] \;\&\; \varphi[\overline{\varphi_i}/\overline{e_i}] \cup \varphi''}$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\textbf{int}} \;\&\; \varphi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\textbf{int}} \;\&\; \varphi_2}{\Gamma; \Delta \vdash t_1 \oplus t_2 \hookrightarrow t_1' \oplus t_2' : \widehat{\textbf{bool}} \;\&\; \varphi_1 \cup \varphi_2}\ [\text{T-Op}] \qquad \frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow}{\Gamma; \Delta \vdash t_1\ \textbf{seq}\ t_2 \hookrightarrow t_1'\ \textbf{seq}\ t_2' : \widehat{\tau}_2 \;\&$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\textbf{bool}} \;\&\; \varphi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \varphi_2 \quad \Gamma; \Delta \vdash t_3 \hookrightarrow t_3' : \widehat{\tau}_3 \;\&\; \varphi_3}{\Gamma; \Delta \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \hookrightarrow \textbf{if}\ t_1'\ \textbf{then}\ t_2'\ \textbf{else}\ t_3' : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \;\&\; \varphi_1 \cup \varphi_2 \cup \varphi_3}\ [\text{T-If}]$$

$$\frac{}{\Gamma; \Delta \vdash []_\tau \hookrightarrow []_\tau : \bot_\tau \;\&\; \emptyset}\ [\text{T-Nil}] \qquad \frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : \widehat{\tau}_1 \;\&\; \varphi_1 \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \left[\widehat{\tau}_1'\langle\varphi_1'\rangle\right] \;\&\; \varphi_2}{\Gamma; \Delta \vdash t_1 :: t_2 \hookrightarrow t_1' :: t_2' : \left[\widehat{\tau}_1 \sqcup \widehat{\tau}_1'\langle\varphi_1 \cup \varphi_1'\rangle\right] \;\&\; \varphi_2}\ [\text{T-Co}$$

$$\frac{\Gamma; \Delta \vdash t_1 \hookrightarrow t_1' : [\tau_1\langle\varphi_1\rangle] \;\&\; \varphi_1' \quad \Gamma; \Delta \vdash t_2 \hookrightarrow t_2' : \widehat{\tau}_2 \;\&\; \varphi_2 \quad \Gamma, x_1 : \widehat{\tau}_1 \;\&\; \varphi_1, x_2 : [\tau_1\langle\varphi_1\rangle] \;\&\; \varphi_1'; \Delta \vdash t_3}{\Gamma; \Delta \vdash \textbf{case}\ t_1\ \textbf{of}\ \{[] \mapsto t_2; x_1 :: x_2 \mapsto t_3\} \hookrightarrow \textbf{case}\ t_1'\ \textbf{of}\ \{[] \mapsto t_2'; x_1 :: x_2 \mapsto t_3'\} : \widehat{\tau}_2 \sqcup \widehat{\tau}_3 \;\&\; \varphi_1' \cup$$

Figure 1.9: Syntax-directed type elaboration system ($\Gamma; \Delta \vdash t \hookrightarrow t' : \widehat{\tau} \;\&\; \varphi$)

## 1.5 Completion

### 1.5.1 Exception types

- To DO.$e \in$ **ExnVar**
- To DO.Well-formedness of exception types: embed conservativity / full-flexibility?
- To DO.Can we roll UNIV and ARR into a single construct: $\forall e : \kappa.\widehat{\tau}_1\langle e\rangle \to \widehat{\tau}_2\langle\varphi(e)\rangle$? Still need to deal with the well-formedness of $\widehat{\tau}_1$... Also may need to quantify over more than one variable simultaneously...

The syntax of well-formed exception types are given in Figure 1.10 and Figure 1.11. 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 $[\widehat{\tau}\langle\varphi\rangle]$ and effect $\psi$, the type $\widehat{\tau}$ of the elements in the list is *annotated* with an exception set expression $\varphi$ of kind EXN. This expression gives a set of exceptions which may be raised when an element of the list is forced. The effect gives a set of exceptions may be raised when a constructor forming the spine of the list is forced.

For a function with exception type $\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle$ and effect $\psi$, the argument of type $\widehat{\tau}_1$ is annotated with an exception set expression $\varphi_1$ 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 $\varphi_2$ that gives the set of exceptions that may be raised when the result of the function is forced. The effect $\psi$ gives the set of exceptions that may be raised if the function closure is forced.

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

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

The exception set expressions $\varphi$ and their kinds $\kappa$ are an instance of the $\lambda^{\cup}$-calculus, where exception set expressions are terms and kinds are the

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

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.

– To do.Some more complicated examples

$$
\begin{array}{rcll}
\kappa \in \textbf{Kind} & ::= & \textsc{exn} & \text{(exception set)} \\
& | & \kappa_1 \Rightarrow \kappa_2 & \text{(exception operator)} \\
\\
\varphi, \psi \in \textbf{Exn} & ::= & e & \text{(exception variables)} \\
& | & \lambda e : \kappa.\varphi & \text{(exception abstraction)} \\
& | & \varphi_1\ \varphi_2 & \text{(exception application)} \\
& | & \emptyset & \text{(empty exception set)} \\
& | & \{\ell\} & \text{(singleton exception)} \\
& | & \varphi_1 \cup \varphi_2 & \text{(exception set union)} \\
\\
\widehat{\tau} \in \textbf{ExnTy} & ::= & \forall e :: \kappa.\widehat{\tau} & \text{(exception quantification)} \\
& | & \widehat{\text{bool}} & \text{(boolean type)} \\
& | & [\widehat{\tau}\langle\varphi\rangle] & \text{(list type)} \\
& | & \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle & \text{(function type)}
\end{array}
$$

Figure 1.10: Exception types: syntax

### 1.5.2 Conservativeness

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 in its behavior.

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

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

$$\frac{\Delta \vdash \widehat{\tau}_1\ \text{wff} \quad \Delta \vdash \varphi_1 : \text{exn} \quad \Delta \vdash \widehat{\tau}_2\ \text{wff} \quad \Delta \vdash \varphi_2 : \text{exn}}{\Delta \vdash \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle\ \text{wff}}\ [\text{W-Arr}]$$

Figure 1.11: Exception types: well-formedness

If we furthermore want out type system to be modular—allowing type checking and inference to work on indivildual 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 known 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 1.**

$$\mathit{tail} : \forall e_1\ e_2.\ [\widehat{\mathbf{bool}}\langle e_1\rangle]\ \langle e_2\rangle \to [\widehat{\mathbf{bool}}\langle e_1\rangle]\ \langle e_2 \cup \mathit{EMPTY}\rangle$$

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

[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 will be 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 2.** *Consider the higher-order function apply that applies its first argument to the second.*

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

*The first, functional, argument of apply has exception type* $\forall e_1 :: \textsc{exn}.\widehat{\mathbf{boo}}\mathbf{l}\langle e_1\rangle \to \widehat{\mathbf{boo}}\mathbf{l}\langle e_3\ e_1\rangle$ *and effect* $e_2$*. It can be instantiated with any function that accepts an argument annotated with any exception set effect, and produces a result annotated with some exception set effect depending on the exception set effect of the argument; the function closure itself may raise any exception. All functions of underling 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* $e_2 \cup e_3\ e_4$ *in the result. The exception set operator* $e_3$ *is now applied to* $e_4$*, as the application* $f\ x$ *will instantiate the quantified exception set variable* $e_1$ *to* $e_4$*.*

### 1.5.3 Exception type completion

During exception type inference will want to compute the least constraint exception type that erases to a given underlying type.

### 1.5.4 Type inference algorithm

– In R-App and R-Fix: check that the fresh variables generated by $\mathcal{I}$ are subsituted away by the substitution $\theta$ created by $\mathcal{M}$. Also, we don't need

$$\frac{}{\overline{e_i :: \kappa_i} \vdash \mathbf{bool} : \widehat{\mathrm{bool}} \;\&\; e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \mathbf{EXN}}\ [\text{C-Bool}] \qquad \frac{\overline{e_i :: \kappa_i} \vdash \tau : \widehat{\tau} \;\&\; \varphi \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash [\tau] : [\widehat{\tau}\langle\varphi\rangle] \;\&\; e\ \overline{e_i} \triangleright e :: \overline{\kappa_i} \Rightarrow \mathbf{EXN}}$$

$$\frac{\vdash \tau_1 : \widehat{\tau}_1 \;\&\; \varphi_1 \triangleright \overline{e_j :: \kappa_j} \quad \overline{e_i :: \kappa_i}, \overline{e_j :: \kappa_j} \vdash \tau_2 : \widehat{\tau}_2 \;\&\; \varphi_2 \triangleright \overline{e_j :: \kappa_j}}{\overline{e_i :: \kappa_i} \vdash \tau_1 \to \tau_2 : \forall \overline{e_j :: \kappa_j}.\, (\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle) \;\&\; e\ \overline{e_i} \triangleright e :: \overline{\kappa_j} \Rightarrow \mathbf{EXN}, \overline{e_k :: \kappa_k}}\ [\text{C-Arr}]$$

Figure 1.12: Type completion ($\Gamma \vdash \tau : \widehat{\tau} \;\&\; \varphi \triangleright \Gamma'$)

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.

### 1.5.5 Subtyping

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

## 1.6 Interesting observations

- Exception types are not invariant under $\eta$-reduction.

## 1.7 Metatheory

### 1.7.1 Declarative type system

**Lemma 2** (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 $[\widehat{\tau}\langle\varphi\rangle]$, then $\widehat{v}$ is either $[]$, $t :: t'$, or $\lightning^{\ell}$.*
4. *If $\widehat{v}$ is a possibly exceptional value of type $\widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle$, then $\widehat{v}$ is either $\lambda x : \widehat{\tau}_1 \;\&\; \varphi_1.t'$ 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 $\bot_\tau$ cannot give us a type of the form $\forall e : \kappa.\widehat{\tau}$. □

To do.: Say something about T-Sub?

**Theorem 1** (Progress)**.** *If $\Gamma; \Delta \vdash t : \widehat{\tau} \;\&\; \varphi$ 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 $\Gamma; \Delta \vdash t : \widehat{\tau} \;\&\; \varphi$.

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 $\Gamma; \Delta \vdash t_1 : \widehat{\tau}_2\langle\varphi_2\rangle \to \widehat{\tau}\langle\varphi\rangle \;\&\; \varphi$, 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 $t_1 = \lambda x : \widehat{\tau}_2 \;\&\; \varphi_2.t_1'$ 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. □

**Lemma 3** (Annotation substitution)**.**

1. *If* $\Delta, e : \kappa' \vdash \varphi : \kappa$ *and* $\Delta \vdash \varphi' : \kappa'$ *then* $\Delta \vdash \varphi[\varphi'/e] : \kappa$.
2. *If* $\Delta, e : \kappa' \vdash \varphi_1 \leqslant \varphi_2$ *and* $\Delta \vdash \varphi' : \kappa'$ *then* $\Delta \vdash \varphi_1[\varphi'/e] \leqslant \varphi_2[\varphi'/e]$.
3. *If* $\Delta, e : \kappa' \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}_2$ *and* $\Delta \vdash \varphi' : \kappa'$ *then* $\Delta \vdash \widehat{\tau}_1[\varphi'/e] \leqslant \widehat{\tau}_2[\varphi'/e]$.
4. *If* $\Gamma; \Delta, e : \kappa' \vdash t : \widehat{\tau}$ & $\varphi$ *and* $\Delta \vdash \varphi' : \kappa'$ *then* $\Gamma; \Delta \vdash t[\varphi'/e] : \widehat{\tau}[\varphi'/e]$ & $\varphi$.

To do.: In part 4, either we need the assumption $e \notin \mathrm{fv}(\varphi)$ (which seems to be satisfied everywhere we want to apply this lemma), or we also need to apply the substitution to $\varphi$ (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 $\varphi$....

*Proof.* 1. By induction on the derivation of $\Delta, e : \kappa' \vdash \varphi : \kappa$. 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 $\Delta, e : \kappa' \vdash \{\ell\} : \textsc{exn}$ to $\Delta \vdash \{\ell\} : \textsc{exn}$ 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 $\Gamma; \Delta, e : \kappa' \vdash t : \widehat{\tau}$ & $\varphi$. 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 4** (Term substitution)**.** *If* $\Gamma, x : \widehat{\tau}'$ & $\varphi'; \Delta \vdash t : \widehat{\tau}$ & $\varphi$ *and* $\Gamma; \Delta \vdash t' : \widehat{\tau}'$ & $\varphi'$ *then* $\Gamma; \Delta \vdash t[t'/x] : \widehat{\tau}$ & $\varphi$.

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

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

Case T-Abs: Our assumptions are

$$\Gamma, x : \widehat{\tau}' \;\&\; \varphi', y : \widehat{\tau}_1 \;\&\; \varphi_1; \Delta \vdash t : \widehat{\tau}_2 \;\&\; \varphi_2 \tag{1.1}$$

$$\Gamma; \Delta \vdash t' : \widehat{\tau}' \;\&\; \varphi'. \tag{1.2}$$

By the Barendregt convention we may assume that $y \neq x$ and $y \notin \mathrm{fv}(t')$. We need to show that $\Gamma; \Delta \vdash (\lambda y : \widehat{\tau}_1 \;\&\; \varphi_1.t)[t'/x] : \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \;\&\; \emptyset$, which by definition of substitution is equal to

$$\Gamma; \Delta \vdash \lambda y : \widehat{\tau}_1 \;\&\; \varphi_1.t[t'/x] : \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \;\&\; \emptyset. \tag{1.3}$$

We weaken (1.2) to $\Gamma, y : \widehat{\tau}_1 \;\&\; \varphi_1; \Delta \vdash t' : \widehat{\tau}' \;\&\; \varphi'$ and apply the induction hypothesis on this and (1.1) to obtain

$$\Gamma, y : \widehat{\tau}_1 \;\&\; \varphi_1; \Delta \vdash t[t'/x] : \widehat{\tau}_2 \;\&\; \varphi_2. \tag{1.4}$$

The desired result (1.3) can be constructed from (1.4) using T-Abs.

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

Case T-App: Our assumptions are

$$\Gamma, x : \widehat{\tau}' \;\&\; \varphi'; \Delta \vdash t_1 : \widehat{\tau}_2\langle\varphi_2\rangle \to \widehat{\tau}\langle\varphi\rangle \;\&\; \varphi \tag{1.5}$$

$$\Gamma, x : \widehat{\tau}' \;\&\; \varphi'; \Delta \vdash t_2 : \widehat{\tau}_2 \;\&\; \varphi_2. \tag{1.6}$$

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

$$\Gamma; \Delta \vdash \big(t_1[t'/x]\big)\ \ \big(t_2[t'/x]\big) : \widehat{\tau} \;\&\; \varphi. \tag{1.7}$$

By applying the induction hypothesis to (1.5) respectively (1.6) we obtain

$$\Gamma; \Delta \vdash t_1[t'/x] : \widehat{\tau}_2\langle\varphi_2\rangle \to \widehat{\tau}\langle\varphi\rangle \;\&\; \varphi \tag{1.8}$$

$$\Gamma; \Delta \vdash t_2[t'/x] : \widehat{\tau}_2 \;\&\; \varphi_2. \tag{1.9}$$

The desired result (1.7) can be constructed by applying T-App to (1.8) and (1.9).

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

**Lemma 5** (Inversion)**.**

1. *If* $\Gamma; \Delta \vdash \lambda x : \widehat{\tau} \;\&\; \varphi.t : \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \;\&\; \varphi_3$*, then*
   - $\Gamma, x : \widehat{\tau} \;\&\; \varphi; \Delta \vdash t : \widehat{\tau}' \;\&\; \varphi'$,
   - $\Delta \vdash \widehat{\tau}_1 \leqslant \widehat{\tau}$ *and* $\Delta \vdash \varphi_1 \leqslant \varphi$,
   - $\Delta \vdash \widehat{\tau}' \leqslant \widehat{\tau}_2$ *and* $\Delta \vdash \varphi' \leqslant \varphi_2$.
2. *If* $\Gamma; \Delta \vdash \Lambda e : \kappa.t : \forall e : \kappa.\widehat{\tau} \;\&\; \varphi$*, then*
   - $\Gamma; \Delta, e : \kappa \vdash t : \widehat{\tau}' \;\&\; \varphi'$,
   - $\Delta, e : \kappa \vdash \widehat{\tau}' \leqslant \widehat{\tau}$,
   - $\Delta \vdash \varphi' \leqslant \varphi$.
   - To do. $e \notin \mathrm{fv}(\varphi)$ *and/or* $e \notin \mathrm{fv}(\varphi')$.

*Proof.* 1. By induction on the typing derivation.

Case T-Abs: We have $\widehat{\tau} = \widehat{\tau}_1$, $\varphi = \varphi_1$ and take $\widehat{\tau}' = \widehat{\tau}_2$, $\varphi' = \varphi_2$, the result then follows immediately from the assumption $\Gamma, x : \widehat{\tau} \;\&\; \varphi; \Delta \vdash t : \widehat{\tau}_2 \;\&\; \varphi_2$ and reflexivity of the subtyping and subeffecting relations.

Case T-Sub: We are given the additional assumptions

$$\Gamma; \Delta \vdash \lambda x : \widehat{\tau} \;\&\; \varphi.t : \widehat{\tau}'_1\langle\varphi'_1\rangle \to \widehat{\tau}'_2\langle\varphi'_2\rangle \;\&\; \varphi'_3, \tag{1.10}$$
$$\Delta \vdash \widehat{\tau}'_1\langle\varphi'_1\rangle \to \widehat{\tau}'_2\langle\varphi'_2\rangle \leqslant \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle, \tag{1.11}$$
$$\Delta \vdash \varphi'_3 \leqslant \varphi_3. \tag{1.12}$$

Applying the induction hypothesis to (1.10) gives us

$$\Gamma, x : \widehat{\tau} \;\&\; \varphi; \Delta \vdash t : \widehat{\tau}''_2 \;\&\; \varphi''_2, \tag{1.13}$$
$$\Delta \vdash \widehat{\tau}'_1 \leqslant \widehat{\tau}, \quad \Delta \vdash \varphi'_1 \leqslant \varphi, \tag{1.14}$$
$$\Delta \vdash \widehat{\tau}''_2 \leqslant \widehat{\tau}'_2, \quad \Delta \vdash \varphi''_2 \leqslant \varphi'_2. \tag{1.15}$$

Inversion of the subtyping relation on (1.11) gives us

$$\Delta \vdash \widehat{\tau}'_1 \leqslant \widehat{\tau}, \quad \Delta \vdash \varphi'_1 \leqslant \varphi, \tag{1.16}$$
$$\Delta \vdash \widehat{\tau}''_2 \leqslant \widehat{\tau}'_2, \quad \Delta \vdash \varphi''_2 \leqslant \varphi'_2. \tag{1.17}$$

The result follows from (1.13) and combining (1.16) with (1.14) and (1.15) with (1.17) using the transitivity of the subtyping and subeffecting relations.

2. By induction on the typing derivation.

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

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

□

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

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

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. □

### 1.7.2 Syntax-directed type elaboration

### 1.7.3 Type inference algorithm

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

*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. □

$$
\begin{array}{l}
\mathcal{R} : \mathbf{TyEnv} \times \mathbf{KiEnv} \times \mathbf{Tm} \to \mathbf{ExnTy} \times \mathbf{Exn} \\
\mathcal{R}\ \Gamma\ \Delta\ x \qquad\qquad = \Gamma_x \\
\mathcal{R}\ \Gamma\ \Delta\ c_\tau \qquad\qquad = \langle \bot_\tau ; \emptyset \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ \lightning^\ell_\tau \qquad\qquad = \langle \bot_\tau ; \{\ell\} \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (\lambda x : \tau . t) = \\
\quad \textbf{let}\ \langle \widehat{\tau}_1 ; e_1 ; \overline{e_i : \kappa_i} \rangle = \mathcal{C}\ \emptyset\ \tau \\
\qquad \langle \widehat{\tau}_2 ; \varphi_2 \rangle \qquad\qquad = \mathcal{R}\ (\Gamma, x : \widehat{\tau}_1\ \&\ e_1)\ (\Delta, \overline{e_i : \kappa_i})\ t \\
\quad \textbf{in}\ \langle \forall \overline{e_i : \kappa_i} . \widehat{\tau}_1 \langle e_1 \rangle \to \widehat{\tau}_2 \langle \varphi_2 \rangle ; \emptyset \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (t_1\ t_2) \quad = \\
\quad \textbf{let}\ \langle \widehat{\tau}_1 ; \varphi_1 \rangle \qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\tau}_2 ; \varphi_2 \rangle \qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\qquad \langle \widehat{\tau}'_2 \langle e'_2 \rangle \to \widehat{\tau}' \langle \varphi' \rangle ; \overline{e_i : \kappa_i} \rangle = \mathcal{I}\ \widehat{\tau}_1 \\
\qquad \theta \qquad\qquad = [e'_2 \mapsto \varphi_2] \circ \mathcal{M}\ \emptyset\ \widehat{\tau}_2\ \widehat{\tau}'_2 \\
\quad \textbf{in}\ \langle [\![ \theta \widehat{\tau}' ]\!]_\Delta ; [\![ \theta \varphi' \cup \varphi_1 ]\!]_\Delta \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (\textbf{fix}\ t) = \\
\quad \textbf{let}\ \langle \widehat{\tau} ; \varphi \rangle \qquad\qquad = \mathcal{R}\ \Gamma\ \Delta\ t \\
\qquad \langle \widehat{\tau}' \langle e' \rangle \to \widehat{\tau}'' \langle \varphi'' \rangle ; \overline{e_i : \kappa_i} \rangle = \mathcal{I}\ \widehat{\tau} \\
\quad \textbf{in}\ \langle \widehat{\tau}_0 ; \varphi_0 ; i \rangle \qquad \leftarrow \langle \bot_{\lfloor \widehat{\tau}' \rfloor} ; \emptyset ; 0 \rangle \\
\qquad \textbf{do}\ \theta \qquad\qquad \leftarrow [e' \mapsto \varphi_i] \circ \mathcal{M}\ \emptyset\ \widehat{\tau}_i\ \widehat{\tau}' \\
\qquad\quad \langle \widehat{\tau}_{i+1} ; \varphi_{i+1} ; i \rangle \leftarrow \langle [\![ \theta \widehat{\tau}'' ]\!]_\Delta ; [\![ \theta \varphi'' ]\!]_\Delta ; i + 1 \rangle \\
\qquad \textbf{until}\ \langle \widehat{\tau}_i ; \varphi_i \rangle \equiv \langle \widehat{\tau}_{i-1} ; \varphi_{i-1} \rangle \\
\qquad \textbf{return}\ \langle \widehat{\tau}_i ; [\![ \varphi \cup \varphi_i ]\!]_\Delta \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (t_1 \oplus t_2) = \\
\quad \textbf{let}\ \langle \widehat{\textbf{int}} ; \varphi_1 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\textbf{int}} ; \varphi_2 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\quad \textbf{in}\ \langle \widehat{\textbf{bool}} ; [\![ \varphi_1 \cup \varphi_2 ]\!]_\Delta \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (t_1\ \textbf{seq}\ t_2) = \\
\quad \textbf{let}\ \langle \widehat{\tau}_1 ; \varphi_1 \rangle \quad = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\tau}_2 ; \varphi_2 \rangle \quad = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\quad \textbf{in}\ \langle \widehat{\tau}_2 ; [\![ \varphi_1 \cup \varphi_2 ]\!]_\Delta \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3) = \\
\quad \textbf{let}\ \langle \widehat{\textbf{bool}} ; \varphi_1 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\tau}_2 ; \varphi_2 \rangle \quad = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\qquad \langle \widehat{\tau}_3 ; \varphi_3 \rangle \quad = \mathcal{R}\ \Gamma\ \Delta\ t_3 \\
\quad \textbf{in}\ \langle [\![ \widehat{\tau}_2 \sqcup \widehat{\tau}_3 ]\!]_\Delta ; [\![ \varphi_1 \cup \varphi_2 \cup \varphi_3 ]\!]_\Delta \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ [\,]_\tau \qquad\qquad = \langle [\bot_\tau \langle \emptyset \rangle] ; \emptyset \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (t_1 :: t_2) = \\
\quad \textbf{let}\ \langle \widehat{\tau}_1 ; \varphi_1 \rangle \qquad = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle [\widehat{\tau}_2 \langle \varphi'_2 \rangle] ; \varphi_2 \rangle = \mathcal{R}\ \Gamma\ \Delta\ t_2 \\
\quad \textbf{in}\ \langle [\![ [(\widehat{\tau}_1 \sqcup \widehat{\tau}_2) \langle \varphi_1 \cup \varphi'_2 \rangle] ]\!]_\Delta ; \varphi_2 \rangle \\
\mathcal{R}\ \Gamma\ \Delta\ (\textbf{case}\ t_1\ \textbf{of}\ \{[\,] \mapsto t_2 ; x_1 :: x_2 \mapsto t_3\}) = \\
\quad \textbf{let}\ \langle [\widehat{\tau}_1 \langle \varphi'_1 \rangle] ; \varphi_1 \rangle \qquad = \mathcal{R}\ \Gamma\ \Delta\ t_1 \\
\qquad \langle \widehat{\tau}_2 ; \varphi_2 \rangle = \mathcal{R}\ (\Gamma, x_1 : \widehat{\tau}_1\ \&\ \varphi'_1, x_2 : [\widehat{\tau}_1 \langle \varphi'_1 \rangle]\ \&\ \varphi_1)\ \Delta\ t_2
\end{array}
$$

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

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

$$\frac{\begin{array}{c}\Delta \vdash \widehat{\tau}_1' \leqslant \widehat{\tau}_1 \quad \Delta \vdash \varphi_1' \leqslant \varphi_1 \\ \Delta \vdash \widehat{\tau}_2 \leqslant \widehat{\tau}_2' \quad \Delta \vdash \varphi_2 \leqslant \varphi_2'\end{array}}{\Delta \vdash \widehat{\tau}_1\langle\varphi_1\rangle \to \widehat{\tau}_2\langle\varphi_2\rangle \leqslant \widehat{\tau}_1'\langle\varphi_1'\rangle \to \widehat{\tau}_2'\langle\varphi_2'\rangle}\ [\text{S-Arr}]$$

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

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

Figure 1.14: Subtyping

$$\begin{array}{rll}
e[\varphi/e] & \equiv \varphi & \\
e'[\varphi/e] & \equiv e' & \text{if } e \neq e' \\
\{\ell\}[\varphi/e] & \equiv \{\ell\} & \\
\varnothing[\varphi/e] & \equiv \varnothing & \\
\left(\lambda e' : \kappa.\varphi'\right)[\varphi/e] & \equiv \lambda e' : \kappa.\varphi'[\varphi/e] & \text{if } e \neq e' \text{ and } e' \notin \mathrm{fv}(\varphi) \\
(e_1\ e_2)[\varphi/e] & \equiv (e_1[\varphi/e])\ (e_2[\varphi/e]) & \\
(e_1 \cup e_2)[\varphi/e] & \equiv e_1[\varphi/e] \cup e_2[\varphi/e] &
\end{array}$$

Figure 1.15: Annotation substitution

$$
\begin{array}{rll}
x[t/x] & \equiv t & \\
x'[t/x] & \equiv x' & \text{if } x \neq x' \\
c_\tau[t/x] & \equiv c_\tau & \\
\left(\lambda x' : \widehat{\tau}.t'\right)[t/x] & \equiv \lambda x' : \widehat{\tau}.t'[t/x] & \text{if } x \neq x' \text{ and } x' \notin \mathrm{fv}(t) \\
& \cdots &
\end{array}
$$

Figure 1.16: Term substitution

# Appendix A

# Old stuff

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

**Values** Values $v$ are terms of the form

$$\lambda x_1 : \tau_1 . \cdots \lambda x_i : \tau_i . \{c_1\} \cup (\cdots \cup (\{c_j\} \cup (x_1\ v_{11} \cdots v_{1m} \cup (\cdots \cup x_k\ v_{k1} \cdots v_{kn}))))$$

### A.1.1 Reduction relation (wrong!)

- To do. Do not match the rules in the prototype (those are sensitive to the order in which they are tried).
- To do. In the second rule only one term is applied; contrast this with the other rules involing applications.
- To do. Should make use of the fact the everything is fully applied (and $\eta$-expanded/-long?): all atoms are of the form $k\ \overline{t_i}$, where $k$ is $c$ or $x$ and the number of arguments fixed by the arity of $k$. Then try to factor out the commutativity rules by taking "sets" of these atoms. That might simplify stuff a whole lot...
- To do. Can we restrict the typing rule T-Union to only allow sets and not functions on both sides? This would remove the 2nd and 3rd rewrite rules and make the system a more traditional higher-order rewrite system: it's "just" higher-order pattern E-unification (decidable), boolean

rings are easy to integrate, and higher-ranked dimension types becomes higher-order E-unification (semi-decidable). Open question: how to represent e.g. $U(e_2(e_1), e_1) = [e_2 \mapsto \lambda e_1.e_1]$ without abstractions? (Reinterpret $e_1$ as $f(e_1)$ with $f = id$?)

**Definition 1.** *Let $\prec$ be a strict total order on* $\mathbf{Con} \cup \mathbf{Var}$*, with $c \prec x$ for all $c \in$* **Con** *and $x \in$* **Var***.*

$$
\begin{aligned}
(\lambda x : \tau.t_1)\ t_2 &\longrightarrow t_1[t_2/x] && (\beta\text{-reduction})\\
(t_1 \cup t_2)\ t_3 &\longrightarrow t_1\ t_3 \cup t_2\ t_3 &&\\
(\lambda x : \tau.t_1) \cup (\lambda x : \tau.t_2) &\longrightarrow \lambda x : \tau.\,(t_1 \cup t_2) && (\text{congruences})\\
x\ t_1 \cdots t_n \cup x\ t'_1 \cdots t'_n &\longrightarrow x\ \left(t_1 \cup t'_1\right) \cdots \left(t_n \cup t'_n\right) &&\\
(t_1 \cup t_2) \cup t_3 &\longrightarrow t_1 \cup (t_2 \cup t_3) && (\text{associativity})\\
\emptyset \cup t &\longrightarrow t && (\text{unit})\\
t \cup \emptyset &\longrightarrow t &&\\
x \cup x &\longrightarrow x && (\text{idempotence})\\
x \cup (x \cup t) &\longrightarrow x \cup t &&\\
\{c\} \cup \{c\} &\longrightarrow \{c\} &&\\
\{c\} \cup (\{c\} \cup t) &\longrightarrow \{c\} \cup t &&\\
x\ t_1 \cdots t_n \cup \{c\} &\longrightarrow \{c\} \cup x\ t_1 \cdots t_n && (\text{A.1})\\
x\ t_1 \cdots t_n \cup (\{c\} \cup t) &\longrightarrow \{c\} \cup (x\ t_1 \cdots t_n \cup t) && (\text{A.2})\\
x\ t_1 \cdots t_n \cup x\ t'_1 \cdots t'_n &\longrightarrow x\ t'_1 \cdots t'_n \cup x\ t_1 \cdots t_n \quad \text{if } x' \prec x && \\
& && (\text{A.3})\\
x\ t_1 \cdots t_n \cup \left(x\ t'_1 \cdots t'_n \cup t\right) &\longrightarrow x\ t'_1 \cdots t'_n \cup (x\ t_1 \cdots t_n \cup t) \quad \text{if } x' \prec x && (\text{A.4})\\
\{c\} \cup \{c'\} &\longrightarrow \{c'\} \cup \{c\} \quad \text{if } c' \prec c && (\text{A.5})\\
\{c\} \cup \left(\{c'\} \cup t\right) &\longrightarrow \{c'\} \cup (\{c\} \cup t) \quad \text{if } c' \prec c && (\text{A.6})
\end{aligned}
$$

**Conjecture 1.** *The reduction relation $\longrightarrow$ preserves meaning.*

**Conjecture 2.** *The reduction relation $\longrightarrow$ is strongly normalizing.*

**Conjecture 3.** *The reduction relation $\longrightarrow$ is locally confluent.*

**Corollary 1.** *The reduction relation $\longrightarrow$ is confluent.*

*Proof.* Follows from SN, LC and Newman's Lemma. □

**Corollary 2.** *The $\lambda^{\cup}$-calculus has unique normal forms.*

**Corollary 3.** *Equality of $\lambda^{\cup}$-terms can be decided by normalization.*

### A.1.2 Semantics

- To do.Combine the lemma and the theorem and make the "extensionally" explicit.
- To do.Is the case for applications rigorous? Relies on the monotonicity of $\varphi : V_{\tau_1} \to V_{\tau_2}$ (separate lemma, require in the denotational semantics?); this might fail for anything other than set union?

**Lemma 6.** $[\![t]\!]_{\rho[x\mapsto v_1]} \cup [\![t]\!]_{\rho[x\mapsto v_2]} \subseteq [\![t]\!]_{\rho[x\mapsto v_1\cup v_2]}$ *(extensionally).*

*Proof.* By induction on the term $t$.

Case "$t = x$": $[\![x]\!]_{\rho[x\mapsto v_1]} \cup [\![x]\!]_{\rho[x\mapsto v_2]} = \rho[x \mapsto v_1](x) \cup \rho[x \mapsto v_2](x) = v_1 \cup v_2 = \rho[x \mapsto v_1 \cup v_2](x) = [\![x]\!]_{\rho[x\mapsto v_1\cup v_2]}$.

Case "$t = y\ (y \neq x)$": $[\![y]\!]_{\rho[x\mapsto v_1]} \cup [\![y]\!]_{\rho[x\mapsto v_2]} = \rho[x \mapsto v_1](y) \cup \rho[x \mapsto v_2](y) = \rho(y) \cup \rho(y) = \rho(y) = \rho[x \mapsto v_1 \cup v_2](y) = [\![y]\!]_{\rho[x\mapsto v_1\cup v_2]}$.

Case "$t = t_1\ t_2$":

$$
\begin{aligned}
&[\![t_1\ t_2]\!]_{\rho[x\mapsto v_1]} \cup [\![t_1\ t_2]\!]_{\rho[x\mapsto v_2]} \\
&= \bigcup \left\{ \varphi([\![t_2]\!]_{\rho[x\mapsto v_1]}) \mid \varphi \in [\![t_1]\!]_{\rho[x\mapsto v_1]} \right\} \cup \bigcup \left\{ \varphi([\![t_2]\!]_{\rho[x\mapsto v_2]}) \mid \varphi \in [\![t_1]\!]_{\rho[x\mapsto v_2]} \right\} \\
&\overset{!}{\subseteq} \bigcup \left\{ \varphi([\![t_2]\!]_{\rho[x\mapsto v_1]}) \cup \varphi([\![t_2]\!]_{\rho[x\mapsto v_2]}) \mid \varphi \in [\![t_1]\!]_{\rho[x\mapsto v_1]} \cup [\![t_1]\!]_{\rho[x\mapsto v_2]} \right\} \\
&\overset{!}{\subseteq} \bigcup \left\{ \varphi([\![t_2]\!]_{\rho[x\mapsto v_1]} \cup [\![t_2]\!]_{\rho[x\mapsto v_2]}) \mid \varphi \in [\![t_1]\!]_{\rho[x\mapsto v_1]} \cup [\![t_1]\!]_{\rho[x\mapsto v_2]} \right\} \\
&\overset{\text{i.h.}}{\subseteq} \bigcup \left\{ \varphi([\![t_2]\!]_{\rho[x\mapsto v_1\cup v_2]}) \mid \varphi \in [\![t_1]\!]_{\rho[x\mapsto v_1\cup v_2]} \right\} \\
&= [\![t_1 \cup t_2]\!]_{\rho[x\mapsto v_1\cup v_2]}
\end{aligned}
$$

Case "$t = \emptyset$": $[\![\emptyset]\!]_{\rho[x\mapsto v_1]} \cup [\![\emptyset]\!]_{\rho[x\mapsto v_2]} = \emptyset = [\![\emptyset]\!]_{\rho[x\mapsto v_1\cup v_2]}$.

Case "$t = \{c\}$": $[\![\{c\}]\!]_{\rho[x\mapsto v_1]} \cup [\![\{c\}]\!]_{\rho[x\mapsto v_2]} = \{c\} = [\![\{c\}]\!]_{\rho[x\mapsto v_1\cup v_2]}$.

Case "$t = t_1 \cup t_2$": $[\![t_1 \cup t_2]\!]_{\rho[x\mapsto v_1]}[\![t_1 \cup t_2]\!]_{\rho[x\mapsto v_2]} = [\![t_1]\!]_{\rho[x\mapsto v_1]} \cup [\![t_1]\!]_{\rho[x\mapsto v_2]} \cup [\![t_2]\!]_{\rho[x\mapsto v_1]} \cup [\![t_2]\!]_{\rho[x\mapsto v_2]} \overset{\text{i.h.}}{\subseteq} [\![t_1]\!]_{\rho[x\mapsto v_1\cup v_2]} \cup [\![t_2]\!]_{\rho[x\mapsto v_1\cup v_2]} = [\![t_1 \cup t_2]\!]_{\rho[x\mapsto v_1\cup v_2]}$. □

The inequality of Lemma 6 in not an equality.

**Counterexample 1.** *Let* app $= \lambda f.\lambda x.f\ x$, *then* app $(\lambda x.\emptyset)\ \{C\} \cup$ app $(\lambda x.x)\emptyset \rightsquigarrow \emptyset$, *but* app $((\lambda x.\emptyset) \cup (\lambda x.x))\ (\{C\} \cup \emptyset) \rightsquigarrow \{C\}$.

### A.1.3 Normalization (with widening)

– 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 $\beta$-reduction rule of the simply typed $\lambda$-calculus with rewrite rules that deal with the associativity, commutativity, idempotence and identity (ACI1) properties of set-union operator.

If a term $t$ is $\eta$-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 $\eta$-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 $\beta$-redex.[1] For each $k_i, k_j$ with $i < j$ we must also

[1] Technically, terms that bind at least one variable would form a $\beta$-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.

have that $k_i < k_j$ for some total order on $\mathbf{Var} \cup \mathbf{Con}$. Not only does this imply that each ter m $k_i(v_{i1}, ..., v_{iq_i})$ 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 $i \neq j$.

$$
\begin{array}{l}
\quad \text{-- normalization of terms} \\
[\![\cdot]\!] : \mathbf{Tm} \to \mathbf{Nf} \\
[\![\lambda x_1 \cdots x_n.T]\!] = \\
\quad \lambda x_1 \cdots x_n.\wr\{ \lfloor f_i([\![t_{i1}]\!], ..., [\![t_{iq_i}]\!]) \rfloor \mid f_i(t_{i1}, ..., t_{iq_i}) \in T \}\wr \\
\quad \text{-- } \beta\text{-reduction} \\
\lfloor k(v_1, ..., v_q) \rfloor \\
\quad = k(v_1, ..., v_q) \\
\lfloor (\lambda y_1 \cdots y_q.T)\ (v\_1, \cdots, v\_q) \rfloor \\
\quad = \mathit{SUBST}\ x\ y\ z \\
\quad \text{-- set-rewriting} \\
\wr\{ \cdots, k\_i\ (\cdots), \cdots, k\_j\ (\cdots), \cdots \}\wr \\
\quad \mid k\_j < k\_i = \wr\{ \cdots, k\_i\ (\cdots), \cdots, k\_j\ (\cdots), \cdots \}\wr \\
\wr\{ \cdots, k\ (\cdots), k\ (\cdots), \cdots \}\wr \\
\quad = \wr\{ \cdots, k\ (\cdots), \cdots \}\wr \\
\wr T \wr \\
\quad = T
\end{array}
$$

Figure A.1: Normalization algorithm for $\lambda^{\cup}$-terms.

## A.2 Completion

The completion procedure as an algorithm:

$$
\begin{array}{l}
\mathcal{C} :: \mathbf{Env} \times \mathbf{Ty} \to \widehat{\tau} \times \varphi \times \mathbf{Env} \\
\mathcal{C}\ \overline{e_i :: \kappa_i}\ \mathbf{bool} = \\
\quad \mathbf{let}\ e\ \mathit{be\ fresh} \\
\quad \mathbf{in}\ \ \langle \widehat{\mathrm{bool}}; e\ \overline{e_i}; e :: \overline{\kappa_i \Rightarrow} \ \textsc{exn} \rangle
\end{array}
$$

# Bibliography

S. Holdermans and J. 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, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.