Proving the Properties of a System that uses Row Annotations to Represent Effects

Tudor Ferariu

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh
2019

Abstract

This project continues the investigation of 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 [6], and is based on a fragment of the ML language, specialized for exception analysis that does not attempt to capture any other possible effects.

With the intention of proving several properties of the system, an implementation in the Coq proof assistant is provided and described. In the process of proving Type Soundness, several flaws are found, discussed, and partially remedied. A limitation of our proposed implementation comes to light due to some recurring problems within some of the proofs, but believable, partial or completed proofs, are achieved for all important qualities of the system.

Overall the proposed system shows potential, seemingly fulfills its stated goals, and could, if modernized and expanded upon, be applied as an extension to the ML language.

Acknowledgements

Special thanks to Don Sannella for his time spent supervising this project and the immense amount of help provided. I would also like to thank Wen Kokke and Daniel Hillerström, for their previous help with the implementation, as well as Ian Stark, Samuel Lindley and Brian Campbell for their feedback. Finally, a sincere thank you to all my friends and family, who helped me directly or indirectly throughout my studies.

Table of Contents

Chapter 1

Introduction

1.1 Overview

A vast majority of modern programming languages feature statements that generate effects, as well as possibly returning values or results. Function effects include but are not limited to transfer of control, input/output operations, and centrally to this project, the raising of exceptions. In order to handle application errors, exceptions are often used, in conjunction with exception handlers. Once an exception is raised, it will automatically be propagated upwards through the program code, until it encounters the nearest exception handler that is equipped to deal with it, ultimately fully interrupting program execution, if no such handler is met.

The main drawback of using exceptions for error reporting is the fact that it is exceptionally easy to neglect catching, and subsequently handling, the error at the right place during execution. In the case of uncaught exceptions, the ML compiler generates no errors, with faults only manifesting during actual testing. Currently, there exists no method to exhaustively test most programs, not even those of moderate complexity, particularly when conditions for the error being examined are rare or difficult to intentionally reproduce.

One possible solution, which is applied by Java and similar languages, is to implement checked and unchecked exceptions. Here, the compiler requires programmers to declare the specific checked exceptions that might escape a function or method, so that they can be caught at the right place. The accuracy of these declarations can then be statically checked during compliation. Unchecked exceptions on the other hand do not get checked at compile-time, and consist of run-time errors such as Null Pointer and Index out of Bound exceptions, usually the fault of the programmer. These are generally avoidable, but numerous and can occur anywhere within a program, so declaring them for every method would greatly reduce program clarity. While the idea works suitably well for monomorphic, first-order languages, it is entirely incompatible with higher-order polymorphic programming, seen in functional languages such as ML.

Consider the common *map* iterator. It takes a function and a list, and applies the given function to every element of the provided list. The type of such an operation commonly takes the following shape: $(\alpha \to \beta) \to (list\ \alpha) \to (list\ \beta)$. Seeing as the use of *map* is equivalent to multiple applications of its given function $f : (\alpha \to \beta)$, it is clear that exceptions possibly raised by *map* are determined by the nature of f.

Consequently, no fixed set of exceptions E will be sufficient, as the exceptions that might escape *map* are only properly represented by the set of all possible exceptions in our system. This subsequently results in losing all precision for our exception analysis.

"Type-Based Analysis of Uncaught Exceptions" (Leroy & Pessaux, 2000) [6] proposes an alternative, which claims the ability to "infer the effects of escaping exceptions from unannotated ML source code by detecting potentially escaping exceptions as a static debugging problem". We attempt to perform an in-depth analysis of this system in order to assert the validity of its claims. This is accomplished by first implementing a streamlined variant of said system in the Coq proof assistant, and then seeking to prove its most important properties, investigating any interesting aspects found along the way.

1.2 Background

The system proposed by Leroy and Pessaux is based on the effect polymorphism portrayed in (Lucassen & Gifford, 1988) [8], with an added extension provided by (Talpin & Jouvelot, 1994) [13]. Supposedly, it is able to track exception effects that might escape during evaluation, while also providing a better prediction of possible expression values.

This is accomplished through annotating functions that generate effects, with "rows" that represent the very effects which might occur when they are applied. Since the focus is on exception analysis, the system is introduced as a set of inference rules for the type and effect of expressions, that treats a cut-down subset of ML, where the only values are integers. The so called "latent effect", is, in this case, the set of exceptions that might be raised.

We denote functions from τ to τ' with potentially escaping exception φ as $(\tau \xrightarrow{\varphi} \tau')$, so applying such a function to a term of type τ would result in an expression of τ/φ (type τ and effect φ). 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.

So as to maintain the size and scope of the Coq metalanguage implementation within feasible bounds, several other simplifications were carried out. Parametrized exceptions have been removed, in favor of using just two constant exceptions C_1 and C_2. Two is exactly the right number for interesting interactions to arise through juggling the exceptions. Having less would not allow this, while adding more would not result in any significant contribution to the expressiveness of the system. Integers have also been replaced with natural numbers, and their effects removed, since the main use of integer effects was distinguishing between parametrized exceptions, such as $syserror(6)$, and this feature is no longer necessary. Some basic arithmetic operations

were also added, as the original design did not explicitly mention which ones were included.

Following the implementation, proofs for several of the interesting properties of the system were attempted, with various degrees of success, resulting in the discovery of several flaws in the initial design. Solutions to the problems encountered are proposed, but only partially successful, resulting in a thorough investigation of several intriguing aspects of both the original system and the presented implementation for it.

1.3 Contributions

Given that this is the second part of a 2-year project, several aspects of the implementation and proofs have already been accomplished and discussed in a previous paper[3]. Most notably, the initial implementation of the system, first modifications to the reduction rules, and subsequent complete proof of progress were achieved previously. As the overarching goal is shared, there will be some overlap with previous findings, but the focus will be on the new accomplishments.

The main aim is examining, and ultimately trying to prove the properties of, the proposed type and effect system. This focuses around the proofs of Type Soundness (Theorem 1) and Correctness of Exception Analysis (Theorem 3). Generally speaking, the following milestones have been achieved:

- Modified the existing implementations of substitution and the Proof of Progress in order to account for mistakes in the definition of substitution for match expressions, and the environment of the proof, respectively.

- Rephrased the formalisations of several key theorems and lemmas in order to better reflect the desired properties, especially Type Soundness itself.

- Discovered two faults in the proof of the Substitution Lemma. Firstly, the typing rules allowed the value of *raise* to be overwritten, which, in turn, gave rise to a contradiction. Secondly, the original proposed solution and helper Lemmas were found insufficient to obtain the needed result, so an alternative path was proposed.

- Formulated a new property concerning the stability of typing under extended environments (Lemma 4), as part of the proposed alternative proof to the Substitution Lemma.

- Added a new subset environment relationship and several properties of it, necessary for demonstrating the new property mentioned above.

- Made an extension to the implementation of reduction rules in order to account for the requirements of Correctness of Pattern Subtraction (Lemma 8).

- Tackled various other proofs to helper lemmas of Subject Reduction (Theorem 2) and discovered a recurring issue surrounding variable naming, in relation to the intstantiation and generalization rules.

- Achieved partial proofs of Theorem 1 and 3, that are predicated on some sensible assumptions, as well as complete proofs of several other lemmas.
- Discussed and analyzed the overall results.

As can be seen in Chapter 3, quite a few of the lemmas have complete proofs, and all those that do not, are partially proven only because they either run into the aforementioned variable naming issue themselves, or depend on another lemma which does. Given that in-depth investigation of the issue did not result in the discovery of proper counterexamples, it is believed that the problem resides with the metalanguage implementation, rather than the system itself. Regrettably, a clear solution for this has not been found, so lemmas with only this issue have been considered admissible without a complete proof, which we believe is not a major issue.

Additionally, several properties of asymmetric concatenation and subset environments are assumed true and subsequently used, as they are not of significant interest, and straightforward to see, but tedious to prove.

Chapter 2

The Effect System

Note that some parts of this chapter are highly derivative with respect to the first part of this project[3], which was written last year. Much of what was previously written is still relevant to the system at hand, but significant additions have also been made.

2.1 The Source Language

As previously mentioned, the language in (Leroy & Pessaux, 2000) [6] is a cut-down fragment of ML, with a typing system that attempts to also track the possible effects of functions during run-time. Effects can then be inferred with respect to this system as an extension of normal ML polymorphic type inference.

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

Figure 2.1: The Source Language

In this language, with syntax given in figure 2.1, natural numbers and exceptions are the only data types, and we have access to common operations, such as abstraction, application and 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. Addition and other operations are partly implemented and can be added, but seeing as the main focuses of the system are exception analysis and effect inference, two simple operations and a more complex one were considered enough to show the expressiveness of the system with respect to arithmetic. The original paper does not explicitly mention any operations with its integer values, and since we mostly use these in examples, to test typing derivations and term reduction, they are not to be considered particularly relevant.

All terms, from identifiers to exception handlers, are taken directly from (Leroy & Pessaux, 2000) [6], with the caveat that parametrized exceptions have been removed and the only constant exceptions are C_1 and C_2. This was done in order to simplify the implementation and proofs alike, while still leaving enough power to the system for it to be able to express interesting statements, with particular care being taken to preserve the polymorphic aspect of exception analysis.

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 for expressing 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 bound in the E_0 environment that all type derivations start in, with the raising of an exception being represented as the application of this *raise* function to an 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 can done in the following manner:

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

Having *raise* be bound in the environment is an intriguing choice. On one hand, this means there is no need to contend with specific reduction and typing rules for it, which technically reduces the complexity of the overall system structure, but on the other, as we will later see, it also causes some amount of grief when it comes to proving several of the desired properties. More detail on this can be found in sections 3.3 and 3.5.

2.2 Reduction Rules

Originally, the system also included evaluation contexts as a substitute for all the rules necessary in order to evaluate each individual term until it became a value, but here, all of them have been made explicit, seeing as they are essential for the implementation, and subsequent proofs. 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) [15]; 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, C represents either of the two constant exceptions, p represents arbitrary patterns, x is an arbitrary identifier, while y is specifically a variable pattern (functionally the same as an identifier), and $a\{x \leftarrow v\}$ represents binding or substitution of value v to identifier x in term a.

$$
\begin{aligned}
(\lambda x.a)(v) &\Rightarrow a\{x \leftarrow v\} && (1)\\
(\lambda x.a)(raise(v)) &\Rightarrow raise(v) && (2)\\
a_1(a_2) &\Rightarrow a_1'(a_2) \quad \textit{if } a_1 \Rightarrow a_1' && (3)\\
v(a_2) &\Rightarrow v(a_2') \quad \textit{if } a_2 \Rightarrow a_2' && (4)\\
(raise(v_1))(a_2) &\Rightarrow raise(v_1) && (5)\\
v_1(raise(v_2)) &\Rightarrow raise(v_2) && (6)\\
let\ x = a_1\ in\ a_2 &\Rightarrow let\ x = a_1'\ in\ a_2 \quad \textit{if } a_1 \Rightarrow a_1' && (7)\\
let\ x = v\ in\ a &\Rightarrow a\{x \leftarrow v\} && (8)\\
let\ x = raise(v)\ in\ a &\Rightarrow raise(v) && (9)\\
match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow match\ a_1'\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \quad \textit{if } a_1 \Rightarrow a_1' && (10)\\
match\ raise(v)\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow raise(v) && (11)\\
match\ v\ with\ y \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow a_2\{y \leftarrow v\} && (12)\\
match\ C\ with\ C \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow a_2 && (13)\\
match\ n\ with\ n \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow a_2 && (14)\\
match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 &\Rightarrow a_3\{x \leftarrow v\} \quad \textit{if } (p,v)\ undefined && (15)\\
try\ a_1\ with\ x \rightarrow a_2 &\Rightarrow try\ a_1'\ with\ x \rightarrow a_2 \quad \textit{if } a_1 \Rightarrow a_1' && (16)\\
try\ v\ with\ x \rightarrow a_2 &\Rightarrow v && (17)\\
try\ raise(v)\ with\ x \rightarrow a_2 &\Rightarrow a_2\{x \leftarrow v\} && (18)\\
n+1 &\Rightarrow succ(n) && (19)\\
n-1 &\Rightarrow pred(n) && (20)\\
n*m &\Rightarrow mult(n)(m) && (21)\\
(raise(v))+1 &\Rightarrow raise(v) && (22)\\
(raise(v))-1 &\Rightarrow raise(v) && (23)\\
(raise(v))*m &\Rightarrow raise(v) && (24)\\
n*(raise(v)) &\Rightarrow raise(v) && (25)
\end{aligned}
$$

Figure 2.2: The Reduction Rules

The undefined relation is given by:

$$
\begin{aligned}
(p,v)\ undefined \iff & \forall y, p \neq y \wedge \\
& \forall n, (p = n \rightarrow v \neq n) \wedge \\
& \forall C, (p = C \rightarrow v \neq C)
\end{aligned}
$$

In order to achieve the best results, our implementation had to enact some changes upon the original ruleset proposed by Leroy and Pessaux [6]. Firstly, one of the rules for application has been modified, for it 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) appears slightly suspicious. Although it has exactly the same form as the one proposed by Leroy and Pessaux [6], it is inconsistent with the rest of the rules, which seem to use the "call by value" approach. Still, it is the correct choice in this case, because 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 so as 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 3.2.

The next point of interest is the *match* expression. Rules (10) and (11) are the same as seen in (Leroy & Pessaux, 2000) [6], and serve the general purpose of reducing a_1 to a value, and propagating an uncaught exception, respectively. On the other hand, rules (12)-(15) are new, and have been designed in order to bypass the need for a separate pattern matching function, by making the four cases explicit. Previously the same would have been accomplished by the following two rules:

$$\begin{aligned} &match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow f(a_2) \quad if\ f = M(v,p)\ is\ defined \\ &match\ v\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 \Rightarrow a_3\{x \leftarrow v\} \quad if\ M(v,p)\ is\ undefined \end{aligned}$$

together with the pattern-matching function *M(v,p)*:

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

This pattern-matching function 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, instead of dealing with the $M(v,p)$ function, we propose rules (12)-(14), which represent the three possible cases in which the pattern matching function is defined, and rule (15) for when it would not be defined.

Initially[3], it was thought that there was no need to put any restriction on rule (15), as the existence of the other three would cause it not to be used unless neither of the other applied, which essentially guaranteed the property which has now been formalized as (p,v) *undefined*. While it is true that the rule is never incorrectly applied for term reduction and does not affect progress, its unrestricted form proved to pose some issues during the induction necessary for Subject Reduction in section 3.7. The (p,v) *undefined* relation has been named such, mostly due to its relation to the case of the original pattern matching function definition that it replaces. It is rather simple, and purely imposes that rule (15) is never to be applied in a case where (12)-(14) would be required. This is done by enforcing that the pattern p is neither a pattern variable y, nor the same natural number n or exception C present in v.

Finally, we have the three arithmetic operations, which are rather self-explanatory; they only work on natural number constant terms, and call specific, separately defined functions, that do the actual operation. If passed an uncaught exception, these functions will simply continue to propagate the exception. Seeing as these are natural numbers, $pred(0)$ is a special case which returns 0. This is done mostly for the sake of simplicity, but it could easily be adapted in such a way that $pred(0)$ would instead reduce to $raise(C_1)$, which would be an interesting method to organically raise an exception when attempting to execute an illegal operation.

2.3 The Type Algebra

As initially proposed in (Talpin & Jouvelot, 1994) [13], 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 equational constraints on the structure of rows, one of which becomes irrelevant due to the removal of the integer effects to which it pertains. The other attempts to express that the order of elements in a row 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. This, however, is very cumbersome. Consequently, we propose the following alternative, which is easier to follow, although considerably more verbose:

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

Either way, neither version of this invariant is needed in our reduced system, as the only two possible elements are $C_1 : \pi$ and $C_2 : \pi$, which allows for these equational 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 whose type is an instance of $\forall \rho.\ \mathit{nat} \xrightarrow{\rho} \mathit{nat}$ or $\forall \delta, \rho.\ \mathit{nat} \xrightarrow{C_1:\delta;\rho} \mathit{nat}$ at the end of its typing derivation, can not raise any exception, while one with type $\forall \rho.\ \mathit{nat} \xrightarrow{C_1:\mathit{Pre};\rho} \mathit{nat}$ can possibly raise exception C_1.

2.4 Kinding of Rows

Using ideas from previous work on kinds and record types in (Ohori, 1995) [9] and (Rémy, 1994) [12], kinds are added to the system in (Leroy & Pessaux, 2000) [6], 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 results in there being no need for them to be enforced any longer. 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.

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.

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

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 **??**.

2.5 The Typing Rules

As in the original effect system from (Leroy & Pessaux, 2000) [6], 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:

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

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

$$\frac{E \vdash \tau_1 wf \quad E \oplus \{x : \tau_1\} \vdash a : \tau_2/\varphi' \quad \vdash \varphi :: EXN(\emptyset) \quad x \notin Dom(E_0)}{E \vdash \lambda x.a : (\tau_1 \xrightarrow{\varphi'} \tau_2)/\varphi} \ (3)$$

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

$$\frac{E \vdash\ a_1 : \tau_1/\varphi \quad E \oplus \{x : Gen(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi \quad x \notin Dom(E_0)}{E \vdash let\ x = a_1\ in\ a_2 : \tau/\varphi} \ (5)$$

$$\frac{\begin{array}{c} x \notin Dom(E_0) \quad p \notin Dom(E_0) \\ E \vdash\ a_1 : \tau_1/\varphi \quad \vdash p : \tau_1 \Rightarrow E' \quad \vdash \tau_1 - p \rightsquigarrow \tau_2 \quad E \oplus E' \vdash a_2 : \tau/\varphi \quad E \oplus \{x : \tau_2\} \vdash a_3 : \tau/\varphi \end{array}}{E \vdash match\ a_1\ with\ p \rightarrow a_2 \mid x \rightarrow a_3 : \tau/\varphi} \ (6)$$

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

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

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

$$\frac{E \vdash n : nat/\varphi}{E \vdash n + 1 : nat/\varphi} \ (10)$$

$$\frac{E \vdash n : nat/\varphi}{E \vdash n - 1 : nat/\varphi} \ (11)$$

$$\frac{E \vdash n_1 : nat/\varphi \quad E \vdash n_2 : nat/\varphi}{E \vdash n_1 * n_2 : nat/\varphi} \ (12)$$

Typing of patterns:

$$\vdash x : \tau \Rightarrow \{x : \tau\} \ (13)$$

$$\vdash n : nat \Rightarrow \{\} \ (14)$$
$$\vdash C : exn[C : \pi; \varphi] \Rightarrow \{\} \ (15)$$
$$\vdash C : exn[C' : \pi'; C : \pi; \varphi] \Rightarrow \{\} \ (16)$$

Pattern Subtraction:

$$\vdash nat - n \rightsquigarrow nat (17)$$
$$\vdash exn[C : \pi; \varphi] - C \rightsquigarrow exn[C : \pi'; \varphi] (18)$$
$$\vdash exn[C' : \pi'; C : \pi; \varphi] - C \rightsquigarrow exn[C' : \pi'; C : \pi''; \varphi] (19)$$
$$\frac{\vdash \tau' \ wf}{\vdash \tau - x \rightsquigarrow \tau'} \ (20)$$

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

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:

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

Firstly, 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 of the above derivation, this allows us to break row invariant 2, since 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.

Additionally, when compared to our previous implementation[3], new pre-requisites have been added for rules (3), (5) and (6), in the form of restrictions to the variables that the λ-abstraction, *let* and *match* can bind in the environment. Namely, none of the variables that these functions can use should be named the same as those in E_0, which in this case is only *raise*. Obviously, patterns are not only variables, but also natural numbers and exceptions, but defining an entirely new relation for this single case would be superfluous, so the same $\notin$ operator was used. When dealing with patterns, $\notin$ acts exactly as expected for variables, and is true for any environment E in the case

where the pattern is either a natural number or exception, as these are not present in environments in the first place. The full discussion on why this was done is present in section 3.3.

Having discarded integer effects and parametrized exceptions allows us to remove several rules from all three categories, as well as make simplifications to rules (2), (14), and (17). Rules (8), (16), and (19) have been added in the interest of circumventing the need for the equational constraint on rows, mentioned in section 2.3. This is done by taking advantage of the fact that there are only two possible row elements in this reduced system.

Typing rules for the arithmetic operations have also been added, respectively, rules (10), (11) and (12). They are very simple, but nonetheless integral to the system. All 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 most 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 (17)-(20) 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.

2.6 Implementation

The entirety of the implementation is done in the Coq proof solver and is built upon the simply-typed lambda calculus module from "Software Foundations" (Pierce, 2017) [11]. Every rule and definition mentioned in this chapter 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 [15] is imitated through the use of small-step operational semantics, also based on the aforementioned book by Pierce. The relevant implementation code has been attached in Appendices A-B, which is also where all the figures mentioned in this chapter are located.

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 that they pertain to. 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, the three lists are simply left empty, which signifies that there are no variables to be universally quantified for said type.

The precise implementation of type schemes is particularly relevant with respect to instantiation. In our code, instantiation iterates over the bound lists of presence

variables, rows and types, in that order, and substitutes them with the necessary ones to form the desired type from its respective scheme. This causes many of the proofs to be extremely tedious, as we need to contend with inducting on three lists of variables in order to reach the final point of the instantiation judgment from its initial state, especially when all three lists are arbitrarily long. The ordering in which variables are being substituted was chosen so as not to cause certain issues with proving Lemma 10 in Section 3.5, as it used to be different in our previous implementation of this system[3]. Even then, several problems, which are further discussed in Sectons 3.6 and 4.1, still occur. Regardless, this remains our choice of implementation, because it is the most intuitive and correct possible, for the given definition, within the current overall system.

Another problem emerges 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. Given that environments are set up to map identifiers to type schemes, the *raise* term must also be represented by an identifier. This causes naming conflicts in conjunction with the original forms of typing rules (3), (5) and (6), which is discussed in detail in Section 3.3, and represents the very reason for the changes that have been mentioned in Section 2.5. Having special rules for *raise* would have probably been preferable, since binding it in the environment causes more harm than good, but in the interest of faithfulness to the original design, this choice was kept.

A major hurdle was the K function that maps row variables to rows, mentioned in section 2.4. 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 2.5, since it is the metalanguage of Coq that requires this compromise, rather than it being a fault with the system itself.

An example of this can be seen here, for the simpler case of incrementing a number, as well as in figure A.8 for the entirety of the rule-set. They are written as:

$$T\_Succ : \forall E, a_1, \varphi, Kmap.$$
$$Kmap \;\&\; E \vdash a_1 : TNat \;/\; \varphi \rightarrow Kmap \;\&\; E \vdash tsucc(a_1) : TNat \;/\; \varphi$$

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 so as 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:

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

Many of the rules do not need to use *Kmap* themselves, as they do not present any kinding judgments as premises, nonetheless, it is necessary for it to be passed to all subterms throughout the derivation, for those that will inevitably need to apply it in some fashion. Since the typing rules preserve effect φ, in at least one of their components, one or more kinding judgments using *Kmap* must occur, for all successful derivations

In order to cope with some of the new requirements presented by Lemmas 2 and 4, a new relationship was also implemented:

Subset Environments:

$$E_1 \subset E_2 \iff \forall x, ((\exists \sigma, E_1(x) = \sigma) \rightarrow E_1(x) = E_2(x))$$

Recall that σ represents an arbitrary type scheme. Thus, $E_1 \subset E_2$ if all variables that are bound in E_1 are also bound to the same type scheme in E_2. Of course, E_2 can contain any number of other bindings, as long as their respective variables are not present in E_1. The precise reason and use of this relation will be expanded upon in Section 3.3. We also define the following properties:

$$E \subset E \qquad (Symmetry)$$
$$E_1 \subset E_2 \wedge x \notin Dom(E_1) \rightarrow E_1 \subset E_2 \oplus \{x : \sigma\} \qquad (Extend\ E_2)$$
$$E_1 \subset E_2 \wedge x \notin Dom(E_2) \rightarrow x \notin Dom(E_1) \qquad (Not\ in\ Dom(E_1))$$
$$E_1 \subset E_2 \rightarrow E_1 \oplus \{x : \sigma\} \subset E_2 \oplus \{x : \sigma\} \qquad (Extend\ Both)$$

All these properties are evident by the definition of subset environments and have been admitted without proof, since they are not particularly interesting and effort was best directed elsewhere.

Of minor note is also the fact that kinds κ should be exceptions stored in a set, but for simplicity, our code simply stores them in a list, and the constraints of sets are applied afterwards. Namely, the two properties of sets which are important to us:

1. Elements can not be duplicated.

2. The order of elements does not matter.

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 perform substitution of terms, types, rows 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 2.5. Some minor flaws from the previous iteration[3] were found and corrected, but most are not worth mentioning, with the exception of the incorrect substitution on match terms, which was faulty and previously allowed a bound variable to be overwritten.

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), instantiation (figure A.10) and subset environments (figure A.13).

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 the interest of facilitating such things as 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.

2.7 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, a battery of tests was run on several expressions, so as to ascertain whether or not they reduce and type-check correctly. This included, most notably, a bigger derivation, discussed in more detail during our previous iteration of this system[3], which uses all of the important rules at least once and thus serves as a good way to evaluate overall system performance.

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

The expression itself can be seen above, as well as alongside the code for all other examples, in Appendix figures B.2 and B.3. While the proof of reduction was quite simple in all 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 other 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 processes all the premises from left to right also does little to help. 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 $C_1 : Pre; \varphi'$.

Chapter 3

Proofs

The code for all proofs in this chapter can be found in Appendix C, and a compendium containing all Lemma definitions in Appendix D.

3.1 Type Soundness

Seeing as the system we are examining aims to infer the effect of expressions as a part of standard type inference, the critical theorem to prove in order to validate the authors' claims is Type Soundness. We aim to do this in the same way that Wright and Felleisen[15] proposed, by using the Progress Lemma and Subject Reduction Theorem, of which Type Soundness is an immediate corollary.

The formulation of Type Soundness offered by the original paper is:

THEOREM (TYPE SOUNDNESS). *Let a be a complete program. Assume* $E_0 \vdash a : \tau/\varphi$. *If* $a \overset{*}{\Rightarrow} a'$ *and* a' *is in normal form with respect to the reduction rules, then* a' *is either a value v or an uncaught exception raise(v).*

Note that $\overset{*}{\Rightarrow}$ simply represents multiple applications of the reduction operation. Unfortunately, this is both hard to follow, and does not seem to properly express the desired property, so instead we propose the following:

THEOREM 1 (TYPE SOUNDNESS). *If* $E_0 \vdash a : \tau/\varphi$, *then either a is an uncaught exception raise(v), a is a value v, or* $\forall a'$, $a \Rightarrow a'$ *implies* $E_0 \vdash a' : \tau/\varphi$.

This serves to say that any typed term is either a value, an uncaught exception, or can step to a term of the same type and effect. Provided that the term eventually steps to a final form that can no longer be reduced, it is also guaranteed to be either a value or uncaught exception. The advantage of our version is that it also covers the case for non-terminating terms, such as $(\lambda x.xx)(\lambda x.xx)$ which reduces to itself according to reduction rule (1) The implemented formalization can be see in figure C.1.

As this is a corollary, its proof is trivial provided that the aforementioned two requirements of Progress and Subject Reduction are in place, which is where some problems start showing.

3.2 Progress

The proof of progress was attempted first, and it is identical to that from (Leroy & Pessaux, 2000) [6], as stated below. Note that this section is almost identical to its iteration in the previous paper[3].

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 being either further reducible, values, or uncaught exceptions. The proof is tackled by structural induction on the typing judgment, as can be seen in figure C.2.

The proof itself proceeds quite smoothly for all the unmodified cases, as well as the added arithmetic operations and modified *match* expression rules. Several additions to the proof have been made since the last version[3]. Most are due to the inclusion of the (p, v) $undefined$ relation to the final *match* rule, (15), which require a rather verbose, but otherwise uncomplicated extension for that particular case. Furthermore, it was noticed that the original proof had been done in the *empty* environment, while it is required to hold for E_0; this has been quickly amended.

A problem arose, however, when dealing with the application term. Here, let us consider $a_1(a_2)$ the application of term 1 to term 2. Since this is an inductive proof, we iterate through all possible combinations of a_1 and a_2, with them being, as the lemma implies, an uncaught exception, a value, or there existing some a' that they step to.

First, we consider if a_1 is an uncaught exception $raise(v)$, in this case, regardless of what form a_2 takes, we can apply substitution rule (5) in order to reduce $(raise(v))(a_2)$ to $raise(v)$, thus, $\exists.raise(v),\ s.t.\ (raise(v))(a_2) \Rightarrow raise(v)$. This perfectly fine, but as mentioned in section 2.1, it was thought that rule (5) was inconsistent with the "call by value" nature of the other rules, prompting a subsequent change to the following:

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

This proved to be incorrect, as in the above step, once a_1 is an uncaught exception $raise(v_1)$, we have to perform induction on a_2 as well. In the cases where it is either a value v, or $\exists\, a_2'\ s.t.\ a_2 \Rightarrow a_2'$, 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.

Continuing the induction, when we have that a_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 a_1 above, by performing induction on a_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 $raise(raise(v_2))$ fulfil the required criteria. The first

is trivial with the above rule, but the second 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, since *raise* is clearly a value of the correct type for the first term 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, $\exists\ raise(v_2)\ s.t.\ a_1(a_2) \Rightarrow raise(v_2)$. All other cases can be solved straightforwardly by applications of the reduction cases. This gives us "half" of what we need for the proof of Soundness.

3.3 Substitution Lemma

Before we can even begin to tackle Subject Reduction though, we must prove the Substitution Lemma, which is an integral part of it, and turned out to be the most significant hurdle in the entire process. Substantial additions have been made to the original variant proposed by (Leroy & Pessaux, 2000)[6], and it is now stated as:

LEMMA 2 (SUBSTITUTION LEMMA). *Assuming* $E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\} \vdash a : \tau/\varphi$, *and for a value* v, $E_0 \vdash v : \tau'/\varphi$, *if* $E_0 \subset E$ *and* $x \notin E_0$, *then* $E \vdash a\{x \leftarrow v\} : \tau/\varphi$

Previously, the constraints $E_0 \subset E$ and $x \notin E_0$ were not included, and the reasons for their addition will soon become apparent.

The proof is done by structural induction on a, but the base case, $a = x'$ is immediately problematic. If $x' \neq x$, then the proof is straightforward, but in the case where $x' = x$, our desired outcome becomes $E \vdash x\{x \leftarrow v\} : \tau/\varphi$ which is equivalent after substitution with $E \vdash v : \tau/\varphi$. There are numerous probems when trying to show that $E \vdash v : \tau/\varphi$ only from our two assumptions:

$$E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\} \vdash x : \tau/\varphi \quad and \quad E_0 \vdash v : \tau'/\varphi$$

Firstly, there is no inductive hypothesis to make use of in this case. Secondly, this is blatantly untrue if we allow $x \in E_0$. Consider the following two assumptions:

$$E_0 \oplus \{raise : nat\} \oplus \{x : \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha\} \vdash x : exn[\varphi] \xrightarrow{\varphi} \tau'/\varphi$$

$$E_0 \vdash raise : exn[\varphi] \xrightarrow{\varphi} \tau'/\varphi$$

We could apply the Substitution Lemma, with $E = E_0 \oplus \{raise : nat\}$, on these two typing judgments and obtain

$$E_0 \oplus \{raise : nat\} \vdash raise : exn[\varphi] \xrightarrow{\varphi} \tau'/\varphi$$

simply by substituting our value, *raise*, into our term x. But by application of the first typing rule, and by the definition of asymmetric concatenation in environment $E_0 \oplus \{raise : nat\}$, *raise* should have type *nat*, not $exn[\varphi] \xrightarrow{\varphi} \tau'$. This is a contradiction, which proves that, in the form that it was originally written, the Substitution Lemma does not stand. At least not as long as overwriting variables bound in E_0 is allowed, something that is never explicitly prohibited in the original system. For this very reason, the $x \notin E_0$ constraint was added as a requirement to the Lemma, as well as a pre-requisite for the relevant typing rules, that can result in binding variables to the environment.

One might notice that, for the purposes of the proof, $x \notin E_0$ can also be derived from the as-of-yet undiscussed term $E_0 \subset E$, so it is technically redundant. While this is true, obtaining $x \notin E_0$ in this manner is quite laborious, so it was left in, for simplification purposes. Note that it is still very much necessary for $x \notin E_0$ to be made a part of the typing rule requirements, because otherwise the subset relationship wouldn't be guaranteed hold when in the case of three affected rules, since λ-abstractions, let and try expressions could overwrite *raise* again.

In order to show that this counterexample can arise from using the system, consider the following typing judgment:

$$E_0 \vdash let\ raise = 5\ in\ \lambda x.(x\, C_1) + 1 : \tau/\varphi$$

The full typing derivation will not be given, but within it, we will obtain the typing judgement for variable x as

$$E_0 \oplus \{raise : nat\} \oplus \{x : \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha\} \vdash x : exn[\varphi] \xrightarrow{\varphi} nat/\varphi$$

from typing it in the first term of application $(x\, C_1)$. We know that it must be a function that takes an $exn[\varphi]$ argument since it is applied to C_1, and also that the resulting type must be *nat* due to the subsequent increment by 1. Row φ here is of the form $C_1 : Pre; \varphi'$, but this is irrelevant to what we are attempting to show and thus the value was not substituted in the judgment in order to keep it more legible. Recalling that $E_0 = \{raise : \forall \alpha, \rho.exn[\rho] \xrightarrow{\rho} \alpha\}$ easily gives us the second assumption needed, about the typing of *raise*, which proves that this can indeed cause a contradiction.

Adding the $x \notin E_0$ restriction to our definition removes this problem and allows us to continue with the proof, but we still need to show that:

$$(E \oplus \{x : \forall \vec{\alpha_i} \vec{\rho_i} \vec{\delta_i}.\tau'\} \vdash x : \tau/\varphi \ \wedge\ E_0 \vdash v : \tau'/\varphi) \rightarrow E \vdash v : \tau/\varphi$$

Here, the first of the two typing judgments in the premise is entirely useless after the substitution is finished, so all we have to prove our desired conclusion is the second one. Leroy and Pessaux[6] claimed to accomplish this by using the following lemma:

LEMMA (TYPING IS STABLE BY SUBSTITUTION). *Let* θ *be a substitution. If* $E \vdash a : \tau/\varphi$, *then* $\theta(E) \vdash a : \theta(\tau)/\theta(\varphi)$.

While it is true that a substitution could allow us to go from $E_0 \vdash v : \tau'/\varphi$ to $E_0 \vdash v : \tau/\varphi$ for the purposes of this proof, there is no substitution that could ever give us

an environment E with more elements than *raise*, when applied to E_0. Since this proof demands the property holds $\forall E$, so E_0 is woefully insufficient for our requirements.

In an attempt to resolve this issue we have searched for a different method that could achieve the desired results within the confines of this system. Firstly, we adapted the previous Lemma as follows:

LEMMA 3 (TYPING IS STABLE BY SUBSTITUTION). *If for a value v,* $E \vdash v : \tau'/\varphi$ *and* $\tau \leq \forall \overrightarrow{\alpha_i}\overrightarrow{\rho_i}\overrightarrow{\delta_i}.\tau'$*, then* $E \vdash v : \tau/\varphi$.

Unfortunately, the proof for Lemma 3 has only been partially completed. While quite tedious to accomplish, this fact should follow easily by induction over the lists of type, row and presence variables universally quantified in the type scheme of τ'. The implementation of instantiation causes some issues here though, since it requires we empty all three lists of variables in the type scheme before obtaining the form of τ. This causes some issues, as we will see when we scrutinize instantiation more carefully in Sections 3.5 and 3.6. Regardless, it should be undeniable that if we know that v can be typed as τ', and τ is an instance of τ', that we can also type v as the less general type τ, so Lemma 3 was accepted as a plausible property without a complete proof in order to allow investigation of further issues.

This permits us, as mentioned previously, to go from $E_0 \vdash v : \tau'/\varphi$ to $E_0 \vdash v : \tau/\varphi$. So as to also take care of the environment mismatch, we pursued the proof of an entirely new lemma, making use of the Subset Environment relationship mentioned in Section 2.6. This is also why the $E_0 \subset E$ requirement was added to the Substitution Lemma. We propose the following:

LEMMA 4 (TYPING IS STABLE UNDER EXTENDED ENVIRONMENT). *If* $E' \vdash a : \tau/\varphi$ *and* $E' \subset E$*, then* $E \vdash a : \tau/\varphi$.

Recall that $E' \subset E$ implies that any variable bound in E' is also bound to the same type scheme in E, with E possibly having other variables bound in it. It stands to reason then, that if we can type term a in environment E', then whatever other variables are bound in E are irrelevant to the derivation, and that the derivation would hold in E' as well. Furthermore, we have already seen what happens if we allow *raise* to be overwritten during derivations, so all environments E arising during derivations should fulfill $E_0 \subset E$, given our new restrictions on typing rules, that prevent this very problem.

The proof of this theorem is attempted by structural induction on a, and proceeds smoothly, until the case where term a is a *let* expression. For achieving this, we need to utilize, among other things the inductive proof pertaining to the second term of the let binding, for which, in turn, we require that:

$$E' \oplus \{x : Gen(\tau_1, E', \varphi)\} \subset E \oplus \{x : Gen(\tau_1, E, \varphi)\}$$

In all other cases we were able to obtain similar relations from the premise $E' \subset E$ together with property (4) of subset environments, which allows us to extend both with an x that is bound to the same type scheme, but this does not appear to be true in this situation. Here, the fact that E' is a subset environment of E works against us, since

E contains more instances, that could, in turn, contain free variables not present in E'. This would then reduce the universally quantified variables in the type scheme on the right side, but not on the left, invalidating our desired relationship.

To bypass this, it was first thought to limit our terms a to values v, since it would then be the case that we do not even need to worry about *let* expressions, and proving this property for values only would still be enough to apply it successfully in the Substitution Lemma proof, where it is needed. Unfortunately, doing this makes the inductive hypothesis too strong, and we can no longer prove the case for lambda abstractions, which are values themselves, but do not guarantee that the term inside them also is.

It would then appear that we have found a contradiction to this proposed Lemma. Surprisingly though, unlike the case for Progress and the initial form of the Substitution Lemma, this snag in the proof did not result in any convincing counterexamples that would actually disprove the validity of the desired property. Several interesting possibilities for counterexamples were considered, but the only ones that showed promise required the system to also contain some combination of if statements, pair types, boolean types and operators such as equality. For any typing judgment that was thought would present this issue, it immediately became apparent that a simple renaming of variables that were not in any way forced to be the same would comfortably remove any contradictions.

This is purely speculation at the moment, as nothing that was attempted has managed to reconcile the contradiction within the proof assistant itself, but it would seem that a situation where such a thing would happen is practically impossible within the current bounds of the system. Whenever a variable that is in E and not in E' could cause an issue with the generalization of τ_1, it can invariably be renamed into one that does not, without compromising the integrity of any parts of the overall typing derivation.

The entire problem originates from the rather old-fashioned methods employed for dealing with variables in our system. At this point, since no definitive solution could be found for either proving or disproving this claim, the particular case for *let* was admitted without a proof in order to continue investigating other properties as well. The rest of the proof for Lemma 4 is smooth after this.

Let us not forget that this was all done so as to deal with the first case of our structural induction on term a within the proof of the Substitution Lemma. To try and circumvent the need of dealing with a generic environment E entirely, an attempt was done to prove a variant of the Substitution Lemma where all environments were E_0 instead, meaning:

LEMMA (SUBSTITUTION LEMMA'). *Assuming* $E_0 \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\} \vdash a : \tau/\varphi$, *and for a value* v, $E_0 \vdash v : \tau'/\varphi$, *if* $E_0 \subset E_0$ *and* $x \notin E_0$, *then* $E_0 \vdash a\{x \leftarrow v\} : \tau/\varphi$

Although not immediately noticeable, this modified lemma would be strong enough to prove the desired properties of Subject Reduction, and would certainly mean not having to deal with Lemma 4 in our first inductive case. Regrettably, it is also the case that the inductive hypothesis is now much too weak to prove any of the other interesting cases, and immediately fails to deal with λ-abstractions, so this idea was quickly

discarded.

It turns out though, that the first case of the Substitution Lemma is by far the most troublesome, as most others are proven without tremendous hassle. Just as in (Leroy & Pessaux, 2000) [6], the *let* case is the second most interesting. Their proposed proof, while satisfactory, is rather over-complicated and has been simplified. The issue arises when dealing with the second term of $a = (letx' = a_1 \ in \ a_2)$, for which we need to use the inductive hypothesis and subsequently prove, assuming $E' = E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\}$, that knowing

$$E' \oplus \{x' : Gen(\tau_1, E', \varphi)\} \vdash a_2 : \tau/\varphi$$

it is also true that:

$$E' \oplus \{x' : Gen(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi$$

The nature of τ_1 pertains to the typing of term a_1, but the specifics of this are irrelevant for the larger proof. What is important is that, by the definition of the Generalization function, all instances x' of $Gen(\tau_1, E', \varphi)$ will also be instances of $Gen(\tau_1, E, \varphi)$. This is a rather obvious property, since environment E' has at least as many free variables as E, possibly more, which will cause the resulting type scheme to always be less general than that arising from E. Seeing as this is not a particularly interesting property, it was assumed true in our system, due to its interaction with the instantiation rule, which is further elaborated on in Section 3.6. Formally we propose the simplified version:

LEMMA 5 (TYPING IS STABLE UNDER MORE GENERAL HYPOTHESIS). *Assuming* $E' = E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\}$. *If* $E' \oplus \{x' : Gen(\tau_1, E', \varphi)\} \vdash a_2 : \tau/\varphi$, *then,* $E' \oplus \{x' : Gen(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi$.

This is much more restrictive than the variant proposed originally, but it accomplishes the same goal in a much more straightforward way. Two other Lemmas were also proven successfully and used to simplify some of the other cases, namely:

LEMMA 6 (TYPING ARE WELL KINDED). *If* $E \vdash a : \tau/\varphi$, *then* $\vdash \varphi :: EXN(\emptyset)$.

LEMMA 7 (VALUES HAVE NO EFFECTS). *Assuming* $\vdash \varphi' :: EXN(\emptyset)$. *For a value* v, *if* $E \vdash v : \tau/\varphi$, *then* $E \vdash v : \tau/\varphi'$.

Both of these proofs are trivial to show by inducting on the typing derivation from the premise, and have been fully completed. It is probably because of the fact that these properties are so simple, that their use has not been mentioned in the original proof of the Substitution Lemma, even though equivalent lemmas exist. Regardless, they can be used here to simplify certain aspects, and are also relevant in several subsequent proofs. The code for all of the relevant lemmas and helper functions can be found in figures C.3-C.7.

3.4 Correctness of Subtraction

Aside from the Substitution Lemma, we also need another major lemma in order to deal with the *match* rule and the way in which its typing judgment behaves with respect to patterns.

LEMMA 8 (CORRECTNESS OF SUBTRACTION). *For a value v, if $E \vdash v : \tau/\varphi$, $\vdash \tau - p \rightsquigarrow \tau'$, and (p, v) undefined, then $E \vdash v : \tau'/\varphi$.*

Essentially, this Lemma, found in figure C.8, assures us of the proper typing judgment needed for the third term of the *match* expression, when in the case that the patterns did not match. It is here that we need the (p, v) *undefined* relation to carry the necessary information about pattern p from from its appropriate reduction rule to the typing environment. Knowledge about the connection between value v and pattern p is mandatory in this case, as we need to obtain relevant information of type τ' from the pattern subtraction judgment.

This is required only for reduction rule (15), but is crucial for maintaining the desired functionality of our system, namely in relation to exception analysis, which is done via a combination of the *try* and *match* rules. The proof is performed by induction and subsequent case analysis on pattern p. While the cases for variable patterns and natural numbers are straightforward, and we do not need to deal with parametrized exceptions, the typing rule that was added to simplify the row invariants slightly complicates taking care of the constant exceptions. Firstly, we need the help of another lemma, namely:

LEMMA 9 (SHAPE OF VALUES BY TYPE). *For a value v, if $E \vdash v : \tau/\varphi$ and $\tau = exn[\varphi]$, then $v = C$ and C belongs to φ.*

The original version of this theorem included extra cases for integers, which are no longer needed due to lack of integer effects, as well as arrow types, the result for which is simply never used in any of the other proofs. The name was kept for the simplified version though, so as to better relate Lemma 9 to the source material, since it still fulfills the same role. Simple case analysis on the possible values v reveals that the result holds, without much hassle.

Of note is the fact that saying *C belongs to* φ means that whichever constant exception is contained within v is also present in row φ. Sadly, due to the simplifications made initially in order to more easily represent our only two constant exceptions within rows, the formalization of this property is rather cumbersome, but is otherwise perfectly fine. This can be seen in the code from figure C.9.

Now we can, within the confines of Lemma 8, do case analysis on the two possible constant exceptions C_1 and C_2, followed in both by a further case analysis on the typing rule used. The desired result is easily obtained with the help of Lemma 9.

3.5 Raise Value Has Any Type

The third most important lemma needed is the following:

LEMMA 10 (RAISE VALUE HAS ANY TYPE). *For a value v, if* $E_0 \vdash raise\ v : \tau/\varphi$ *and* $\vdash \tau'\ wf$*, then* $E_0 \vdash raise\ v : \tau'/\varphi$.

This is required in order to deal with all the cases of Subject Reduction where we are simply propagating an uncaught exception, since the type of an uncaught exception can be anything. It should be clear, given that *raise* is bound to $\forall\alpha,\rho.exn[\rho] \xrightarrow{\rho} \alpha$ in E_0, that no matter what exception is contained in v, the resulting type of the overall uncaught exception will be whatever the larger context demands through other typing judgments, as long as that type is well formed.

The proof itself is rather straightforward and follows by case analysis on value v, and while it seems self-evident at first, there is one small problem with our previous implementation[3]. When typing term *raise v*, due to our rules and implementation, we need to go through the following simplified steps:

$$E_0 \vdash raise\ v : \tau_2/\varphi\ (1)$$

$$E_0 \vdash raise : \tau_1 \xrightarrow{\varphi} \tau_2/\varphi\ (2)$$

$$\tau_1 \xrightarrow{\varphi} \tau_2 \leq \forall\alpha,\rho.exn[\rho] \xrightarrow{\rho} \alpha\ (3)$$

$$\tau_1 \xrightarrow{\varphi} \tau_2 \leq \forall\rho.(exn[\rho] \xrightarrow{\rho} \alpha)\{\alpha \leftarrow \tau_2\}\ (4)$$

$$exn[\varphi] \xrightarrow{\varphi} \tau_2 \leq \forall\emptyset.(exn[\rho] \xrightarrow{\rho} \tau_2)\{\rho \leftarrow exn[\varphi]\}\ (5)$$

$$exn[\varphi] \xrightarrow{\varphi} \tau_2 = exn[\varphi] \xrightarrow{\varphi} \tau_2\ (6)$$

Within the transition from step (5) to step (6) lies a complication. Here, τ_2 can be anything, a surrogate for the τ' in the lemma above, otherwise we would not be able to obtain the desired proof. At the point where we are doing substitution in step (5), ρ could be a free variable in τ_2, which would cause the substitution to give us some other result than (6), and making our proof impossible.

While it is true that the desired property should hold for "any" τ_2, no counterexample arises from this, because we can always rename the variable ρ in the type scheme of *raise* to something else, without compromising anything else in the derivation. There will never be a case where the derivation enforces the same variable to be used in the definition of *raise* as well as in some other type scheme, without being able to freely rename one of them and resolve the collision.

This problem is a combination between the system specification and the implementation chosen, as the use of α and ρ is simply a naming conceit. Notice that this is very similar to the problem discussed during the proof of Lemma 4, but this time we have a solution. Technically, preventing these two identifiers from being used in any type scheme other than *raise* would be a possibility, this is in fact what we did for *raise* itself, but attempting to enforce this in the Coq proof assistant is a logistic nightmare, if at all possible.

Instead, we slightly redefine the instantiation rule in order to reconcile this conflict. Swapping the order that the lists of variables are processed from *types* $\rightarrow$ *rows* $\rightarrow$ *presence variables* to the opposite, *presence variables* $\rightarrow$ *rows* $\rightarrow$ *types* is enough to resolve the issue, since we are now starting from the "smallest" variables. In this manner, uses of instantiation will have less issues with variable naming conflicts, and the rest of the proof for this Lemma can be completed, as is seen in figure C.10.

3.6 Typings Are Well Formed

The final lemma that is significant for the proof of Subject Reduction is:

LEMMA 11 (TYPINGS ARE WELL FORMED). *Assuming E wf, if $E \vdash a : \tau/\varphi$, then $\vdash \tau\ wf$.*

Where:

$$\begin{aligned} E\ wf \iff \forall x, x \notin Dom(E) \lor \\ & (x \in Dom(E) \land \\ & \exists \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}\tau,\ E(x) = \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau \land \\ & \vdash \tau\ wf) \end{aligned}$$

While this might look like a convoluted definition, what we are trying to say is quite simple, an environment E is well formed if all identifiers bound in it are bound to a well formed type. This property is necessary for environments that we do our derivations in, otherwise we would experience problems when dealing with the typing rule for λ-abstraction, as well as any required application of Lemma 10. Notably, even though we have omitted it here for simplicity, the *Kmap* also needs to be passed to the $E\ wf$ property, because it is needed in the well-formedness judgment for arrow types, as well as exception types.

We approach the proof by inducting on the typing derivation, which leads to uncomplicated proofs of most cases, except the ever-present variable case where $a = x$. For this we have written a sub-lemma:

LEMMA 12 (INSTANCES OF WELL FORMED TYPES). *If E wf, and $\tau \leq E(x)$, then τ wf.*

This should be rather obvious, as any type scheme present in E is well formed, so naturally any instance of them will also be well formed. Regrettably, this runs into the similar issues as those discussed for Lemmas 3, 5 and 10, again due to the nature of the instantiation rule and how it interacts with our current implementation of variables and bindings.

A simple induction on the instantiation rule should normally afford us the desired result, but unlike in Lemma 10, we are dealing with arbitrary lists of variables, not just one type and one row variable. Changing the order of going through the respective

lists resolved Lemma 10, but it can not overcome this hurdle, because we are required to substitute an arbitrary number of rows, and rows contain other rows. It can then be the case that a one of the newly substituted rows contains another row, which can then later be substituted by an identically named row variable still in the list. This causes a mismatch in the expected row structure and prevents the inductive hypothesis from being applied where it is required.

Similarly to Lemma 10, this does not actually reflect the existence of a counterexample, because convenient renaming variables can resolve all such conflicts. This time though, it was impossible to find a solution that would convince the proof assistant itself of such a property. Regardless, on paper examination and common sense would seem to indicate that this lemma, as well as all others that face the same problems of dealing with instantiation should hold, so, in light of the aforementioned facts, the proof has been admitted with only a partial proof, in order to continue and investigate some of the other aspects of the system.

Also for Lemma 11, with respect to well formed environments, we prove and subsequently use the property that has been formalized as:

LEMMA 13 (EXTEND WELL FORMED ENVIRONMENTS). *If E wf, and $\vdash \tau$ wf, then $\forall \vec{\alpha_i}, \vec{\rho_i}, \vec{\delta_i}, x.\ E \oplus \{x : \forall \vec{\alpha_i}, \vec{\rho_i}, \vec{\delta_i}.\tau\}$ wf.*

A rather simple property, extending a well formed environment by binding another variable to a type scheme based on a well formed type will also result in a well formed environment. The proof for this is straightforward by the definition of well formed environments. This now allows us to build up well formed environments from smaller ones, which is essential in other proofs. Furthermore, we prove

LEMMA 14 (E_0 IS WELL FORMED). *E_0 wf.*

which is simple and straightforward, by direct application of the definition of well formed environments. Having it be a separate lemma of its own simplifies other proofs where it is needed a great deal, at least in terms of amount of code. All lemmas mentioned in this chapter can be found in figures C.11-C.13.

3.7 Subject Reduction

We have now assembled all the necessary parts for Subject Reduction:

THEOREM 2 (SUBJECT REDUCTION). *If $E_0 \vdash a : \tau/\varphi$ and $a \Rightarrow a'$, then $E_0 \vdash a' : \tau/\varphi$.*

This is to say, term reduction preserves typing and effect. We show this by case analysis on the reduction rule used, as can be seen in figure C.14. Many of the cases are trivial and will not be discussed in detail, concentrating instead on the more interesting ones.

For rules (1), (8), (12), (15) and (18) which feature substitution, we require Lemma

2, in conjunction with Lemma 8 for rule (15) specifically. The other *match* terms proceed without much issue, since we have simplified the pattern matching function and parametrized exceptions, nullifying the need for a Pattern Substitution Lemma, which was necessary in the original system. Rules (2), (5), (6), (9) and (11) are taken care of using Lemma 10. Lemmas 6, 7, 11 and 13 are also used throughout, when they are needed on more minor cases.

The original proof hand-waves away many of the cases of this induction as trivial, which we have discovered is far from the truth. Unlike in the case of the Substitution Lemma, the proof is mostly achievable with the help of Lemmas that were already mentioned by Leroy and Pessaux [6], but not explicitly stated to be used in the proof. The exceptions from this are Lemma 10, and other lemmas that were necessary due to modifications made in the attempt of resolving issues with Lemmas 1 and 2.

Intriguing is the fact that for the equivalents of rules (2), (5), (6), (9) and (11), the original paper concerns itself with the effects φ in the derivation, and claims the use of Lemma 14 (described in Section 3.8). This is inconsistent with the problems we encountered, which revolved around the types of the terms instead. Any kinding judgment for effects that was needed as part of sub-proofs, followed immediately from application of Lemma 6. It might be the case that the results of Lemma 10 were considered trivial within the bounds of the paper proof, and thus not worth mentioning, but this was not the case for us, as it did reveal a flaw with our implementation.

Provided we assume that all mentioned lemmas are considered proven, the proof for Subject Reduction and consequently, Type Soundness can be achieved. It is, however, quite a stretch to consider the pre-requiste lemmas as proven true, given the faults found in Lemmas 12 and 4, but this is further elaborated in Section 4.1

3.8 Correctness Exception Analysis

This is quite an important property of the system that only gets briefly mentioned in the original paper, but not expanded upon.

THEOREM 3 (CORRECTNESS OF EXCEPTION ANALYSIS). *If $E_0 \vdash a : \tau/\varphi$ and $a \overset{*}{\Rightarrow}$ raise v, then $v = C$ and C belongs to φ.*

Possibly the second most relevant property of the entire system, this states that uncaught exceptions are properly kinded and analyzed by our system. As with the reduction rules, C represents either of the two constant exceptions. Theorem 3 is a corollary of Theorem 2 and Lemma 15:

LEMMA 15 (EFFECTS OF EXCEPTIONS). *For a value v, if $E_0 \vdash$ raise $v : \tau/\varphi$, then $v = C$ and C belongs to φ.*

Which, in turn, follows trivially from use of typing rule (4) for applications, in conjunction with Lemma 9. Both of these proofs can be found in figure C.15.

As the name would imply, Theorem 3 gives us the fact that uncaught exceptions are properly analyzed by the system and attributed the correct effect within the row

that kinds it, an essential property for a system whose primary focus is that of inferring exception effects. However, this desirable property is also predicated on the use of Theorem 2, an important linchpin, currently without a believable, complete proof.

3.9 Observations

Unexpectedly, our worries from the experiments in section 2.7 were unfounded, as neither the kind mapping *Kmap*, nor the left-to-right order of execution, caused any complications to arise within our desired proofs. Simply using a *Kmap* that maps all kind variables to the empty kind, $EXN(\emptyset)$, was sufficient, as the kinding of most relevant rows was assured through the typing judgments present in the premises. Similarly, the order of execution did not pose an obstacle, with the exception of slightly prolonging some of the proofs; all needed properties were derivable from the premises, together with the appropriate inductive hypothesis, and never depended on parts of subsequent terms.

Of minor note is the fact that in the original paper, the equivalents for Theorems 1 and 3 mention that term a is a complete program. It is unsure what that even means, as there is no reference to this anywhere within the paper itself, or any of the relevant reference, including Pessaux's original thesis[10]. All possible definitions for complete programs, gathered from other pieces of the literature such as [1], either did not fit with our system, or did not provide any meaningful information that would be relevant to the theorems it was mentioned in. Since their variants of the proofs do not attempt to deal with the non-terminating cases which we have mentioned previously, it is speculated that the definition for a complete program would at least entail termination. Whether or not other important properties are included is unknown. It is unlikely that this is particularly relevant, as our versions of these Theorems accomplish the same goal, while not requiring any constraints on term a.

Additionally throughout the proof of the Substitution Lemma we use two properties of asymmetric concatenation:

$$E \oplus \{x : \sigma\} \oplus \{x : \sigma'\} \vdash a : \tau/\varphi \rightarrow E \oplus \{x : \sigma'\} \vdash a : \tau/\varphi \qquad (Overwrite)$$

$$x \neq y \wedge E \oplus \{x : \sigma\} \oplus \{y : \sigma'\} \vdash a : \tau/\varphi \rightarrow E \oplus \{y : \sigma'\} \oplus \{x : \sigma\} \vdash a : \tau/\varphi \qquad (Swap)$$

The first property is the fact that only the most recent binding of the same variable in the environment matters, while the second asserts that ordering of bindings does not matter as long as the variables bound are different. Both of these are quite plain to see, but were accepted with only partial proof, as they also suffer from the tedium and potential issues that come with the current, questionable, implementation of variables and instantiation.

Chapter 4

Discussion and Conclusion

4.1 Discussion

Due to faults found with both the Substitution and Progress Lemmas, we can safely say that the original system by Leroy and Pessaux [6] does not accomplish its desired goal. That being said, all problems that occur within the proof of Subject Reduction pertain to type and environment mismatches, with effects being consistent throughout. It stands to reason then, that a modified version of Theorem 2 could feasibly be proven, one that only concerns itself with retaining only the effects of terms on reduction, which, in conjunction with Lemma 15, would provide a definite proof for Theorem 3 (Correctness of Exception Analysis), where the type of the term is irrelevant. All in all, this is a good sign with respect to the "row annotations as effects" part of the proposed system, as it successfully accomplishes its stated goals.

Problematic is the fact that the aforementioned effects were to be inferred alongside types, through a standard inference algorithm. Such an algorithm would, unfortunately, not work properly on a system where Lemma 1 (Progress) and Theorem 1 (Type Soundness) do not hold. While our solution to Progress fixes the issue we discovered, a way to fully remedy the inconsistencies of Subject Reduction has not been found. Our proposed approach seems sensible, and within the confines of a paper proof would probably be considered acceptable, but is inadequate for a proof assistant.

All problems with the proposed alternative proof stem from our implementation of variables. Naming conflicts plague some of the central lemmas necessary, most notably Lemmas 4 and 12, which are required for the the Substitution Lemma and Subject Reduction, respectively. It appears that solving these issues concerning variables would probably require fully redesigning several of the system aspects, including at least: variables, environments, substitution, and the instantiation rule, if not more.

It is rather clear now, that the initial intention, of adapting the proposed system as directly as possible, was naive, especially when the end goal was a formalized, mechanized proof. Since the original paper is quite old, better techniques have been developed and used in order to deal with exactly these types of complications. One idea that stood out was to use something akin to De Brujin indices [2], which would bypass the need to deal with named variables entirely. Regrettably, such a change would require a significant amount of work, which was considered outside the reasonable scope of this project.

Any future continuation would be required to perform such an extensive change though, as it is clear that the original system has flaws that can not be resolved within its proposed structure. As mentioned previously, the idea to use row annotations to encapsulate effects seems sound, and is not where the problems with this system arise. Consequently, it would be interesting to compare it with other systems that attempt to do similar things, such as (D. Leijen, 2014)[5], (P. Wadler, 2014)[14], and (S. Lindley and J. Cheney, 2012)[7], with the goal of combining some of the ideas and coming up with some meaningful extensions or changes to the existing implementation.

Although complications with the proofs themselves established system extensions as infeasible within the scope of the current project, they are still an interesting consideration for future work. This is, of course, provided that the new implementation manages to solve the current inconsistencies in the proposed proofs. Firstly, reverting the simplifications to parametrized exceptions and integer effects would allow extending the proofs to the entirety of the original system, which would significantly increase expressive power. Similarly, the addition of several terms, such as *if* expressions, boolean types and operators, and possibly even loops, would pose an equally interesting challenge. Aside from that, the authors suggest several possible extensions, out of which the most fascinating would be tuples, records and mutable data structures. Concerning records and mutable data structures, particular interest is directed towards investigating and integrating ideas from a paper by Flanagan [4], which raises several points of interest, especially with regards to a formal proof.

4.2 Conclusion

In this project we implemented a simplified version of the system proposed by Leroy and Pessaux [6], with the aim of proving that it indeed possessed the qualities claimed by the authors. The original idea was to extend the type system of ML with row annotations, that would represent effects. Both would then be derived, at the same time, through standard polymorphic type inference. Specifically, the focus was directed towards exception analysis, which ML does not traditionally deal with very well.

Due to the complexity of the original design, several simplifications were made, most relevant being the removal of parametrized exceptions and integer effects, as well as reducing the number of constant exceptions to two. This number alloed for the exceptions to be juggled and create interesting behaviours, since more would not have generated a meaningful increase in expressiveness, while less would have had a significant negative impact. Coq was chosen as the proof assistant, and this new, streamlined system was translated into the relevant metalanguage. Several experiments were performed, as described in Section 2.7, which illustrate that the reduced system still has significant expressive power, although it currently lacks many of the types and terms that one would expect of a finished, usable programming language.

The most significant properties were determined to be Theorem 1 (Type Soundness) and Theorem 3 (Correctness of Exception Analysis), so most work was concentrated around investigating and obtaining their proofs. This follows the proof of Progress achieved in the first year of the project. While the path to both of these has been laid, there are still some holes along the way. Two important cornerstones on our path were the Progress and Substitution Lemmas. A problem with the Progress Lemma

was discovered initially, and quickly fixed, but the Substitution Lemma proved much more troublesome, and imposed changes on both the typing rules and the proposed proposed proof structure.

Several approaches were tried in an attempt to resolve the problems that arose, with the proposed one bringing us very close, but still barely out of reach of a complete, comprehensive solution. Having said that, complete proofs exist for a significant number important properties, and all other lemmas and theorems have partial proofs that hinge only on dealing with the variable naming conflicts.

Our implementation, while faithful to the original design, is insufficiently equipped to deal with the variable name clashes and falls slightly short of attaining the desired functionality. Using alternative formulations, such as de Brujin indices, for the same concepts, could help considerably, albeit at significant technical cost. Nevertheless, even as it stands, our code has managed to locate, interesting, relevant faults within the original system, and has at the very least shown that there is significant potential to be realized behind the initial idea.

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
        (if patcomp p x then t2
                        else (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

```
Definition undefined p v :=
      (forall y, p <> pvar y) /\
      (forall n, (p = pnat n -> v <> tnat n)) /\
      (forall e, (p = pexn e -> v <> texn e)).
```

```
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_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 ->
    undefined p 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
```

```
  | 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)

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

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),
      not (inDom E0 x) ->
      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),
      not (inDom E0 x) ->
      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),
      not (inDom E0 x) ->
      (forall x', (p = pvar x') -> not (inDom E0 x')) ->
      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),
      not (inDom E0 x) ->
      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

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

```
Definition subset E1 E2 :=
  forall x, ( ( exists t, ( apply E1 x = Some t ) ) -> apply E2 x = apply E1 x ).

Lemma SubsetSymm: forall E,
  subset E E.
Proof.
  intros. unfold subset.
  intros. reflexivity.
Qed.

Lemma SubsetExtensionNID : forall E1 E2 x ts,
  subset E1 E2 ->
  not (inDom E1 x) ->
  subset E1 (EUpdate x ts E2).
Proof. Admitted.

Lemma NIDofSubset : forall E1 E2 x,
  subset E1 E2 ->
  not (inDom E2 x) ->
  not (inDom E1 x).
Proof. Admitted.

Lemma SubsetExtension : forall E1 E2 x t,
  subset E1 E2 ->
  subset (EUpdate x t E1) (EUpdate x t E2).
Proof.
  Admitted.
```

Figure A.13: Subset Environments

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

Proofs

```
Theorem TypeSoundness: forall a tau phi,
  t_empty (EXN nil) \and E0 |- a \: tau // phi ->
  (uncaught a \/ value a \/
   exists a', ( a ==> a' /\ t_empty (EXN nil) \and E0 |- a' \: tau // phi ) ).
Proof.
  intros. assert (t_empty (EXN []) \and E0 |- a \: tau // phi).
  assumption. eapply progress in H. inversion H...
  - left. assumption.
  - inversion H1...
    * right. left. assumption.
    * inversion H2... right. right.
      exists x. split. apply H3.
      eapply SubjectReduction.
      apply H0. apply H3.
Qed.
```

Figure C.1: Type Soundness

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

```
Lemma progress : forall t T phi,
     t_empty (EXN nil) \and E0 |- t \: T // phi ->
     uncaught t \/ value t \/ exists t', t ==> t'.
Proof with eauto.
  intros t T phi Ht.
  remember E0 as Gamma.
  generalize dependent HeqGamma.
  induction Ht; intros HeqGamma; subst.
  - (* T_Var *)
    unfold apply in H. destruct (beq_idP x raise).
    + subst. right. left...
    + 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)...
    }
(* let *)
- destruct IHHt1...
  * (*t1 uncaught*)
    destruct H0.
    right. right. exists (tapp (tvar raise) v)...
  * { destruct H0.
      - (*t1 value*)
        right. right. exists (subst x t1 t2)...
      - (*t1 steps*)
        right. right. destruct H0 as [t1' Hstp]. exists (tlet x t1' t2)...
    }
```

```
(* match *)
- destruct IHHt1...
  * (*t1 uncaught*)
    destruct H3; subst. right. right. exists (tapp (tvar raise) v)...
  * destruct H3.
    + (*t1 value*)
      { destruct H3.
        - inversion Ht1;subst;simpl;inversion H2;subst;simpl...
        - inversion Ht1;subst;simpl;inversion H2;subst;simpl...
          destruct (beq_natP n1 n).
          * subst...
          * assert (undefined (pnat n) (tnat n1)). unfold undefined.
            split. intros. unfold Logic.not. intros. inversion H3.
            split. intros;inversion H3;subst;simpl... unfold Logic.not.
            intros. inversion H4. unfold Logic.not in n0.
            apply n0 in H7. assumption. intros. inversion H3.
            right. right. exists ([x:=(tnat n1)]t3). apply ST_MatchNot...
        - inversion Ht1;subst;simpl;inversion H2;subst;simpl...
          + destruct (beq_exnP e C).
            * subst...
            * assert (undefined (pexn C) (texn e)). unfold undefined.
              split. intros. unfold Logic.not. intros. inversion H3.
              split. intros;inversion H3;subst;simpl... intros...
              unfold Logic.not. intros. inversion H3. inversion H5.
              unfold Logic.not in n. rewrite <- H8 in H9.
              apply n in H9. assumption. right. right.
              exists ([x:=(texn e)]t3). apply ST_MatchNot...
```

```
      + destruct (beq_exnP e C').
        * subst...
        * assert (undefined (pexn C') (texn e)). unfold undefined.
          split. intros. unfold Logic.not. intros. inversion H3.
          split. intros;inversion H3;subst;simpl... intros...
          unfold Logic.not. intros. inversion H3. inversion H5.
          unfold Logic.not in n. rewrite <- H8 in H9.
          apply n in H9. assumption. right. right.
          exists ([x:=(texn e)]t3). apply ST_MatchNot...
    - right. right. inversion H2;subst;simpl...
      + assert (undefined (pnat n) (tvar raise)). unfold undefined.
        split. intros. unfold Logic.not. intros. inversion H3.
        split. intros. unfold Logic.not. intros. inversion H4.
        intros. inversion H3. exists ([x:=(tvar raise)]t3). apply ST_MatchNot..
      + assert (undefined (pexn C) (tvar raise)). unfold undefined.
        split. intros. unfold Logic.not. intros. inversion H3.
        split. intros. unfold Logic.not. intros. inversion H4.
        intros. unfold Logic.not. intros. inversion H4.
        exists ([x:=(tvar raise)]t3). apply ST_MatchNot...
      + assert (undefined (pexn C) (tvar raise)). unfold undefined.
        split. intros. unfold Logic.not. intros. inversion H3.
        split. intros. unfold Logic.not. intros. inversion H4.
        intros. unfold Logic.not. intros. inversion H4.
        exists ([x:=(tvar raise)]t3). apply ST_MatchNot...
  }
+ (*t1 steps*)
  right. right. destruct H3 as [t1' Hstp]. exists (tmatch t1' p t2 x t3)...
```

```
(*exn*)
- right. left...
(*exn2*)
- right. left...
(*try*)
- destruct IHHt1...
  * (*t1 uncaught*)
    destruct H0; subst. right. right. exists (subst x v t2)...
  * destruct H0.
    + (*t1 value*)
      right. right. exists t1...
    + (*t1 steps*)
      right. right. destruct H0 as [t1' Hstp]. exists (ttry t1' x 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;subst.
          inversion H9;subst. inversion H0;subst.
          inversion H3;subst. inversion H12;subst.
          inversion H5;subst. inversion H7.
      }
    * (* 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;subst.
          inversion H9;subst. inversion H0;subst.
          inversion H3;subst. inversion H12;subst.
          inversion H5;subst. inversion H7.
      }
    * (* 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;subst.
              inversion H10;subst. inversion H1;subst.
              inversion H4;subst. inversion H13;subst.
              inversion H6;subst. inversion H8.
              inversion Ht1; subst. inversion H2;subst.
              inversion H10;subst. inversion H1;subst.
              inversion H4;subst. inversion H13;subst.
              inversion H6;subst. inversion H8.
            + (*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)...
    }
Qed.
```

Figure C.2: Progress

```
Lemma SubstitutionLemma : forall Kmap x l1 l2 l3 tau tau' a v phi E,
      Kmap \and (EUpdate x (TScheme l1 l2 l3 tau') E) |- a \: tau // phi ->
      subset E0 E ->
      value v ->
      Kmap \and E0 |- v \: tau' // phi ->
      not (inDom E0 x) ->
      Kmap \and E |- [x:=v]a \: tau // phi.
Proof with eauto.
  intros. generalize dependent tau. generalize dependent phi.
  generalize dependent l1. generalize dependent l2. generalize dependent l3.
  generalize dependent E. induction a; intros; simpl; inversion H; subst...
  - rename i into y. unfold apply in H5.
    destruct (beq_idP x y).
    * assert (H': beq_id y x = true).
      { symmetry. rewrite -> e. apply beq_id_refl. }
      rewrite -> H' in H5. eapply TypingStableSubstitution.
      eapply TSUEE. apply H2. apply H0.
      assumption. apply H5.
    * assert (H': beq_id y x = false).
      { rewrite -> beq_symm. apply false_beq_id in n. assumption. }
      rewrite -> H' in H5...
  - rename i into y. destruct (beq_idP x y).
    * subst. eapply T_Abs... eapply EnvOverwrite in H10. apply H10.
    * eapply T_Abs...
      eapply IHa.
      + eapply SubsetExtensionNID...
      + apply TypingsWellKinded in H10. eapply val_no_eff.
        assumption. apply H2. assumption.
      + apply SwapBindingOrder in H10. apply H10.
        rewrite <- beq_id_false_iff. rewrite beq_symm.
        rewrite <- beq_id_false_iff in n. assumption.
```

```
- rename i into y. destruct (beq_idP x y).
  * subst. eapply T_Let... eapply TyStableMoreGeneralHyp in H13.
    eapply EnvOverwrite in H13. assumption.
  * eapply T_Let... eapply IHa2...
    + eapply SubsetExtensionNID. assumption. apply H9.
    + eapply TyStableMoreGeneralHyp in H13. apply SwapBindingOrder in H13. apply H13.
      rewrite <- beq_id_false_iff. rewrite beq_symm.
      rewrite <- beq_id_false_iff in n. assumption.
- rename i into y. destruct (beq_idP x y).
  * eapply T_Match...
    + inversion H16;subst;simpl... destruct (beq_idP y x0);
      subst;unfold aconcat in H18;simpl...
      apply EnvOverwrite in H18. apply H18.
      eapply IHa2. eapply SubsetExtensionNID...
      assumption. apply SwapBindingOrder in H18. apply H18.
      rewrite <- beq_id_false_iff. rewrite beq_symm.
      rewrite <- beq_id_false_iff in n. assumption.
    + subst. apply EnvOverwrite in H19. assumption.
  * eapply T_Match...
    + inversion H16;subst;simpl... destruct (beq_idP x x0);
      subst;unfold aconcat in H18;simpl...
      apply EnvOverwrite in H18. apply H18.
      eapply IHa2. eapply SubsetExtensionNID...
      assumption. apply SwapBindingOrder in H18. apply H18.
      rewrite <- beq_id_false_iff. rewrite beq_symm.
      rewrite <- beq_id_false_iff in n0. assumption.
    + eapply IHa3... eapply SubsetExtensionNID.
      assumption. assumption.
      eapply SwapBindingOrder. apply H19.
      rewrite <- beq_id_false_iff. rewrite beq_symm.
      rewrite <- beq_id_false_iff in n. assumption.
'
- rename i into y. destruct (beq_idP x y);subst.
  * eapply T_Try...
    + eapply IHa1. assumption. eapply val_no_eff.
      assumption. apply H2. apply TypingsWellKinded in H12.
      apply H12. apply H12.
    + apply EnvOverwrite in H13. apply H13.
  * eapply T_Try...
    + eapply IHa1. assumption. eapply val_no_eff.
      assumption. apply H2. apply TypingsWellKinded in H12.
      apply H12. apply H12.
    + eapply IHa2... eapply SubsetExtensionNID...
      eapply SwapBindingOrder in H13. apply H13.
      rewrite <- beq_id_false_iff. rewrite beq_symm.
      rewrite <- beq_id_false_iff in n. assumption.
Qed.
```

Figure C.3: Substitution Lemma

```
Lemma TSUEE : forall E E' Kmap a tau phi,
  Kmap \and E' |- a \: tau // phi ->
  subset E' E ->
  Kmap \and E  |- a \: tau // phi.
Proof with eauto.
  intros. generalize dependent E. generalize dependent E'.
  generalize dependent phi. generalize dependent tau.
  induction a;intros;subst;simpl...
  - inversion H;subst;simpl. eapply T_Var. induction H0 with i.
    assumption. destruct (apply E' i).
    exists t. reflexivity. inversion H2. assumption.
  - inversion H. eapply T_Nat. assumption.
  - inversion H;subst;simpl... eapply T_Abs...
    eapply IHa. apply H7. eapply SubsetExtension. assumption.
  - inversion H. eapply T_App...
  - inversion H;subst;simpl... eapply T_Let.
    assumption. eapply IHa1. apply H9. assumption.
    eapply IHa2. apply H10. eapply RenameVarEnv. assumption.
  - inversion H;subst;simpl... eapply T_Match...
    * inversion H13;subst... eapply IHa2. apply H15.
      eapply SubsetExtension. assumption.
    * eapply IHa3. apply H16. eapply SubsetExtension. assumption.
  - inversion H;subst;simpl...
  - inversion H;subst;simpl... eapply T_Try.....
    eapply IHa2. apply H10. eapply SubsetExtension. assumption.
  - inversion H;subst;simpl...
  - inversion H;subst;simpl...
  - inversion H;subst;simpl...
Qed.
```

Figure C.4: Lemma 4

```
Axiom RenameVarEnv : forall E E' tau phi x,
  subset E' E ->
  subset (EUpdate x (Gen tau E' phi) E') (EUpdate x (Gen tau E phi) E).

Lemma TypingStableSubstitution : forall Kmap l1 l2 l3 tau tau' v phi E,
  Kmap \and E |- v \: tau' // phi ->
  value v ->
  tau \leq Some (TScheme l1 l2 l3 tau') \with Kmap ->
  Kmap \and E |- v \: tau // phi.
Proof. Admitted.

Lemma TyStableMoreGeneralHyp: forall Kmap y tau1 x ts E a tau phi,
Kmap \and EUpdate y (Gen tau1 (EUpdate x ts E) phi) (EUpdate x ts E) |- a \: tau // phi ->
Kmap \and EUpdate y (Gen tau1 E phi) (EUpdate x ts E) |- a \: tau // phi.
Proof. Admitted.
```

Figure C.5: Lemmas 3 and 5

```
Lemma TypingsWellKinded: forall t Kmap E tau phi,
  Kmap \and E |- t \: tau // phi ->
  phi \:: EXN [] \in Kmap.
Proof with eauto.
  intros.
  induction H;subst;simpl...
Qed.
```

Figure C.6: Lemma 6

```
Lemma val_no_eff : forall v Kmap E0 tau phi phi',
    value v ->
    Kmap \and E0 |- v \: tau // phi ->
    phi' \:: (EXN []) \in Kmap ->
    Kmap \and E0 |- v \: tau // phi'.
Proof with eauto.
  intros.
  inversion H.
  - (*tabs*)
    subst; inversion H0; subst.
    eapply T_Abs. assumption.
    assumption. assumption. assumption.
  - (*tnat*)
    subst; inversion H0; subst.
    eapply T_Nat. assumption.
  - (*texn*)
    subst; inversion H0; subst.
    * eapply T_Exn.
      assumption.
      assumption.
    * eapply T_Exn2.
      assumption.
      assumption.
  - (*tvar*)
    subst; inversion H0; subst.
    eapply T_Var.
    assumption.
    assumption.
Qed.
```

Figure C.7: Lemma 7

```
Lemma CorrSub : forall v tau phi tau' p Kmap,
  value v ->
  Kmap \and E0 |- v \: tau // phi ->
  tau \- p \~> tau' \for Kmap ->
  undefined p v ->
  Kmap \and E0 |- v \: tau' // phi.
Proof.
  intros.
  induction p.
  - inversion H2. induction H3 with i. reflexivity.
  - inversion H1; subst; simpl... assumption.
  - destruct e; inversion H1; subst; simpl...
    * inversion H2. inversion H4.
      eapply (SoVbT Kmap0 v (rel (elem C1 pi) phi0) phi) in H. inversion H.
        + inversion H7. induction H6 with C1. reflexivity. assumption.
        + inversion H7. rewrite -> H8. rewrite -> H8 in H0. inversion H0;subst;simpl...
          eapply T_Exn2. assumption. assumption.
        + assumption.
    * inversion H2. inversion H4.
      eapply (SoVbT Kmap0 v (rel (elem C' pi') (rel (elem C1 pi) phi0)) phi) in H. inversion H.
        + inversion H7. induction H6 with C1. reflexivity. assumption.
        + inversion H7. rewrite -> H8. rewrite -> H8 in H0. inversion H0;subst;simpl...
          eapply T_Exn. inversion H17; subst; simpl... eapply K_Exn.
          assumption. assumption. assumption.
        + assumption.
    * inversion H2. inversion H4.
      eapply (SoVbT Kmap0 v (rel (elem C2 pi) phi0) phi) in H. inversion H.
        + inversion H7. rewrite -> H8. rewrite -> H8 in H0. inversion H0;subst;simpl...
          eapply T_Exn2. assumption. assumption.
        + inversion H7. induction H6 with C2. reflexivity. assumption.
        + assumption.
    * inversion H2. inversion H4.
      eapply (SoVbT Kmap0 v (rel (elem C' pi') (rel (elem C2 pi) phi0)) phi) in H. inversion H.
        + inversion H7. rewrite -> H8. rewrite -> H8 in H0. inversion H0;subst;simpl...
          eapply T_Exn. inversion H17; subst; simpl... eapply K_Exn.
          assumption. assumption. assumption.
        + inversion H7. induction H6 with C2. reflexivity. assumption.
        + assumption.
Qed.
```

Figure C.8: Lemma 8

```
Lemma SoVbT : forall Kmap v rho' phi,
  value v ->
  Kmap \and E0 |- v \: TExn rho' // phi ->
  (v = texn C1  /\ ((exists phi', (rho' = (rel (elem C1 pre) phi'))) \/
   (exists phi' pi C',  rho' = (rel (elem C' pi) (rel (elem C1 pre) phi'))))) \/
  (v = texn C2 /\ ((exists phi', (rho' = (rel (elem C2 pre) phi'))) \/
   (exists phi' pi C',  rho' = (rel (elem C' pi) (rel (elem C2 pre) phi'))))).
Proof.
  intros.
  inversion H; subst; simpl...
  - inversion H0.
  - inversion H0.
  - destruct e; inversion H0; subst; simpl...
    * left. split. reflexivity. left. exists phi'. reflexivity.
    * left. split. reflexivity. right. exists phi'. exists pi. exists C'. reflexivity.
    * right. split. reflexivity. left. exists phi'. reflexivity.
    * right. split. reflexivity. right. exists phi'. exists pi. exists C'. reflexivity.
  - inversion H0; subst; simpl...
    unfold apply in H2. simpl... assert (H': beq_id raise raise = true).
    { apply beq_id_true_iff. reflexivity. }
    rewrite -> H' in H2. inversion H2; subst; simpl... inversion H10. destruct H1.
    inversion H3; subst; simpl... destruct H12. destruct H4. inversion H6; subst; simpl...
Qed.
```

Figure C.9: Lemma 9

```
Lemma RaiseValAnyType : forall v tau tau' phi,
  t_empty (EXN nil) \and E0 |- tapp (tvar raise) v \: tau // phi ->
  value v ->
  tau' wf t_empty (EXN nil) ->
  t_empty (EXN nil) \and E0 |- tapp (tvar raise) v \: tau' // phi.
Proof.
  intros.
  inversion H0;subst;simpl;inversion H;subst;simpl...
  - inversion H6;subst;simpl... inversion H3;subst;simpl...
    inversion H13;subst;simpl... inversion H2;subst;simpl...
    inversion H5;subst;simpl... inversion H16;subst;simpl...
    inversion H8;subst;simpl... inversion H11;subst;simpl...
    inversion H9.
  - inversion H6;subst;simpl... inversion H3;subst;simpl...
    inversion H13;subst;simpl... inversion H2;subst;simpl...
    inversion H5;subst;simpl... inversion H16;subst;simpl...
    inversion H8;subst;simpl... inversion H11;subst;simpl...
    inversion H9.
  - inversion H6;subst;simpl... inversion H3;subst;simpl...
    inversion H13;subst;simpl... inversion H2;subst;simpl...
    inversion H5;subst;simpl... inversion H16;subst;simpl...
    inversion H8;subst;simpl... inversion H11;subst;simpl...
    eapply T_App. eapply T_Var. unfold apply; simpl...
    eapply I_Row. exists x0. split. assumption.
    apply I_Type. exists tau'. split. assumption. simpl. eapply I_Eq.
    unfold t_empty in H8. apply H8. assumption.
  - inversion H6;subst;simpl... inversion H3;subst;simpl...
    inversion H13;subst;simpl... inversion H2;subst;simpl...
    inversion H5;subst;simpl... inversion H16;subst;simpl...
    inversion H8;subst;simpl... inversion H11;subst;simpl...
    inversion H9;subst;simpl... inversion H14;subst;simpl...
    inversion H23;subst;simpl... inversion H12;subst;simpl...
    inversion H17;subst;simpl... inversion H26;subst;simpl...
    inversion H19;subst;simpl... inversion H21;subst;simpl...
Qed.
```

Figure C.10: Lemma 10

```
Lemma TypingsWellFormed: forall t Kmap E tau phi,
  envWF E Kmap->
  Kmap \and E |- t \: tau // phi ->
  tau wf Kmap.
Proof with eauto.
  intros.
  induction H0.
  - apply WFInst in H0. assumption. assumption.
  - eapply WF_Nat.
  - eapply WF_Arrow. assumption. apply TypingsWellKinded in H2. assumption.
    assert (H': envWF (EUpdate x (TScheme [] [] [] tau1) E) Kmap0).
    { eapply EWF_Extend. assumption. apply H1. }
    apply IHhas_type in H'. assumption.
  - apply IHhas_type1 in H. inversion H;subst;simpl...
  - assert (H': envWF (EUpdate x (Gen tau1 E phi) E) Kmap0).
    { eapply EWF_Extend. assumption. apply IHhas_type1 in H. apply H. }
    apply IHhas_type2 in H'. assumption.
  - inversion H2;simpl;subst... unfold aconcat in IHhas_type2.
    assert (H': envWF E Kmap0). {assumption. }
    apply IHhas_type1 in H. assert (H'': envWF (EUpdate x0 (TScheme [] [] [] tau1) E) Kmap0).
    { eapply EWF_Extend. assumption. apply H. }
    apply IHhas_type2 in H''. assumption.
  - eapply WF_Exn. eapply K_Exn. unfold not. intros. inversion H2. assumption.
  - eapply WF_Exn. eapply K_Exn. unfold not. intros. inversion H2.
    eapply K_Exn. unfold not. intros. inversion H2;subst;simpl...
    apply SetInvariant1 in H0. inversion H0. apply SetInvariant2 in H0. assumption.
  - apply IHhas_type1 in H. assumption.
  - eapply WF_Nat.
  - eapply WF_Nat.
  - eapply WF_Nat.
Qed.
```

Figure C.11: Lemma 11

```
Definition envWF E Kmap :=
   forall (x:id), (inDom E x /\
   (exists l1 l2 l3 tau, (apply E x = Some (TScheme l1 l2 l3 tau) /\
   tau wf Kmap )) \/ (not (inDom E x)).

Lemma EWF_Extend : forall E tau' l1 l2 l3 Kmap x,
      envWF E Kmap ->
      tau' wf Kmap ->
      envWF (EUpdate x (TScheme l1 l2 l3 tau') E) Kmap.
Proof.
  intros. unfold envWF in H. unfold envWF. intros. induction H with x0.
  - inversion H1. unfold inDom in H2. unfold inDom.
    unfold apply in H2. unfold apply. destruct (beq_id x0 x).
    * left. split. reflexivity.
      exists l1. exists l2. exists l3. exists tau'.
      split. reflexivity. apply H0.
    * left. split. assumption. fold apply.
      inversion H3. inversion H4. inversion H5.
      inversion H6. inversion H7.
      exists x1. exists x2. exists x3. exists x4.
      split. assumption. assumption.
  - unfold not in H1. unfold inDom in H1. unfold inDom.
    unfold apply. destruct (beq_id x0 x).
    * left. split. reflexivity.
      exists l1. exists l2. exists l3. exists tau'.
      split. reflexivity. apply H0.
    * right. assumption.
Qed.
```

Figure C.12: Well Formed Environments and Lemma 13

```
Lemma WFInst : forall E tau Kmap x,
  envWF E Kmap->
  tau \leq apply E x \with Kmap ->
  tau wf Kmap.
Proof.
  intros. unfold envWF in H. induction H with x.
  inversion H1. inversion H3. inversion H4.
  inversion H5. inversion H6. inversion H7.
  rewrite -> H8 in H0. induction x0,x1,x2.
  - inversion H0;subst;simpl... assumption.
  - Admitted.

Lemma E0WF :
  envWF E0 (t_empty (EXN nil)).
Proof.
  intros. unfold envWF. intros.
  destruct (beq_idP x raise).
  - left. split;unfold inDom;unfold apply;subst.
    * assert (H: (beq_id raise raise) = true). apply beq_symm.
      rewrite -> H. reflexivity.
    * assert (H: (beq_id raise raise) = true). apply beq_symm.
      rewrite -> H. exists [alpha]. exists [rho]. exists [].
      exists (TArrow (TExn (rvar rho)) (rvar rho) (TVar alpha)).
      split. reflexivity. eapply WF_Arrow.
      + eapply WF_Exn. eapply K_K. reflexivity.
      + eapply K_K. reflexivity.
      + eapply WF_Var.
  - right. unfold not;unfold inDom;unfold apply.
    assert (H': beq_id x raise = false).
    { apply false_beq_id in n. assumption. }
    rewrite -> H'. intros. inversion H.
Qed.
```

Figure C.13: Lemmas 12 and 14

```
Theorem SubjectReduction: forall t t' T phi,
  t_empty (EXN nil) \and E0 |- t \: T // phi ->
  t ==> t' ->
  t_empty (EXN nil) \and E0 |- t' \: T // phi.
Proof with eauto.
  intros t t' T phi HT.
  remember (t_empty (EXN [])) as Kmap. generalize dependent HeqKmap.
  remember E0 as E. generalize dependent HeqE.
  generalize dependent t'.
  induction HT;
  intros t' HeqE HeqKmap HE; subst; inversion HE; subst...
  - eapply SubstitutionLemma...
    inversion HT1... apply SubsetSymm. inversion HT1;subst;simpl...
  - inversion HT2; subst.
    eapply RaiseValAnyType in HT2. apply HT2. assumption.
    assert (H': t_empty (EXN []) \and E0 |- tapp (tabs x t0) (tapp (tvar raise) v2) \: tau // phi).
    { eapply T_App. apply HT1. apply HT2. }
    apply TypingsWellFormed in H'. assumption. apply E0WF.
  - inversion HT1; subst.
    eapply RaiseValAnyType in HT1. apply HT1. assumption.
    assert (H': t_empty (EXN []) \and E0 |- tapp (tapp (tvar raise) v1) t2 \: tau // phi).
    { eapply T_App. apply HT1. apply HT2. }
    apply TypingsWellFormed in H'. assumption. apply E0WF.
  - inversion HT2; subst.
    eapply RaiseValAnyType in HT2. apply HT2. assumption.
    assert (H': t_empty (EXN []) \and E0 |- tapp t1 (tapp (tvar raise) v2) \: tau // phi).
    { eapply T_App. apply HT1. apply HT2. }
    apply TypingsWellFormed in H'. assumption. apply E0WF.
  - eapply SubstitutionLemma... apply SubsetSymm.
  - inversion HT1; subst.
    eapply RaiseValAnyType in HT1. apply HT1. assumption.
    assert (H': t_empty (EXN []) \and E0 |- tlet x (tapp (tvar raise) v1) t2 \: tau // phi).
    { eapply T_Let. assumption. apply HT1. apply HT2. }
    apply TypingsWellFormed in H'. assumption. apply E0WF.
```

```
- inversion HT1; subst.
  eapply RaiseValAnyType in HT1. apply HT1. assumption.
  assert (H': t_empty (EXN []) \and E0 |- tmatch (tapp (tvar raise) v1) p t2 x t3  \: tau // phi).
  { eapply T_Match. assumption. apply H0. apply HT1. apply H1. apply H2. apply HT2. apply HT3. }
  apply TypingsWellFormed in H'. assumption. apply E0WF.
- eapply SubstitutionLemma.
  * inversion H1. subst. simpl...
  * apply SubsetSymm.
  * assumption.
  * assumption.
  * apply H0. reflexivity.
- inversion H1; subst; simpl...
- inversion H1; subst; simpl...
- eapply SubstitutionLemma.
  * apply HT3.
  * apply SubsetSymm.
  * assumption.
  * eapply CorrSub. assumption. apply HT1.
    apply H2. assumption.
  * assumption.
- eapply SubstitutionLemma.
  * apply HT2.
  * apply SubsetSymm.
  * assumption.
  * inversion H4;subst;simpl;inversion HT1;subst;simpl...
    + inversion H5;subst;simpl... inversion H1;subst;simpl...
      inversion H12;subst;simpl... inversion H0;subst;simpl...
      inversion H3;subst;simpl... inversion H15;subst;simpl...
      inversion H7;subst;simpl... inversion H10;subst;simpl...
      inversion H8.
    + inversion H5;subst;simpl... inversion H1;subst;simpl...
      inversion H12;subst;simpl... inversion H0;subst;simpl...
      inversion H3;subst;simpl... inversion H15;subst;simpl...
      inversion H7;subst;simpl... inversion H10;subst;simpl...
      inversion H8.
```

```
      + inversion H5;subst;simpl... inversion H1;subst;simpl...
        inversion H12;subst;simpl... inversion H0;subst;simpl...
        inversion H3;subst;simpl... inversion H15;subst;simpl...
        inversion H7;subst;simpl... inversion H10;subst;simpl...
        eapply val_no_eff in H8. apply H8. assumption.
        apply TypingsWellKinded in HT2. assumption.
      + inversion H5;subst;simpl... inversion H1;subst;simpl...
        inversion H12;subst;simpl... inversion H0;subst;simpl...
        inversion H3;subst;simpl... inversion H15;subst;simpl...
        inversion H7;subst;simpl... inversion H10;subst;simpl...
        eapply val_no_eff in H8. apply H8. assumption.
        apply TypingsWellKinded in HT2. assumption.
    * assumption.
  - inversion H4;subst;simpl...
    * inversion HT1;subst;simpl... eapply T_Abs.
      assumption. assumption. assumption. induction phi. eapply K_K. unfold t_empty.
      reflexivity... apply TypingsWellKinded in HT2. assumption.
    * inversion HT1;subst;simpl... eapply T_Nat.
      apply TypingsWellKinded in HT2. assumption.
    * inversion HT1;subst;simpl...
      eapply T_Exn. assumption.
      apply TypingsWellKinded in HT2. assumption.
      eapply T_Exn2. assumption.
      apply TypingsWellKinded in HT2. assumption.
    * inversion HT1;subst;simpl... eapply T_Var.
      assumption. apply TypingsWellKinded in HT2. assumption.
  - eapply T_Nat. inversion HT. assumption.
  - eapply T_Nat. inversion HT. assumption.
  - eapply T_Nat. inversion HT1. assumption.
Qed.
```

Figure C.14: Theorem 2

```
Theorem ExceptionAnalysis : forall a tau phi v,
  t_empty (EXN nil) \and E0 |- a \: tau // phi ->
  a ==>* (tapp (tvar raise) v) ->
  value v ->
  (v = texn C1  /\ ((exists phi', (phi = (rel (elem C1 pre) phi'))) \/
   (exists phi' pi C',  phi = (rel (elem C' pi) (rel (elem C1 pre) phi'))))) \/
  (v = texn C2 /\ ((exists phi', (phi = (rel (elem C2 pre) phi'))) \/
   (exists phi' pi C',  phi = (rel (elem C' pi) (rel (elem C2 pre) phi'))))).
Proof.
  intros. eapply reduceStep in H.
  eapply eff_of_exn. apply H. assumption. apply H0.
Qed.

Lemma eff_of_exn : forall Kmap v tau phi,
  Kmap \and E0 |- (tapp (tvar raise) v) \: tau // phi ->
  value v ->
  (v = texn C1  /\ ((exists phi', (phi = (rel (elem C1 pre) phi'))) \/
   (exists phi' pi C',  phi = (rel (elem C' pi) (rel (elem C1 pre) phi'))))) \/
  (v = texn C2 /\ ((exists phi', (phi = (rel (elem C2 pre) phi'))) \/
   (exists phi' pi C',  phi = (rel (elem C' pi) (rel (elem C2 pre) phi'))))).
Proof.
  intros.
  inversion H;subst;simpl... inversion H5;subst;simpl...
  inversion H2;subst;simpl... inversion H12;subst;simpl...
  inversion H1;subst;simpl... inversion H4;subst;simpl...
  inversion H15;subst;simpl... inversion H7;subst;simpl...
  inversion H10;subst;simpl... eapply SoVbT. assumption. apply H8.
Qed.
```

Figure C.15: Theorem 3 and Lemma 15

```
Lemma EnvOverwrite : forall Kmap x ts1 ts2 phi tau a E,
Kmap \and EUpdate x ts1 (EUpdate x ts2 E) |- a \: tau // phi ->
Kmap \and EUpdate x ts1 E |- a \: tau // phi.
Proof.
  intros. generalize dependent E. generalize dependent phi.
  generalize dependent x. generalize dependent ts1. generalize dependent tau.
  generalize dependent ts2. induction a;intros;subst;simpl...
  - eapply T_Var; inversion H;subst;simpl...  unfold apply. unfold apply in H1.
    destruct (beq_idP i x). assumption. assumption. assumption.
  - inversion H. eapply T_Nat. assumption.
  - inversion H;subst;simpl... eapply T_Abs...
    Admitted.

Lemma SwapBindingOrder: forall x y ts ts' E tau phi a Kmap,
  Kmap \and EUpdate x ts (EUpdate y ts' E) |- a \: tau // phi ->
  x <> y ->
  Kmap \and EUpdate y ts' (EUpdate x ts E) |- a \: tau // phi.
Proof.
  Admitted.
```

Figure C.16: Other Helper Properties

Appendix D

List of Theorems and Lemmas

THEOREM 1 (TYPE SOUNDNESS). *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 $a \Rightarrow a'$ and $E_0 \vdash a' : \tau/\varphi$.*

THEOREM 2 (SUBJECT REDUCTION). *If $E_0 \vdash a : \tau/\varphi$ and $a \Rightarrow a'$, then $E_0 \vdash a' : \tau/\varphi$.*

THEOREM 3 (CORRECTNESS OF EXCEPTION ANALYSIS). *If $E_0 \vdash a : \tau/\varphi$ and $a \Rightarrow$ raise v, then $v = C$ and C belongs to φ.*

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 $a \Rightarrow a'$.*

LEMMA 2 (SUBSTITUTION LEMMA). *Assuming $E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\} \vdash a : \tau/\varphi$, and for a value v, $E_0 \vdash v : \tau'/\varphi$, if $E_0 \subset E$ and $x \notin E_0$, then $E \vdash a\{x \leftarrow v\} : \tau/\varphi$*

LEMMA 3 (TYPING IS STABLE BY SUBSTITUTION). *If for a value v, $E \vdash v : \tau'/\varphi$ and $\tau \leq \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'$, then $E \vdash v : \tau/\varphi$.*

LEMMA 4 (TYPING IS STABLE UNDER EXTENDED ENVIRONMENT). *If $E' \vdash a : \tau/\varphi$ and $E' \subset E$, then $E \vdash a : \tau/\varphi$.*

LEMMA 5 (TYPING IS STABLE UNDER MORE GENERAL HYPOTHESIS). *Assuming $E' = E \oplus \{x : \forall \vec{\alpha_i}\vec{\rho_i}\vec{\delta_i}.\tau'\}$. If $E' \oplus \{x' : Gen(\tau_1, E', \varphi)\} \vdash a_2 : \tau/\varphi$, then, $E' \oplus \{x' : Gen(\tau_1, E, \varphi)\} \vdash a_2 : \tau/\varphi$.*

LEMMA 6 (TYPING ARE WELL KINDED). *If $E \vdash a : \tau/\varphi$, then $\vdash \varphi :: EXN(\emptyset)$.*

LEMMA 7 (VALUES HAVE NO EFFECTS). *Assuming $\vdash \varphi' :: EXN(\emptyset)$. For a value v, if $E \vdash v : \tau/\varphi$, then $E \vdash v : \tau/\varphi'$.*

LEMMA 8 (CORRECTNESS OF SUBTRACTION). *For a value v, if $E \vdash v : \tau/\varphi$, $\vdash \tau - p \rightsquigarrow \tau'$, and (p, v) undefined, then $E \vdash v : \tau'/\varphi$.*

LEMMA 9 (SHAPE OF VALUES BY TYPE). *For a value v, if* $E \vdash v : \tau/\varphi$ *and* $\tau = exn[\varphi]$*, then* $v = C$ *and C belongs to* φ.

LEMMA 10 (RAISE VALUE HAS ANY TYPE). *For a value v, if* $E_0 \vdash raise\ v : \tau/\varphi$ *and* $\vdash \tau'\ wf$*, then* $E_0 \vdash raise\ v : \tau'/\varphi$.

LEMMA 11 (TYPINGS ARE WELL FORMED). *Assuming E wf, if* $E \vdash a : \tau/\varphi$*, then* $\vdash \tau\ wf$.

LEMMA 12 (INSTANCES OF WELL FORMED TYPES). *If E wf, and* $\tau \leq E(x)$*, then* τ *wf.*

LEMMA 13 (EXTEND WELL FORMED ENVIRONMENTS). *If E wf, and* $\vdash \tau\ wf$*, then* $\forall \vec{\alpha_i}, \vec{\rho_i}, \vec{\delta_i}, x.\ E \oplus \{x : \forall \vec{\alpha_i}, \vec{\rho_i}, \vec{\delta_i}.\tau\}\ wf$.

LEMMA 14 (E_0 IS WELL FORMED). E_0 *wf.*

LEMMA 15 (EFFECTS OF EXCEPTIONS). *For a value v, if* $E_0 \vdash raise\ v : \tau/\varphi$*, then* $v = C$ *and C belongs to* φ.

Bibliography

[1] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology*, page 36–50, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[2] N.g De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[3] Tudor Ferariu. Proving the soundness of a system that can infer effects. Honours project report, School of Informatics, University of Edinburgh, 2018.

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

[5] Daan Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126, Jun 2014.

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

[7] Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 91–102, New York, NY, USA, 2012. ACM.

[8] 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.

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

[10] François Pessaux. Detection statique d'exceptions non rattrapees en objective caml. 1999. PhD thesis, Université Paris.

[11] 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.

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

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

[14] Philip Wadler. The marriage of effects and monads. *SIGPLAN Not.*, 34(1):63–74, September 1998.

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