Automatic Amortized Resource Analysis for Exception Handling

Yiyang Guo

CMU-CS-23-133
August 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Type Systems, Resource Analysis, Exception Handling, Amortized Analysis

Abstract

Automatic amortized resource analysis (AARA) is a type-based technique for inferring symbolic resource bounds for programs at compile time. Since its first introduction, the technique has been extended to the analysis to different resource metrics, evaluation strategies, non-linear bounds, and various language features.

This thesis builds upon AARA. The contribution consists of two parts. First, we present a new soundness proof of the type system of AARA with respect to a small-step, operational cost semantics on an abstract machine that makes control flow explicit. Compared to the big-step, structural cost semantics adopted in the previous works, it leads to a more concise type soundness proof that is amenable to extension to complex language features, such as polymorphism and nonstandard control flows. Second, we extend the technique of AARA to a language with exception handling in the style of Standard ML. We present a type system, prove its soundness by extending the small-step soundness proof, and show resource safety as a corollary of the type soundness theorem. We discuss how type inference can be automated to achieve, for the first time, automatic amortized resource analysis for programs with exception handling.

Acknowledgments

The world of programming languages has inspired me in an indescribable way. It has been a privilege to have encountered and explored the field and I am eternally grateful for those who made my journey possible.

First and foremost, I want to thank my advisor, Jan Hoffmann. His insights and character greatly enlightened not only my work, including this thesis, but also my personal growth outside of the research. I cannot thank enough for his selfless guidance throughout my study. I want to thank Professor Robert Harper for insightful feedback on this thesis, for 15-312 which introduced me to the world of programming languages. I am extremely grateful for his patience and for inspiring me in the field throughout the years. I would also like to thank Professor Stephanie Balzer, for her mentorship and for encouraging me at many critical points during my time at CMU.

I would also like to shoutout to friends in the Principles of Programming group for fun and fruitful discussions, and to Tracy Farbacher for her kindness and help over the course of my Master's program.

Last but not least, I want to thank my parents and sister for their unconditional love throughout my life.

Contents

Chapter 1

Introduction

1.1 AARA

Automatic amortized resource analysis (AARA) is a type-based technique for inferring symbolic resource bounds for programs at compile time [6]. The central pieces of this technique are the following:

- A type system, consisting of local inference rules, codifies the resource bounds of programs. Type checking certifies resource bounds. The derivation tree is the proof. Type inference generates resource bounds.
- The type system is usually a combination of the standard type system for the underlying language and some form of resource annotations. Type checking and inference then amount to a traditional checking/inference method such as Hindley–Milner–Damas and a process of deriving resource annotations. Resource annotations are derived by solving a numeric optimization problem, usually a linear program induced by the constraints in the typing rules.
- The type system incorporates potential method (as in amortized analysis), which gives symbolic, amortized resource bounds of programs.
- The analysis is proven sound by showing the soundness of the type system, with respect to a cost semantics that associates closed programs with an evaluation cost.

The type-based approach makes AARA compositional by nature and extensible to various programming paradigms. The technique has been developed to support different resource metrics, evaluation strategies, non-linear bounds, and various language features[6]. This thesis develops some of the missing pieces in AARA: (1) a new soundness proof that would be helpful in the presence of complex language features (2) an extension of AARA to exception handling.

1.2 Static Analysis for Exception Handling

Most of the static analysis works for exceptions focus on estimating uncaught exceptions using control flow analysis and/or type and effect systems.[7][8][3][4][2][1]. The focus of this line of works is to achieve both precision and efficiency of the analysis. It is motivated by software engineering applications where uncaught exceptions cause programs to terminate abruptly with little information for debugging.

However, exception handling is much more than exiting program execution and reporting errors. Exceptions, especially in functional languages such as Standard ML, are oftentimes used by programmers as a way to utilize nonstandard control flows. For example, in a program that multiplies a list of numbers, an exception can be raised when the program encounters a zero in the list. Its handler can then bypass normal control flows and return zero immediately. In this sense, exceptions are not 'exceptional" (i.e. rare cases of failures, anomalies), but rather a mechanism with which programmers use to transfer control flows when desired, akin to the role of continuations and algebraic effects.

Under this view, program analysis for exceptions should not make assumptions about how and why exceptions are used. Resource bounds when the exceptions are raised (and potentially handled), are as relevant as when other constructs are used. To our knowledge, this thesis is the first work of resource analysis in the presence of exception handling.

1.3 Outline

The rest of this thesis is organized as follows:

1. In Chapter 2, we describe a soundness proof of AARA with respect to a small-step, operational semantics on an abstract machine that we call K-machine.
2. In Chapter 3, we present a type system for a language with exception handling, where exceptions are implemented to be a globally fixed type. We prove its soundness by adapting the proof in Chapter 2.
3. In Chapter 4, we extend Chapter 3 to allow exceptions to be dynamically classified values, as implemented in Standard ML. We show how the type system can be adapted to accommodate such for AARA.
4. In Chapter 5, we summarize the results, discuss automation of the type system, and suggest future directions for this work.

Chapter 2

Soundness Proof via K-machine

In this chapter, we describe a soundness proof of AARA with respect to a small-step, operational cost semantics. We define our semantics using an abstract machine, which we call K-machine, that makes control flows explicit in its state.

2.1 Language

Consider the following language with a base type of unit, general recursive functions, and lists.

$$\begin{array}{rcll}
v & ::= & x & \text{variable} \\
 & \mid & \langle\rangle & \text{unit} \\
 & \mid & \mathtt{fun}(f.x.e) & \text{function} \\
 & \mid & \mathtt{nil} & \text{empty list} \\
 & \mid & \mathtt{cons}(v_1, v_2) & \text{list construction} \\
 & & & \\
e & ::= & \mathtt{ret}(v) & \text{return} \\
 & \mid & v_1(v_2) & \text{function application} \\
 & \mid & \mathtt{let}(e_1; x.e_2) & \text{sequence} \\
 & \mid & \mathtt{match}(v)(e_0; x.y.e_1) & \text{match list} \\
 & \mid & \mathtt{tick}(q) & \text{consume/free up resource}
\end{array}$$

The syntax separates *expressions* from *values*. *Values* are data that can carry *potential* (which we will explain below). They include variables, units, recursive functions, empty lists, and inductive cases of list construction. *Expressions* are computation that can consume or free up potential when executed. They include returns, which lift values to expressions, function applications, let-binds that sequence two expressions, and pattern matchers for lists. In addition, we have $\mathtt{tick}(q)$, which has no computational meaning, but signals resource changes. $\mathtt{tick}(q)(q \geq 0)$ says the program consumes q amount of resource; $\mathtt{tick}(q)(q \leq 0)$ says the program frees up q amount of resource. We assume they have been inserted by programmers before typing and analysis.

Despite being minimal, this language is sufficient to showcase the key ideas of AARA and our extension presented in this thesis. We believe many developments of AARA, including non-linear cost bounds and other standard functional language constructs (such as products, sums, and recursive types), are orthogonal to the ideas below. The technique presented below can be easily extended to those features and constructs. For the remainder of this thesis, we develop our type systems, proofs, and exception handling constructs on top of this small language.

2.2 Cost Semantics

K-machine

Now, to define the cost semantics of our language, we follow the formulation in [5] to introduce an abstract machine, called K-machine. K-machine includes an explicit control stack that records the work that remains to be done after an instruction is executed.

During program execution, a *state* $s; q$ of a computation says:

- We have q amount of resource available.
- s is one of the following two:
  - $k \triangleright e$. This says a closed expression e is being evaluated on a *stack* k.
  - $k \triangleleft v$. This says a closed value v is being returned to a *stack* k.

A *stack* k consists of a list of *frames*. Each *frame* f is a let-binder $x.e$ that sequences computation. The full syntax is summarized below:

$$\begin{array}{rcll} f & ::= & x.e & \text{sequence} \\ k & ::= & \epsilon & \text{empty stack} \\ & \mid & k; f & \text{frame sequence} \\ s & ::= & k \triangleright e & \text{evaluate expression} \\ & \mid & k \triangleleft v & \text{return value} \end{array}$$

Transitions

With this representation, we now define the dynamics of our language.
Initial and final states of computation are:

$$\frac{}{\epsilon \triangleright e; q \quad \textsf{initial}}\ \textsc{D-Init} \qquad \frac{}{\epsilon \triangleleft v; q \quad \textsf{final}}\ \textsc{D-Final}$$

Transition of states is given by:

$\boxed{s; q \mapsto s'; q'}$

$$\frac{p \geq q}{k \triangleright \texttt{tick}(q); p \mapsto k \triangleleft \langle\rangle; p - q}\ \textsc{D-Tick} \qquad \frac{}{k \triangleright \texttt{ret}(v); q \mapsto k \triangleleft v; q}\ \textsc{D-Ret}$$

$$\frac{}{k \triangleright \texttt{fun}(f.x.e)(v_2); q \mapsto k \triangleright [\texttt{fun}(f.x.e), v_2/f, x]e; q}\ \textsc{D-Fun}$$

$$\frac{}{k \triangleright \texttt{let}(e_1; x.e_2); q \mapsto k; x.e_2 \triangleright e_1; q}\ \text{D-LET} \qquad \frac{}{k; x.e \triangleleft v; q \mapsto k \triangleright [v/x]e; q}\ \text{D-SEQ}$$

$$\frac{}{k \triangleright \texttt{match}(\texttt{nil})(e_0; x.y.e_1); q \mapsto k \triangleright e_0; q}\ \text{D-NIL}$$

$$\frac{}{k \triangleright \texttt{match}(\texttt{cons}(v_1, v_2))(e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q}\ \text{D-CONS}$$

Rule D-TICK implies when program execution intends to consume q amount of resource, the state of the program must have at least q amount of resource for it to proceed. The rest of the rules is straightforward. This semantics presents a standard dynamics, subject to a quantitative resource constraint.

To show the soundness of AARA under this semantics, it suffices to show the following theorem:
Theorem 2.2.1 (Resource safety)**.**
For a closed expression e, if it is well-typed with some resource bound q according to the AARA type system (presented below), then starting with q amount of resource (i.e. initial state $\epsilon \triangleright e; q$) the program execution does not get stuck.

2.3 Type System

Now we present our type system that would give rise to a sound AARA. It is in spirit the same as the ones presented in the previous AARA works [6], but reformulated to account for the syntactic separation of values from expressions and our use of K-machine.

We have two classes of types for values and expressions respectively: τ for values, and annotated type $A = \langle \tau, q \rangle$ for expressions. $\langle \tau, q \rangle$ means a value of type τ plus q potential. $\langle \tau_1, q_1 \rangle \to \langle \tau_2, q_2 \rangle$ is a function type $\tau_1 \to \tau_2$ where q_1 potential is given with the input and q_2 potential is guarenteed upon function returns. $L(\langle \tau, q \rangle)$ is a list type, where each element is of type τ and each carries q potential.

$$\begin{array}{rcll} \tau & ::= & \texttt{unit} & \text{unit} \\ & | & A \to B & \text{function} \\ & | & L(A) & \text{list} \\ \\ A & ::= & \langle \tau, q \rangle & \text{potential} \end{array}$$

Affine Types

Values carry *potential*, in the same sense in amortized analysis. Potentials are morally "tokens" of resource (or to make the name more fit, the "ability" to do work(consume resource)). Therefore, the type system treats potential in an affine manner, so that the resource is not duplicated out of nowhere in the analysis.

To put it concretely, the typing rules treat all variables as affine resource by default. Contraction is only allowed when the potential involved is properly shared (See 2.3).

Value Typing

Values typing stipulates the exact potential each value carries.
$\boxed{\Gamma; q \vdash v : \tau}$ says under context Γ, a value v of type τ carries q potential.

$$\frac{}{x : \tau; 0 \vdash x : \tau}\ \textsc{T-var} \qquad \frac{}{\cdot; 0 \vdash \langle\rangle : \texttt{unit}}\ \textsc{T-triv} \qquad \frac{p \geq 0}{\cdot; 0 \vdash \texttt{nil} : L(\langle \tau, p\rangle)}\ \textsc{T-nil}$$

$$\frac{q = p + q_1 + q_2 \quad \Gamma_1; q_1 \vdash v_1 : \tau \quad \Gamma_2; q_2 \vdash v_2 : L(\langle \tau, p\rangle) \quad \Gamma = \Gamma_1, \Gamma_2}{\Gamma; q \vdash \texttt{cons}(v_1, v_2) : L(\langle \tau, p\rangle)}\ \textsc{T-list}$$

Variables carry zero potential before being substituted. $\langle\rangle$ carries zero potential. A list of type $L(\langle \tau, p\rangle)$ carries p potential per element, in addition to the potential carried by the element itself.

$$\frac{|\Gamma| = \Gamma \quad \Gamma, f : A \to B, x : \tau; q \vdash e : B \quad A = \langle \tau, q\rangle \quad A \to B <: A' \to B'}{\Gamma; 0 \vdash \texttt{fun}(f.x.e) : A' \to B'}\ \textsc{T-fun}$$

Functions take in a value and some potential, and if they terminate, return some value and some potential. We chose to treat functions as values that do not carry potential. This way, we can reuse/reinvoke functions as needed. This means in the typing rule, we need to require potential of the function and the context Γ to be zero. $|\Gamma|$ is a point-wise application of $|\cdot|$ to every type in Γ, where $|\cdot|$ is defined by the following:

$$\begin{aligned} |\texttt{unit}| &= \texttt{unit} \\ |L(A)| &= L(|A|) \\ |A \to B| &= A \to B \\ |\langle \tau, q\rangle| &= \langle |\tau|, 0\rangle \end{aligned}$$

The constraint $|\Gamma| = \Gamma$ essentially says functions are not allow to use potential carrying variables from the context. Notice this means we cannot type check any code as it is, for example, a function defined in a context with a variable $x : L(A)$. But it does not limit the language's expressiveness, since we can transform the function to take in an extra argument of $x : L(A)$, and have the callsites responsible for the potential.
In addition, we have explicit subtyping in T-FUN rule to be able to relax the resource bound of a function. Subtyping relaxes the resource bound represented in a type to a looser one. The formal definition of subtyping is the following:
$\boxed{\tau_1 <: \tau_2}$ $\boxed{A_1 <: A_2}$

$$\frac{}{\texttt{unit} <: \texttt{unit}}\ \textsc{SB-unit} \qquad \frac{q' \leq q \quad \tau_1 <: \tau_2}{\langle \tau_1, q\rangle <: \langle \tau_2, q'\rangle}\ \textsc{SB-pot}$$

$$\frac{A_2 <: A_1 \quad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2}\ \textsc{SB-fun} \qquad \frac{A_1 <: A_2}{L(A_1) <: L(A_2)}\ \textsc{SB-list}$$

This explicit subtyping of function values in T-FUN turned out to be necessary to have type preservation in our language. To see why, consider the following piece of code:

```
f1: <unit,0> -> <unit,0>
f2: <(<unit,1> -> <unit,0>),0> -> <unit,0>

f1(x:unit) = match [] with
                |  nil => return x;
                |  y::ys => f1(<>);f1(<>);

let
    x = ret(f1)
in
    f2(x)
end
```
The expression `f2(f1)` is ill-typed but the code above (which steps to `f2(f1)`) is well-typed, since we can subtype the expression `ret(f1)` (see T-SUB below). Retyping function `f1` to type `<unit,1> -> <unit,0>` is not possible without explicit subtyping on function values.

Although type preservation is not necessary for type soundness in some cases, we adopt this patch here so as not to disrupt the proof strategy to type soundness.

Expression Typing

Expressions consume or free up potential and, if they terminate, evaluate to values.
$\boxed{\Gamma; q \vdash e : \langle \tau, p \rangle}$ says under context Γ, an expression e, if it terminates, evaluates to a value of type τ and the starting q potential is enough to cover potential carried in the value of type τ plus extra p potential.

$$\frac{\Gamma; q \vdash v : \tau}{\Gamma; q \vdash \texttt{ret}(v) : \langle \tau, 0 \rangle}\ \text{T-RET} \qquad \frac{\Gamma_1; q \vdash e_1 : \langle \tau_1, q_1 \rangle \quad \Gamma_2, x : \tau_1; q_1 \vdash e_2 : B}{\Gamma_1, \Gamma_2; q \vdash \texttt{let}(e_1; x.e_2) : B}\ \text{T-LET}$$

$$\frac{q = q_1 + q_2 \quad \Gamma_1; 0 \vdash v_1 : \langle \tau, q_1 \rangle \to B \quad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; q \vdash v_1(v_2) : B}\ \text{T-APP} \qquad \frac{q \geq 0}{\cdot; q \vdash \texttt{tick}(q) : \langle \texttt{unit}, 0 \rangle}\ \text{T-TICK}$$

$$\frac{q = q_1 + q_2 \quad \Gamma_1; q_1 \vdash v : L(\langle \tau, p \rangle) \quad \Gamma_2; q_2 \vdash e_0 : B \quad \Gamma_2, x : \tau, y : L(\langle \tau, p \rangle); q_2 + p \vdash e_1 : B}{\Gamma_1, \Gamma_2; q \vdash \texttt{match}(v)(e_0; x.y.e_1) : B}\ \text{T-MATCH}$$

All the rules above treat the variables linearly. We add the following structural rules to type programs that are well-typed under usual cost-agnostic structural type systems.

$$\frac{\Gamma; p \vdash e : \langle \tau, p' \rangle \quad q \geq p \quad q - q' \geq p - p'}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ \text{T-RELAX} \qquad \frac{\Gamma; q \vdash e : B}{\Gamma, x : \tau; q \vdash e : B}\ \text{T-WEAK}$$

T-RELAX and T-WEAK relax type and potential to be affine, instead of linear.

$$\frac{\Gamma; q \vdash e : \langle \tau', q' \rangle \quad \tau' <: \tau}{\Gamma; q \vdash e : \langle \tau, q' \rangle}\ \text{T-SUB} \qquad \frac{\Gamma, x : \tau; q \vdash e : B \quad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B}\ \text{T-SUPER}$$

T-SUB and T-SUPER allow us to type `match` expressions.

$$\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : B \quad \tau \curlyvee (\tau_1, \tau_2)}{\Gamma, x : \tau; q \vdash [x, x/x_1, x_2]e : B}\ \text{T-CONTRACT}$$

Lastly, contraction is allowed by properly sharing the potential involved, i.e. potential is not duplicated.

Sharing

$\curlyvee$ is formally defined to be: $\boxed{\tau \curlyvee (\tau_1, \tau_2)}$, which says the potential carried by a value of type τ equals to that of a value of type τ_1, plus that of a value of type τ_2.
$\boxed{A \curlyvee (A_1, A_2)}$

$$\frac{}{\texttt{unit} \curlyvee (\texttt{unit}, \texttt{unit})}\ \text{SH-UNIT} \qquad \frac{}{A \to B \curlyvee (A \to B, A \to B)}\ \text{SH-FUN}$$

$$\frac{q = q_1 + q_2 \quad q_1 \geq 0 \quad q_2 \geq 0 \quad \tau \curlyvee (\tau_1, \tau_2)}{\langle \tau, q \rangle \curlyvee (\langle \tau_1, q_1 \rangle, \langle \tau_2, q_2 \rangle)}\ \text{SH-POT} \qquad \frac{A \curlyvee (A_1, A_2)}{L(A) \curlyvee (L(A_1), L(A_2))}\ \text{SH-LIST}$$

Stack/State Typing

Now with the type system above, we have specified the set of closed, well-typed expressions and their resource bounds q. (when $\cdot; q \vdash e : B$). To show the resource safety theorem 2.2.1, it suffices to show a program exexution starting from $\epsilon \triangleright e; q$ does not get stuck.

To that end, we define a notion of well-formedness for program states that would be preserved during execution.
$\boxed{k \lhd\!: A}$. This says k is good given a value and potential of A.

$$\frac{}{\epsilon \lhd\!: A}\ \text{FR-EMP} \qquad \frac{k \lhd\!: B \quad x : \tau; q \vdash e : B}{k; x.e \lhd\!: \langle \tau, q \rangle}\ \text{FR-BND}$$

$\boxed{q \vdash s}$. A program execution state is *well-formed* when:

$$\frac{\cdot; q \vdash e : B \quad k \lhd\!: B}{q \vdash k \triangleright e}\ \text{ST-EXP} \qquad \frac{q = q_1 + q_2 \quad \cdot; q_1 \vdash v : \tau \quad k \lhd\!: \langle \tau, q_2 \rangle}{q \vdash k \lhd v}\ \text{ST-VAL}$$

2.4 Soundness

The remaining pieces to show 2.2.1 amount to proving type soundness via progress and preservation mediated by such well-formedness. The key lemmas and theorems are:

Lemma 2.4.1 (Substitution)**.**
If $\cdot; q_1 \vdash v_1 : \tau_1$*:*
(A) If $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$*, then* $\Gamma; q_1 + q_2 \vdash [v_1/x_1]v : \tau$*.*
(B) If $\Gamma, x_1 : \tau_1; q_2 \vdash e : B$*, then* $\Gamma; q_1 + q_2 \vdash [v_1/x_1]e : B$*.*

Theorem 2.4.2 (Progress)**.**
If $q \vdash s$*, then either* s *final or* $s; q \mapsto s'; q'$*.*

Theorem 2.4.3 (Preservation)**.**
If $q \vdash s$ *and* $s; q \mapsto s_0; q_0$ *then* $q_0 \vdash s_0$*.*

Since the language here is a subset of the one in Chapter 3, the proof is subsumed by the type soundness proof there. We omit the full proof for brevity.

Chapter 3

Exception Handling

In this chapter, we extend AARA to a language with exception handling. We first start with a setup where exception type is simply a globally fixed type (let $\tau_{\texttt{exn}}$ be this type) in our language. In the next chapter, we relax this to match the exceptions in Standard ML.

3.1 Language

On top of the small language presented in Chapter 2, we add two additional constructs in the class of expressions, similar to the setup in [5]. $\texttt{raise}(v)$ raises an exception with value v. For example, if $\tau_{\texttt{exn}} = \texttt{Int}$, $\texttt{raise}(0)$ would be one valid way to raise an exception and the value 0 would be passed to the handler. $\texttt{try}(e_1; x.e_2)$ sets up exception handlers. It first evaluates expression e_1. If it raises an exception v, the control flow would be transfered to the handler, and v would be bound to variable x, e_2 would be evaluated; otherwise, the handler is ignored.

$$
\begin{array}{rcll}
e & ::= & ... & \\
 & | & \texttt{raise}(v) & \text{raise exception} \\
 & | & \texttt{try}(e_1; x.e_2) & \text{exception handler}
\end{array}
$$

Again, following [5], we extend the semantics for exception handling. A stack k is now a list of $x.e$ and/or $\texttt{try}(\_; x.e)$. A program state can now also be $k \blacktriangleleft v$, which says an exception v is thrown to the stack k.

$$
\begin{array}{rcll}
f & ::= & x.e & \text{sequence} \\
 & | & \texttt{try}(\_; x.e) & \text{exception handler} \\
k & ::= & \epsilon & \text{empty stack} \\
 & | & k; f & \text{frame sequence} \\
s & ::= & k \triangleright e & \text{evaluate expression} \\
 & | & k \triangleleft v & \text{return value} \\
 & | & k \blacktriangleleft v & \text{exception}
\end{array}
$$

Initial states are still $\epsilon \triangleright e; q$ and final states are now:

$$
\frac{}{\epsilon \triangleleft v; q \quad \textsf{final}}\ \textsc{D-final}
\qquad
\frac{}{\epsilon \blacktriangleleft v; q \quad \textsf{final}}\ \textsc{D-final-exn}
$$

Transition relation is extended with:

$$\frac{}{k \triangleright \texttt{raise}(v); q \mapsto k \blacktriangleleft v; q}\ \textsc{D-raise} \qquad \frac{}{k; \texttt{try}(\_; x.e) \blacktriangleleft v; q \mapsto k \triangleright [v/x]e; q}\ \textsc{D-handle}$$

$$\frac{}{k; x.e \blacktriangleleft v; q \mapsto k \blacktriangleleft v; q}\ \textsc{D-exn} \qquad \frac{}{k; \texttt{try}(\_; x.e) \triangleleft v; q \mapsto k \triangleleft v; q}\ \textsc{D-normal}$$

$$\frac{}{k \triangleright \texttt{try}(e_1; x.e); q \mapsto k, \texttt{try}(\_; x.e) \triangleright e_1; q}\ \textsc{D-try}$$

3.2 Type System

Statics of exception handling is enriched with resource analysis via potential. $\texttt{raise}(v)$ is similar to $\texttt{ret}(v)$, it needs potential q to pay for the potential stored in exception value being thrown to the stack. The resulting type can be any arbitrary B, since no normal value would be returned and bound to the following computation. $\texttt{try}(e_1; x.e_2)$ needs potential to account for both cases of handler being invoked or not. In either case, the resulting type should be that of the entire `try` expression.

$$\frac{\Gamma; q \vdash v : \tau_{\texttt{exn}}}{\Gamma; q \vdash \texttt{raise}(v) : B}\ \textsc{T-raise}$$

$$\frac{q = q_1 + q_2 \quad \Gamma_1; q_1 \vdash e_1 : B \quad \Gamma_2, x : \tau_{\texttt{exn}}; q_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2; q \vdash \texttt{try}(e_1; x.e_2) : B}\ \textsc{T-handle}$$

Problem in Soundness Proof

This type system is in fact sound. However, its proof in the style of progress and preservation sketched in Chapter 2 demands a more precise version of well-typedness. To observe why, let us revisit T-HANDLE and T-RELAX. Consider a case where the expression $\texttt{try}(e_1; x.e_2)$ is typed with the following derivation:

$$\frac{q' \geq q \quad q' - r' \geq q - r \quad \dfrac{q = q_1 + q_2 \quad \Gamma_1; q_1 \vdash e_1 : \langle \tau, r \rangle \quad \Gamma_2, x : \tau_{\texttt{exn}}; q_2 \vdash e_2 : \langle \tau, r \rangle}{q \vdash \texttt{try}(e_1; x.e_2) : \langle \tau, r \rangle}\ \textsc{T-handle}}{q' \vdash \texttt{try}(e_1; x.e_2) : \langle \tau, r' \rangle}\ \textsc{T-relax}$$

The use of T-RELAX looks unsound at the first sight. Raising the starting potential from q to q' is only enough to leave an additional $r' - r$ potential, but if we blend T-RELAX into T-HANDLE, both q_1 and q_2 need to have additional $q' - q$ potential. Since $q = q_1 + q_2$, the total q should be raised with $2 * (q' - q)$!

However, this rule is in fact sound, since e_1 and e_2 might both get evaluated, but never normally return at the same time. If e_1 evaluates to completion without raising an exception, the handler $x.e_2$ would be skipped, therefore we don't need potential for that. We have exactly enough for e_1 to leave extra $r' - r$ potential. If the handler is invoked, we know e_1 raised an exception. Since $q_1 \vdash e_1 : \langle \tau, r \rangle$, we know morally q_1 is enough for e_1 to reach the exception, then intuitively we should be able to derive $q_1 \vdash e_1 : \langle \tau, r' \rangle$, since the resulting type of an exception-raising expression is irrelevant according to T-RAISE.

Notice similar reasoning is not necessary for `match` statements, where only one of the two

branches of `match` will be evaluated, so the typing rule simply says, if in both branches, q potential is enough to fuel through evaluation, leaving r potential, then the entire `match` can evaluate to completion, starting with q potential, leaving r potential.
This use of T-HANDLE and T-RELAX blocks the preservation proof, when we need to show the stepping $k \triangleright \texttt{try}(e_1; x.e_2); q \;\mapsto\; k, \texttt{try}(\_; x.e_2) \triangleright e_1; q$ preserves well-formedness. The well-formedness would mean e_1 is well-typed, the stack $k, \texttt{try}(\_; x.e_2)$ is well-formed, and with resource q, the two compose "well". The assumption tells us e_1 is well-typed ($q_1 \vdash e_1 : \langle \tau, p \rangle$), which only tells us q_1 potential is enough to cover *either* e_1 evaluating to $v : \tau$ plus potential p, *or* e_1 raising an exception. It does not tell us if e_1 raises an exception, if there is any remaining potential we could (also need to) use to evaluate the handler $x.e_2$ and the computation left on the stack k.

3.3 A More Precise Type System

With this intuition, we develop the following type system that leads to the final proof of type soundness.

Expressions are now typed to $\langle \tau, q \rangle \oplus p$, which means the expression evaluates to a value and potential of $\langle \tau, q \rangle$, *or* raises an exception, leaving p potential. The syntax of types is adjusted accordingly to:

$$\begin{array}{rcll} \tau & ::= & \texttt{unit} & \text{unit} \\ & | & A \to (B \oplus p) & \text{function} \\ & | & L(A) & \text{list} \\ \\ A & ::= & \langle \tau, q \rangle & \text{potential} \end{array}$$

Subtyping and Sharing

Subtyping and sharing relations on types remain morally the same. $\oplus p$ is treated contravariant in the function return type, similar to q in $\langle \tau, q \rangle$.

$\boxed{\tau \curlyvee (\tau_1, \tau_2)}\ \boxed{A \curlyvee (A_1, A_2)}$

$$\frac{}{\texttt{unit} \curlyvee (\texttt{unit}, \texttt{unit})}\ \text{SH-UNIT} \qquad \frac{}{A \to (B \oplus p) \curlyvee (A \to (B \oplus p), A \to (B \oplus p))}\ \text{SH-FUN}$$

$$\frac{q = q_1 + q_2 \quad q_1 \geq 0 \quad q_2 \geq 0 \quad \tau \curlyvee (\tau_1, \tau_2)}{\langle \tau, q \rangle \curlyvee (\langle \tau_1, q_1 \rangle, \langle \tau_2, q_2 \rangle)}\ \text{SH-POT} \qquad \frac{A \curlyvee (A_1, A_2)}{L(A) \curlyvee (L(A_1), L(A_2))}\ \text{SH-LIST}$$

$\boxed{\tau_1 <: \tau_2}\ \boxed{A_1 <: A_2}$

$$\frac{}{\texttt{unit} <: \texttt{unit}}\ \text{SB-UNIT} \qquad \frac{q' \leq q \quad \tau_1 <: \tau_2}{\langle \tau_1, q \rangle <: \langle \tau_2, q' \rangle}\ \text{SB-POT}$$

$$\frac{A_2 <: A_1 \quad B_1 <: B_2 \quad p_2 \leq p_1}{A_1 \to (B_1 \oplus p_1) <: A_2 \to (B_2 \oplus p_2)}\ \text{SB-FUN} \qquad \frac{A_1 <: A_2}{L(A_1) <: L(A_2)}\ \text{SB-LIST}$$

Value Typing

Similarly, value typing remains the same. Explicit subtyping in T-FUN is updated for $\oplus p$.
$\boxed{\Gamma; q \vdash v : \tau}$

$$\frac{}{x : \tau; 0 \vdash x : \tau}\ \text{T-VAR} \qquad \frac{}{\cdot; 0 \vdash \langle\rangle : \texttt{unit}}\ \text{T-TRIV} \qquad \frac{p \geq 0}{\cdot; 0 \vdash \texttt{nil} : L(\langle \tau, p \rangle)}\ \text{T-NIL}$$

$$\frac{|\Gamma| = \Gamma \quad \Gamma, f : A \to (B \oplus p), x : \tau; q \vdash e : B \oplus p \quad A = \langle \tau, q \rangle \quad A \to (B \oplus p) <: A' \to (B' \oplus p')}{\Gamma; 0 \vdash \texttt{fun}(f.x.e) : A' \to (B' \oplus p')}\ \text{T-FUN}$$

$$\frac{q = p + q_1 + q_2 \quad \Gamma_1; q_1 \vdash v_1 : \tau \quad \Gamma_2; q_2 \vdash v_2 : L(\langle \tau, p \rangle) \quad \Gamma = \Gamma_1, \Gamma_2}{\Gamma; q \vdash \texttt{cons}(v_1, v_2) : L(\langle \tau, p \rangle)}\ \text{T-LIST}$$

Expression Typing

$\boxed{\Gamma; q \vdash e : B \oplus p}$ says e, if it terminates, evaluates to a value and potential of B, *or* raises an exception, leaving p potential. Notice the symmetry between T-LET and T-HANDLE. In T-LET, normal return composes between e_1 and e_2, while exceptional flow $\oplus p$ falls through between them. In T-HANDLE, normal return falls through e_1 and e_2, while exceptional flow composes between.

$$\frac{\Gamma; q \vdash v : \tau}{\Gamma; q \vdash \texttt{ret}(v) : \langle \tau, 0 \rangle \oplus 0}\ \text{T-RET} \qquad \frac{\Gamma_1; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p \quad \Gamma_2, x : \tau_1; q_1 \vdash e_2 : B \oplus p}{\Gamma_1, \Gamma_2; q \vdash \texttt{let}(e_1; x.e_2) : B \oplus p}\ \text{T-LET}$$

$$\frac{q = q_1 + q_2 \quad \Gamma_1; 0 \vdash v_1 : \langle \tau, q_1 \rangle \to (B \oplus p) \quad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; q \vdash v_1(v_2) : B \oplus p}\ \text{T-APP} \qquad \frac{}{\cdot; q \vdash \texttt{tick}(q) : \langle \texttt{unit}, 0 \rangle \oplus 0}\ \text{T-TICK}$$

$$\frac{q = q_1 + q_2 \quad \Gamma_1; q_1 \vdash v : L(\langle \tau, r \rangle) \quad \Gamma_2; q_2 \vdash e_0 : B \oplus p \quad \Gamma_2, x : \tau, y : L(\langle \tau, r \rangle); q_2 + r \vdash e_1 : B \oplus p}{\Gamma_1, \Gamma_2; q \vdash \texttt{match}(v)(e_0; x.y.e_1) : B \oplus p}\ \text{T-MATCH}$$

$$\frac{\Gamma; q \vdash v : \tau_{\texttt{exn}}}{\Gamma; q \vdash \texttt{raise}(v) : B \oplus 0}\ \text{T-RAISE}$$

$$\frac{\Gamma_1; q \vdash e_1 : B \oplus p_1 \quad \Gamma_2, x : \tau_{\texttt{exn}}; p_1 \vdash e_2 : B \oplus p}{\Gamma_1, \Gamma_2; q \vdash \texttt{try}(e_1; x.e_2) : B \oplus p}\ \text{T-HANDLE}$$

Structural rules are morally the same. T-RELAX also relaxes $\oplus p$.

$$\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2; q \vdash e : B \oplus p \quad \tau \curlyvee (\tau_1, \tau_2)}{\Gamma, x : \tau; q \vdash [x, x/x_1, x_2]e : B \oplus p}\ \text{T-CONTRACT} \qquad \frac{\Gamma; q \vdash e : B \oplus p}{\Gamma, x : \tau; q \vdash e : B \oplus p}\ \text{T-WEAK}$$

$$\frac{\Gamma; q' \vdash e : \langle \tau, r' \rangle \oplus p' \quad q \geq q' \quad q - r \geq q' - r' \quad q - p \geq q' - p'}{\Gamma; q \vdash e : \langle \tau, r \rangle \oplus p}\ \text{T-RELAX}$$

$$\frac{\Gamma; q \vdash e : \langle \tau', r \rangle \oplus p \quad \tau' <: \tau}{\Gamma; q \vdash e : \langle \tau, r \rangle \oplus p}\ \text{T-SUB} \qquad \frac{\Gamma, x : \tau; q \vdash e : B \oplus p \quad \tau' <: \tau}{\Gamma, x : \tau'; q \vdash e : B \oplus p}\ \text{T-SUPER}$$

Stack/State Typing

With $\oplus p$, we can now define a precise-enough notion of well-formedness for program states, in the presence of exceptions. A stack is well-formed when:
$\boxed{k \lhd\!: A \oplus p}$. This says k is good, given either a value and potential of A, or an exception and potential p.

$$\frac{p \geq 0}{\epsilon \lhd\!: A \oplus p}\ \textsc{FR-emp} \qquad \frac{k \lhd\!: B \oplus p \quad x : \tau; q \vdash e : B \oplus p}{k; x.e \lhd\!: \langle \tau, q \rangle \oplus p}\ \textsc{FR-bnd}$$

$$\frac{k \lhd\!: B \oplus p_0 \quad x : \tau_{\texttt{exn}}; p \vdash e : B \oplus p_0}{k; \texttt{try}(\_; x.e) \lhd\!: B \oplus p}\ \textsc{FR-exn}$$

Finally, a program execution state is well-formed when:
$\boxed{q \vdash s}$

$$\frac{\cdot; q \vdash e : B \oplus p \quad k \lhd\!: B \oplus p}{q \vdash k \rhd e}\ \textsc{ST-exp} \qquad \frac{q = q_1 + q_2 \quad \cdot; q_1 \vdash v : \tau \quad k \lhd\!: \langle \tau, q_2 \rangle \oplus p}{q \vdash k \lhd v}\ \textsc{ST-val}$$

$$\frac{q = q_1 + p \quad \cdot; q_1 \vdash v : \tau_{\texttt{exn}} \quad k \lhd\!: B \oplus p}{q \vdash k \blacktriangleleft v}\ \textsc{ST-exn}$$

3.4 Soundness

We now present the full proof of type soundness.

Lemma 3.4.1.
If $\Gamma; q \vdash v : \tau$*, then* $q \geq 0$.

Lemma 3.4.2.
If $\Gamma; q \vdash v : \tau$ *and* $|\tau| = \tau$*, then* $q = 0$.

Proof. Induct on $\Gamma; q \vdash v : \tau$.

- T-VAR: $q = 0$.
- T-TRIV: $q = 0$.
- T-NIL: $q = 0$.
- T-FUN: q.
- T-LIST: $v = \texttt{cons}(v_1, v_2), \tau = L(\langle \tau', p \rangle)$.
  By the assumption, $q = p + q_1 + q_2$, $\Gamma_1; q_1 \vdash v_1 : \tau'$, $\Gamma_2; q_2 \vdash v_2 : \tau$, $\Gamma = \Gamma_1, \Gamma_2$.
  Since $|L(\langle \tau', p \rangle)| = L(|\langle \tau', p \rangle|) = L(\langle |\tau'|, 0 \rangle) = L(\langle \tau', p \rangle)$, we have $p = 0$ and $|\tau'| = \tau'$.
  Using the IH, $q_1 = 0$ and $q_2 = 0$.
  Then $q = p + q_1 + q_2 = 0$.

□

Lemma 3.4.3 (Sharing)**.**
If $\tau \curlyvee (\tau_1, \tau_2)$ *and* $\cdot; q \vdash v : \tau$*, then* $\cdot; q_1 \vdash v : \tau_1$, $\cdot; q_2 \vdash v : \tau_2$*, where* $q = q_1 + q_2, q_1 \geq 0, q_2 \geq 0$.

Proof. Induction on $\tau \curlyvee (\tau_1, \tau_2)$.

- SH-UNIT: $\tau = \tau_1 = \tau_2 = \texttt{unit}$. Induct on $\cdot; q \vdash v : \texttt{unit}$, we get $v = \langle\rangle, q = 0$. Then let $q_1 = q_2 = 0$.
- SH-FUN: Similarly, $\tau = \tau_1 = \tau_2$, $q = q_1 = q_2 = 0$.
- SH-POT: Vacuous.
- SH-LIST: $\tau = L(A)$, $L(A) \curlyvee (L(A_1), L(A_2))$. By the assumption, $A \curlyvee (A_1, A_2)$. Induct on $\cdot; q \vdash v : \tau$.
  - T-NIL: $v = \texttt{nil}, q = 0$, $\cdot; 0 \vdash \texttt{nil} : L(\langle \tau, p \rangle)$ for any $p \geq 0$, which is guaranteed by $A \curlyvee (A_1, A_2)$
  - T-LIST: $v = \texttt{cons}(v_1, v_2)$, $A = \langle \tau', p \rangle$, $q = p + r_1 + r_2$, and $\cdot; r_1 \vdash v_1 : \tau'$, $\cdot; r_2 \vdash v_2 : L(\langle \tau', p \rangle)$.
    Invert $A \curlyvee (A_1, A_2)$, get $A_1 = \langle \tau_1', p_1 \rangle, A_2 = \langle \tau_2', p_2 \rangle, \tau' \curlyvee (\tau_1', \tau_2')$, and $p = p_1 + p_2$, $p_1 \geq 0, p_2 \geq 0$.
    Using the inner IH, $\cdot; r_1' \vdash v_1 : \tau_1'$, $\cdot; r_1'' \vdash v_1 : \tau_2'$, where $r_1 = r_1' + r_1'', r_1' \geq 0, r_1'' \geq 0$.
    Also, $\cdot; r_2' \vdash v_2 : L(\langle \tau_1', p_1 \rangle)$, $\cdot; r_2'' \vdash v_2 : L(\langle \tau_2', p_2 \rangle)$, where $r_2 = r_2' + r_2'', r_2' \geq 0, r_2'' \geq 0$.
    By T-LIST, $\cdot; r_1' + r_2' + p_1 \vdash \texttt{cons}(v_1, v_2) : L(\langle \tau_1', p_1 \rangle)$, and $\cdot; r_1'' + r_2'' + p_2 \vdash \texttt{cons}(v_1, v_2) : L(\langle \tau_2', p_2 \rangle)$, where $r_1' + r_2' + p_1 + r_1'' + r_2'' + p_2 = q$.

□

Lemma 3.4.4 (Transitivity of <:)**.**
Subtyping is transitive, i.e. if $\tau_1 <: \tau_2$, $\tau_2 <: \tau_3$, then $\tau_1 <: \tau_3$.
Lemma 3.4.5 (Value Subtyping)**.**
If $\cdot; q \vdash v : \tau$ and $\tau <: \tau'$, then $\cdot; q' \vdash v : \tau'$ where $q' \leq q$.

Proof. Induction on $\tau <: \tau'$.

- SB-UNIT: $\tau = \tau' = \texttt{unit}$, let $q' = q$.
- SB-POT: /
- SB-FUN: $\tau = A_1 \to (B_1 \oplus p_1)$, $\tau' = A_2 \to (B_2 \oplus p_2)$. $A_2 <: A_1$, $B_1 <: B_2$, and $p_2 \leq p_1$.
  Induct on $\cdot; q \vdash v : \tau$ to see $q = 0$, $v = \texttt{fun}(f.x.e)$. And $f : A_1' \to (B_1' \oplus p_1'), x : \tau_0; q_0 \vdash e : B_1'$, $A_1' = \langle \tau_0, q_0 \rangle$, where $A_1 <: A_1', B_1' <: B_1, p_1 \leq p_1'$.
  By transitivity of subtyping, $A_2 <: A_1', B_1' <: B_2$. Also $p_2 \leq p_1'$.
  By T-FUN, $\cdot; 0 \vdash v : A_2 \to (B_2 \oplus p_2)$.
- SB-LIST: $\tau = L(A_1)$, $\tau' = L(A_2)$, $A_1 <: A_2$.
  Induct on $\cdot; q \vdash v : \tau$. Either $v = \texttt{nil}$, then let $q' = q = 0$.
  Or $v = \texttt{cons}(v_1, v_2)$, where $A_1 = \langle \tau_0, p \rangle$ $q = q_1 + q_2 + p$, $\cdot; q_1 \vdash v_1 : \tau_0$, $\cdot; q_2 \vdash v_2 : L(\langle \tau_0, p \rangle)$.
  Also, $A_1 <: A_2$, so $A_2 = \langle \tau_0', p' \rangle$, $\tau_0 <: \tau_0'$, $p \geq p'$.
  By IH, $\cdot; q_1' \vdash v_1 : \tau_0'$, $\cdot; q_2' \vdash v_2 : L(\langle \tau_0', p' \rangle)$ with $q_1' \leq q_1, q_2' \leq q_2$.
  By T-LIST, $\cdot; p' + q_1' + q_2' \vdash v : L(\langle \tau_0', p' \rangle)$ and $p' + q_1' + q_2' \leq p + q_1 + q_2$.

□

Lemma 3.4.6. *If $\Gamma; q \vdash e : \langle \tau, r \rangle \oplus p$, then $\Gamma; q + d \vdash e : \langle \tau, r + d \rangle \oplus p + d$ when $d \geq 0$.*

Lemma 3.4.7. *If $k \lhd: \langle \tau, q\rangle \oplus p$, then $k \lhd: \langle \tau, q+d\rangle \oplus p+d$ when $d \geq 0$.*
Lemma 3.4.8 (State relaxing)**.** *If $q \vdash s$, then $q' \vdash s$ when $q' \geq q$.*

Lemma 3.4.9 (Substitution)**.**
If $\cdot; q_1 \vdash v_1 : \tau_1$:
(A) If $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$, then $\Gamma; q_1 + q_2 \vdash [v_1/x_1]v : \tau$.
(B) If $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \oplus p$, then $\Gamma; q_1 + q_2 \vdash [v_1/x_1]e : B \oplus p$.

Proof. Induction on $\cdot; q_1 \vdash v_1 : \tau_1$:

- T-VAR: /
- T-TRIV, T-FUN, T-NIL, T-LIST:
  (A): Induction on $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$
  - T-VAR: $q_2 = 0, v = x_1, \Gamma = \cdot, \tau_1 = \tau$. Then $[v_1/x_1]x_1 = v_1$, and $\cdot; q_1 \vdash v_1 : \tau_1$.
  - T-TRIV: /
  - T-FUN: $q_2 = 0, v = \texttt{fun}(f.x.e)(x_1 \neq x), \tau' <: \tau, \tau' = \langle \tau_x, q'\rangle \to (B \oplus p)$.
    $[v_1/x_1]\texttt{fun}(f.x.e) = \texttt{fun}(f.x.[v_1/x_1]e)$.
    By the assumption, $\Gamma, x_1 : \tau_1, f : \langle \tau_x, q'\rangle \to (B \oplus p), x : \tau_x; q' \vdash e : B \oplus p$, and $|\Gamma, x_1 : \tau_1| = \Gamma, x_1 : \tau_1$ (so $|\tau_1| = \tau_1$).
    Then by lemma 3.4.2, $q_1 = 0$.
    Using IH(B), $\Gamma, f : \langle \tau_x, q'\rangle \to (B \oplus p), x : \tau_x; q' + q_1 \vdash [v_1/x_1]e : B$, where $q_1 = 0$, $|\Gamma| = \Gamma$.
    By T-FUN, $\Gamma; 0 \vdash \texttt{fun}(f.x.[v_1/x_1]e) : \tau$.
  - T-NIL: /
  - T-LIST: $v = \texttt{cons}(v_1', v_2'), \tau = L(\langle \tau_0, p\rangle), \Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$.
    By the assumption $q_2 = p + q_1' + q_2'$, $\Gamma_1; q_1' \vdash v_1' : \tau_0$, and $\Gamma_2; q_2' \vdash v_2' : L(\langle \tau_0, p\rangle)$.
    Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
    - If $x_1 : \tau_1 \in \Gamma_1$, using IH(A), $\Gamma_1 \setminus (x_1 : \tau_1); q_1' + q_1 \vdash [v_1/x_1]v_1' : \tau_0$.
      By T-LIST, $\Gamma; q_2 + q_1 \vdash \texttt{cons}([v_1/x_1]v_1', v_2') : L(\langle \tau_0, p\rangle)$.
    - Otherwise, $x_1 : \tau_1 \in \Gamma_2$, using IH(A), $\Gamma_2 \setminus (x_1 : \tau_1); q_2' + q_1 \vdash [v_1/x_1]v_2' : L(\langle \tau_0, p\rangle)$.
      By T-LIST, $\Gamma; q_2 + q_1 \vdash \texttt{cons}(v_1', [v_1/x_1]v_2') : L(\langle \tau_0, p\rangle)$.

  (B): Induction on $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \oplus p$
  - T-RET: $e = \texttt{ret}(v'), B = \langle \tau, 0\rangle, p = 0$.
    By the assumption, $\Gamma, x_1 : \tau_1; q_2 \vdash v' : \tau$.
    Using IH(A), $\Gamma; q_2 + q_1 \vdash [v_1/x_1]v' : \tau$.
    By T-RET, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\texttt{ret}(v') : \langle \tau, 0\rangle \oplus 0$.
  - T-LET: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$ and $e = \texttt{let}(e_1; x.e_2)$.
    By the assumption, $\Gamma_1; q_2 \vdash e_1 : \langle \tau, q_3\rangle \oplus p$ and $\Gamma_2, x : \tau_1; q_3 \vdash e_2 : B \oplus p (x \neq x_1)$.
    Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
    - If $x_1 : \tau_1 \in \Gamma_1$, by IH(B), $\Gamma_1 \setminus (x_1 : \tau_1); q_2 + q_1 \vdash [v_1/x_1]e_1 : \langle \tau, q_3\rangle \oplus p$.
      By T-LET, $\Gamma; q_2 + q_1 \vdash \texttt{let}([v_1/x_1]e_1; x.e_2) : B \oplus p$.
    - Otherwise $x_1 : \tau_1 \in \Gamma_2$, then by IH(B), $\Gamma_2 \setminus (x_1 : \tau_1), x : \tau; q_3 + q_1 \vdash [v_1/x_1]e_2 : B \oplus p$.

Using T-RELAX on $\Gamma_1; q_2 \vdash e_1 : \langle \tau, q_3 \rangle \oplus p$, we have $\Gamma_1; q_2 + q_1 \vdash e_1 : \langle \tau, q_3 + q_1 \rangle \oplus p$.
By T-LET, $\Gamma; q_2 + q_1 \vdash \texttt{let}(e_1; x.[v_1/x_1]e_2) : B \oplus p$.

- T-APP: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2, e = v_1'(v_2')$.
  By the assumption, $q_2 = p_1 + p_2$. $\Gamma_1; 0 \vdash v_1' : \langle \tau, p_1 \rangle \to (B \oplus p)$ and $\Gamma_2; p_2 \vdash v_2' : \tau$.
  Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
  - If $x_1 : \tau_1 \in \Gamma_1$, by IH(A), $\Gamma_1 \setminus (x_1 : \tau_1); q_1 \vdash [v_1/x_1]v_1' : \langle \tau, p_1 \rangle \to (B \oplus p)$.
    Since $|\langle \tau, p_1 \rangle \to (B \oplus p)| = \langle \tau, p_1 \rangle \to (B \oplus p)$, by lemma 3.4.2, $q_1 = 0$.
    By T-APP, $\Gamma; q_2 \vdash [v_1/x_1]v_1'(v_2') : B \oplus p$.
  - Otherwise $x_1 : \tau_1 \in \Gamma_2$, then by IH(A), $\Gamma_2 \setminus (x_1 : \tau_1); p_2 + q_1 \vdash [v_1/x_1]v_2' : \tau$.
    By T-APP, $\Gamma; q_2 + q_1 \vdash v_1'([v_1/x_1]v_2') : B \oplus p$.
- T-TICK: /
- T-MATCH: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$ and $e = \texttt{match}(v)(e_0; x.y.e_1)$.
  By the assumption, $q_2 = q_1' + q_2'$, $\Gamma_1; q_1' \vdash v : L(\langle \tau_0, r \rangle)$, $\Gamma_2; q_2' \vdash e_0 : B \oplus p$, $\Gamma_2, x : \tau_0, y : L(\langle \tau_0, r \rangle); q_2' + r \vdash e_1 : B \oplus p$.
  Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
  - If $x_1 : \tau_1 \in \Gamma_1$, by IH(A), $\Gamma_1 \setminus (x_1 : \tau_1); q_1 + q_1' \vdash [v_1/x_1]v : L(\langle \tau_0, r \rangle)$.
    By T-MATCH, $\Gamma; q_1 + q_2 \vdash \texttt{match}([v_1/x_1]v)(e_0; x.y.e_1) : B \oplus p$.
  - Otherwise $x_1 : \tau_1 \in \Gamma_2$, then by IH(B), $\Gamma_2 \setminus (x_1 : \tau_1); q_2' + q_1 \vdash [v_1/x_1]e_0 : B \oplus p$, and $\Gamma_2 \setminus (x_1 : \tau_1), x : \tau_0, y : L(\langle \tau_0, r \rangle); q_2' + q_1 + r \vdash [v_1/x_1]e_1 : B \oplus p$.
    By T-MATCH, $\Gamma; q_1 + q_2 \vdash \texttt{match}(v)([v_1/x_1]e_0; x.y.[v_1/x_1]e_1) : B \oplus p$.
- T-RAISE: By the assumption, $p = 0$, and $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau_{\texttt{exn}}$.
  Using IH(A), $\Gamma; q_1 + q_2 \vdash [v_1/x_1]v : \tau_{\texttt{exn}}$.
  By T-RAISE, $\Gamma; q_1 + q_2 \vdash [v_1/x_1]\texttt{raise}(v) : B \oplus 0$.
- T-HANDLE: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$. Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
  By the assumption, $\Gamma_1; q_2 \vdash e_1 : B \oplus p_0$, $\Gamma_2, x : \tau_{\texttt{exn}}; p_0 \vdash e_2 : B \oplus p$.
  - If $x_1 : \tau_1 \in \Gamma_1$,using IH(B), $\Gamma_1 \setminus (x_1 : \tau_1); q_1 + q_2 \vdash [v_1/x_1]e_1 : B \oplus p_0$.
    By T-HANDLE, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\texttt{try}(e_1; x.e_2) : B \oplus p$.
  - Otherwise, $x_1 : \tau_1 \in \Gamma_2$, using IH(B), $\Gamma_2 \setminus (x_1 : \tau_1), x : \tau_{\texttt{exn}}; q_1 + p_0 \vdash e_2 : B \oplus p$.
    Using T-RELAX on $\Gamma_1; q_2 \vdash e_1 : B \oplus p_0$ to get $\Gamma_1; q_2 + q_1 \vdash e_1 : B \oplus p_0 + q_1$.
    By T-HANDLE, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\texttt{try}(e_1; x.e_2) : B \oplus p$.
- T-CONTRACT: $\Gamma, x_1 : \tau_1 = \Gamma', x : \tau$ and $e = [x, x/y_1, y_2]e' : B \oplus p$.
  Then $x_1 = x$ xor $x_1 \neq x$.
  By the assumption, $\Gamma', y_1 : \tau_1', y_2 : \tau_2'; q_2 \vdash e' : B \oplus p$ and $\tau \curlyvee (\tau_1', \tau_2')$.
  - If $x_1 = x$, $\Gamma = \Gamma'$, $x_1 \notin \Gamma'$ and $\tau_1 = \tau \curlyvee (\tau_1', \tau_2')$.
    By the sharing lemma 3.4.3, $\cdot; p_1 \vdash v_1 : \tau_1'$ and $\cdot; p_2 \vdash v_1 : \tau_2'$, where $q_1 = p_1 + p_2$.
    Using IH(B) once for y_1 and y_2 respectively, we get $\Gamma'; q_2 + p_1 + p_1 \vdash [v_1/y_2][v_1/y_1]e' : B \oplus p$.
    $x_1 \notin v_1$, so $[v_1/y_2][v_1/y_1]e' = [v_1/x][x, x/y_1, y_2]e' = [v_1/x_1]e$.
    Then $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.
  - Otherwise $x_1 \neq x$, $\Gamma' = \Gamma'', x_1 : \tau_1$. $x_1 \neq y_1$, $x_2 \neq y_2$.
    Using IH(B), $\Gamma'', y_1 : \tau_1', y_2 : \tau_2'; q_2 + q_1 \vdash [v_1/x_1]e' : B \oplus p$.

By T-CONTRACT, $\Gamma'', x : \tau; q_2 + q_1 \vdash [x, x/y_1, y_2][v_1/x_1]e' : B \oplus p$, which is $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.

- T-WEAK: $\Gamma, x_1 : \tau_1 = \Gamma', x : \tau$.
  Then $x = x_1$ xor $x \neq x_1$. By the assumption, $\Gamma'; q_2 \vdash e : B \oplus p$.
  - If $x = x_1$, then $\Gamma = \Gamma'$, $x_1 \notin \Gamma'$, therefore $x_1 \notin e$. Then $[v_1/x_1]e = e$. Then $\Gamma; q_2 \vdash [v_1/x_1]e : B \oplus p$ by the assumption.
    By T-RELAX, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$ ($q_1 \geq 0$ by lemma 3.4.1).
  - Otherwise $x \neq x_1$, $x_1 \in \Gamma'$. By the assumption, $\Gamma'; q_2 \vdash e : B \oplus p$.
    Using IH(B), $\Gamma' \setminus (x_1 : \tau_1); q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.
    By T-WEAK, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.
- T-RELAX: $B = \langle \tau, r \rangle$.
  By the assumption, $\Gamma, x_1 : \tau_1; q' \vdash e : \langle \tau, r' \rangle \oplus p'$ and $q_2 \geq q'$, $q_2 - r \geq q' - r'$, $q_2 - p \geq q' - p'$.
  Using IH(B), $\Gamma; q' + q_1 \vdash [v_1/x_1]e : \langle \tau, r' \rangle \oplus p'$.
  By T-RELAX, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : \langle \tau, r \rangle \oplus p$
- T-SUB: $B = \langle \tau, r \rangle$.
  By the assumption $\tau' <: \tau$ and $\Gamma, x_1 : \tau_1; q_2 \vdash e : \langle \tau', r \rangle \oplus p$.
  Using IH(B), $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : \langle \tau', r \rangle \oplus p$.
  By T-SUB, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : \langle \tau, r \rangle \oplus p$.
- T-SUPER: $\Gamma, x_1 : \tau_1 = \Gamma', x : \tau'$.
  Then $x = x_1$ xor $x \neq x_1$.
  By the assumption, $\tau' <: \tau$, and $\Gamma', x : \tau; q_2 \vdash e : B \oplus p$.
  - If $x = x_1$, then $\Gamma = \Gamma', \tau' = \tau_1$.
    Since $\cdot; q_1 \vdash v_1 : \tau_1$, $\tau_1 <: \tau$, by the value subtyping lemma 3.4.5, $\cdot; q_1' \vdash v_1 : \tau$ with $q_1' \leq q_1$.
    Using IH(B), $\Gamma; q_2 + q_1' \vdash [v_1/x_1]e : B \oplus p$.
    By T-RELAX, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.
  - Otherwise $x \neq x_1$, $x_1 \in \Gamma'$.
    So the assumption gives $\Gamma'', x_1 : \tau_1, x : \tau; q_2 \vdash e : B \oplus p$, where $\Gamma'', x_1 : \tau_1 = \Gamma'$.
    Using IH(B), $\Gamma'', x : \tau; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$.
    By T-SUPER, $\Gamma'', x : \tau'; q_2 + q_1 \vdash [v_1/x_1]e : B \oplus p$, where $\Gamma'', x : \tau' = \Gamma$.

□

Preservation

Theorem 3.4.10 (Preservation)**.**
If $q \vdash s$ *and* $s; q \mapsto s_0; q_0$ *then* $q_0 \vdash s_0$.

Proof. Induct on $s; q \mapsto s_0; q_0$.

- D-TICK: $k \triangleright \texttt{tick}(r); q \mapsto k \triangleleft \langle\rangle; q_0$ and $q \geq r, q_0 = q - r$.
  $q \vdash s$, so $\cdot; q \vdash \texttt{tick}(r) : B \oplus p$ and $k \triangleleft : B \oplus p$.
  We want to show $q - r \vdash k \triangleleft \langle\rangle$.

Let $B = \langle \tau, x \rangle$. Induct on $\cdot; q \vdash \texttt{tick}(r) : \langle \tau, x \rangle \oplus p$ to show $q - x \geq r$ and $\texttt{unit} <: \tau$. Only 3 cases are possible:

- T-TICK: $q = r$, $\tau = \texttt{unit}$, $x = 0$. Good.
- T-RELAX: By the assumption, $\cdot; q' \vdash \texttt{tick}(r) : \langle \tau, x' \rangle \oplus p'$ and $q \geq q'$, $q - x \geq q' - x'$, $q - p \geq q' - p'$.
  Using the inner IH, get $\texttt{unit} <: \tau$ and $q' - x' \geq r$. So $q - x \geq r$.
- T-SUB: By the assumption, $\cdot; q \vdash \texttt{tick}(r) : \langle \tau', x \rangle \oplus p$, where $\tau' <: \tau$.
  Using the inner IH, $q - x \geq r$, $\texttt{unit} <: \tau'$, by transitivity of subtyping, $\texttt{unit} <: \tau$.

By T-TRIV, $\cdot; 0 \vdash \langle\rangle : \texttt{unit}$. By value subtyping lemma 3.4.5, $\cdot; y \vdash \langle\rangle : \tau$ where $y \leq 0$.
We have $k \triangleleft : \langle \tau, x \rangle \oplus p$, then by ST-VAL, $x + y \vdash k \triangleleft \langle\rangle$.
$x + y \leq q - r + 0 = q - r$. By state relaxing lemma 3.4.8, $q - r \vdash k \triangleleft \langle\rangle$.

- D-FUN: $k \triangleright \texttt{fun}(f.x.e)(v_2); q \mapsto k \triangleright [\texttt{fun}(f.x.e), v_2/f, x]e; q$.
  $q \vdash s$, so $k \triangleleft : B \oplus p$, $\cdot; q \vdash \texttt{fun}(f.x.e)(v_2) : B \oplus p$.
  It suffices to show $\cdot; q \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : B \oplus p$.
  Induct on $\cdot; q \vdash \texttt{fun}(f.x.e)(v_2) : B \oplus p$. Only 3 cases are possible:
  - T-APP:
    By the assumption, $\cdot; 0 \vdash \texttt{fun}(f.x.e) : \langle \tau, q_1 \rangle \to (B \oplus p)$, $\cdot; q_2 \vdash v_2 : \tau$, $q = q_1 + q_2$.
    Invert, get $f : \langle \tau', q_1' \rangle \to (B' \oplus p'), x : \tau'; q_1' \vdash e : B' \oplus p'$, $B' <: B$, $\langle \tau, q_1 \rangle <: \langle \tau', q_1' \rangle$, $p \leq p'$ so $\tau <: \tau'$, $q_1' \leq q_1$.
    Then by T-FUN, $\cdot; 0 \vdash \texttt{fun}(f.x.e) : \langle \tau', q_1' \rangle \to (B' \oplus p')$.
    By value subtyping lemma 3.4.5, $\cdot; q_2' \vdash v_2 : \tau'$ with $q_2' \leq q_2$.
    Using the substitution lemma 3.4.9 twice on f, x, $\cdot; q_1' + q_2' \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : B' \oplus p'$.
    By T-RELAX and T-SUB, $\cdot; q_1 + q_2 \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : B \oplus p$, where $q = q_1 + q_2$.
  - T-RELAX: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q' \vdash \texttt{fun}(f.x.e)(v_2) : \langle \tau, r' \rangle \oplus p'$, where $q \geq q'$, $q - r \geq q' - r'$, $q - p \geq q' - p'$.
    Using the inner IH, $\cdot; q' \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : \langle \tau, r' \rangle \oplus p'$.
    By T-RELAX, $\cdot; q \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : \langle \tau, r \rangle \oplus p$.
  - T-SUB: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q \vdash \texttt{fun}(f.x.e)(v_2) : \langle \tau', r \rangle \oplus p$ and $\tau' <: \tau$.
    Using the inner IH, $\cdot; q \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : \langle \tau', r \rangle \oplus p$.
    By T-SUB, $\cdot; q \vdash [\texttt{fun}(f.x.e), v_2/f, x]e : \langle \tau, r \rangle \oplus p$.
- D-RET: $k \triangleright \texttt{ret}(v); q \mapsto k \triangleleft v; q$.
  $q \vdash k \triangleright \texttt{ret}(v)$, so $k \triangleleft : B \oplus p$ and $\cdot; q \vdash \texttt{ret}(v) : B \oplus p$.
  Let $B = \langle \tau, r \rangle$. Induct on $\cdot; q \vdash \texttt{ret}(v) : \langle \tau, r \rangle \oplus p$ to show there exists $\tau' <: \tau$, where $\cdot; q_1 \vdash v : \tau'$ and $q - r \geq q_1$.
  Only 3 cases are possible:
  - T-RET: $B = \langle \tau, 0 \rangle$, and the assumption gives $\cdot; q \vdash v : \tau$ ($r = 0, q_1 = q, \tau = \tau'$).
  - T-RELAX: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q' \vdash \texttt{ret}(v) : \langle \tau, r' \rangle \oplus p'$, where $q \geq q'$, $q - r \geq q' - r'$, $q - p \geq q' - p'$.
    Using the inner IH, we have: $\cdot; q_1 \vdash v : \tau'$ with $\tau' <: \tau$ and $q' - r' \geq q_1$.

Then $q - r \geq q_1$.

  - T-SUB: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q \vdash \texttt{ret}(v) : \langle \tau', r \rangle \oplus p$, and $\tau' <: \tau$.
    Using the inner IH, we have $q - r \geq q_1$, where $\cdot; q_1 \vdash v : \tau''$, and $\tau'' <: \tau'$.
    By transitivity of subtyping, $\tau'' <: \tau$.

  By value subtyping lemma 3.4.5, $\cdot; q_1' \vdash v : \tau$ with $q_1' \leq q_1 \leq q - r$.
  By ST-VAL, $q_1' + r \vdash k \lhd v$.
  $q_1' + r \leq q - r + r = q$, so $q \vdash k \lhd v$ by state relaxing lemma 3.4.8.

- D-LET: $k \rhd \texttt{let}(e_1; x.e_2); q \mapsto k; x.e_2 \rhd e_1; q$.
  By the assumption, $q \vdash s$, so $\cdot; q \vdash \texttt{let}(e_1; x.e_2) : B \oplus p$, $k \lhd: B \oplus p$.
  Induct on $\cdot; q \vdash \texttt{let}(e_1; x.e_2) : B \oplus p$ to show:
  for some τ_1, q_1, we have $\cdot; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p$ and $x : \tau_1; q_1 \vdash e_2 : B \oplus p$.
  Only 3 cases are possible:

  - T-LET: Immediate by the assumption.
  - T-RELAX: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q' \vdash \texttt{let}(e_1; x.e_2) : \langle \tau, r' \rangle \oplus p'$, and $q \geq q'$, $q - r \geq q' - r'$, $q - p \geq q' - p'$.
    Using the inner IH, get τ_1, q_1, such that $\cdot; q' \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p'$ and $x : \tau_1; q_1 \vdash e_2 : \langle \tau, r' \rangle \oplus p'$.
    By T-RELAX, $\cdot; q \vdash e_1 : \langle \tau_1, q_1 + (q - q') \rangle \oplus p$ and $x : \tau_1; q_1 + (q - q') \vdash e_2 : \langle \tau, r \rangle \oplus p$.
  - T-SUB:$B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q \vdash \texttt{let}(e_1; x.e_2) : \langle \tau', r \rangle \oplus p$, and $\tau' <: \tau$.
    Using the inner IH, we have τ_1, q_1, such that $\cdot; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p$, $x : \tau_1; q_1 \vdash e_2 : \langle \tau', r \rangle \oplus p$.
    By T-SUB, $x : \tau_1; q_1 \vdash e_2 : \langle \tau, r \rangle \oplus p$.

  By FR-BND, $k; x.e_2 \lhd: \langle \tau_1, q_1 \rangle \oplus p$.
  We have $\cdot; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p$. By ST-EXP, $q \vdash k; x.e_2 \rhd e_1$.

- D-SEQ: $k; x.e \lhd v; q \mapsto k \rhd [v/x]e; q$.
  By the assumption, $q \vdash k; x.e \lhd v$. Invert, get $q = q_1 + q_2$, $\cdot; q_1 \vdash v : \tau$ and $k; x.e \lhd: \langle \tau, q_2 \rangle \oplus p$.
  Invert, $x : \tau; q_2 \vdash e : B \oplus p$ and $k \lhd: B \oplus p$.
  By the substitution lemma 3.4.9, $\cdot; q_2 + q_1 \vdash [v/x]e : B \oplus p$.
  By ST-EXP, $q_2 + q_1 \vdash k \rhd [v/x]e$ ($q = q_1 + q_2$).

- D-NIL: $k \rhd \texttt{match}(\texttt{nil})(e_0; x.y.e_1); q \mapsto k \rhd e_0; q$.
  By the assumption, $k \lhd: B \oplus p$, $\cdot; q \vdash \texttt{match}(\texttt{nil})(e_0; x.y.e_1) : B \oplus p$.
  It suffices to show $\cdot; q \vdash e_0 : B \oplus p$.
  Induct on $\cdot; q \vdash \texttt{match}(\texttt{nil})(e_0; x.y.e_1) : B \oplus p$. Only 3 cases are possible:

  - T-MATCH: By the assumption, $\cdot; q_2 \vdash e_0 : B \oplus p$, $\cdot; q_1 \vdash \texttt{nil} : L(A)$ where $q = q_1 + q_2$.
    $q_1 \geq 0$ by lemma 3.4.1, then by T-RELAX, $\cdot; q \vdash e_0 : B \oplus p$.
  - T-RELAX: By the assumption, $B = \langle \tau, r \rangle$ and $\cdot; q' \vdash \texttt{match}(\texttt{nil})(e_0; x.y.e_1) : \langle \tau, r' \rangle \oplus p'$ where $q \geq q'$, $q - r \geq q' - r'$, $q - p \geq q' - p'$.
    Using the inner IH, $\cdot; q' \vdash e_0 : \langle \tau, r' \rangle \oplus p'$. Then by T-RELAX, $\cdot; q \vdash e_0 : B \oplus p$.

  - T-SUB: By the assumption, $B = \langle\tau, r\rangle$ and $\cdot; q \vdash \mathtt{match}(\mathtt{nil})(e_0; x.y.e_1) : \langle\tau', r\rangle \oplus p$ where $\tau' <: \tau$.
    Using the inner IH, $\cdot; q \vdash e_0 : \langle\tau', r\rangle \oplus p$.
    Then by T-SUB, $\cdot; q \vdash e_0 : B \oplus p$.

- D-CONS: $k \triangleright \mathtt{match}(\mathtt{cons}(v_1, v_2))(e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q$.
  By the assumption, $k \triangleleft: B \oplus p_0$, $\cdot; q \vdash \mathtt{match}(\mathtt{cons}(v_1, v_2))(e_0; x.y.e_1) : B \oplus p_0$.
  It suffices to show $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \oplus p_0$.
  Induct on $\cdot; q \vdash \mathtt{match}(\mathtt{cons}(v_1, v_2))(e_0; x.y.e_1) : B \oplus p_0$. Only 3 cases are possible:

  - T-MATCH:
    By the assumption, $x : \tau, y : L(\langle\tau, p\rangle); q_2 + p \vdash e_0 : B \oplus p_0$, where $\cdot; q_1 \vdash \mathtt{cons}(v_1, v_2) : L(\langle\tau, p\rangle)$, $q = q_1 + q_2$.
    Invert $\cdot; q_1 \vdash \mathtt{cons}(v_1, v_2) : L(\langle\tau, p\rangle)$ to get $q_1 = q_1' + q_2' + p$, $\cdot; q_1' \vdash v_1 : \tau$, $\cdot; q_2' \vdash v_2 : L(\langle\tau, p\rangle)$.
    Using substitution lemma 3.4.9, $\cdot; q_2 + p + q_1' + q_2' \vdash [v_1, v_2/x, y]e_1 : B \oplus p_0$, which is $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \oplus p_0$.
  - T-RELAX: $B = \langle\tau, r\rangle$.
    By the assumption, $\cdot; q' \vdash \mathtt{match}(\mathtt{cons}(v_1, v_2))(e_0; x.y.e_1) : \langle\tau, r'\rangle \oplus p'$ where $q \geq q'$, $q - r \geq q' - r'$, $q - p_0 \geq q' - p'$.
    Using the inner IH, $\cdot; q' \vdash [v_1, v_2/x, y]e_1 : \langle\tau, r'\rangle \oplus p'$.
    Then by T-RELAX, $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \oplus p_0$.
  - T-SUB: $B = \langle\tau, r\rangle$.
    By the assumption, $\cdot; q \vdash \mathtt{match}(\mathtt{cons}(v_1, v_2))(e_0; x.y.e_1) : \langle\tau', r\rangle \oplus p_0$ where $\tau' <: \tau$.
    Using the inner IH, $\cdot; q \vdash [v_1, v_2/x, y]e_1 : \langle\tau', r\rangle \oplus p_0$.
    Then by T-SUB, $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \oplus p_0$.

- D-RAISE: $k \triangleright \mathtt{raise}(v); q \mapsto k \blacktriangleleft v; q$.
  By the assumption, $\cdot; q \vdash \mathtt{raise}(v) : B \oplus p$ and $k \triangleleft: B \oplus p$.
  Induct on $\cdot; q \vdash \mathtt{raise}(v) : B \oplus p$ to show $\cdot; q_1 \vdash v : \tau_{\mathtt{exn}}$ with $q_1 \leq q - p$. Only 3 cases are possible:
  - T-RAISE: immediate by the assumption, $q = q_1$, $p = 0$, $q_1 \vdash v : \tau_{\mathtt{exn}}$.
  - T-RELAX: By the assumption, $B = \langle\tau, r\rangle$, $\cdot; q' \vdash \mathtt{raise}(v) : \langle\tau, r'\rangle \oplus p'$ where $q' \leq q$, $q - r \geq q' - r'$, $q - p \geq q' - p'$.
    Using inner IH, get $\cdot; q_1 \vdash v : \tau_{\mathtt{exn}}$ with $q_1 \leq q' - p' \leq q - p$.
  - T-SUB: By the assumption, $B = \langle\tau, r\rangle$, $\cdot; q \vdash \mathtt{raise}(v) : \langle\tau', r\rangle \oplus p$ where $\tau' <: \tau$.
    Using inner IH, get $\cdot; q_1 \vdash v : \tau_{\mathtt{exn}}$ with $q_1 \leq q - p$.

  By ST-EXN, $q_1 + p \vdash k \blacktriangleleft v$. $q_1 + p \leq q - p + p = q$.
  By state relaxing lemma 3.4.8, $q \vdash k \blacktriangleleft v$.
- D-HANDLE: $k; \mathtt{try}(\_; x.e) \blacktriangleleft v; q \mapsto k \triangleright [v/x]e; q$.
  By the assumption, $q = q_1 + p_1$, $\cdot; q_1 \vdash v : \tau_{\mathtt{exn}}$, and $k; \mathtt{try}(\_; x.e) \triangleleft: B \oplus p_1$.
  Invert, get $k \triangleleft: B \oplus p$ and $x : \tau_{\mathtt{exn}}, p_1 \vdash e : B \oplus p$.
  Using substitution lemma 3.4.9, $\cdot; q_1 + p_1 \vdash [v/x]e : B \oplus p$.
  By ST-EXP, $q_1 + p_1 \vdash k \triangleright [v/x]e$, which is $q \vdash k \triangleright [v/x]e$.

- D-EXN: $k; x.e \blacktriangleleft v; q \mapsto k \blacktriangleleft v; q$.
  By the assumption, $q = q_1 + p$ where $\cdot; q_1 \vdash v : \tau_{\texttt{exn}}$ and $k; x.e \triangleleft\!: B \oplus p$.
  Invert, get $k \triangleleft\!: A \oplus p$ for some A.
  By ST-EXN, $q_1 + p = q \vdash k \blacktriangleleft v$.
- D-NORMAL: $k; \texttt{try}(\_; x.e) \triangleleft v; q \mapsto k \triangleleft v; q$.
  By the assumption, $q = q_1 + q_2$, $\cdot; q_1 \vdash v : \tau$, $k; \texttt{try}(\_; x.e) \triangleleft\!: \langle \tau, q_2 \rangle \oplus p$.
  Invert, get $k \triangleleft\!: \langle \tau, q_2 \rangle \oplus p'$ for some p'.
  By ST-VAL, $q_1 + q_2 = q \vdash k \triangleleft v$.
- D-TRY: $k \triangleright \texttt{try}(e_1; x.e); q \mapsto k; \texttt{try}(\_; x.e) \triangleright e_1; q$.
  By the assumption, $k \triangleleft\!: B \oplus p$ and $\cdot; q \vdash \texttt{try}(e_1; x.e) : B \oplus p$.
  Induct on $\cdot; q \vdash \texttt{try}(e_1; x.e) : B \oplus p$ to show there exists p_0 such that $\cdot; q \vdash e_1 : B \oplus p_0$, $x : \tau_{\texttt{exn}}; p_0 \vdash e : B \oplus p$.
  Only 3 cases are possible:
  - T-HANDLE: Immediate by the assumption.
  - T-RELAX: $B = \langle \tau, r \rangle$.
    By the assumption, $\cdot; q' \vdash \texttt{try}(e_1; x.e) : \langle \tau, r' \rangle \oplus p'$. where $q' \leq q$, $q - r \geq q' - r'$, $q - p \geq q - p'$.
    Using the inner IH, get p_0, such that $\cdot; q' \vdash e_1 : \langle \tau, r' \rangle \oplus p_0$, $x : \tau_{\texttt{exn}}; p_0 \vdash e : \langle \tau, r' \rangle \oplus p'$.
    By T-RELAX, $\cdot; q \vdash e_1 : \langle \tau, r \rangle \oplus p_0 + (q - q')$, and $x : \tau_{\texttt{exn}}; p_0 + (q - q') \vdash e : \langle \tau, r \rangle \oplus p$.
  - T-SUB: By the assumption, $B = \langle \tau, r \rangle$, $\cdot; q \vdash \texttt{try}(e_1; x.e) : \langle \tau', r \rangle \oplus p$. where $\tau' <: \tau$.
    Using the inner IH, get p_0, such that $\cdot; q \vdash e_1 : \langle \tau', r \rangle \oplus p_0$ and $x : \tau_{\texttt{exn}}; p_0 \vdash e : \langle \tau', r \rangle \oplus p$.
    By T-SUB, $\cdot; q \vdash e_1 : B \oplus p_0$ and $x : \tau_{\texttt{exn}}; p_0 \vdash e : B \oplus p$.

  Then by FR-EXN, $k; \texttt{try}(\_; x.e) \triangleleft\!: B \oplus p_0$.
  By ST-EXP, $q \vdash k; \texttt{try}(\_; x.e) \triangleright e_1$.

□

Progress

Lemma 3.4.11 (Canonical Forms)**.**
If $\cdot; q \vdash v : \tau$, and
- *$\tau = L(A)$, then either $v = \texttt{nil}$ or $v = \texttt{cons}(v_1, v_2)$.*
- *$\tau = A \to (C \oplus p)$, then $v = \texttt{fun}(f.x.e')$.*

Corollary 3.4.12.
If $\cdot; q \vdash e : B \oplus p$, and
- *$e = \texttt{match}(v)(e_0; x.y.e_1)$, then either $v = \texttt{nil}$ or $v = \texttt{cons}(v_1, v_2)$.*
- *$e = v_1(v_2)$, then $v_1 = \texttt{fun}(f.x.e')$.*

Theorem 3.4.13 (Progress)**.**
If $q \vdash s$, then either s final or $s; q \mapsto s'; q'$.

Proof. Induct on $q \vdash s$.

- ST-VAL: $q \vdash k \lhd v$.
  $k = \epsilon$, or $k = k'; x.e$, or $k = k'; \texttt{try}(\_; x.e)$.
  If $k = \epsilon$, $k \lhd v$ is final.
  Otherwise if $k = k'; x.e$, by D-SEQ, $k'; x.e \lhd v; q \mapsto k' \rhd [v/x]e; q$.
  Otherwise, $k = k'; \texttt{try}(\_; x.e)$, by D-NORMAL, $k'; \texttt{try}(\_; x.e) \lhd v; q \mapsto k' \lhd v; q$.
- ST-EXN: $q \vdash k \blacktriangleleft v$.
  $k = \epsilon$, or $k = k'; x.e$, or $k = k'; \texttt{try}(\_; x.e)$.
  If $k = \epsilon$, $k \blacktriangleleft v$ is final.
  Otherwise if $k = k'; x.e$, by D-EXN, $k'; x.e \blacktriangleleft v; q \mapsto k' \blacktriangleleft v; q$.
  Otherwise, $k = k'; \texttt{try}(\_; x.e)$, by D-HANDLE, $k'; \texttt{try}(\_; x.e) \blacktriangleleft v; q \mapsto k' \rhd [v/x]e; q$.
- ST-EXP: $q \vdash k \rhd e$.
  By the assumption, $\cdot; q \vdash e : B \oplus p$, $k \lhd : B \oplus p$.
  Induct on the structure of e.
  - e is not $\texttt{tick}(r)$. Then $k \rhd e; q \mapsto s'; q$ for some s' by the canonical forms lemma and rules D-RET, D-FUN,D-LET, D-SEQ, D-NIL, D-CONS, D-RAISE, D-TRY.
  - $e = \texttt{tick}(r)$.
    Induct on $\cdot; q \vdash \texttt{tick}(r) : B \oplus p$ to show $q \geq r$. Only 3 cases are possible:
    - T-TICK: $q = r$.
    - T-RELAX: $\cdot; q \vdash \texttt{tick}(r) : \langle \tau, x \rangle \oplus p$.
      By the assumption, $\cdot; q' \vdash \texttt{tick}(r) : \langle \tau, x' \rangle \oplus p'$, and $q \geq q'$.
      Using the inner IH, $q' \geq r$, then $q \geq r$.
    - T-SUB: $q \geq r$ by the IH.

    $q \geq r$ then b D-TICK, $k \rhd \texttt{tick}(r); q \mapsto k \lhd \langle\rangle; q - r$.

□

Soundness of AARA

Finally, as a corollary to progress and preservation, we have our main theorem of this thesis:
Theorem 3.4.14 (Resource safety)**.**
If $\cdot; q \vdash e : B \oplus p$, then starting with q amount of resource (i.e. initial state $\epsilon \rhd e; q$), the program execution does not get stuck.

Notice we can show the following theorem:
Theorem 3.4.15.
If $\cdot; q \vdash e : B$ (from section 3.2), then $\cdot; q \vdash e : B \oplus 0$ (from section 3.3).
Therefore the type system in Section 3.2 is also sound with respect to resource safety.

Chapter 4

Exception Handling using Dynamic Classification

This chapter extends the previous chapter to accomodate a more modular, robust, flexible implementation of exceptions.

4.1 Exceptions and Dynamic Classfication

In Chapter 3, we assumed the exception type is some globally fixed type in our system. In other words, we adopted a *closed-world* view of exceptions: the entire program agrees beforehand on possible exception classes and their types [5]. However, it is oftentimes beneficial to dynamically generate classes of exceptions, as it is implemented in Standard ML. This gives better modularity, since the entire program does not need to agree upon possible classes of exceptions beforehand. Dynamic class generation also ensures exceptions are handled only by the intended handler, rather than by some alien code.

The key behavior of dynamically classified exception is best exemplified by the following code:

```
let exception Fail in
    let exception Fail in
        raise Fail
    end
    handle Fail => ...
end
```

Upon execution of the statement `exception Fail` in line 2, a new class of exception is generated. The new class is distinct from existing classes of exceptions, despite having the same exception name `Fail`. The handler in line 5 would not catch the exception raised in line 4. In applications, this implies that the alien code could never "guess" the dynamic class and intercept the classified data/exception.

Dynamic classification can be realized using *symbols*. See [5] for implementation details and a formal account of symbols.

4.2 Language

We extend the language in Chapter 3 with dynamically classified type, which we call $\tau_{\texttt{exn}}$. We equip the external language with limited access to dynamic classification. Specifically, we have constructs to declare/generate a new class denoted by a, and to classify and declassify data with a. Externally, such a is not a first-class value that we can pass around and compute with. Internally, a is a variable of type class reference ($\texttt{cls}(A)$). Declaration generates a new class and binds the reference to the new class to a.
Syntax of the internal language is the following:

$$
\begin{array}{rcll}
\tau & ::= & ... & \\
 & | & \texttt{cls}(A) & \text{class reference} \\
 & | & \tau_{\texttt{exn}} & \text{exception/classified type} \\
A & ::= & \langle \tau, q \rangle & \text{annotated type} \\
v & ::= & ... & \\
 & | & \&s & \text{class reference} \\
 & | & \texttt{classify}(v_1; v_2) & \text{classified value} \\
e & ::= & ... & \\
 & | & \texttt{exn}[A](a.e) & \text{declare exception} \\
 & | & \texttt{isin}(v_1)(v_2; x.e_1; e_2) & \text{declassify/match exceptions}
\end{array}
$$

In the external language, we can declare an exception (`exception F of tau in e`);within `e`, we can raise it (`raise F v`) and handle it(`try e where handle F x => e'`). These external constructs can be elaborated to expressions in our internal language as the following:

```
// r is a constant number
exception F of <tau,r> in e          ---->  exn[<tau,r>](F.e)

// v:tau
raise F v                           ---->  raise(classify(F; v))

// x:tau_exn, y:tau
try e where handle F x => e'        ---->  try(e;y.isin(F)(y;x.e';raise(y)))
```

Dynamics is extended with a symbol context Σ which keeps track of all the symbols declared. Declaring an exception generates a fresh symbol and passes a reference to it. Declassifying an exception checks for the equality of symbols. The other rules in Chapter 3 simply propagate Σ.

$$\frac{s \text{ is fresh with respect to } \Sigma}{k \triangleright \texttt{exn}[A](x.e); q; \Sigma \mapsto k \triangleright [\&s/x]e; q; \Sigma, s \sim A} \ \textsc{D-dcl}$$

$$\frac{s = s' \text{ in } \Sigma}{k \triangleright \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2); q; \Sigma \mapsto k \triangleright [v_2/x]e_1; q; \Sigma} \ \textsc{D-isin1}$$

$$\frac{s \neq s' \text{ in } \Sigma}{k \triangleright \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2); q; \Sigma \mapsto k \triangleright e_2; q; \Sigma} \ \textsc{D-isin2}$$

4.3 Type System

Accordingly, we modify our type system to keep around the symbol context Σ. Notice to type check expressions in the external language (do not need T-SYMREF), Σ is irrelevant.

$\boxed{\Gamma; q \vdash_\Sigma v : \tau}$

$$\frac{}{\cdot; 0 \vdash_{\Sigma, s \sim A} \&s : \texttt{cls}(A)}\ \textsc{T-symref} \qquad \frac{\Gamma_1; 0 \vdash_\Sigma v_1 : \texttt{cls}(\langle \tau, r \rangle) \quad \Gamma_2; q \vdash_\Sigma v_2 : \tau}{\Gamma_1, \Gamma_2; q + r \vdash_\Sigma \texttt{classify}(v_1; v_2) : \tau_{\texttt{exn}}}\ \textsc{T-classify}$$

A symbol reference does not carry potential. A classified exception carries the potential carried by the exception value v_1, as well as extra r potential that we declare this class to have.

$\boxed{\Gamma; q \vdash_\Sigma e : B \oplus p}$

$$\frac{\Gamma, x : \texttt{cls}(A); q \vdash_\Sigma e : B \oplus p}{\Gamma; q \vdash_\Sigma \texttt{exn}[A](x.e) : B \oplus p}\ \textsc{T-dcl}$$

$$\frac{\begin{array}{c} q = q_1 + q_2 \quad \Gamma_1; 0 \vdash_\Sigma v_1 : \texttt{cls}(\langle \tau, r \rangle) \quad \Gamma_2; q_1 \vdash_\Sigma v_2 : \tau_{\texttt{exn}} \\ \Gamma_3, x : \tau; q_2 + r \vdash_\Sigma e_1 : B \oplus p \quad \Gamma_3; q_2 \vdash_\Sigma e_2 : B \oplus p \end{array}}{\Gamma_1, \Gamma_2, \Gamma_3; q \vdash_\Sigma \texttt{isin}(v_1)(v_2; x.e_1; e_2) : B \oplus p}\ \textsc{T-isin}$$

Declaring an exception creates a variable binding x, where x is a class reference. Declassifying data needs to pay for the potential carried by the classified exception, and the potential needed to evaluate either branch of the declassification. In the case where the exception is declassifed, we get extra r potential that we packaged into the exception upon classification.

Stack/State typing judgements $k \triangleleft : A \oplus p$ and $q \vdash s$ are modified to be $k \triangleleft :_\Sigma A \oplus p$ and $q \vdash_\Sigma s$. They simply propagate Σ to value and expression typing.

We also need to define sharing and subtyping for the two types we added. $\texttt{cls}(A)$ does not carry potential, so it can be freely shared. $\tau_{\texttt{exn}}$ carries a classified amount of potential, so it cannot be shared.

$$\frac{}{\texttt{cls}(A) <: \texttt{cls}(A)}\ \textsc{SB-symref} \qquad \frac{}{\tau_{\texttt{exn}} <: \tau_{\texttt{exn}}}\ \textsc{SB-exn}$$

$$\frac{}{\texttt{cls}(A) \curlyvee (\texttt{cls}(A), \texttt{cls}(A))}\ \textsc{SH-symref}$$

And finally, $|\cdot|$ is extended to say $\tau_{\texttt{exn}}$ is not potential-free, $\texttt{cls}(A)$ is.

$$|\tau_{\texttt{exn}}| = ☺$$
$$|\texttt{cls}(A)| = \texttt{cls}(A)$$

4.4 Soundness

Again, soundness of the type system can be proved by showing progress and preservation. We briefly go over the proof in Chapter 3 to show how it can be extended.

Lemma 4.4.1.
If $\Gamma; q \vdash_\Sigma v : \tau$, then $q \geq 0$.
Lemma 4.4.2.
If $\Gamma; q \vdash_\Sigma v : \tau$ and $|\tau| = \tau$, then $q = 0$.
Lemma 4.4.3 (Sharing)**.**
If $\tau \curlyvee (\tau_1, \tau_2)$ and $\cdot; q \vdash_\Sigma v : \tau$, then $\cdot; q_1 \vdash_\Sigma v : \tau_1$, $\cdot; q_2 \vdash_\Sigma v : \tau_2$, where $q = q_1 + q_2, q_1 \geq 0, q_2 \geq 0$.
Lemma 4.4.4 (Transitivity of <:)**.**
Subtyping is transitive, i.e. if $\tau_1 <: \tau_2$, $\tau_2 <: \tau_3$, then $\tau_1 <: \tau_3$.
Lemma 4.4.5 (Value Subtyping)**.**
If $\cdot; q \vdash_\Sigma v : \tau$ and $\tau <: \tau'$, then $\cdot; q' \vdash_\Sigma v : \tau'$ where $q' \leq q$.
Lemma 4.4.6. *If $\Gamma; q \vdash_\Sigma e : \langle \tau, r \rangle \oplus p$, then $\Gamma; q + d \vdash_\Sigma e : \langle \tau, r + d \rangle \oplus p + d$ when $d \geq 0$.*
Lemma 4.4.7. *If $k \lhd{:}_\Sigma \langle \tau, q \rangle \oplus p$, then $k \lhd{:}_\Sigma \langle \tau, q + d \rangle \oplus p + d$ when $d \geq 0$.*
Lemma 4.4.8 (State relaxing)**.** *If $q \vdash_\Sigma s$, then $q' \vdash_\Sigma s$ when $q' \geq q$.*

We can check the lemmas above still hold.
Lemma 4.4.9 (Substitution)**.**
If $\cdot; q_1 \vdash_\Sigma v_1 : \tau_1$:
(A) If $\Gamma, x_1 : \tau_1; q_2 \vdash_\Sigma v : \tau$, then $\Gamma; q_1 + q_2 \vdash_\Sigma [v_1/x_1]v : \tau$.
(B) If $\Gamma, x_1 : \tau_1; q_2 \vdash_\Sigma e : B \oplus p$, then $\Gamma; q_1 + q_2 \vdash_\Sigma [v_1/x_1]e : B \oplus p$.

Proof.
(A): Induction on $\Gamma, x_1 : \tau_1; q_2 \vdash_\Sigma v : \tau$

- ...
- T-SYMREF: vacuous.
- T-CLASSIFY: $v = \texttt{classify}(v_3; v_4)$, $q_2 = q + r$, $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$, $\Gamma_1; 0 \vdash_\Sigma v_3 : \texttt{cls}(\langle \tau', r \rangle)$, and $\Gamma_2; q \vdash_\Sigma v_4 : \tau'$.
  Then $x_1 : \tau_1 \in \Gamma_1$ xor $x_1 : \tau_1 \in \Gamma_2$.
  - If $x_1 : \tau_1 \in \Gamma_1$, using IH(A), So $\Gamma_1 \setminus (x_1 : \tau_1); q_1 \vdash_\Sigma [v_1/x_1]v_3 : \texttt{cls}(\langle \tau', r \rangle)$.
    Since $|\texttt{cls}(\langle \tau', r \rangle)| = \texttt{cls}(\langle \tau', r \rangle)$, $q_1 = 0$ by lemma 4.4.2.
    By T-CLASSIFY, $\Gamma; q_2 + q_1 \vdash_\Sigma [v_1/x_1]v : \tau_{\texttt{exn}}$.
  - Otherwise, $x_1 : \tau_1 \in \Gamma_2$, using IH(A), $\Gamma_2 \setminus (x_1 : \tau_1); q + q_1 \vdash_\Sigma [v_1/x_1]v_4 : \tau'$.
    By T-CLASSIFY, $\Gamma; q_2 + q_1 \vdash_\Sigma [v_1/x_1]v : \tau_{\texttt{exn}}$.

(B): Induction on $\Gamma, x_1 : \tau_1; q_2 \vdash_\Sigma e : B \oplus p$

- ...
- T-DCL: follows directly from IH(B).
- T-CLASSIFY: follows from IH(A), IH(B), and lemma 4.4.2.

□

Lemma 4.4.10.
If $\Gamma; q \vdash_\Sigma v : \tau$ and s is fresh with respect to Σ, then $\Gamma; q \vdash_{\Sigma, s \sim A} v : \tau$.
If $\Gamma; q \vdash_\Sigma e : B \oplus p$ and s is fresh with respect to Σ, then $\Gamma; q \vdash_{\Sigma, s \sim A} e : B \oplus p$.

Theorem 4.4.11 (Preservation)**.**
If $q \vdash_\Sigma s$ *and* $s; q; \Sigma \mapsto s_0; q_0; \Sigma_0$ *then* $q_0 \vdash_{\Sigma_0} s_0$.

Proof. Induct on $s; q; \Sigma \mapsto s_0; q_0; \Sigma_0$.

- D-DCL: $k \triangleright \texttt{exn}[A](x.e); q; \Sigma \mapsto k \triangleright [\&s/x]e; q; \Sigma, s \sim A$, where s fresh w.r.t. Σ.
  By the assumption, $\cdot; q \vdash_\Sigma \texttt{exn}[A](x.e) : B \oplus p$ and $k \triangleleft :_\Sigma B \oplus p$. It suffices to show $\cdot; q \vdash_{\Sigma, s\sim A} [\&s/x]e : B \oplus p$.
  Induct on $\cdot; q \vdash_\Sigma \texttt{exn}[A](x.e) : B \oplus p$. Only three cases are possible.
  - T-DCL: By the assumption, $x : \texttt{cls}(A); q \vdash_\Sigma e : B \oplus p$. By lemma 4.4.10, $x : \texttt{cls}(A); q \vdash_{\Sigma, s\sim A} e : B \oplus p$, where s fresh w.r.t. Σ.
    By T-SYM, we have $\cdot; 0 \vdash_{\Sigma, s\sim A} \&s : \texttt{cls}(A)$.
    By the substituition lemma 4.4.9, $\cdot; q \vdash_{\Sigma, s\sim A} [\&s/x]e : B \oplus p$.
  - T-RELAX: follows from the inner IH.
  - T-SUB: follows from the inner IH.
- D-ISIN1: $k \triangleright \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2); q; \Sigma \mapsto k \triangleright [v_2/x]e_1; q; \Sigma$, where $s = s'$.
  By the assumption, $\cdot; q \vdash_\Sigma \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2) : B \oplus p$ and $k \triangleleft :_\Sigma B \oplus p$. It suffices to show $\cdot; q \vdash_\Sigma [v_2/x]e_1 : B \oplus p$.
  Induct on $\cdot; q \vdash_\Sigma \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2) : B \oplus p$. Only three cases are possible.
  - T-ISIN: By the assumption, $q = q_1 + q_2$, $\cdot; 0 \vdash_\Sigma \&s' : \texttt{cls}(\langle \tau, r \rangle)$, $\cdot; q_1 \vdash_\Sigma \texttt{classify}(\&s; v_2) : \tau_{\texttt{exn}}$, and $x : \tau; q_2 + r \vdash_\Sigma e_1 : B \oplus p$.
    Induct on $\cdot; q_1 \vdash_\Sigma \texttt{classify}(\&s; v_2) : \tau_{\texttt{exn}}$ to get $q_1 = q' + r$, $\cdot; q' \vdash_\Sigma v_2 : \tau'$, $\cdot; 0 \vdash_\Sigma \&s : \texttt{cls}(\langle \tau', r' \rangle)$.
    Invert $\cdot; 0 \vdash_\Sigma \&s : \texttt{cls}(\langle \tau', r' \rangle)$ and $\cdot; 0 \vdash_\Sigma \&s' : \texttt{cls}(\langle \tau, r \rangle)$ to get $s \sim \langle \tau', r' \rangle \in \Sigma$ and $s' \sim \langle \tau, r \rangle \in \Sigma$.
    Since $s = s'$, so $\tau = \tau'$ and $r = r'$.
    By the substituition lemma 4.4.9, $\cdot; q_2 + r + q' \vdash_\Sigma [v_2/x]e_1 : B \oplus p$, which is $\cdot; q \vdash_\Sigma [v_2/x]e_1 : B \oplus p$.
  - T-RELAX: follows from the inner IH.
  - T-SUB: follows from the inner IH.
- D-ISIN2: $k \triangleright \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2); q; \Sigma \mapsto k \triangleright e_2; q; \Sigma$, where $s \neq s'$.
  By the assumption, $\cdot; q \vdash_\Sigma \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2) : B \oplus p$ and $k \triangleleft :_\Sigma B \oplus p$. It suffices to show $\cdot; q \vdash_\Sigma e_2 : B \oplus p$.
  Induct on $\cdot; q \vdash_\Sigma \texttt{isin}(\&s')(\texttt{classify}(\&s; v_2); x.e_1; e_2) : B \oplus p$. Only three cases are possible.
  - T-ISIN:By the assumption, $q = q_1 + q_2$, $\cdot; 0 \vdash_\Sigma \&s' : \texttt{cls}(\langle \tau, r \rangle)$, $\cdot; q_1 \vdash_\Sigma \texttt{classify}(\&s; v_2) : \tau_{\texttt{exn}}$, and $\cdot; q_2 \vdash_\Sigma e_2 : B \oplus p$.
    By lemma 4.4.1, $q_1 \geq 0$.
    By T-RELAX, $\cdot; q \vdash_\Sigma e_2 : B \oplus p$.
  - T-RELAX: follows from the inner IH.
  - T-SUB: follows from the inner IH.

□

Lemma 4.4.12 (Canonical Forms).
If $\cdot; q \vdash_\Sigma v : \tau$, *and*

- $\tau = L(A)$, *then either* $v = \texttt{nil}$ *or* $v = \texttt{cons}(v_1, v_2)$.
- $\tau = A \to (C \oplus p)$, *then* $v = \texttt{fun}(f.x.e')$.
- $\tau = \texttt{cls}(A)$, *then* $v = \&s$ *and* $s \in \Sigma$.
- $\tau = \tau_{\texttt{exn}}$, *then* $v = \texttt{classify}(v_1; v_2)$ *where* $v_1 = \&s$ *with* $s \in \Sigma$

Corollary 4.4.13.
If $\cdot; q \vdash_\Sigma e : B \oplus p$, *and*

- $e = \texttt{match}(v)(e_0; x.y.e_1)$, *then either* $v = \texttt{nil}$ *or* $v = \texttt{cons}(v_1, v_2)$.
- $e = v_1(v_2)$, *then* $v_1 = \texttt{fun}(f.x.e')$.
- $e = \texttt{isin}(v_1)(v_2; x.e_1; e_1)$, *then* $v_1 = \&s'$, $v_2 = \texttt{classify}(\&s; v_2')$, *with* s, s' *both valid in* Σ.

Theorem 4.4.14 (Progress).
If $q \vdash_\Sigma s$, *then either* s *final or* $s; q; \Sigma \mapsto s'; q'; \Sigma'$.

Proof. Follows from preservation and lemma 4.4.13. □

Chapter 5

Discussion

5.1 Future Work

5.1.1 Automation

Although not fully explained in this thesis, the type system in Chapter 3 was developed with automation in mind. We believe it is possible to automate resource bound inference using the type system, in a similar way other AARA works do [6].

First notice, from the type system we developed, if we erase all potential annotations and rules that only touch potential annoatations (T-RELAX, T-SUB, T-SUPER) , we get a standard, plain type system for the language. Also notice we can incorporate all the structural rules in each of the non-structural rules to get an equivalent, syntax-directed type system.

Weakening T-WEAK is allowed by modifying the base cases of the type system (T-VAR, T-TICK, T-TRIV, T-NIL). For example, T-VAR would now be:

$$\frac{}{\Gamma, x : \tau; 0 \vdash x : \tau} \text{ T-VAR}$$

Contraction T-CONTRACT is left untouched. It does not hinder algorithmic type inference: everytime we see an expression with $x(x \geq 2)$ uses of the same variable, we apply T-CONTRACT $x - 1$ times.

Rules T-SUB, T-SUPER, T-RELAX that make potential affine can be blended in the expression typing rules. For example, the rule T-LET from 3 was:

$$\frac{\Gamma_1; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p \quad \Gamma_2, x : \tau_1; q_1 \vdash e_2 : B \oplus p}{\Gamma_1, \Gamma_2; q \vdash \texttt{let}(e_1; x.e_2) : B \oplus p} \text{ T-LET}$$

One algorithmic version of the rule would be:

$$\frac{\begin{array}{c}\Gamma_1; q \vdash e_1 : \langle \tau_1, q_1 \rangle \oplus p \quad \Gamma_2, x : \tau_1'; q_1' \vdash e_2 : \langle \tau, r \rangle \oplus p \\ \tau_1 <: \tau_1' \quad q_1 \geq q_1' \quad q' \geq q \quad q' - r' \geq q - r \quad q' - p' \geq q - p\end{array}}{\Gamma_1, \Gamma_2; q' \vdash \texttt{let}(e_1; x.e_2) : \langle \tau', r' \rangle \oplus p'} \text{ T-LET}$$

It then suffices to infer potential annotation in the deirvation tree. We can imagine a procedure where we start with all potential annotations left unspecified. We then collect all the constraints on those annotations from each typing rule invoked. Since all the constraints on those potential annotations are linear (as in linear algebra), the inference amounts to solving a linear optimization problem induced by those constraints. The objective of the linear optimization problem can be set to minimizing the resouce bound of interest.

5.1.2 Advanced Control Construct: Continuations

Another piece of future work is to extend AARA to more advanced constructs that bypass standard control flows, such as `letcc/callcc` in functional languages. Continuation, captured by `letcc`, is exactly what is on the control stack k of the K-machine, so the soundness proof we developed using K-machine is clearly relevant. However, the exact type system that would give rise to a sound AARA is unclear. We summarize the intuition that we have collected so far below.

At first sight, continuations appear like functions. Both pass around pieces of computation as values (no matter whether we syntactically treat values separately or not). A function type $\langle \tau, q \rangle \to A$ says given a value of type τ and q potential, the function can safely evaluate to A. In section 2.3, we had $k \triangleleft\colon \langle \tau, q \rangle$ to say the stack(continuation) k can safely evaluate to completion given a value of τ and q potential. Somewhat arbitrarily, we chose to make functions not carry potentials. Cost of the function body is "paid" at the call sites. Alternatively, we could allow functions to carry potential (at least some cost is "paid" upon creation of the functions) and treat them as linear values. Similarly, continuations can also be set up cost-free or not, the cost of executing the continuation can be "paid" when we throw to it, or when we capture it.

However, the two are, at least different, perhaps incomparable. Unlike functions which are defined by the code $x.e$, continuations are not defined by the list of frames. The intensional identity of continuations is not determined until runtime. This seems to make continuations a different type of animal. Consider the following code that exemplifies some interesting cost semantics of continuations.

```
let
    f = letcc(k. ret(fun _ => throw (fn <> => <>) to k))
in
    let
        _ = tick(1)
    in
        f(<>)
    end
end
```

`letcc` captures a continuation that consumes 1 resource and calls the function f that was thrown to the continuation. Within the letcc, it returns a suspended computation of throwing an identity function to the captured continuation. The entire expression costs 2 to evaluate.

Intuitively and informally, the captured k expects a function f with cost x to run on input <> and costs $x + 1$ to execute the continuation. In the current setup of AARA, although the same value/expression can be typed to different types (e.g. identity function can be typed to consume 0, 1, or any positive amount of resource), once the typing derivation is presented, each value/expression is stipulated to a constant resource annotation. If we follow this setup and attempt to type the continuation k to expect a fixed type and potential, we would assign the identity function to be a function with a cost of constant number $a (a \geq 0)$. Then the entire `letcc` expression would be typed to be a function of cost $1 + a$(constant). The `letcc` expression should also match the hole of the continuation, which is a function of cost a. Then $a \approx 1 + a$, which has no solution in our setup.

We are still hopeful that we can develop a static semantics of continuations in the style of AARA, but it appears that we need a principled view of continuations, in juxtaposition to values and expressions, informed by an adequate type theory that accounts for cost semantics. Potentially, we also need to reconsider our treatment of functions that bridge expressions and values, in that framework.

5.2 Conclusion

The type-based approach of Automatic Amortized Resource Analysis(AARA) makes the technique extensible to many language features and analysis requirements. This thesis presents a new soundness proof of AARA with respect to a cost semantics via an abstract machine that makes control stack explicit. We showcase the power of this new semantics, proof, and AARA, by extending them to support exception handling.

Bibliography

[1] Byeong-Mo Chang and Kwanghoon Choi. A review on exception analysis. *Information and Software Technology*, 77:1–16, 2016. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2016.05.003. URL `https://www.sciencedirect.com/science/article/pii/S0950584916300830`. 1.2

[2] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In Pascal Van Hentenryck, editor, *Static Analysis*, pages 114–126, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69576-9. 1.2

[3] J Guzmán and Ascánder Suárez. An extended type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, volume 135, 1994. 1.2

[4] J Guzmán and Ascánder Suárez. An extended type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, volume 135, 1994. 1.2

[5] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016. ISBN 1107150302. 2.2, 3.1, 4.1

[6] Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, 32(6):729–759, 2022. doi: 10.1017/S0960129521000487. 1.1, 2.3, 5.1.1

[7] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, mar 2000. ISSN 0164-0925. doi: 10.1145/349214.349230. URL `https://doi.org/10.1145/349214.349230`. 1.2

[8] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in standard ml programs. *Theoretical Computer Science*, 277(1):185–217, 2002. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(00)00317-0. URL `https://www.sciencedirect.com/science/article/pii/S0304397500003170`. Static Analysis. 1.2