Proving the Soundness of a System that can Infer Effects

Tudor Ferariu

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh
2018

Abstract

This project presents a system that attempts to use annotated types, in combination with standard type inference, in order to approximate the effects that might occur as a byproduct of executing a certain expression. The system is built upon a similar one, proposed in a paper by Xavier Leroy and François Pessaux [2]. It is based on a fragment of the ML language, specialized for exception analysis and does not attempt to capture any other possible effects.

With the goal of proving the type soundness of this system, we provide an implementation in the Coq proof assistant. Concrete examples of typing derivations and expression reduction show that the implemented version has promise and seems to exhibit the desired properties.

A proof of progress is completed as a halfway point towards the proof of type soundness. This involved the discovery of several intricacies that are harder to implement properly, and not very well explained in the original system. A mistake in the original system that renders the progress lemma invalid was also found and corrected.

Overall, the ideas presented seem to produce the desired results and once the proof of subject reduction, and thus soundness, is completed, the system could be augmented to encompass a larger fragment of the ML language.

Acknowledgements

Special thanks to Don Sannella for his time spent supervising this project and the amount of help provided. I would also like to thank Wen Kokke and Daniel Hillerström, for agreeing to meet me on short-notice and help answer some of my questions, as well as friends and family, who helped me directly or indirectly.

Table of Contents

Chapter 1

Introduction

1.1 Overview

Most programming languages used in the modern day feature statements that produce effects, as well as possibly returning a result. Function effects include such things as performing input and output operations, transfer of control to a different process, and most importantly for this project, the raising of exceptions. Exceptions serve as a method of handling errors in applications, by being automatically propagated upwards through the code until the nearest exception handler that can deal with it is reached, eventually interrupting the program execution entirely, if no such handler is encountered.

Unfortunately, using exceptions for error reporting has the drawback that it is exceedingly easy to forget to catch the exception and handle the error at the right place. The ML compiler generates no errors in the case of uncaught exceptions, and the mistake can only show up in testing. There is no reliable method of exhaustively testing most programs, even those of average complexity, especially for the cases where the error condition is itself either uncommon or hard to intentionally reproduce. Languages like Java try to deal with this by having both unchecked and checked exceptions, whereby the compiler enforces that the programmer declares, for every function or method, the particular checked exceptions that might escape from it, in order to prevent these exceptions from not being caught at the right place in the program. This approach works well enough for monomorphic, first-order languages, but is entirely unsuited for the higher-order, polymorphic programming present in most functional languages, such as ML. For example, take the common *map* iterator, that applies a given function to every element of a given list. Its type would be something along the lines of $(\alpha \to \beta) \to (list\ \alpha) \to (list\ \beta)$, and since the application of *map* given a function f and a list is equivalent to applying f to every element of the list, it is clear that the exceptions a *map* function application might raise are determined by the $f : (\alpha \to \beta)$ function. Consequently we can never declare some fixed set E as the exceptions *map* might raise, unless E represents the set of all possible exceptions, but in this case we lose all precision in our system's exception analysis.

An alternative is proposed in "Type-Based Analysis of Uncaught Exceptions" (Leroy & Pessaux, 2000) [2], which claims to infer the effects of escaping exceptions from unannotated ML source code by detecting potentially escaping exceptions as a static

debugging problem. This project attempts to further look into the claimed properties of this system by implementing a simplified version in the Coq proof assistant, in order to test its properties and prove the theorems claimed to be true by the authors.

1.2 Background

Based on the effect polymorphism described in (Lucassen & Gifford, 1988) [3] and using the extension provided in (Talpin & Jouvelot, 1994) [7], the system proposed by Leroy and Pessaux allegedly not only keeps track of the exception effects that might escape during evaluation, but also enhances traditional ML types in order to get a better prediction of possible expression values. This is done by annotating any function that produces effects with the effect which might occur when the function is applied or term otherwise evaluated. The so called "latent effect", is, in this case, the set of exceptions that might be raised, or values the expression might evaluate to. The system presents itself as a set of inference rules for the type and effect of expressions, and treats a cut-down subset of ML, where the only values are integers.

In order to keep the size and scope of the Coq implementation within manageable bounds, some additional simplifications to the system were made. While said original system includes parametrized exceptions, these were entirely removed in favor of using only two constant exceptions C_1 and C_2. Having two constant exceptions allows for interesting interactions when juggling the two exceptions, which would not be possible with only a single one, while adding a third wouldn't really add any more expressiveness to the system than we already have.

Furthermore, integers were replaced with natural numbers, and their effects have also been removed. This was done in light of the fact that integer effects were mainly used to distinguish between parametrized exceptions, which have been removed, as well as occasionally providing more precise information on the possible values that expressions might evaluate to, which is not of much interest when the main focus is the exception analysis. Some basic arithmetic operations were also added, as the original design did not explicitly mention which ones were included.

In this system, we denote functions from τ to τ' with potentially escaping exception φ as $(\tau \xrightarrow{\varphi} \tau')$. Going back to the previous example of the *map* function, it can now be properly captured with the following type scheme:

$$map : \forall \alpha, \beta, \varphi.(\alpha \xrightarrow{\varphi} \beta) \xrightarrow{\emptyset} (list\ \alpha \xrightarrow{\varphi} list\ \beta)$$

where α and β range over types and φ ranges over sets of exceptions. Clearly, any application of the *map* function in this system will have the same effect as applying function $\alpha \xrightarrow{\varphi} \beta$ to each individual element in the list, which is exactly the desired behavior.

1.3 Contributions

The main aim of this project is to prove the properties and soundness of this type system for exception analysis. The steps taken towards achieving this objective are as follows:

1. Understood and explained the existing effect system.
2. Adapted and implemented the proposed simplified version of the original system in Coq.
3. Tested the implemented system through the use of concrete examples.
4. Achieved a full proof of progress towards the end-goal of proving soundness
5. Discussed and analyzed the findings.

Chapter 2

The Effect System

2.1 The Source Language

As previously mentioned, the language in (Leroy & Pessaux, 2000) [2] is a cut-down fragment of ML, with a typing system that attempts to also track the possible effects of functions during runtime. Effects can then be inferred with respect to this system as an extension of normal ML polymorphic type inference.

In this language, observed in figure 2.1, natural numbers and exceptions are the only data types, and we have access to the common operations such as abstraction, application, let bindings, as well as the pattern-matching and exception handlers made specifically for this task. The only arithmetic operations are obtaining a successor or predecessor, as well as multiplying two numbers.

All terms from identifiers to exception handlers are taken directly from (Leroy & Pessaux, 2000) [2], with the caveat that parametrized exceptions have been removed and the only constant exceptions are C_1 and C_2.

The $match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3$ construct attempts to match the value of a_1 with the pattern p, and evaluates a_2 if they match, or a_3 with the value of a_1 bound to x if they do not. Cascading match expressions are required in order to express multiple pattern-matches. Similarly, the $try\ a_1\ with\ x \rightarrow a_2$ constructor attempts to reduce a_1 to a value, but if an exception is raised during the evaluation, then a_2 is evaluated with the value of the exception bound to x.

A predefined *raise* function is assumed to exist in the environment, with the raising of an exception being represented as the application of this *raise* function to the exception value. Since the try constructor catches any and all exceptions, it must be combined with a pattern-matching operation in order to catch a specific exception C. This is done in the following manner:

$$try\ a_1\ with\ x \rightarrow match\ x\ with\ C \rightarrow a_2 \mid y \rightarrow raise(y)$$

Originally, the system included evaluation contexts as a substitute for all the rules necessary in order to evaluate each individual term until it became a value, but all of these rules have been made explicit, seeing as they are essential for the implementation

Terms:	$a ::= x$	identifier
	$\mid n$	natural number constant
	$\mid \lambda x.a$	abstraction
	$\mid a_1(a_2)$	application
	$\mid let\ x = a_1\ in\ a_2$	the *let* binding
	$\mid match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3$	pattern-matching
	$\mid C_1 \mid C_2$	exception constructors
	$\mid try\ a_1\ with\ x \rightarrow a_2$	exception handler
	$\mid n+1$	successor of n
	$\mid n-1$	predecessor of n
	$\mid n*m$	product of n and m
Patterns:	$p ::= x$	variable pattern
	$\mid n$	natural number pattern
	$\mid C_1 \mid C_2$	exception patterns
Values:	$p ::= n \mid C_1 \mid C_2 \mid \lambda x.a \mid raise$	

Figure 2.1: The Source Language

and a proper proof of progress. Many of these reduction rules are identical to the ones proposed by Leroy and Pessaux, and are similarly written in the style of (Wright and Felleisen, 1994) [8]; they can be seen in figure 2.2.

In the aforementioned rules, a represents arbitrary terms, v represents terms that have reduced to a value, p represents arbitrary patterns, x is an identifier, while y is a variable pattern (also represented as an identifier), and $a\{x \leftarrow v\}$ represents binding the value of v to identifier x in term a. The three arithmetic operations are self-explanatory; they only work on natural number constant terms, and call specific in-built functions that do the actual operation. If passed an uncaught exception, these functions will simply continue to propagate the exception. As these are natural numbers, $pred(0)$ is a special case which returns 0. This is done mostly for simplicity, but it could easily be adapted such that $pred(0)$ would instead reduce to $raise(C_1)$, which would be an interesting way to organically raise an exception when attempting to execute an illegal operation.

The reduction rules (12) and (13) for *match* expressions make use of the pattern-matching function $M(v, p)$, so as to properly represent the difference between executing branch a_2 when the matching is successful, and the failed case where a_3 is executed. While this pattern-matching function is still necessary to properly illustrate the desired

$$(\lambda x.a)(v) \Rightarrow a\{x \leftarrow v\} \quad (1)$$
$$(\lambda x.a)(raise(v)) \Rightarrow raise(v) \quad (2)$$
$$a_1(a_2) \Rightarrow a_1'(a_2) \quad \textit{if } a_1 \Rightarrow a_1' \quad (3)$$
$$v(a_2) \Rightarrow v(a_2') \quad \textit{if } a_2 \Rightarrow a_2' \quad (4)$$
$$(raise(v_1))(a_2) \Rightarrow raise(v_1) \quad (5)$$
$$v_1(raise(v_2)) \Rightarrow raise(v_2) \quad (6)$$
$$let\ x = a_1\ in\ a_2 \Rightarrow let\ x = a_1'\ in\ a_2 \quad \textit{if } a_1 \Rightarrow a_1' \quad (7)$$
$$let\ x = v\ in\ a \Rightarrow a\{x \leftarrow v\} \quad (8)$$
$$let\ x = raise(v)\ in\ a \Rightarrow raise(v) \quad (9)$$
$$match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow match\ a_1'\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \quad \textit{if } a_1 \Rightarrow a_1' \quad (10)$$
$$match\ raise(v)\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow raise(v) \quad (11)$$
$$match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow f(a_2) \quad \textit{if } f = M(v,p) \textit{ is defined} \quad (12)$$
$$match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_3\{x \leftarrow v\} \quad \textit{if } M(v,p) \textit{ is undefined} \quad (13)$$
$$try\ a_1\ with\ x \rightarrow a_2 \Rightarrow try\ a_1'\ with\ x \rightarrow a_2 \quad \textit{if } a_1 \Rightarrow a_1' \quad (14)$$
$$try\ v\ with\ x \rightarrow a_2 \Rightarrow v \quad (15)$$
$$try\ raise(v)\ with\ x \rightarrow a_2 \Rightarrow a_2\{x \leftarrow v\} \quad (16)$$
$$n+1 \Rightarrow succ(n) \quad (17)$$
$$n-1 \Rightarrow pred(n) \quad (18)$$
$$n*m \Rightarrow mult(n)(m) \quad (19)$$
$$(raise(v))+1 \Rightarrow raise(v) \quad (20)$$
$$(raise(v))-1 \Rightarrow raise(v) \quad (21)$$
$$(raise(v))*m \Rightarrow raise(v) \quad (22)$$
$$n*(raise(v)) \Rightarrow raise(v) \quad (23)$$

The pattern-matching function *M(v,p)*:

$$M(v,y) = \{y \leftarrow v\} \quad M(n,n) = \textit{identity function} \quad M(C,C) = \textit{identity function}$$

Figure 2.2: The Reduction Rules

functionality, it is quite cumbersome and mostly served as a method to deal with the complications that arose when needing to match parametrized exceptions. Given that parametrized exceptions have been removed from the system, and with some help from the properties of Coq, a simpler approach is used in the actual implementation, which bypasses the need for calling an auxiliary function and checking if it is defined, by adding two more rules which assimilate the functionality of the pattern-matching function. More details on this in section 3.1.

Finally, one of the rules for application has been changed in order to properly reflect the desired behavior, namely rule (6), which was previously more restrictive, and had

the form:

$$(\lambda x.a)(raise(v)) \Rightarrow raise(v)$$

One might also notice that rule (5) is also slightly suspicious. Although it has exactly the same form as the one proposed by Leroy and Pessaux [2], it is inconsistent with the rest of the rules, which seem to use the "call by value" approach. This is still a correct approach though, as whenever the first term of an application is an uncaught exception, the entire expression should reduce to said uncaught exception, irrespective of what the second term evaluates to, so there is no need to reduce the second term to a value. It is, in fact, necessary for both rules to have this form in order to obtain the proof of progress. A full description detailing the reasoning behind the forms of rules (5) and (6) can be found in section 3.3.

2.2 The Type Algebra

As initially proposed in (Talpin & Jouvelot, 1994) [7], the function types $\tau_1 \xrightarrow{\varphi} \tau_2$ are annotated by a row φ which in this case represents the set of exceptions it might raise when applied, or more generally, its latent effect. Exception types $exn[\varphi]$ are also annotated in a similar manner, in order to restrict the values an expression of that type could have. A row is represented as a sequence of row elements $\varepsilon_1 ... \varepsilon_n$ terminated by a row variable ρ. Row elements take the form $C : \pi$, with presence annotation π taking the value *Pre* if the element is actually present, or that of a presence variable δ when it is absent.

Type expressions:	$\tau ::= \alpha$	type variable
	$\mid nat$	natural number type
	$\mid exn[\varphi]$	exception type
	$\mid \tau_1 \xrightarrow{\varphi} \tau_2$	function type
Type schemes:	$\sigma ::= \forall \alpha_i, \rho_j, \delta_k.\tau$	
Rows:	$\varphi ::= \rho$	row variable
	$\mid \varepsilon;\varphi$	the element ε followed by row φ
Row Elements:	$\varepsilon ::= C : \pi$	constant exception element
Presence Annotations:	$\pi ::= \delta$	presence variable
	$\mid Pre$	element is present

The paper by Leroy & Pessaux imposes two equation constraints on the structure of rows, one of which becomes irrelevant due to the removal of integer effects. The other attempts to express that the order of elements in a set does not matter, in the following way:

$$\varepsilon_1;\varepsilon_2;\varphi \;=\; \varepsilon_2;\varepsilon_1;\varphi$$

While this is technically a correct way of expressing the desired property, it initially appears to only say that the first two elements of a row can be swapped. This is not

correct, as any two elements in the middle of a row are themselves a row, and thus can be swapped, eventually allowing for the swapping of any two elements through repeated permutations. We propose the following alternative, which is easier to follow, although considerably more verbose:

$$\forall \varphi = \varepsilon_1; ...\varepsilon_n; \rho \ \textit{and} \ 1 \leq i \leq j \leq n, \ ...\varepsilon_i; ...\varepsilon_j; ...; \rho \ = \ ...\varepsilon_j; ...\varepsilon_i; ...; \rho$$

Either way, neither is needed in our reduced system, as the only two possible elements are $C_1 : \pi$ and $C_2 : \pi$, which allows for these equation constraints to be removed completely in the modified system and instead incorporated in the typing rules, as seen in the next section.

A peculiarity of this system is the way it deals with effects that an expression may **not** produce. As seen for rows and row elements, there is no constant for an empty row, or annotation for an absent element. Instead, a row variable ρ represents the empty row, but may also be replaced by some $\varepsilon; \rho'$ in order to satisfy unification constraints. Similarly, a presence variable δ signifies the absence of whatever exception element it is attached to, but may be replaced by *Pre*. This is accomplished through the use of universally quantified row and presence variables in type schemes. For example, an expression of type $\forall \rho.\ nat \xrightarrow{\rho} nat$ or $\forall \delta, \rho.\ nat \xrightarrow{C_1:\delta;\rho} nat$ cannot raise any exception, while one with type $\forall \rho.\ nat \xrightarrow{C_1:Pre;\rho} nat$ can possibly raise exception C_1.

2.3 Kinding of Rows

Using ideas from previous work on kinds and record types in (Ohori, 1995) [4] and (Rémy, 1994) [6], kinds are added to the system in (Leroy & Pessaux, 2000) [2], in order to enforce two structural invariants necessary for ensuring the existence of principal unifiers and typings, as well as to simplify the typing rules. While the original paper proposes four such invariants, the removal of integer effects and rows from this simplified version removes the need for them to be enforced. The two that remain are:

1. Any given exception constructor can never occur more than once in a row. As an example, $(C_1 : \pi\ ;\ C_1 : \pi';\ \rho)$ is not well formed.

2. A row variable ρ must always be preceded by the same set of exception constructors in all rows where it occurs. For example, $(C_1 : \pi\ ;\ \rho)$ and $(C_2 : \pi'\ ;\ \rho)$ can never occur in the same derivation.

$$\text{Kinds:} \quad \kappa ::= EXN\{sets\ of\ exceptions\ C\}$$

The exceptions appearing in the set describing the kind are the ones which can not appear in rows of the same kind, due to the fact that they are already present in elements seen preceding these rows somewhere else in the derivation, which would lead to contradicting invariant 1. In order to ensure the correctness invariant 2, it is assumed that for every derivation there exists a function K that assigns kinds to row

variables. This guarantees that any row variable ρ that terminates a row anywhere in the derivation will have the same kind in all of those rows, and consequently, be preceded by exactly the same elements in all such rows.

Here we define two judgments, $\vdash \varphi :: \kappa$ (row φ has kind κ), and $\vdash \tau\ wf$ (type τ is well formed). The rules for both of these can be seen in figure 2.3, although many of the original kinding rules have been removed since they pertained to rows containing either integer or parametrized exception elements, which are no longer part of the system.

Kinding Rules:

$$\vdash \rho :: K(\rho)\ (1)$$

$$\frac{C \notin S \quad \vdash (C : \pi; \varphi) :: EXN(S \cup \{C\})}{\vdash \varphi :: EXN(S)}\ (2)$$

Well-Formedness Rules:

$$\vdash \alpha\ wf\ (1)$$

$$\vdash nat\ wf\ (2)$$

$$\frac{\vdash \varphi :: EXN(\emptyset)}{\vdash exn[\varphi]\ wf}\ (3)$$

$$\frac{\vdash \tau_1\ wf \quad \vdash \varphi :: EXN(\emptyset) \quad \vdash \tau_2\ wf}{\vdash \tau_1 \xrightarrow{\varphi} \tau_2\ wf}\ (4)$$

Figure 2.3: The Kinding and Well-Formedness Rules

This mapping K is one of the more interesting concepts and is in essence passed along as an argument for any and all typing, kinding and well-formedness rules as these might eventually require the kinding of a terminating row variable ρ, which can only be shown using kinding rule (1), the application of K. This is discussed in further detail in section 3.1.

2.4 The Typing Rules

As in the original effect system from (Leroy & Pessaux, 2000) [2], the typing rules define the judgment $E \vdash a : \tau/\varphi$, with E being the typing environment, a the term we are interested in typing, τ the type of a, and φ the row expressing what exceptions may be raised during evaluation. Environments act as a map from identifiers to type schemes, which will simply appear without any universally quantified variables when they are representing an exact type. The initial environment all type derivations are assumed to start in is the following:

$$E_0 = \{raise : \forall \alpha, \rho . exn[\rho] \xrightarrow{\rho} \alpha\}$$

Additionally we define the operation of asymmetric concatenation for environments, written $E_1 \oplus E_2$, as:

$$(E_1 \oplus E_2)(x) = E_2(x) \qquad if\ x \in Dom(E_2)$$
$$(E_1 \oplus E_2)(x) = E_1(x) \qquad if\ x \in Dom(E_1) \backslash Dom(E_2)$$

The collection of typing rules, as well as the ones for pattern matching and pattern subtraction can be seen in the following figure, 2.4, alongside the definitions for instantiation and generalization.

Typing of expressions:

$$\frac{\tau \leq E(x) \quad \vdash \varphi :: EXN(\emptyset)}{E \vdash x : \tau/\varphi} \ (1)$$

$$\frac{\vdash \varphi :: EXN(\emptyset)}{E \vdash n : nat/\varphi} \ (2)$$

$$\frac{E \vdash \tau_1 wf \quad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \quad \vdash \varphi :: EXN(\emptyset)}{E \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi} \ (3)$$

$$\frac{E \vdash\ a_1 : (\tau' \xrightarrow{\varphi} \tau)/\varphi \quad E \vdash a_2 : \tau'/\varphi}{E \vdash a_1(a_2) : \tau/\varphi} \ (4)$$

$$\frac{E \vdash\ a_1 : \tau_1/\varphi \quad E \oplus \{x : Gen(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi}{E \vdash let\ x = a_1\ in\ a_2 : \tau/\varphi} \ (5)$$

$$\frac{E \vdash\ a_1 : \tau_1/\varphi \quad \vdash p : \tau_1 \Rightarrow E' \quad \vdash \tau_1 - p \rightsquigarrow \tau_2 \quad E \oplus E' \vdash a_2 : \tau/\varphi \quad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi}{E \vdash match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi} \ (6)$$

$$\frac{\vdash \varphi' :: EXN(\{C\}) \quad \vdash \varphi :: EXN(\emptyset)}{E \vdash C : exn[C : Pre; \varphi'] : \tau/\varphi} \ (7)$$

$$\frac{\vdash \varphi' :: EXN(\{C', C\}) \quad \vdash \varphi :: EXN(\emptyset)}{E \vdash C : exn[C' : \pi; C : Pre; \varphi'] : \tau/\varphi} \ (8)$$

$$\frac{E \vdash\ a_1 : \tau/\varphi_1 \quad E \oplus \{x : exn[\varphi_1]\} \vdash a_2 : \tau/\varphi}{E \vdash try\ a_1\ with\ x \rightarrow a_2 : \tau/\varphi} \ (9)$$

Typing of patterns:

$$\vdash x : \tau \Rightarrow \{x : \tau\} \ (10)$$

$$\vdash n : nat \Rightarrow \{\} \ (11)$$

$$\vdash C : exn[C : \pi; \varphi] \Rightarrow \{\} \ (12)$$

$$\vdash C : exn[C' : \pi'; C : \pi; \varphi] \Rightarrow \{\} \ (13)$$

Pattern Subtraction:

$$\vdash nat - n \rightsquigarrow nat (14)$$

$$\vdash exn[C : \pi; \varphi] - C \rightsquigarrow exn[C : \pi'; \varphi] (15)$$

$$\vdash exn[C' : \pi'; C : \pi; \varphi] - C \rightsquigarrow exn[C' : \pi'; C : \pi''; \varphi] \quad (16)$$

$$\frac{\vdash \tau' \; wf}{\vdash \tau - x \rightsquigarrow \tau'} \quad (17)$$

Instantiation and generalization:
$\tau' \leq \forall \alpha_i \rho_j \delta_k . \tau \Leftrightarrow \exists \, \tau_i, \varphi_j, \pi_k$ s.t. $\vdash \tau_i \; wf$ and $\vdash \varphi_j :: K(\rho_j)$ and
$\tau' = \tau\{\alpha_i \leftarrow \tau_i, \rho_j \leftarrow \varphi_j, \delta_k \leftarrow \pi_k\}$
$Gen(\tau, E, \varphi)$ is $\forall \alpha_i \rho_j \delta_k . \tau$ where $\{\alpha_i, \rho_j, \delta_k\} = FV(\tau) \backslash (FV(E) \cup FV(\varphi))$

Figure 2.4: The Typing Rules

Typing rules for the arithmetic operations have not been included in this list, as they are very simple and would just take up space, they will be mentioned when discussing implementation in chapter 3.

Rule (1) for variables has been changed to include a kinding of the row φ representing the possible effects of whatever is bound to x in the environment. The previous rule had the form:

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

This does not respect the proper format for typing judgments, which is defined at the start of this section, but also, when the conclusion is rewritten as $E \vdash x : \tau / \varphi$ (according to the aforementioned proper format for judgments), allows for the derivation of the following contrived example, which the system should not allow:

$$\frac{exn[C_1 : Pre; \varphi'] \leq \forall \pi . exn[C_1 : \pi; \varphi']}{E_0 \oplus \{x : \forall \pi . exn[C_1 : \pi; \varphi']\} \vdash x : exn[C_1 : Pre; \varphi'] / (C_2 : Pre; \varphi')}$$

The incomplete rule (1) variant would seem to imply that the possible effect φ of variable x is irrelevant, but we can clearly see that, in the case the above derivation, this allows us to break row invariant 2, as φ' in this derivation is preceded by both C_1 and C_2. This is easily solved by applying the same technique seen in rules (2), (3), (7) and (8) in order to ensure proper kinding by adding $\vdash \varphi :: EXN(\emptyset)$ to the necessary assumptions.

Once again, having discarded integer effects and parametrized exceptions allows for the removal of several rules from all three categories, as well as the simplifications made to rules (2), (11), and (14). Rules (8), (13), and (16) have been added in order to remove the need for the equational constraint on rows mentioned in section 2.2. This is done by taking advantage of the fact that there are only two possible row elements in this reduced system.

All the other rules are adapted as seen in the original design, most of which follow a standard for effect systems. Rule (5) for let bindings is more unusual, because it generalizes over all three kinds of variables (type, row and annotation) by using the *Gen* predicate defined at the bottom of figure 2.4. This is what allows for polymorphism in the proposed system.

The other interesting rules are (9), for exception handling, and (6), which deals with pattern matching and is essential for the exception analysis. Rule (6) is of particular note, since it uses the pattern subtraction rules (14)-(17) to ensure that the type of

values that can be bound to x and flow into the second branch a_3 are exactly the type of those in a_1 from which those matching pattern p have been excluded.

2.5 Typing Derivation Example

In order to have a better look at how the proposed system works, let us examine a full type derivation of the following expression, which will showcase the use of most of the interesting rules:

$$let\ test = \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y) in\ test(C_1)$$

Using typing rule (5) for let bindings:

$$\frac{\begin{array}{c} E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash test(C_1) : \tau/\varphi \\ E_0 \vdash \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y) : \tau_1/\varphi \end{array}}{E_0 \vdash let\ test = \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ in\ test(C_1) : \tau/\varphi} \quad (1)$$

In order to apply the lambda abstraction (3) rule: $\boxed{\tau_1 = (\tau_2 \xrightarrow{\varphi'} \tau_3)}$

$$\frac{\begin{array}{c} \vdash \tau_2\ wf \quad \vdash \varphi :: EXN(\emptyset) \\ E_0 \oplus \{e : \tau_2\} \vdash try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ : \tau_3/\varphi' \end{array}}{E_0 \vdash \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ : (\tau_2 \xrightarrow{\varphi'} \tau_3)/\varphi} \quad (2)$$

Continuing down this path we apply the exception handler rule (9) to the try expression:

$$\frac{\begin{array}{c} E_0 \oplus \{e : \tau_2\} \vdash raise(e) : \tau_3/\varphi_1 \\ E \oplus \{e : \tau_2\} \oplus \{x : exn[\varphi_1]\} \vdash match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ : \tau_3/\varphi' \end{array}}{E_0 \oplus \{e : \tau_2\} \vdash try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ : \tau_3/\varphi'} \quad (3)$$

Here we can continue the derivation for the match expression with rule (6) and abbreviate the environment to $E_1 = E_0 \oplus \{e : \tau_2\} \oplus \{x : exn[\varphi_1]\}$ to increase readability. The derivation is as follows:

$$\frac{\begin{array}{ccc} E_1 \vdash x : \tau_4/\varphi' & \vdash C_1 : \tau_4 \Rightarrow E' & \vdash \tau_4 - C_1 \rightsquigarrow \tau_5 \\ E_1 \oplus E' \vdash 1 : \tau_3/\varphi' & E_1 \oplus \{y : \tau_5\} \vdash raise(y)\ : \tau_3/\varphi' & \end{array}}{E_1 \vdash match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y)\ : \tau_3/\varphi'} \quad (5)$$

Looking at $E_1 \oplus E' \vdash 1 : \tau_3/\varphi'$ and applying rule (2) we can conclude that $\boxed{\tau_3 = nat}$ and $\boxed{\varphi' :: EXN(\emptyset)}$. For the typing of patterns in $\vdash C_1 : \tau_4 \Rightarrow E'$ we can apply both (12) and (13), but (12) is sufficient here as C_2 does not appear in the expression at any point. We must then have $\boxed{\tau_4 = exn[C_1 : \pi; \varphi_2]}$ and $\boxed{E' = \{\}}$. Similarly, from $\vdash \tau_4 - C_1 \rightsquigarrow \tau_5$, since we now know τ_4, with the use of rule (15) we obtain $\boxed{\tau_5 = exn[C_1 : \pi'; \varphi_2]}$.

Now looking at the derivation of $E_1 \vdash x : \tau_4/\varphi'$ by using rule (1) and replacing the variables we know about and extracting the bound value of x from the environment:

$$\frac{exn[C_1 : \pi; \varphi_2] \leq exn[\varphi_1] \quad \varphi' :: EXN(\emptyset)}{E_0 \oplus \{e : \tau_2\} \oplus \{x : exn[\varphi_1]\} \vdash x : exn[C_1 : \pi; \varphi_2]/\varphi'} \quad (6)$$

From which we must now also have that $\boxed{\varphi_1 = C_1 : \pi; \varphi_2}$. With this new information we can go back to the raise expression in the assumptions of derivation (5) and by substituting some of the now known variables get:

$$\frac{\dfrac{(\tau_6 \xrightarrow{\varphi'} \tau_3) \leq \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha \quad \vdash \varphi' :: EXN(\emptyset)}{E_1 \oplus \{y : exn[C_1 : \pi'; \varphi_2]\} \vdash raise : (\tau_6 \xrightarrow{\varphi'} \tau_3)/\varphi'} \quad \dfrac{\tau_6 \leq exn[C_1 : \pi'; \varphi_2] \quad \varphi' :: EXN(\emptyset)}{E_1 \oplus \{y : exn[C_1 : \pi'; \varphi_2]\} \vdash y : \tau_6/\varphi'}}{E_1 \oplus \{y : exn[C_1 : \pi'; \varphi_2]\} \vdash raise(y) \ : \tau_3/\varphi'} \quad (7)$$

On the far left side we ascertain that $\boxed{\tau_6 = exn[\varphi']}$, which then propagates to the right derivation of the variable y to also reveal that $\boxed{\varphi' = C_1 : \pi'; \varphi_2}$. We can then return to the raise expression in the assumptions of derivation (3) and with the new information find:

$$\frac{\dfrac{(\tau_7 \xrightarrow{(C_1:\pi;\varphi_2)} nat) \leq \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha \quad \varphi_1 :: EXN(\emptyset)}{E_0 \oplus \{e : \tau_2\} \vdash raise : (\tau_7 \xrightarrow{(C_1:\pi;\varphi_2)} nat)/(C_1 : \pi; \varphi_2)} \quad \dfrac{\tau_7 \leq \tau_2 \quad \vdash \varphi_1 :: EXN(\emptyset)}{E_0 \oplus \{e : \tau_2\} \vdash e : \tau_7/(C_1 : \pi; \varphi_2)}}{E_0 \oplus \{e : \tau_2\} \vdash raise(e) : nat/(C_1 : \pi; \varphi_2)} \quad (8)$$

Once again, looking on the far left side, we can see that $\boxed{\tau_7 = exn[C_1 : \pi; \varphi_2]}$, which then implies that we can consider $\boxed{\tau_2 = exn[C_1 : \pi; \varphi_2]}$, as we don't need it to be more general than that.

As an aside, if we go and do a kinding derivation for $\vdash \varphi_1 :: EXN\{\emptyset\}$:

$$\frac{C_1 \notin \emptyset \quad \dfrac{\vdash \varphi_2 :: K(\varphi_2)}{\vdash \varphi_2 :: EXN(\{C_1\})}}{\vdash (C_1 : \pi; \varphi_2) :: EXN(\emptyset)}$$

Which constrains our global mapping function K to have $K(\varphi_2) = EXN(\{C_1\})$, but this is perfectly legal, and also ensures the truth of all other kinding derivations that have and will appear in the entire derivation, as well as ensure that $\tau_2 \ wf$, an assumption required by derivation (2):

$$\frac{\vdash (C_1 : \pi; \varphi_2) :: EXN(\emptyset)}{exn[C_1 : \pi; \varphi_2]}$$

The assumption being proven exactly as shown in the kinding above.

Finally, returning to the application of $test(C_1)$ from the very first derivation, we now have enough information to put together that $\boxed{\tau_1 = (exn[C_1 : \pi; \varphi 2] \xrightarrow{(C_1:\pi';\varphi 2)} nat)}$ and thus properly generalize the type of *test* in order to complete the derivation:

$$\frac{E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash test : (\tau' \xrightarrow{\varphi} \tau)/\varphi \quad E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash C_1 : \tau'/\varphi}{E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash test(C_1) : \tau/\varphi} \quad (9)$$

First, on the right side we have $E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash C_1 : \tau'/\varphi$ which can be derived with rule (7) or (8), but (7) is once again sufficient here, as there is no reason to consider C_2 given our initial expression. This essentially implies that $\boxed{\tau' = exn[C_1 : Pre; \varphi'']}$, since we need:

$$\frac{\vdash \varphi'' :: EXN(C_1) \quad \vdash \varphi :: EXN(\emptyset)}{E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash C_1 : (C_1 : Pre; \varphi'')/\varphi} \quad (10)$$

Then, taking this new information and utilizing the expanded form of the generalization of variable *test* we find that:

$$\frac{(exn[C_1 : Pre; \varphi''] \xrightarrow{\varphi} \tau) \leq \forall \pi, \pi', \rho_2, \varphi'.exn[C_1 : \pi; \varphi_2] \xrightarrow{(C_1:\pi';\varphi 2)} nat \quad \varphi :: EXN(\emptyset)}{E_0 \oplus \{test : Gen(\tau_1, E_0, \varphi)\} \vdash test : (exn[C_1 : Pre; \varphi''] \xrightarrow{\varphi} \tau)/\varphi} \quad (11)$$

This instantiation is correct for $\boxed{\varphi'' = \varphi_2}$ and $\boxed{\pi = Pre}$, which implies that $\boxed{\tau = nat}$ and $\boxed{\varphi = (C_1 : \pi'; \varphi_2)}$, which gives the final typing of the initial expression that we were deriving in this example as:

$$E_0 \vdash let\ test = \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_1 \rightarrow 1 \mid y \rightarrow raise(y) in\ test(C_1) : nat/C_1 : \pi'; \varphi_2$$

This conveys the fact that the expression will have type *nat* and cannot possibly let any exceptions escape. On the other hand, if we replace the pattern in the match expression with C_2 instead of C_1, through a very similar derivation we will instead arrive to:

$$E_0 \vdash let\ test = \lambda e.try\ raise(e)\ with\ x \rightarrow\ match\ x\ with\ C_2 \rightarrow 1 \mid y \rightarrow raise(y) in\ test(C_1) : nat/C_1 : Pre; C_2 : \pi'; \varphi_2$$

In this case the type of the expression is still nat, but it will never yield a value of that type, as it will instead raise the C_1 exception during evaluation.

Chapter 3

Implementation and Proofs

The entirety of the implementation is done in the Coq proof solver and is built upon the the simply-typed lambda calculus module from "Software Foundations" (Pierce, 2017) [5]. Every rule and definition from chapter 2 has been translated as directly as possible in order to preserve the desired functionality of the system, with everything from terms and types to operations and rules implemented from scratch with some use of built-in Coq libraries. For the reduction rules, the style of Wright and Felleisen [8] is imitated through the use of of small-step operational semantics, also based on the aforementioned book by Pierce. The relevant implementation code has been attached in Appendices A-C, which is where all the figures mentioned in this chapter are located.

3.1 Description

In order to emulate the original architecture as best as possible, as well as maintain the desired functionality, some choices had to be made during the implementation.

Type schemes, for example, are formally defined as $\forall \alpha_i, \rho_j, \delta_k.\tau$, and were implemented as a constructor that takes three lists of identifiers, each representing the possible universally quantified type, row, or presence variables, and the type τ. Consequently, environments were designed such that they map certain identifiers bound in the expression to their respective type scheme. In the case where it is supposed to be a simple type and not a type scheme, then the three lists are simply left empty, to signify that there are no variables to be universally quantified for said type. A problem arises when you consider the *raise* term. Leroy and Pessaux described *raise* as a predefined function with type scheme $\forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha$, but they accomplish this by having each derivation happen in the E_0 environment where *raise* is bound to this value. Unfortunately, if environments are set up to map identifiers to type schemes, the *raise* term can not be bound in an environment without being represented by an identifier. At that point, new rules would have to be added in order to deal with the raise term that now carries an identifier, and they would be little more than small variations of typing rule (1) for variables. In order to solve this issue in the simplest manner, *raise* was made into a globally reserved identifier, such that it can be bound in the E_0 environment as suggested in the original paper. This both solves the problem and does not impose much change on the overall architecture, with the small exception that reduction rules and list of values had to be adapted in order to recognize the term containing identifier

raise, as opposed to a dedicated *raise* term. An alternative that was briefly considered was to change the way environments work entirely, so that they would be more adaptable and deal with this situation as well, but that did not yield any good results, and was scrapped in favor of the simpler approach mentioned above.

A major hurdle was the K function that maps row variables to rows, mentioned in section 2.3. An initial attempt was made to derive it alongside the typing derivation by adding to the typing rules, but that did not lead to any solutions, because it appears to be impossible to build K this way. However it also cannot be made into a true global function, as K is unique for each individual derivation. The current solution has been to existentially quantify K at the beginning of every typing derivation, and also add it to every type rule in the following form "$K\&E \vdash a_1 : \tau/\varphi$", as opposed to the previous "$E \vdash a_1 : \tau/\varphi$". A very similar approach is utilized to pass K through the rules for pattern subtraction, kinding and well-formedness, albeit using a slightly different format for each of the three. This has been done simply so that the rules can eventually pass K to the part of the derivation that requires its use, since Coq does not allow for rules to reference undeclared variables that are to be defined later, and there is no way to define it globally when it needs to be unique to each derivation. The rules themselves are still essentially identical to those seen in section 2.4, since it is the metalanguage of Coq that requires this compromise of adding K to the implementation of the rules.

An example of this can be seen in figure A.8, where we can also inspect the newly added rules for arithmetic expressions, which were mentioned in section 2.4. They are written as:

$$
\begin{aligned}
&T\_Succ : \forall E, t_1, \varphi, Kmap. \\
&Kmap \;\&\; E \vdash t_1 : TNat \;//\; \varphi \rightarrow Kmap \;\&\; E \vdash tsucc(t_1) : TNat \;//\; \varphi
\end{aligned}
$$

Here $Kmap$ represents the K mapping, with $TNat$ being the type of natural numbers as defined in figure A.6. When inspecting the appendix, one might notice that the unadapted Coq code uses many backslashes in order to avoid conflicts with certain keywords (such as "and", "in", ":"), and is overall quite hard to read, which is why we will refer to it using more elegant pseudo-code, akin to that seen in chapter 2, whenever possible. Here, this rule would translate to:

$$\frac{K \;\&\; E \vdash n : nat/\varphi}{K \;\&\; E \vdash n+1 : nat/\varphi}$$

Similarly for the other two operations we would have:

$$\frac{K \;\&\; E \vdash n : nat/\varphi}{K \;\&\; E \vdash n-1 : nat/\varphi}$$

$$\frac{K \;\&\; E \vdash n : nat/\varphi \quad K \;\&\; E \vdash m : nat/\varphi}{K \;\&\; E \vdash n * m : nat/\varphi}$$

In the scope of these rules we do not need any kinding judgment as premise, since any application of these typing rules always preserves the effect φ and type *nat*, which means that eventually this will reduce to the application of typing rule (2) for natural numbers, which will then impose the proper kinding on φ.

As was previously mentioned in section 2.4, the substitution rules have been rewritten in the Coq language in order to subsume the pattern-matching function $M(v,p)$. They can be observed in figure A.5, and are as follows:

$$match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow match\ a_1'\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \quad if\ a_1 \Rightarrow a_1' \tag{1}$$
$$match\ raise(v)\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow raise(v) \tag{2}$$
$$match\ v\ with\ y \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_2\{y \leftarrow v\} \tag{3}$$
$$match\ C\ with\ C \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_2 \tag{4}$$
$$match\ n\ with\ n \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_2 \tag{5}$$
$$match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_3\{x \leftarrow v\} \tag{6}$$

The first two rules are the same as those seen before, as they serve a more general purpose, of reducing a_1 to a value, and propagating an uncaught exception, respectively. Instead of the two cases where $M(v,p)$ is defined or not, we have rules (3)-(5), which represent the three possible cases in which the pattern matching function is defined, and rule (6) for when it would not be defined. Here, rule (6) is quite interesting, as it could technically be applied in situations where one of the (3)-(5) rules is valid and should be applied instead, but because it is the last *match* reduction rule defined, it will never be applied unless all other rules will not work, due to the particular way Coq syntax works. This essentially guarantees that we impose the same restrictions on the *match* reduction as the pattern-matching function, in order to emulate the desired effect.

Many helper functions that the system needs but the original does not mention in detail also had to be implemented, as can be seen in appendix A. These include the environments themselves and the asymmetric concatenation of environments operation (figure A.11), recursive functions that either substitute type, row and presence variables (figure A.4), or extract the free type, row and presence variables from a certain context (figure A.12). Generalization is also defined as a function (figure A.10), which uses the type, environment and row it has been given, as well as the aforementioned helper functions, to produce the type scheme that is needed for the *let* typing rule (5). It does so by universally quantifying the variables free in the type, but not free in either the environment or row, as described in section 2.4.

The rest of the important requirements are represented as inductive definitions which return certain properties, denoted as *Prop* in Coq. These properties are then used either recursively, or by other inductive definitions in order to show either concrete examples of how the system would behave, or prove lemmas and theorems. This includes terms (figure A.2), patterns, values (figure A.3), reduction rules (figure A.5), the type algebra (figure A.6), kinding and well-formedness rules (figure A.7), typing rules (figure A.8), typing of patterns and pattern subtraction (figure A.9), and instantiation (figure A.10).

Certain built-in libraries were used as well, such as “Coq.lists.list”, in order to import list utilities without the need to define them from scratch, as they are used for the definitions of kinds and type schemes. The same library also affords us access to the *In* function, which checks whether an element belongs to a list, and is used in kinding rule (2). A few minor lemmas were also implemented in order to facilitate things like the comparison of identifiers and assure properties such as the transitivity

of multi-stepping from an expression to another simpler instance of itself using the reduction rules.

3.2 Experiments

The environment used for basic tests was set up to contain several predefined identifiers for easy use, as well as the E_0 environment that binds the value of *raise* and serves as a starting point for all type derivations (Appendix B, figure B.1). Note also that all examples are conditioned on the existence of the *Kmap* function that assures the mapping of row variables to kinds, an instance of which will always be necessary to assure the proper kinding of rows. Here it is used in a rather lax manner, since it is assumed this *Kmap* has to exist for all derivations, as such, in the case where a row variable is not found in the mapping, the return value simply defaults to kind $EXN\{\emptyset\}$. The mapping has to always return a kind for every possible variable by definition, so we cannot use an optional type or partial map here, but a stricter implementation would probably create an impossible value to return instead, in order to make sure no erroneous proofs can arise.

With the use of the complete implementation, several simple expressions were tested to ascertain whether or not they reduce correctly and type-check. First, testing was done with a simple expression using only the new arithmetic operations, then one to check the functionality of the *let* binding, and finally one for an uncaught exception. All of these can be seen in figure B.2, alongside their respective proofs. This excludes a proof of reduction for the uncaught exceptions, as there is nothing for it to reduce to except itself, which is trivial to show.

After these basic tests, the big derivation in section 2.5 was tackled next, as it uses all of the important rules at least once and thus serves as a good way to evaluate overall system performance. The code for this can be seen in figure B.3, alongside the check for proper reduction of the slightly modified version where the error raised and the one caught by the match expression are different. While the proof of reduction was quite simple in both cases, the type-checking was only done for the main example and is, regrettably, just as laborious as manual derivation. It is rather unfortunate that the abundance of existential quantifiers used in the instantiation rule prevents Coq from automating the type checking. One of the main offenders is the *Kmap* function, for which there currently exists no better alternative in the current version of the system. The fact that the Coq metalanguage also processes all the premises from left to right does not help at all either. Take for example the derivation:

$$\dfrac{\dfrac{(\tau' \xrightarrow{\varphi} \tau) \leq \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha \quad \vdash \varphi :: EXN(\emptyset)}{E_0 \vdash raise : (\tau' \xrightarrow{\varphi} \tau)/\varphi} \qquad \dfrac{\varphi' :: EXN(\{C_1\}) \quad \varphi :: EXN(\emptyset)}{E_0 \vdash C_1 : exn[C_1 : Pre; \varphi']/\varphi}}{E_0 \vdash raise(C_1) \ : \tau/\varphi}$$

It is clear from the right side that $\tau' = exn[C_1 : Pre; \varphi']$, but the Coq proof assistant will always try to solve the premises from left to right. As such, when instantiating the form of $(\tau' \xrightarrow{\varphi} \tau)$ we need to skip directly to the fact that $\tau' = exn[C_1 : Pre; \varphi']$, because while $\tau' = exn[\varphi]$ is also correct, none of the typing rules can actually derive the right

side at this point, and there is no explicit way in this system to substitute φ with its value of $C_1 : Pre; \varphi'$. This is one of the main problems of the system as it presently stands, seeing as it is currently uncertain whether it is just a minor inconvenience when doing type derivations, or might cause issues with respect to the subject reduction proof.

The implementation that ours is based on (Pierce, 2017) [5], does not deal with polymorphism or inference, and thus waves away these problems. In said system, alongside not having to deal with rows and kinds for the effects of expressions, the let binding is little more than a modified lambda abstraction, and the lambda abstractions themselves are annotated with the type of their parameter. For example, the expression $(\lambda(x : nat).x)\ 5$ would be derived as:

$$\frac{\lambda(x : nat).x : nat \to nat \quad 5 : nat}{(\lambda(x : nat).x)\ 5 : nat}$$

Since it is already known that the type of x is nat, there is no need to infer it like in our proposed system. While this does allow for easier proofs and automated type-checking, we simply cannot afford to use a similar approach here, as that would stop us from investigating the desired behaviors of the system. Due to all of these factors, finding a better way to represent polymorphism will become a high priority if subject reduction is deemed unprovable with the current system.

3.3 Proofs

Unlike the various problems encountered with type derivations, all expressions reduced to either a value or an uncaught exception through automatic methods in all tests, so the proof of progress was attempted first. Progress is defined just as in (Leroy & Pessaux, 2000) [2].

LEMMA 1 (PROGRESS). *If $E_0 \vdash a : \tau/\varphi$, then either a is an uncaught exception raise(v), a is a value v, or $\exists\ a'$ such that $a \Rightarrow a'$.*

This is written in the standard way of expressing progress, with the terms either being reducible, values, or the added case for our particular system, where they can also be uncaught exceptions. The proof is tackled in an inductive manner as can be seen in Appendix C, figure C.1, with a Coq definition for terms representing uncaught exceptions right before the lemma.

The proof itself goes quite smoothly for all the unmodified cases, as well as the added arithmetic operations and modified *match* expression rules. The only problem arose when dealing with the application term. Here, let us consider $t_1(t_2)$ the application of term 1 to term 2. Since this is an inductive proof, we iterate through all possible combinations of t_1 and t_2, with them being, as the lemma implies, an uncaught exception, a value, or there existing some t' that they step to.

First, we consider if t_1 is an uncaught exception $raise(v)$, in this case, regardless of what form t_2 takes, we can apply substitution rule (5) in order to reduce $(raise(v))(t_2)$ to $raise(v)$, thus $\exists\ raise(v),\ s.t.\ (raise(v))(t_2) \Rightarrow raise(v)$, so this case is fine according to the definition. This case is interesting, because an initial change made to the system

was to alter rule (5) to:

$$(raise(v_1))(v_2) \Rightarrow raise(v_1)$$

As mentioned in section 2.1, it was thought that rule (5) was inconsistent with the "call by value" nature of the other rules, and subsequently changed to the above variant. This proved to be incorrect, as in the above step, once t_1 is an uncaught exception $raise(v_1)$, we have to perform induction on t_2 as well. In the cases where it is either a value v, or $\exists\, t_2'\ s.t.\ t_2 \Rightarrow t_2'$, a solution can be found, but if it is also a uncaught exception, the application becomes $(raise(v_1))(raise(v_2))$, which is neither an uncaught exception, nor a value, and is irreducible with the current rule set, since $raise(v_2)$ is not a value. Consequently, rule (5) was reverted to its initial form, so that this would not be an issue and we can use the proof written above.

Continuing the induction, when we have that t_1 is a value v_1, the following subcases arise: v_1 is either $(\lambda x.a)$ or *raise* since of all possible values that v_1 could take, only these two fulfill the required typing for term 1 of an application. Originally, as mentioned in section 2.1, rule (6) was:

$$(\lambda x.a)(raise(v)) \Rightarrow raise(v)$$

Considering what is established about t_1 above, now by performing induction on t_2 as well, in the case where it is an uncaught exception $raise(v_2)$, we would need to prove that both $(\lambda x.a)(raise(v_2))$ and $raise(raise(v_2))$. The first one is trivial with the above rule, but the second one is not a value, an uncaught exception, or reducible in any way. As we can see, it is a perfectly correct step in the induction, as *raise* is clearly a value of the correct type for term 1 of an application, and $raise(v_2)$ is an ordinary uncaught exception, yet there is no rule to reduce this expression any further. It is unclear why the authors of the initial paper restricted the above rule to only apply to lambda expressions, but clearly this is insufficient and also means that their claimed proof is invalid. Consequently, the rule was changed to that which is currently present in section 2.1, as well as the implementation, namely:

$$v_1(raise(v_2)) \Rightarrow raise(v_2)$$

In this form, the rule now has sufficient power in order to properly reduce both $(\lambda x.a)(raise(v_2))$ and $raise(raise(v_2))$ to $raise(v_2)$, thus proving that in both cases, $\exists\ raise(v_2)\ s.t.\ t_1(t_2) \Rightarrow raise(v_2)$. All other cases can be solved straightforwardly by applications of the reduction cases.

Due to time constraints, this is all that has been fully proven at present. In order to prove type soundness, both progress and subject reduction will need to be proven, and only a small start towards subject reduction has been made.

Chapter 4

Conclusion

Based on a system proposed in the Leroy and Pessaux [2] paper mentioned throughout this report, this project implemented a cut-down version of said system. The focus was on the way it could be used as an extension to standard ML polymorphic type inference in order to infer the most general derivable effect of expressions, specifically effects concerning possible exceptions that might be raised during runtime.

The original system got much of its power from the annotation of all of its base types and the function type with rows. These annotations were used in order to increase the accuracy of the analysis carried out, by indicating the values a term may evaluate to and the effects that a function application may cause.

In order to be able to both implement the system and attempt a proof of soundness, simplifications were made. These simplifications mostly revolved around the removal of the parametrized exceptions, but the number of constant exceptions was also restricted to two. Integer effects were removed as well, since they served no more purpose in a system focused on exception effects once parametrized exceptions were removed. The new system was then adapted and implemented in the Coq proof assistant.

The experiments performed and described in earlier chapters illustrate that the implemented type and effect system still has considerable power and expressiveness, even after having been significantly cut-down. Much of this power comes, as mentioned above, from the row annotations working together with the kinds and kinding rules. Due to the nature of the system, being a fragment of ML focused on a specific task, most experiments and examples come across as artificial, so considering the applicability of an extension of the described system to an actual programing language could be an interesting thought experiment.

Towards the end-goal of proving soundness, only the halfway point of proving progress has been reached, with only a small start having been made towards subject reduction, the other half of the soundness proof. Even so, some problems with the original system have been discovered and fixed, as their original proof of progress was discovered to be incomplete. It remains to be seen if other changes need to be made in order to complete the proof of soundness, but on initial inspection, it seems like the Coq implementation is a suitable with respect to the proposed system, and could work as is.

Seeing as this is the first part of a two-year project, the immediate goal for next

year will be to finish the proof of soundness as soon as possible. Following that, several extensions will be considered. Firstly, the system might be adapted so as to revert the simplification made and once again include parametrized exceptions and integer constants. This would then allow for the proof of soundness to be extended to the entire system presented by Leroy and Pessaux [2], while also removing some of the previous limitations imposed on its expressiveness and power. Additionally, the authors suggest in their paper how it would be possible to extend the system to the entire ML language. Out of those possible extensions, most feasible to implement within the scope of this project are tuples, records and mutable data structures, with the aim of proving that soundness still holds in this new augmented system, once they have been added. This is in no way mutually exclusive with the first extension, and, if time allows it, a combined system would be the most desirable outcome. With respect to the implementation of records and mutable data structures, we refer especially to a paper by Flanagan [1], which offers detailed descriptions on how this might be done. In the author's own words, "extensions to this project would aim to prove the soundness of the augmented exception effect system, the storage effect system and the combined effect system", which perfectly suits our intentions.

Appendix A

Implementation Code

```
Inductive id : Type :=
  | Id : string -> id.

Notation raise := (Id "raise").

Inductive exn : Type :=
  | C1 : exn
  | C2 : exn.

Definition beq_id x y :=
  match x,y with
    | Id n1, Id n2 => if string_dec n1 n2 then true else false
  end.

Definition total_map (A:Type) := id -> A.

Definition t_empty {A:Type} (v : A) : total_map A :=
  (fun _ => v).

Definition t_update {A:Type} (m : total_map A)
                    (x : id) (v : A) :=
  fun x' => if beq_id x x' then v else m x'.

Definition not (P:Prop) := P -> False.
```

Figure A.1: Useful Definitions

```
Inductive tm : Type :=
  | tvar : id -> tm
  | tnat : nat -> tm
  | tabs : id -> tm -> tm
  | tapp : tm -> tm -> tm
  | tlet : id -> tm -> tm -> tm
  | tmatch : tm -> pat -> tm -> id -> tm -> tm
  | texn : exn -> tm
  | ttry : tm -> id -> tm -> tm
  | tsucc : tm -> tm
  | tpred : tm -> tm
  | tmult : tm -> tm -> tm.
```

Figure A.2: Terms

```
Inductive pat : Type :=
  | pvar : id -> pat
  | pnat : nat -> pat
  | pexn : exn -> pat.

Inductive value : tm -> Prop :=
  | v_abs : forall x t1,
      value (tabs x t1)
  | v_nat : forall n1,
      value (tnat n1)
  | v_exn : forall e,
      value (texn e)
  | v_raise : value (tvar raise)
  .
```

Figure A.3: Patterns and Values

```
Fixpoint tsubst (x:id) (s:ty) (T:ty) : ty :=
  match T with
  | TVar y =>
      if beq_id x y then s else T
  | TNat => TNat
  | TExn rho => TExn rho
  | TArrow tau1 rho tau2 => (TArrow (tsubst x s tau1) rho (tsubst x s tau2))
  end.
Fixpoint substrow (x:id) (s:row) (r:row) : row :=
  match r with
  | rvar y =>
      if beq_id x y then s else r
  | rel e r' => (rel e (substrow x s r'))
  end.
Fixpoint rsubst (x:id) (s:row) (T:ty) : ty :=
  match T with
  | TVar y => TVar y
  | TNat => TNat
  | TExn r => (TExn (substrow x s r))
  | TArrow tau1 rho tau2 => (TArrow (rsubst x s tau1) (substrow x s rho) (rsubst x s tau2)
  end.
Fixpoint substanot (x:id) (s:anot) (r:row) : row :=
  match r with
  | rvar y => rvar y
  | rel (elem ex pre) r' => (rel (elem ex pre) (substanot x s r'))
  | rel (elem ex (avar y)) r' =>
      if beq_id x y then (rel (elem ex s) (substanot x s r')) else (rel (elem ex (avar y))
  end.
Fixpoint asubst (x:id) (s:anot) (T:ty) : ty :=
  match T with
  | TVar y => TVar y
  | TNat => TNat
  | TExn r => (TExn (substanot x s r))
  | TArrow tau1 rho tau2 => (TArrow tau1 (substanot x s rho) tau2)
  end.
```

```
Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar y =>
      if beq_id x y then s else t
  | tabs y t1 =>
      tabs y (if beq_id x y then t1 else (subst x s t1))
  | tapp t1 t2 =>
      tapp (subst x s t1) (subst x s t2)
  | tnat n =>
      tnat n
  | tsucc t1 =>
      tsucc (subst x s t1)
  | tpred t1 =>
      tpred (subst x s t1)
  | tmult t1 t2 =>
      tmult (subst x s t1) (subst x s t2)
  | tlet y t1 t2 =>
      tlet y (subst x s t1) (if beq_id x y then t2 else (subst x s t2))
  | tmatch t1 p t2 y t3 =>
      tmatch (subst x s t1) p (subst x s t2) y
        (if beq_id x y then t3
                       else (subst x s t3))
  | ttry t1 y t2 =>
      ttry (subst x s t1) y
        (if beq_id x y then t2
                       else (subst x s t2))
  | texn e => texn e
  end.

Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).
```

Figure A.4: Substitution

```
Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm -> tm -> Prop :=
  | ST_AppAbs : forall x t1 v2,
         value v2 ->
         (tapp (tabs x t1) v2) ==> [x:=v2]t1
  | ST_AbsRaise : forall x t1 v2,
         value v2 ->
         (tapp (tabs x t1) (tapp (tvar raise) v2)) ==> (tapp (tvar raise) v2)
  | ST_App1 : forall t1 t1' t2,
         t1 ==> t1' ->
         (tapp t1 t2) ==> (tapp t1' t2)
  | ST_AppRaise : forall v1 t2,
         value v1 ->
         tapp (tapp (tvar raise) v1) t2 ==> (tapp (tvar raise) v1)
  | ST_AppRaise2 : forall v1 v2,
         value v1 ->
         value v2 ->
         tapp v1 (tapp (tvar raise) v2) ==> (tapp (tvar raise) v2)
  | ST_App2 : forall v1 t2 t2',
         value v1 ->
         t2 ==> t2' ->
         (tapp v1 t2) ==> (tapp v1 t2')
  | ST_Succ1 : forall t1 t1',
       t1 ==> t1' ->
       (tsucc t1) ==> (tsucc t1')
  | ST_SuccNat : forall n1,
       (tsucc (tnat n1)) ==> (tnat (S n1))
  | ST_SuccRaiseV : forall v1,
         value v1 ->
         tsucc (tapp (tvar raise) v1) ==> (tapp (tvar raise) v1)



  | ST_Pred : forall t1 t1',
       t1 ==> t1' ->
       (tpred t1) ==> (tpred t1')
  | ST_PredNat : forall n1,
       (tpred (tnat n1)) ==> (tnat (pred n1))
  | ST_PredRaiseV : forall v1,
         value v1 ->
         tpred (tapp (tvar raise) v1) ==> (tapp (tvar raise) v1).
  | ST_Mult1 : forall t1 t1' t2,
       t1 ==> t1' ->
       (tmult t1 t2) ==> (tmult t1' t2)
  | ST_Mult2 : forall v1 t2 t2',
       value v1 ->
       t2 ==> t2' ->
       (tmult v1 t2) ==> (tmult v1 t2')
  | ST_MultNats : forall n1 n2,
       (tmult (tnat n1) (tnat n2)) ==> (tnat (mult n1 n2))
  | ST_MultRaise1V : forall v1 t2,
         value v1 ->
         tmult (tapp (tvar raise) v1) t2 ==> (tapp (tvar raise) v1)
  | ST_MultRaise2V : forall t1 v2,
         value v2 ->
         tmult t1 (tapp (tvar raise) v2) ==> (tapp (tvar raise) v2)
```

```
  | ST_Let1 : forall x t1 t1' t2,
    t1 ==> t1' ->
    tlet x t1 t2 ==> tlet x t1' t2
  | ST_Let2 : forall x v1 t2,
    value v1 ->
    tlet x v1 t2 ==>  [ x := v1 ] t2 (*subst x v1 t2*)
  | ST_LetRaise : forall x v1 t2,
    value v1 ->
    tlet x (tapp (tvar raise) v1) t2 ==> (tapp (tvar raise) v1)
  | ST_MatchSt : forall t1 t1' p t2 x t3,
      t1 ==> t1' ->
      tmatch t1 p t2 x t3 ==> tmatch t1' p t2 x t3
  | ST_MatchRaise : forall v1 p t2 x t3,
      value v1 ->
      tmatch (tapp (tvar raise) v1) p t2 x t3 ==> (tapp (tvar raise) v1)
  | ST_MatchVar : forall v1 p t2 x t3,
      value v1 ->
      tmatch v1 (pvar p) t2 x t3 ==> [ p := v1 ] t2





  | ST_MatchExn : forall e t2 x t3,
      tmatch (texn e) (pexn e) t2 x t3 ==> t2
  | ST_MatchNat : forall n t2 x t3,
      tmatch (tnat n) (pnat n) t2 x t3 ==> t2
  | ST_MatchNot : forall v1 p t2 x t3,
      value v1 ->
      tmatch v1 p t2 x t3 ==> [ x := v1 ] t3
  | ST_TrySt : forall t1 t1' x t2,
      t1 ==> t1' ->
      ttry t1 x t2 ==> ttry t1' x t2
  | ST_TryRaise : forall v1 x t2,
      value v1 ->
      ttry (tapp (tvar raise) v1) x t2 ==> [ x := v1 ] t2
  | ST_Try : forall v1 x t2,
      value v1 ->
      ttry v1 x t2 ==> v1

where "t1 '==>' t2" := (step t1 t2).

Inductive multi {X:Type} (R: relation X) : relation X :=
  | multi_refl  : forall (x : X), multi R x x
  | multi_step : forall (x y z : X),
                    R x y ->
                    multi R y z ->
                    multi R x z.

Notation " t '==>*' t' " := (multi step t t') (at level 40).
```

Figure A.5: Reduction Rules

```
Inductive anot : Type :=
  | pre : anot
  | avar : id -> anot.

Inductive el : Type :=
  | elem : exn -> anot -> el.

Inductive row : Type :=
  | rvar : id -> row
  | rel : el -> row -> row.

Inductive kind : Type :=
  | EXN : (list exn) -> kind.

Inductive ty : Type :=
  | TVar   : id -> ty
  | TArrow : ty -> row -> ty -> ty
  | TNat   : ty
  | TExn   : row -> ty.

Inductive ts : Type :=
  | TScheme : (list id) -> (list id) -> (list id) -> ty -> ts.
```

Figure A.6: Type Algebra

```
Definition Kmap := total_map kind.

Reserved Notation "r '\::' K \in Kmap" (at level 40).

Inductive has_kind : row -> kind -> Kmap -> Prop :=
  (* Kinding rules *)
  | K_K : forall rho (Kmap:Kmap) K,
      (Kmap rho) = K ->
      (rvar rho) \:: K \in Kmap
  | K_Exn : forall C S phi pi (Kmap:Kmap),
      (not (In C S)) ->
      phi \:: (EXN (C::S)) \in Kmap ->
      (rel (elem C pi) phi) \:: (EXN S) \in Kmap

where "r '\::' K '\in' Kmap" := (has_kind r K Kmap).

Reserved Notation "ty 'wf' Kmap" (at level 40).

Inductive well_formed : ty -> Kmap -> Prop :=
  | WF_Var : forall alpha (Kmap:Kmap),
    (TVar alpha) wf Kmap
  | WF_Nat : forall (Kmap:Kmap),
    TNat wf Kmap
  | WF_Exn : forall phi (Kmap:Kmap),
    phi \:: (EXN []) \in Kmap ->
    (TExn phi) wf Kmap
  | WF_Arrow : forall tau1 phi tau2 (Kmap:Kmap),
    tau1 wf Kmap ->
    phi \:: (EXN []) \in Kmap ->
    tau2 wf Kmap ->
    (TArrow tau1 phi tau2) wf Kmap

where "ty 'wf' Kmap" := (well_formed ty Kmap).
```

Figure A.7: Kinding and Well-Formedness Rules

```
Reserved Notation "Kmap '\and' E '|-' t '\:' T '//' row" (at level 40).

Inductive has_type : Kmap -> env -> tm -> ty -> row -> Prop :=
  (* Typing rules for proper terms *)
  | T_Var : forall tau E x phi (Kmap:Kmap),
      tau \leq (apply E x) \with Kmap ->
      phi \:: (EXN nil) \in Kmap ->
      Kmap \and E |- (tvar x) \: tau // phi
  | T_Nat : forall phi n1 E (Kmap:Kmap),
      phi \:: (EXN nil) \in Kmap ->
      Kmap \and E |- (tnat n1) \: TNat // phi
  | T_Abs : forall tau1 tau2 x E phi phi' t (Kmap:Kmap),
      tau1 wf Kmap->
      Kmap \and (EUpdate x (TScheme nil nil nil tau1) E) |- t \: tau2 // phi' ->
      phi \:: (EXN nil) \in Kmap ->
      Kmap \and E |- (tabs x t) \: (TArrow tau1 phi' tau2) // phi
  | T_App : forall t1 t2 tau tau' phi E (Kmap:Kmap),
      Kmap \and E |- t1 \: (TArrow tau' phi tau) // phi ->
      Kmap \and E |- t2 \: tau' // phi ->
      Kmap \and E |- (tapp t1 t2) \: tau // phi
  | T_Let : forall tau tau1 phi E x t1 t2 (Kmap:Kmap),
      Kmap \and E |- t1 \: tau1 // phi ->
      Kmap \and (EUpdate x (Gen tau1 E phi) E) |- t2 \: tau // phi ->
      Kmap \and E |- (tlet x t1 t2) \: tau // phi
  | T_Match : forall t1 t2 t3 tau1 tau2 tau phi E E' p x (Kmap:Kmap),
      Kmap \and E |- t1 \: tau1 // phi ->
      p \; tau1 \=> E'->
      tau1 \- p \~> tau2 \for Kmap ->
      Kmap \and (aconcat E E') |- t2 \: tau // phi ->
      Kmap \and (EUpdate x (TScheme nil nil nil tau2) E) |- t3 \: tau // phi ->
      Kmap \and E |- (tmatch t1 p t2 x t3) \: tau // phi


  | T_Exn : forall phi phi' C E (Kmap:Kmap),
      phi' \:: (EXN (C::nil)) \in Kmap ->
      phi \:: (EXN nil) \in Kmap ->
      Kmap \and E |- (texn C) \: (TExn (rel (elem C pre) phi')) // phi
  | T_Exn2 : forall phi phi' C C' E pi (Kmap:Kmap),
      phi' \:: (EXN (C'::(C::nil))) \in Kmap ->
      phi \:: (EXN nil) \in Kmap ->
      Kmap \and E |- (texn C) \: (TExn (rel (elem C' pi) (rel (elem C pre) phi'))) // phi
  | T_Try : forall tau phi phi1 x t1 t2 E (Kmap:Kmap),
      Kmap \and E |- t1 \: tau // phi1 ->
      Kmap \and (EUpdate x (TScheme [] [] [] (TExn phi1)) E) |- t2 \: tau // phi ->
      Kmap \and E |- (ttry t1 x t2) \: tau // phi
  | T_Succ : forall E t1 phi (Kmap:Kmap),
      Kmap \and E |- t1 \: TNat // phi ->
      Kmap \and E |- (tsucc t1) \: TNat // phi
  | T_Pred : forall E t1 phi (Kmap:Kmap),
      Kmap \and E |- t1 \: TNat // phi ->
      Kmap \and E |- (tpred t1) \: TNat // phi
  | T_Mult : forall E t1 t2 phi (Kmap:Kmap),
      Kmap \and E |- t1 \: TNat // phi ->
      Kmap \and E |- t2 \: TNat // phi ->
      Kmap \and E |- (tmult t1 t2) \: TNat // phi

where "Kmap \and E '|-' t '\:' T '//' row" := (has_type Kmap E t T row).
```

Figure A.8: Typing Rules

```
Reserved Notation "p '\;' tau '\=>' E" (at level 40).

Inductive pat_type : pat -> ty -> env -> Prop :=
  | P_Var : forall x tau,
      (pvar x) \; tau \=> EUpdate x (TScheme [] [] [] tau) EEmpty
  | P_Nat : forall n,
      (pnat n) \; TNat \=> EEmpty
  | P_Exn : forall C pi phi,
      (pexn C) \; (TExn (rel (elem C pi) phi)) \=> EEmpty
  | P_Exn2 : forall C C' pi pi' phi,
      (pexn C) \; (TExn (rel (elem C' pi') (rel (elem C pi) phi'))) \=> EEmpty

where "p '\;' tau '\=>' E" := (pat_type p tau E).

Reserved Notation "tau '\-' p '\~>' tau' '\for' Kmap" (at level 40).

Inductive pat_sub : ty -> pat -> ty -> Kmap -> Prop :=
  | S_Nat : forall n (Kmap:Kmap),
      TNat \- (pnat n) \~> TNat \for Kmap
  | S_Exn : forall C pi pi' phi (Kmap:Kmap),
      (TExn (rel (elem C pi) phi)) \- (pexn C)
      \~> (TExn (rel (elem C pi') phi)) \for Kmap
  | S_Exn2 : forall C pi pi' pi'' phi (Kmap:Kmap),
      (TExn (rel (elem C' pi') (rel (elem C pi) phi'))) \- (pexn C)
      \~> (TExn (rel (elem C' pi') (rel (elem C pi'') phi'))) \for Kmap
  | S_Var : forall x tau tau' (Kmap:Kmap),
      tau' wf Kmap ->
      tau \- (pvar x) \~> tau' \for Kmap

where "tau '\-' p '\~>' tau' '\for' Kmap" := (pat_sub tau p tau' Kmap).
```

Figure A.9: Typing of Patterns and Pattern Subtraction

```
Reserved Notation "t '\leq' ts '\with' Kmap" (at level 40).

Inductive inst : ty -> (option ts) -> Kmap -> Prop :=
  | I_Type : forall tau' tau l1 l2 l3 alpha (Kmap:Kmap),
      (exists tau1, (tau1 wf Kmap /\ tau' \leq
      Some (TScheme l1 l2 l3 (tsubst alpha tau1 tau)) \with Kmap)) ->
      tau' \leq Some (TScheme (alpha::l1) l2 l3 tau) \with Kmap
  | I_Row : forall tau' tau l2 l3 rho (Kmap:Kmap),
      (exists (phi1:row), ( phi1 \:: (Kmap rho) \in Kmap /\ tau' \leq
      Some (TScheme [] l2 l3 (rsubst rho phi1 tau)) \with Kmap)) ->
      tau' \leq Some (TScheme [] (rho::l2) l3 tau) \with Kmap
  | I_Anot : forall tau' l3 delta tau (Kmap:Kmap),
      (exists pi1, (tau' \leq
      Some (TScheme [] [] l3 (asubst delta pi1 tau)) \with Kmap)) ->
      tau' \leq Some (TScheme [] [] (delta::l3) tau) \with Kmap
  | I_Eq : forall tau (Kmap:Kmap),
      tau \leq Some (TScheme [] [] [] tau) \with Kmap

where "t '\leq' ts '\with' Kmap " := (inst t ts Kmap).


Definition Gen (tau:ty) (E:env) (phi:row) : ts :=
  (TScheme
  (removeBound (freeTypeE E) (freeType (TScheme nil nil nil tau)))
  (removeBound (freeRowE E ++ frr phi) (freeRow (TScheme nil nil nil tau)))
  (removeBound (freeAnotE E ++ far phi) (freeAnot (TScheme nil nil nil tau)))
  tau).
```

Figure A.10: Instantiation and Generalization

```
Inductive env : Type :=
  | EEmpty : env
  | EUpdate: id -> ts -> env -> env.

Fixpoint apply (E:env) (x:id) : (option ts) :=
  match E with
  | EEmpty => None
  | EUpdate y T E' =>
      if beq_id x y then (Some T)
                    else (apply E' x)
  end.

Fixpoint aconcat (E1 E2: env) : env :=
  match E2 with
  | EEmpty => E1
  | EUpdate x T E2' => aconcat (EUpdate x T E1) E2'
  end.

Fixpoint freeTypeE (E:env) : (list id) :=
  match E with
  | EEmpty => nil
  | EUpdate x T E' => (freeType T) ++ (freeTypeE E')
  end.

Fixpoint freeRowE (E:env) : (list id) :=
  match E with
  | EEmpty => nil
  | EUpdate x T E' => (freeRow T) ++ (freeRowE E')
  end.

Fixpoint freeAnotE (E:env) : (list id) :=
  match E with
  | EEmpty => nil
  | EUpdate x T E' => (freeAnot T) ++ (freeAnotE E')
  end.
```

Figure A.11: Environments

```
Fixpoint removeId (x:id) (l:list id) : list id :=
  match l with
  | nil => nil
  | y::l' => if (beq_id x y) then removeId x l' else y::(removeId x l')
  end.

Fixpoint removeBound (xs : list id) (l : list id) : list id :=
  match xs with
    | nil => l
    | x::xs' => removeBound xs' (removeId x l)
  end.

Fixpoint removeDuplicates (l1 : list id) (l2 : list id) : list id :=
  match l1 with
    | nil => l2
    | y::l1' => removeDuplicates l1' (y::(removeId y l2))
  end.

Fixpoint ftt (T:ty) : list (id) :=
  match T with
  | TVar x => (x::nil)
  | TArrow tau1 phi tau2 => removeDuplicates ((ftt tau1) ++ (ftt tau2)) []
  | TNat => nil
  | TExn r => nil
  end.

Fixpoint freeType (T:ts) : (list id) :=
  match T with
  | TScheme l1 l2 l3 ty => removeBound l1 (ftt ty)
  end.

Fixpoint frr (r:row) : list id :=
  match r with
  | rvar x => (x::nil)
  | rel el r' => frr r'
  end.
```

```
Fixpoint frt (T:ty) : list id :=
  match T with
  | TVar x => nil
  | TArrow tau1 phi tau2 => removeDuplicates ((frt tau1) ++ (frr phi) ++ (frt tau2)) []
  | TNat => nil
  | TExn r => frr r
  end.

Fixpoint freeRow (T:ts) : (list id) :=
  match T with
  | TScheme l1 l2 l3 ty => removeBound l2 (frt ty)
  end.

Fixpoint far (r:row) : list id :=
  match r with
  | rvar x => nil
  | rel (elem ex (avar x)) r' => (x::(far r'))
  | rel (elem ex pre) r' => far r'
  end.

Fixpoint fat (T:ty) : list id :=
  match T with
  | TVar x => nil
  | TArrow tau1 phi tau2 => removeDuplicates ((fat tau1) ++ (far phi) ++ (fat tau2)) []
  | TNat => nil
  | TExn r => far r
  end.

Fixpoint freeAnot (T:ts) : (list id) :=
  match T with
  | TScheme l1 l2 l3 ty => removeBound l3 (fat ty)
  end.
```

Figure A.12: Functions for free variable manipulation

Appendix B

Examples

```
Notation x := (Id "x").
Notation y := (Id "y").
Notation a := (Id "a").
Notation exn := (Id "exn").
Notation alpha := (Id "alpha").
Notation rho := (Id "rho").

Notation E0 := (EUpdate
                raise
                (TScheme (alpha::nil) (rho::nil) nil
                (TArrow (TExn (rvar rho)) (rvar rho) (TVar alpha)))
                EEmpty).
```

Figure B.1: Testing environment

```
Definition nat_test :=
 (tpred
      (tsucc
        (tpred
          (tmult
            (tnat 2)
            (tnat 4))))).

Example nat_reduces :
  test ==>* tnat 7.
Proof.
  unfold test. normalize.
Qed.

Example nat_typechecks :
  exists phi Kmap, Kmap \and E0 |- test \: TNat // phi.
Proof.
  unfold test. exists (rvar x). exists (t_update (t_empty (EXN nil)) x (EXN nil)).
  apply T_Pred. apply T_Succ. apply T_Pred. apply T_Mult.
  - apply T_Nat. apply K_K. unfold t_update. simpl. reflexivity.
  - apply T_Nat. apply K_K. unfold t_update. simpl. reflexivity.
Qed.
```

```
Definition test :=
  tlet
    x
    (tpred (tnat 6))
    (tsucc (tvar x)).

Example reduces :
  test ==>* tnat 6.
Proof. unfold test. normalize. Qed.

Example typechecks :
  exists phi Kmap, Kmap \and E0 |- test \: TNat // (rvar phi).
Proof.
  unfold test. exists (Id "phi").
  exists (t_update (t_empty (EXN nil)) (Id "phi") (EXN nil)).
  eapply T_Let.
    - apply T_Pred. apply T_Nat. apply K_K. unfold t_update. simpl. reflexivity.
    - apply T_Succ. apply T_Var.
      * unfold apply. simpl. unfold Gen. simpl. apply I_Eq.
      * apply K_K. unfold t_update. simpl. reflexivity.
Qed.
```

```
Definition test' :=
  tapp (tvar raise) (texn C2).

Example typechecks2:
  exists tau Kmap,
  Kmap \and E0 |- test' \: tau // (rel (elem C2 pre) (rvar (Id "rho'"))).
Proof.
  exists (TVar (Id "alpha")).
  exists (t_update (t_update (t_empty (EXN nil)) (Id "rho'") (EXN (C2::nil)))
 (Id "rho") (EXN nil)).
  unfold test'. eapply T_App.
  - { apply T_Var.
      - unfold apply. simpl. apply I_Type. exists (TVar (Id "alpha")). split.
        + apply WF_Var.
        + apply I_Row. exists (rel (elem C2 pre) (rvar (Id "rho'"))). simpl. split.
          * unfold t_update. simpl. apply K_Exn.
            unfold not. intros. inversion H. apply K_K. simpl. reflexivity.
          * unfold t_update. simpl. apply I_Eq.
      - apply K_Exn. unfold not. intros. inversion H. apply K_K. simpl. reflexivity.
    }
  - { apply T_Exn.
      - apply K_K. unfold t_update. simpl. reflexivity.
      - apply K_Exn. unfold not. intros. inversion H.
      | apply K_K. unfold t_update. simpl. reflexivity.
    }
Qed.
```

Figure B.2: Simple tests

```
Definition steptest2 :=
  tlet test
    (tabs exn
      (ttry
        (tapp (tvar raise) (tvar exn))
        (x)
        (tmatch
          (tvar x)
          (pexn (C1))
          (tnat 1)
          (y)
          (tapp (tvar raise) (tvar y)))))
    (tapp (tvar test) (texn C2)).

Example step2_reduce :
  steptest2 ==>* (tapp (tvar raise) (texn C2)).
Proof. unfold steptest2. normalize. Qed.

Definition steptest :=
  tlet test
    (tabs exn
      (ttry
        (tapp (tvar raise) (tvar exn))
        (x)
        (tmatch
          (tvar x)
          (pexn (C1))
          (tnat 1)
          (y)
          (tapp (tvar raise) (tvar y)))))
    (tapp (tvar test) (texn C1)).

Example step_reduce :
  steptest ==>* (tnat 1).
Proof. unfold steptest. normalize. Qed.
```

```
Example step_typecheck :
  exists Kmap,
  Kmap \and E0 |- steptest \: TNat // (rel (elem C1 (avar (Id "pi"))) (rvar (Id "rho'"))).
Proof.
  unfold steptest.
  exists (t_update (t_update (t_update (t_empty (EXN nil)) (Id "rho''")
  (EXN (C1::nil))) (Id "rho'") (EXN (C1::nil))) (Id "phi") (EXN (nil))).
  apply T_Let with ((TArrow (TExn (rel (elem C1 pre) (rvar (Id "rho'")))))
  (rel (elem C1 (avar (Id "pi"))) (rvar (Id "rho'"))) (TNat))).
    - apply T_Abs.
        + apply WF_Exn. apply K_Exn. unfold not. intros. inversion H.
          apply K_K. unfold t_update. simpl. reflexivity.
        + apply T_Try with (rel (elem C1 pre) (rvar (Id "rho'"))).
          * { eapply T_App.
              - eapply T_Var.
                + unfold apply. simpl. eapply I_Type. exists TNat.
                  split. apply WF_Nat.
                  eapply I_Row. exists (rel (elem C1 pre) (rvar (Id "rho'"))).
                  split. unfold t_update. simpl. apply K_Exn. unfold not.
                  intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
                  simpl. apply I_Eq.
                + apply K_Exn. unfold not. intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
              - eapply T_Var.
                + unfold apply. simpl. eapply I_Eq.
                + apply K_Exn. unfold not. intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
            }
```

```
        * { eapply T_Match.
            - apply T_Var.
              + unfold apply. simpl. apply I_Eq.
              + apply K_Exn. unfold not. intros. inversion H. apply K_K. reflexivity.
            - apply P_Exn.
            - apply S_Exn.
            - unfold aconcat. simpl. apply T_Nat.
              apply K_Exn. unfold not. intros. inversion H. apply K_K. reflexivity.
            - eapply T_App.
              + apply T_Var.
                * unfold apply. simpl. apply I_Type. exists TNat.
                  split. apply WF_Nat. eapply I_Row.
                  exists (rel (elem C1 (avar (Id "pi"))) (rvar (Id "rho'"))).
                  split. apply K_Exn. unfold not. intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
                  simpl. apply I_Eq.
                * apply K_Exn. unfold not. intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
              + apply T_Var.
                * unfold apply. simpl. apply I_Eq.
                * apply K_Exn. unfold not. intros. inversion H.
                  apply K_K. unfold t_update. simpl. reflexivity.
          }
      + apply K_Exn. unfold not. intros. inversion H.
        apply K_K. unfold t_update. simpl. reflexivity.
  - eapply T_App.
    + apply T_Var.
      * unfold apply. simpl. unfold Gen. simpl.
        apply I_Eq.
      * apply K_Exn. unfold not. intros. inversion H.
        apply K_K. unfold t_update. simpl. reflexivity.
    + eapply T_Exn.
      * apply K_K. reflexivity.
      * apply K_Exn. unfold not. intros. inversion H.
        apply K_K. unfold t_update. simpl. reflexivity.
Qed.
```

Figure B.3: Full derivation from chapter 2.5

Appendix C

Proof of Progress

```
Inductive uncaught : tm -> Prop :=
  | u_exn : forall v,
    value v ->
    uncaught (tapp (tvar raise) v).

Theorem progress : forall t T phi,
     t_empty (EXN nil) \and EEmpty |- t \: T // phi ->
     uncaught t \/ value t \/ exists t', t ==> t'.
Proof with eauto.
  intros t T phi Ht.
  remember EEmpty as Gamma.
  generalize dependent HeqGamma.
  induction Ht; intros HeqGamma; subst.
  - (* T_Var *)
    inversion H.
  - (* T_Nat *)
    right. left...
  - (* T_Abs *)
    right. left...
  - (* T_App *)
    destruct IHHt1; subst...
    + (* t1 is uncaught *)
      destruct IHHt2; subst...
      * (* t2 is uncaught *)
        right. right.
        inversion H; subst. try solve_by_invert.
        exists (tapp (tvar raise) v)...
      * destruct H0.
        { (* t2 is a value *)
          right. right.
          inversion H; subst. try solve_by_invert.
          exists (tapp (tvar raise) v)...
        } (* t2 steps *)
          right. right.
          inversion H; subst. exists (tapp (tvar raise) v)...
```

```
  + { destruct H.
      - (* t1 is a value *)
        destruct IHHt2...
        + (*t2 is uncaught*)
          inversion H; subst; try solve_by_invert.
          * right. right.
            destruct H0; subst. exists (tapp (tvar raise) v)...
          * destruct H0; subst. right. right. exists (tapp (tvar raise) v)...
        + { destruct H0.
            - (*t2 is a value*)
              inversion H; subst; try solve_by_invert.
              * right. right. exists (subst x t2 t0)...
              * left. apply u_exn. assumption.
            - (*t2 is steps*)
              right. right.
              inversion H0 as [t2' Hstp]; subst. exists (tapp t1 t2')...
          }
      - (*t1 steps*)
        inversion H as [t1' Hstp]; subst. right. right. exists (tapp t1' t2)...
    }
- (* T_Succ *)
  right. right.
  destruct IHHt...
  + (* t1 is uncaught *)
    inversion H; subst; try solve_by_invert.
    exists (tapp (tvar raise) v)...
  + destruct H.
    * (* t1 is value *)
      { inversion H; subst; try solve_by_invert.
        - exists (tnat (S n1))...
        - inversion Ht; subst. inversion H1.
      }
    * (* t1 steps*)
      inversion H as [t1' Hstp]; subst; try solve_by_invert.
      exists (tsucc t1')...
```

```
- (* T_Pred *)
  right. right.
  destruct IHHt...
  + (* t1 is uncaught *)
    inversion H; subst; try solve_by_invert.
    exists (tapp (tvar raise) v)...
  + destruct H.
    * (* t1 is value *)
      { inversion H; subst; try solve_by_invert.
        - exists (tnat (pred n1))...
        - inversion Ht; subst. inversion H1.
      }
    * (* t1 steps*)
      inversion H as [t1' Hstp]; subst; try solve_by_invert.
      exists (tpred t1')...
- (* T_Mult *)
  destruct IHHt1...
  + (* t1 is uncaught *)
    destruct IHHt2...
    * (* t2 is uncaught *)
      destruct H; subst; right. right...
    * (* t2 is value*)
      destruct H; subst; right. right...
  + { destruct H.
      - (* t1 is a value *)
        destruct IHHt2...
          * (* t2 is uncaught *)
          destruct H0; subst; right. right...
          * destruct H0...
            + (*t2 is a value*)
              right. right.
              inversion H; subst; try solve_by_invert.
              inversion H0; subst; try solve_by_invert.
              exists (tnat (mult n1 n0))...
              inversion Ht2; subst. inversion H2.
              inversion Ht1; subst. inversion H2.
```

```
            + (*t2 steps*)
              right. right. destruct H0 as [t2' Hstp]. exists (tmult t1 t2')...
        - (*t1 steps*)
          right. right. destruct H as [t1' Hstp]. exists (tmult t1' t2)...
      }
  (* let *)
  - destruct IHHt1...
    * (*t1 uncaught*)
      destruct H. right. right. exists (tapp (tvar raise) v)...
    * { destruct H.
        - (*t1 value*)
          right. right. exists (subst x t1 t2)...
        - (*t1 steps*)
          right. right. destruct H as [t1' Hstp]. exists (tlet x t1' t2)...
      }
  (* match *)
  - destruct IHHt1...
    * (*t1 uncaught*)
      destruct H1; subst. right. right. exists (tapp (tvar raise) v)...
    * destruct H1.
      + (*t1 value*)
        destruct H1...
      + (*t1 steps*)
        right. right. destruct H1 as [t1' Hstp]. exists (tmatch t1' p t2 x t3)...
  (*exn*)
  - right. left...
  (*try*)
  - destruct IHHt1...
    * (*t1 uncaught*)
      destruct H; subst. right. right. exists (subst x v t2)...
    * destruct H.
      + (*t1 value*)
        right. right. exists t1...
      + (*t1 steps*)
        right. right. destruct H as [t1' Hstp]. exists (ttry t1' x t2)...
Qed.
```

Figure C.1: Proof of Progress

Bibliography

[1] Paul Flanagan. Effect systems for exceptions and mutable variables. Technical report, University of Edinburgh, School of Informatics, June 2003.

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

[3] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 47–57, 1988.

[4] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.

[5] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2017.

[6] Didier Rémy. Type inference for records in a natural extension of ml. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, 1994.

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

[8] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.