Handling Exceptions and Effects with Automatic Resource Analysis

ETHAN CHU*, YIYANG GUO*, and JAN HOFFMANN, Carnegie Mellon University, USA

There exist many techniques for automatically deriving parametric resource (or cost) bounds by analyzing the source code of a program. These techniques work effectively for a large class of programs and language features. However, non-local transfer of control as needed for exception or effect handlers has remained a challenge.

This paper presents the first automatic resource bound analysis that supports non-local control transfer between exceptions or effects and their handlers. The analysis is an extension of type-based automatic amortized resource analysis (AARA), which automates the potential method of amortized analysis. It is presented for a simple functional language with lists and linear potential functions. However, the ideas are directly applicable to richer settings and implemented for Standard ML and polynomial potential functions.

Apart from the new type system for exceptions and effects, a main contribution is a novel syntactic type-soundness theorem that establishes the correctness of the derived bounds with respect to a stack-based abstract machine. An experimental evaluation shows that the new analysis is capable of analyzing programs that cannot be analyzed by existing methods and that the efficiency overhead of supporting exception and effect handlers is low.

CCS Concepts: • **Theory of computation → Control primitives**; **Program analysis**; **Type theory**; **Abstract machines**; **Operational semantics**.

Additional Key Words and Phrases: Automatic Resource Bound Analysis, Cost Analysis, Type Systems, Algebraic Effects, Computational Effects, Effect Handlers, Exceptions, Automatic Amortized Resource Analysis, K-Machine

ACM Reference Format:
Ethan Chu, Yiyang Guo, and Jan Hoffmann. 2026. Handling Exceptions and Effects with Automatic Resource Analysis. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 99 (April 2026), 41 pages. https://doi.org/10.1145/3798207

1 Introduction

The goal of *automatic resource bound analysis* is to statically and automatically infer information on the resource use of a program. In this context, resources are quantities used during the execution of the program, such as time, energy, or memory. Such information is useful for users of a software library [62], to schedule jobs in a data center, to ensure safety and security in resource constraint settings such as embedded systems [20, 107] and smart contracts [6, 32], and to automatically detect performance bugs [37]. There are many existing methods for inferring resource information, including type systems [10, 48, 60, 101, 104, 109], deriving and solving recurrence relations [4, 28, 42, 71, 106], and other static analyses [12, 19, 46]. Most techniques are designed to derive (worst-case) upper bounds. However, some analyses focus on (best-case) lower bounds [8, 40, 85] and expected

*The first two authors contributed equally to the paper.

Authors' Contact Information: Ethan Chu, ethanchu@cmu.edu; Yiyang Guo, yiyangg@alumni.cmu.edu; Jan Hoffmann, jhoffmann@cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

ACM 2475-1421/2026/4-ART99
https://doi.org/10.1145/3798207

cost [13, 24, 84]. Automatic resource bound analysis is an undecidable problem, but the work in the area continues to expand the class of bounds that can be derived, to improve the efficiency of analyses, and to expand the set of supported language features.

Non-Local Control. Some of the most challenging programming language features for automatic resource analysis involve non-local transfer of control, such as exception handlers, continuations [38, 91], and effect handlers [15, 89].

Exceptions are a popular language feature that can redirect control flow by dynamically transferring control to an *exception handler*, which often resides in a previously called function. It is comparatively straightforward to support errors or exceptions without handlers [57] when deriving worst-case bounds. The reason is that early termination with an error reduces the resource cost in comparison to the non-error case. Exception handlers are challenging because resource analyses are generally designed around regular control-flow patterns such as structural recursion or simple for loops. We are only aware of one resource analysis that has limited support for intraprocedural exceptions handlers [4], that is, handlers that catch exceptions raised in the same function call. We are not aware of a resource analysis that supports general exceptions where raised exceptions cannot be statically linked to a handler in the same function.

Algebraic effects and the associated paradigm of programming with effect handlers have gained popularity in the programming language community in the past few years [34, 35, 97, 100]. They can be seen as a generalization of exceptions where the user defines a set of operations (or effects), whose implementation is context-dependent and provided by an effect handler. Compared to exception handlers, effect handlers provide additional difficulties for resource analysis: they can access a *delimited continuation* [38, 72] to resume the computation at an operations' call site or to provide a new effect handler for the deferred computation. Analyzing programs with continuations is hard because it is difficult to predict the dynamic control-flow. To the best of our knowledge there are no resource analyses that support continuations, delimited continuations, or effect handlers.

Resource Bounds for Programs with Exception and Effects. This paper introduces the first resource bound analysis for computational effects and their handlers. In addition, it is the first resource bound analysis that fully supports exception handlers and exceptions that can transfer control between different functions. We extend a type-based technique called automatic amortized resource analysis (AARA) [58], which was introduced by Hofmann and Jost [60]. In AARA, types constitute resource bounds, and type derivations are certificates of the soundness of the bound with respect to an operational cost semantics. It is parametric in the resource and derives high-water mark bounds for non-monotone resource metrics. The analysis supports recursive types [44], higher-order functions [63], arrays, and references [80]. AARA can be seen as an automation of the potential methods of amortized analysis, and the use of potential functions is crucial for the treatment of exception handlers.

Contributions. We conservatively extend AARA with typing rules for exceptions, computational effects, and handlers supporting one-shot continuations so that we can account for resource cost that depends on the non-local transfer of control. The cost of executing exception or effect handlers is bounded by a function of the payload of exceptions or effects. The analysis is parametric in the type of this payload. In a nutshell, the idea is that a handler establishes a contract that consists of a potential function, which specifies the amount of potential that needs to be provided by an exception or effect. In this article, we focus on linear worst-case bounds for a simple call-by-value language with computational effects with handlers, higher-order functions, lists, sums, and products. However, extending the techniques to more language features and a larger set of bounds is orthogonal. In fact, we implemented the technique for recursive types and polynomial bounds. We also show how exceptions, exception handlers, and their respective typing rules can be derived from effects and effect handlers. *Besides the design of the resource-aware type system, the main technical contributions are the soundness proof, the implementation, and an experimental evaluation.*

Soundness Proof. A technical innovation of this work is a novel proof technique for establishing the soundness of the derived bounds with respect to a stack-based abstract machine that specifies the cost and behavior of programs. Almost all past soundness proofs are based on big-step cost semantics and expressions in let-normal-form [54, 57, 60, 67]. Here we use a small-step style soundness argument that includes progress and preservation theorems. The small-step approach with an abstract machine supports continuations, which are needed to model computational effects. Instead of let-normal form, we use a modal separation between values and expression in the manner of fine-grain call-by-value [79]. This explicit separation of pure values and possibly effectul computations streamlines the typing rules, evaluation rules, and the proof. Another soundness proof for AARA (without exceptions) based on a small-step semantics has been given by Grosen et al. [44]. The difference is that the proof in this paper is fully syntactic and does not rely on a logical relation, uses a stack-based abstract machine, and a modal separation of expressions and values. Another approach is to use a big-step cost semantics and a logical relation to prove soundness [90].

Implementation and Experimental Evaluation. We have implemented the analysis for Standard ML (SML) programs to evaluate the effectiveness of the analysis, its overhead compared to a prior implementation of AARA without support for exception handlers or effects, and the quality of the derived bounds. The implementation uses the MLton SML compiler [105] as a front end to convert SML programs into an intermediate form. In contrast to the presentation in the paper, we support bounds that are polynomials in sizes of values of recursive types. We evaluate the implemented analysis with 20 benchmarks, which all use either exceptions and effects. Of these, 14 cannot be analyzed with existing techniques, as they either handle exceptions or use effects. We find that the implementation efficiently finds tight worst-case bounds for these benchmarks and that our technique has low overhead compared to existing AARA implementations like RaML [56].

2 Overview

In this section, we describe the basic idea and state-of-the-art of type-based automatic amortized resource analysis (AARA), and the challenges that have hindered these systems from analyzing programs with exceptions. We then discuss the main technical innovations of this work, namely (in Section 2.2) a modular extension of AARA that can be combined with existing type systems to derive bounds for programs with exception handlers and (in Section 2.3) effect handlers.

2.1 Type-Based Automatic Amortized Resource Analysis

Type-based AARA was introduced by Hofmann and Jost [60] to automatically derive *linear bounds* on the heap-memory use of *first-order* functional programs. Its appeal is that

- It derives tight symbolic worst-case bounds for many typical functions,
- It is proven sound with respect to a cost semantics,
- Type derivations are easily-checkable certificates of the derived bounds,
- Type (and therefore bound) inference can be reduced to linear programming (LP), and
- The integration of types with the physicist's method of amortized analysis makes the system naturally compositional.

Maybe surprisingly, these properties of type-based AARA have been preserved in conservative extensions of the original work to arbitrary user-defined resource metrics [64], non-linear bounds [54, 61, 66], general recursive types [44], lower bounds [85], and higher-order functions [63]. A more complete discussion can be found in a recent survey [58]. To focus on the main ideas, we develop the extension of AARA to exceptions and effects based on an eager, higher-order functional language with lists and a type system that only encodes linear resource bounds, which we call linear-bound AARA. However, we claim that this extension can apply to more complex AARA systems, e.g. one capable of deriving polynomial resource bounds, with only minor changes. We omit the discussion of said changes from this paper, but as an example, the evaluation conducted in section 5 uses polynomial AARA extended with exceptions and effects.

Cost Model. Various cost and resource models for functional languages have been defined for monotone resources such as runtime and non-monotone ones such as space use. To abstract from a specific cost model, we follow previous work on resource analysis [43, 52] and equip the language with a tick effect – we can think of the expression tick q as adding the fixed rational number q to some global counter denoting the cost incurred. Even though the tick effect takes a fixed argument, we can simulate non-constant ticks. For example, we can define a function tick_nat such that tick_nat n will ultimately perform n ticks. This is achieved by representing n as a unit list of length n, to which we apply a recursive function that traverses the list and performs tick 1 at each element.

The tick resource model is a well-established and adequate methodology. To define a sound cost model for time (or space), it needs to be justified with respect to some machine model. This is known as a bounded implementation and was pioneered by Blelloch and Greiner [17, 18]. It is well established that constant tick annotations correspond to the execution of an interpreter on a random-access machine (RAM), modulo constant factors. Such cost models do not necessarily incur cost proportional to the theoretical size increase resulting from substitution during beta reduction [29]. Instead, we can leverage an abstract machine model that simulates the lambda calculus more efficiently, such as explicit substitution/sharing as described by Accattoli [1].

In most of the examples throughout this paper, we will instrument the code with ticks to measure the number of *evaluated function calls* and *exception or effect handler evaluations*. This is an interesting resource metric because in a functional program, the number of function calls and handler evaluations performed is proportional to its asymptotic runtime complexity. Additionally, it can be easily extended to be more realistic – there has been work demonstrating how to select the amounts to tick by such that the derived bounds correspond to the actual clock cycles needed to execute the compiled code [63].

An immediate concern readers may have with our chosen resource metric is that it assumes constant cost whenever a handler is run, rather than a dynamic cost, which is often the case when locating the correct effect handler to use. Our type system sidesteps this concerns in two ways. First, our handlers require explicit forwarding [50, 68], which is to say that each handler must handle *all* effects that can be performed by the computation being handled. For the effects that it does not morally handle, it still must explicitly handle, incur cost via tick, then re-perform the effect, thus forwarding it. This ensures that whenever an effect is performed, control flow is always transferred to the closest enclosing effect handler, instead of having to dynamically search for a handler that handles the given effect. Then the tick's triggered at each forwarding site effectively simulate the non-constant cost of identifying a suitable handler. Next, by enforcing a one-shot discipline for delimited continuations in handlers, we ensure that non-handler stack frames are never executed more than once, so the runtime can simply store a pointer to the closest enclosing effect handler instead of having to copy or shuffle around stack frames [96]. Together, this justifies incurring only constant cost at each handler evaluation.

Potential Method. The potential method of amortized analysis was introduced [98] to analyze the cost of a sequence of data structure operations. The key idea is to define a potential function for

```
fun sqdist (v1 : real list, v2 : real list) : real =
   case (v1,v2) of ([],[]) => 0.0
    | (x::xs,y::ys) => (x - y) * (x - y) + (R.tick 1; sqdist (xs, ys))
    | ([],_::_) => raise Emis1
    | (_::_,[]) => raise Emis2
```

Fig. 1. The SML function sqdist computes the square of the Euclidean distance between two n-dimensional vectors (lists of reals). The function R.tick is used to specify resource use.

the data structure and the amortized cost of an operation. Then one has to demonstrate that for all possible executions, the amortized cost and the potential carried by the data structure is sufficient to cover both the actual cost and the potential of the resulting data structure. Then initial potential and the sum of the amortized costs is an upper bound on the resource consumption of the program.

Resource Annotated Types. AARA is based on the potential method of amortized analysis. To automate amortized analysis, it fixes the set of possible potential functions via types. Specifically, types induce potential functions that map data structures of the given type to potential, given as a non-negative numbers. Then, typing rules ensure that terms which introduce or eliminate these types preserve potential in a manner consistent with the potential functions. Note that in AARA, the amortized cost is always zero, so the cost of a program is fully bound by the initial potential.

On top of a simple type system with functions, lists, sums, and products, AARA equips certain types with potential annotations:

(1) List types are annotated with a constant potential carried by each element of the list. For example, the type $L^q(\mathsf{unit})$ encodes a unit list that carries q potential per element. In other words, it defines the potential function $\Phi(v : L^q(\mathsf{unit})) = q|v|$.
(2) Each variant of a sum type is annotated with some constant potential carried by an injection into that variant. For example, the type $\mathsf{unit}^p + \mathsf{unit}^q$ encodes a boolean that carries p potential when true and q potential when false. Concretely, it defines the potential function $\Phi(v : \mathsf{unit}^p + \mathsf{unit}^q) = p$ if $v = \mathsf{inl}(\langle\rangle)$, q if $v = \mathsf{inr}(\langle\rangle)$.
(3) In functions, the argument type is annotated with a constant potential that must be provided when applying the function, and the result type is annotated with a constant potential that gets returned alongside the result. For example, a function of type $\tau_1 \rightarrow^p_q \tau_2$ requires p potential to be provided along with its argument, and returns q potential along with its result.

Example. Consider the function sqdist in Figure 1, which computes the square of the Euclidean distance between two n-dimensional vectors represented as lists of reals ($L(\mathbb{R})$). It recursively iterates down both lists until at least one of the lists is empty, raising exception Emis$i$ if the i-th list is shorter ($i = 1, 2$). Since the resource metric chosen is the *number of evaluated function calls*, the call sqdist(v1,v2) costs min(|v1|, |v2|), which we would like to derive using AARA. Indeed, using the simple and local typing rules of linear-bound AARA, we can justify numerous possible types for sqdist, such as

$$L^1(\mathbb{R}) \times L^0(\mathbb{R}) \rightarrow^0_0 \mathbb{R} \qquad L^0(\mathbb{R}) \times L^1(\mathbb{R}) \rightarrow^3_3 \mathbb{R} \qquad L^{0.5}(\mathbb{R}) \times L^{0.5}(\mathbb{R}) \rightarrow^5_5 \mathbb{R}$$

Given the function call sqdist(v_1, v_2), these types express the cost bounds $|v_1|$, $|v_2| + 3 - 3$, and $0.5|v_1| + 0.5|v_2| + 5 - 5$, respectively. All three bounds are upper bounds on the actual cost of min(|v1|, |v2|). Which bound is the best depends on the context in which the call sqdist(v_1, v_2) appears. To facilitate this, our type system actually expresses function types as a set of possible types. During type inference, this set is governed by linear constraints that we infer. For example, sqdist would be inferred to have the following type:

$$\mathsf{sqdist} : \{L^{q_1}(\mathbb{R}) \times L^{q_2}(\mathbb{R}) \rightarrow^p_{p'} \mathbb{R} \mid q_1 + q_2 \geq 1 \land p \geq p'\}$$

```
1 fun distances_1 (vs : real list list, p : real list) : real list =
2   List.map (fn v ⇒ R.tick 1; sqdist (v, p)) vs
3 fun distances_2 (vs : real list list, p : real list) : real list =
4   let fun f v = (R.tick 1; sqdist (v, p)) handle Emis2 ⇒ (R.tick 1; ~1.0)
5   in List.map f vs end
```

Fig. 2. The functions distances_1 and distances_2 in SML. The exception handler in distances_2 incurs a cost of 1 whenever it is evaluated. The function call List.map f l applies f to each element of l.

Then at each callsite of sqdist, an LP solver will select the best concrete choices for the potential variables q_1, q_2, p, p' to minimize the cost of the larger program in which sqdist is called. In this sense, our type system supports *resource polymorphism*. However, for simplicity's sake, most examples in this paper will only discuss monomorphic type signatures when typing functions.

Compositionality is achieved by assigning potential to result types. For example, if we wish to type the expression sqdist(append(v_1, v_2), v') where $v' : L^0(\mathbb{R})$, then we must construct a typing of append with result type $L^1(\mathbb{R})$. Then assuming that append does not contain tick expressions (and thus has cost 0), we can give it the following type:

$$\text{append} : L^1(\mathbb{R}) \times L^1(\mathbb{R}) \to_0^0 L^1(\mathbb{R})$$

This type reflects that appending two lists that both carry one potential per element produces a list carrying one potential per element.

2.2 Handling Exceptions

While there are some AARA systems that support errors (or raising exceptions), there does not exist a type-based AARA system that supports handling exceptions, which is arguably the non-trivial part of supporting exceptions that this paper develops. In this subsection, we will analyze the functions distances_1 and distances_2 in Figure 2, which call the previously discussed sqdist (from Figure 1) and handle the Emis$i$ exceptions that it raises. For the sake of simplicity, we will ignore the function calls induced by List.map[1], so our cost metric in the following discussion is the number of function calls *to sqdist* only, as well as the number of *exception handler evaluations*.

Consider the function distances_1. It takes a list vs of vectors and a point p (a vector) and calls the standard list map function List.map to compute the distance squared of p to each vector in the list using the function sqdist. Recall that we are not counting the number of function applications induced by List.map, so the cost of an evaluation of distances_1([$v_1 \ldots, v_n$],p) is $n + \sum_{1 \le i \le n} \min(|v_i|, |p|)$. One valid upper bound for this is encoded by the AARA typing

$$\text{distances\_1} : L^1(L^1(\mathbb{R})) \times L^0(\mathbb{R}) \to_0^0 L^0(\mathbb{R}),$$

which corresponds to the bound $n + \sum_{1 \le i \le n} |v_i|$. We can actually arrive at this typing by partially relying on the previously derived typing sqdist : $L^1(\mathbb{R}) \times L^0(\mathbb{R}) \to_0^0 \mathbb{R}$. distances_1 maps sqdist across its first argument, and sqdist requires 1 potential per element of its first argument, so unsurprisingly, distances_1 requires 1 potential per *inner* list element of its first argument. Slightly more interestingly, to pay for the very first call to sqdist in distances_1 (R.tick 1 in the lambda on line 2), disances_1 must carry 1 additional potential per *outer* list element.

Next, consider the function distances_2. It performs the same computation as distances_1 but handles the exception Emis2, which can be raised by sqdist. A R.tick expression in the handler specifies an additional cost of 1 if the handler is evaluated. Would we simply treat the handler as a sequential computation, we would derive the bound $n + \sum_{1 \le i \le n}(1 + |v_i|)$, or $2n + \sum_{1 \le i \le n} |v_i|$, for the cost of evaluating distances_2([$v_1 \ldots, v_n$],p). At first sight, this looks like a tight bound.

[1]When analyzing distances_1 and distances_2 in the evaluation section 5, we do count the function calls induced by List.map. This is a simplification for the overview only.

However, the type system presented in this paper can prove the more precise bound $n + \sum_{1\leq i\leq n} |v_i|$ by deriving the following typing, which is identical to the typing of distances_1. [2]

$$\text{distances\_2} : L^1(L^1(\mathbb{R})) \times L^0(\mathbb{R}) \to_0^0 L^0(\mathbb{R})$$

Before we discuss how we can derive this tight bound, let us first intuit why it is correct. First note that the bound $\sum_{1\leq i\leq n} |v_i|$ is tight if each inner list v_i is fully traversed in the function sqdist (let's informally call this the worst-case). Now consider the case in which the exception Emis2 is raised in the function sqdist. In this case, the second argument of sqdist is shorter than the first, and consequently, the respective inner list v_i has not been fully traversed. So we are not in the previously described "worst-case" situation when the handler gets evaluated: an additional cost of 1 does not exceed the cost of fully traversing v_i and not handling an exception.

How can we express this subtlety in a type system that is simple enough to support type inference? *The idea is to associate potential annotations with exception types, thus establishing a contract between the point in the code at which an exception is raised and the point at which it is handled.* The amount of potential is chosen to fit the cost of the innermost exception handler, where the potential may be consumed. In return, the same amount of potential has to be provided each time an exception is raised.

To see the idea in action, let's try to justify the typing of distances_2 shown above. We can mostly repeat the steps we took the type distances_1, but now we must decide what potentials p, q to annotate the exception type $\text{Emis1}^p + \text{Emis2}^q$ with. There is no handler for the exception Emis1, so it can carry no potential, i.e. we let $p = 0$. In contrast, the handler for Emis2 performs a R.tick 1, which means Emis2 should carry 1 potential, i.e. we let $q = 1$. Thus this exception handler is well-typed.

But on the other end of the contract, we must ensure that whoever raises Emis2 within distances_2 must provide 1 potential. To achieve this, we must now construct a new typing for sqdist, specifically for its callsite in distances_2. Earlier, we typed distances_1 by relying on the typing sqdist : $L^1(\mathbb{R}) \times L^0(\mathbb{R}) \to_0^0 \mathbb{R}$, so let's start there again. The first argument v1 in the sqdist gets the type $L^1(\mathbb{R})$, i.e. it carries 1 potential per element. Next, observe that raise Emis2 occurs in the case when v_1 matches _::_, which means its length, and thus the potential it carries, is ≥ 1 and available for raise Emis2. We can thus justify the following typing of sqdist, now equipped with an annotated exception type:

$$\text{sqdist} : L^1(\mathbb{R}) \times L^0(\mathbb{R}) \to_{0\odot 0}^0 \mathbb{R} \odot (\text{Emis1}^0 + \text{Emis2}^1)$$

Here, we have modified annotated function types to become $\tau_1 \to_{q\odot q_e}^p \tau_2 \odot \tau_e$, where τ_e is the resource annotated exception type, and q_e is additional potential returned alongside the raised exception. [3] Therefore, this typing expresses that in the evaluation of sqdist(v1,v2), |v1| is sufficient to cover the R.tick's within *and* provide 1 additional potential whenever Emis2 is raised.

Our type system is also capable of typing exceptions carrying non-trivial payloads like lists. The subsequent section on handling computational effects will demonstrate such a payload.

2.3 Handling Computational Effects

In this section, we will demonstrate how our type system can analyze programs that perform and handle effects. We will consider programs equipped with a set of named effects, each with an input and output type. For example, the canonical IO effect Print might have input type string and output type unit. An effect EName takes a payload x of its input type, and is performed via the expression do[EName] x, which has the effect's output type. For example, do[Print] "Hello World"

[2] In this example, the difference is only a constant factor. However, such imprecisions can easily multiply in a larger program and lead to asymptotic differences in the derived bounds.

[3] In most of our examples, τ_e is a sum type containing the various exceptions declared in the program, in which case q_e is redundant thanks to the annotations at each variant of a sum type. Our type system, however, has no restriction on τ_e.

```
1  effect Insert : int list ⇒ unit
2  effect Remove : unit ⇒ int list option
3
4  fun hstore (m : unit -> unit) : unit = (
5     m () handle
6       return x ⇒ (fn store ⇒ x)
7     | Insert n  k ⇒ (R.tick 1;
8       fn store ⇒ k () (n :: store))
9     | Remove () k ⇒ (R.tick 1;
10      fn store ⇒
11        case store of
12          [] ⇒ k NONE []
13        | x :: xs ⇒ k (SOME x) xs
14      ) ) []
15
16 fun insert_lists (l : int list list) :
       unit =
17   case l of
18     [] ⇒ ()
19   | x::xs ⇒ (do[Insert] x; insert_lists
       xs)
20
21 fun consume () : unit =
22   case do[Remove] () of
23     NONE ⇒ ()
24   | SOME x ⇒ (traverse x; consume ())
25
26 fun store_lists (l : int list list) :
       unit = hstore (
27   fn () ⇒ insert_lists l; consume ())
```

Fig. 3. Example implementing a stack store with effect handlers. We assume access to traverse : $L(\mathbb{Z}) \rightarrow$ unit.

is an expression that performs the Print effect with the string "Hello World" as its payload, and has type unit, as that is the output type.

To handle effects, we have effect handler expressions of the form e **handle return** a => e1 | EName x k => e2 | … where e is the effectful expression being handled. When e evaluates to a pure value v, then v is substituted for a in e1 to get the final value. However, if some effect EName is performed, the handler branch e2 is executed, given the effect's payload x and a (delimited) continuation k that allows it to resume execution of e from the site where EName was performed. We work in the deep handler setting of Kammar et al. [68], so the continuation k reinstalls the effect handler.

Consider the example in Figure 3. We first define two effects: Insert requires an integer list payload and returns unit, and Remove requires a unit payload and returns either an integer list or nothing. Together, they intend to implement a store data structure. The function hstore takes a thunk m, which may perform the effects Insert and Remove, and handles them. In both branches of the handler, we use R.tick to specify a cost of 1 if the handler is evaluated, just as we did for exceptions. This effect handler uses functions to encode state, threading through the state store. The branch for Insert is a function which calls the continuation k on a unit before updating the state store by prepending the effect payload n. The branch for Remove is a function that cases on the current state store. If store is empty, it calls the continuation k on None followed by passing the empty list as the state. If store is nonempty, it calls the continuation k on Some x where x is the head of store, and updates the state to be the tail xs of store.

Now we move on to the computation we wish to handle, store_lists on line 26. Given a list of lists of integers l, it first calls insert_lists l, which performs the Insert effect on each element of l. Next, it calls consume (), which repeatedly performs the Remove effect until it returns NONE, signifying that the store has been emptied. Importantly, for each SOME x that Remove outputs, we call traverse x, which consumes 1 potential per element of x, i.e. we assume that it has the typing traverse : $L^1(\mathbb{Z}) \rightarrow^0_0$ unit. Our type system is then able to derive that store_lists $[v_1, \dots, v_n]$ has a cost of $1 + 2n + \sum_{1 \le i \le n} |v_i|$, or the following type.

$$\text{store\_lists} : L^2(L^1(\mathbb{Z})) \rightarrow^1_0 \text{unit}$$

We start by arguing why this bound is correct. There are three places that incur cost in store_lists: The call insert_lists $[v_1, \dots, v_n]$ inserts each v_i into the store exactly once, so that incurs n total cost. The call consume () removes elements from the store until (and including when) it is empty, which results in cost $n + 1$. Finally, traverse x is called for every element x removed from the store, which are $v_1, \dots, v_n$, incurring a total cost of $\sum_{1 \le i \le n} |v_i|$.

But how is our type system able to figure this out? The answer lies in having annotated types for effects. We build on the type system for exceptions explained in the Subsection 2.2. Now, the annotated exception type in a function's typing is replaced with an effect signature containing the annotated input and output types of each effect that its body may perform. Similar to annotated exception types, these resource annotated effect signature are contracts between the client code and handler. Let's look at the typings of insert_lists and consume from our example.

$$\mathsf{insert\_lists} : L^2(L^1(\mathbb{Z})) \to_0^0 \mathsf{unit} \odot \{\mathsf{Insert} : L^1(\mathbb{Z}) \Rightarrow_0^2 \mathsf{unit}\}$$

The type of insert_lists promises that when the function performs the Insert effect, it will supply a list payload that carries 1 potential per element, as well as 2 additional potential, while expecting the handler to return 0 potential. Its typing is relatively straightforward – its argument $[v_1, \dots, v_n]$ is a list of lists, each of which carries $2 + |v_i|$ potential, which it simply forwards to the Insert handler via the payload.

$$\mathsf{consume} : \mathsf{unit} \to_0^1 \mathsf{unit} \odot \{\mathsf{Remove} : \mathsf{unit} \Rightarrow_0^1 \left(\mathsf{Some}^1(L^1(\mathbb{Z})) + \mathsf{None}^0\right)\}$$

The type of consume promises to supply 1 potential when the function performs the Remove effect and expects in return either Some x carrying $1 + |x|$ potential or None carrying no potential. It pays for the performance of Remove from its argument carrying 1 potential. Then when casing on the result of the Remove, in the Some case on line 24, $|x|$ potential pays for the call traverse x, while the 1 remaining potential pays for the recursive call consume ().

On the other end of the contract, the effect handler in hstore must be able to handle the Insert and Remove effects with the annotated types above, namely

$$\{\mathsf{Insert} : L^1(\mathbb{Z}) \Rightarrow_0^2 \mathsf{unit}, \mathsf{Remove} : \mathsf{unit} \Rightarrow_0^1 \left(\mathsf{Some}^1(L^1(\mathbb{Z})) + \mathsf{None}^0\right)\}$$

Furthermore, all branches of an effect handler must have the same type – in this handler, all branches have type $L^1(L^1(\mathbb{Z})) \to_0^0$ unit, which can be justified as follows. (1) The return branch on line 6 is a function that does not touch its argument, trivially satisfying the type. (2) The Insert branch on line 7 starts with $|n| + 2$ potental units available, where n is the payload. 1 potential pays for R.tick 1. In the function, store starts with type $L^1(L^1(\mathbb{Z}))$. This type is preserved when we prepend n along with the remaining $|n| + 1$ potentials to the store via n :: store. (3) The Remove branch on line 9 starts with 1 potential, which pays for R.tick 1. In the function, we case on store, which has type $L^1(L^1(\mathbb{Z}))$ again. If empty, we send None to the client without any potential. If nonempty, i.e. it matches x::xs, we send Some x along with the $|x| + 1$ potential available from this head element to the client.

Just like **raise** v for exceptions, we are able to transfer complex potential via payload v from **do**[EName] v to the closest enclosing effect handler. However, unlike exception handlers, effect handlers can also transfer potential back to effectful computations via their delimited continuations.

3 Type System

In this section, we present the syntax of a simple functional language and type rules for deriving resource bounds. Our language starts with exceptions and their handlers only, which we will extend to support effects as well. We also briefly discuss type inference.

3.1 Syntax

We develop our type system on a language with (higher-order) functions, lists, pairs, and binary sums. In our implementation, we use a more complete functional language that includes regular recursive types. We restrict ourselves to the simple setting because it already exhibits all the technical difficulties that arise with exception handlers (and effects).

$$\begin{aligned} v &::= x & e &::= \mathsf{val}(v) \mid \mathsf{let}(e_1; x.e_2) \mid \mathsf{tick}\{q\} \mid \mathsf{app}(v_1; v_2) \\ &\mid \mathsf{fun}(f.x.e) & &\mid \mathsf{casepair}(v; x_1.x_2.e) \\ &\mid \langle\rangle \mid \mathsf{pair}(v_1; v_2) & &\mid \mathsf{casevoid}(v) \mid \mathsf{casesum}(v; x_1.e_1; x_2.e_2) \\ &\mid \mathsf{inl}(v) \mid \mathsf{inr}(v) & &\mid \mathsf{caselist}(v; e_1; x_1.x_2.e_2) \\ &\mid \mathsf{nil} \mid \mathsf{cons}(v_1; v_2) & &\mid \mathsf{raise}(v) \mid \mathsf{try}(e_1; x.e_2) \end{aligned}$$

We syntactically distinguish values v and expressions/computations e using a modal separation based on lax logic [33] and fine-grain call-by-value [79]. Values include the trivial unit value, functions, pairs, injections, and lists. Since we work in a call-by-value setting, variables stand for values and are therefore considered values. In the examples and the implementation, we use SML syntax, which can be translated into the lax syntax.

Subcomponents of expressions are values whenever possible without restricting expressivity. They include function application, as well as pattern matching elimination for pairs, injections, and lists. Sequential evaluation of computations only occurs within let expressions. Values v are lifted to expressions using the syntactic form $\mathsf{val}(v)$. The syntactic form $\mathsf{tick}\{q\}$ specifies the consumption of $q \in \mathbb{Q}$ resources. As usual, they can be automatically inserted in higher-level code based on a resource metric that can model runtime, memory use, or energy consumption. If $q < 0$ then q resources become available. This is used, for example, to mark when memory is freed. Finally, exceptions can be raised using the computation $\mathsf{raise}(v)$. The computation $\mathsf{try}(e_1; x.e_2)$ defines an exception handler e_2 that handles exceptions raised by e_1.

3.2 Resource Annotated Types

Resource annotated types are defined as follows.

$$\tau ::= \mathcal{T} \mid \mathsf{unit} \mid \tau_1 \times \tau_2 \mid \mathsf{void} \mid A_1 + A_2 \mid \mathsf{L}(A) \qquad A, B, C ::= \langle \tau, q \rangle \qquad \mathcal{T} ::= \{A \to B \odot C \mid \Theta\}$$

The idea is that types τ are annotated with non-negative numbers which define a potential function $\Phi(v : \tau)$ that maps values v of type τ to non-negative numbers. The subsequent subsection will formally define the potential functions.

Types τ include arrow types, lists, products, and sums. A, B, C are types with constant potential $q \in \mathbb{Q}_{\geq 0}$ attached to them, allowing us to specify for example $\mathsf{L}(\langle \tau, q \rangle)$, a list of τ's where each element carries q potential. Similarly, we can also specify sum types $\langle \tau_1, q_2 \rangle + \langle \tau_2, q_2 \rangle$ specifying that a left injection will carry q_1 potential in addition to having a payload of type τ_1, and likewise for the right injection. Finally, function types are sets $\mathcal{T}$ of possible types each of the form $A \to B \odot C$ governed by some predicate Θ, typically the conjunction of various linear constraints regarding potential annotations $q \in \mathbb{Q}_{\geq 0}$ that appear within A, B, C. Note that B is the function's normal return type, while C is the type of exceptions it could raise. For a concrete example of a set $\mathcal{T}$ of function types, refer to the resource polymorphic typing of sqdist in the overview section 2.1.

Note that we use a more readable notation for resource annotated types throughout the overview section 2.1. $L^q(\tau)$ is shorthand for $\mathsf{L}(\langle \tau, q \rangle)$, $\tau_1^p + \tau_2^q$ is shorthand for $\langle \tau_1, p \rangle + \langle \tau_2, q \rangle$, and $\tau_1 \to^p_{q \odot q_e} \tau_2 \odot \tau_e$ is shorthand for $\langle \tau_1, p \rangle \to \langle \tau_2, q \rangle \odot \langle \tau_e, q_e \rangle$.

We also introduce the function $|\cdot|$ that takes in a resource annotated type and zeroes all the potential annotations within positive types. It is defined in Figure 4 and will be useful for defining various typing rules in the rest of the paper. We extend it to contexts too, defining $|\Gamma|$ as the pointwise application of $|\cdot|$ to each type in Γ.

$\boxed{|\tau| \quad |A|}$ zeroing

$$|\mathcal{T}| = \mathcal{T} \qquad |\mathsf{unit}| = \mathsf{unit} \qquad |\tau_1 \times \tau_2| = |\tau_1| \times |\tau_2| \qquad |\mathsf{void}| = \mathsf{void} \qquad |A_1 + A_2| = |A_1| + |A_2|$$
$$|\mathsf{L}(A)| = \mathsf{L}(|A|) \qquad |\langle \tau, q\rangle| = \langle |\tau|, 0\rangle$$

Fig. 4. Zeroing Resource Annotated Types

$\boxed{\Gamma; q \vdash v : \tau}$ value typing

$$\text{T-Var}\ \frac{}{x : \tau; 0 \vdash x : \tau} \qquad \text{T-Pair}\ \frac{\Gamma_1; q_1 \vdash v_1 : \tau_1 \qquad \Gamma_2; q_2 \vdash v_2 : \tau_2}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{pair}(v_1; v_2) : \tau_1 \times \tau_2} \qquad \text{T-Inl}\ \frac{\Gamma; p \vdash v : \tau_1}{\Gamma; q_1 + p \vdash \mathsf{inl}(v) : \langle \tau_1, q_1\rangle + A_2}$$

$$\text{T-Fun}\ \frac{\Gamma = |\Gamma| \qquad \forall (\langle \tau, q\rangle \to B \odot C) \in \mathcal{T}.\ \Gamma; f : \mathcal{T}, x : \tau; q \vdash e : B \odot C}{\Gamma; 0 \vdash \mathsf{fun}(f.x.e) : \mathcal{T}} \qquad \text{T-Inr}\ \frac{\Gamma; p \vdash v : \tau_2}{\Gamma; q_2 + p \vdash \mathsf{inr}(v) : A_1 + \langle \tau_2, q_2\rangle}$$

$$\text{T-Unit}\ \frac{}{\cdot; 0 \vdash \langle\rangle : \mathsf{unit}} \qquad \text{T-Nil}\ \frac{}{\cdot; 0 \vdash \mathsf{nil} : \mathsf{L}(A)} \qquad \text{T-cons}\ \frac{\Gamma_1; q_1 \vdash v_1 : \tau \qquad \Gamma_2; q_2 \vdash v_2 : \mathsf{L}(A) \qquad A = \langle \tau, p\rangle}{\Gamma_1, \Gamma_2; q_1 + q_2 + p \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(A)}$$

Fig. 5. Typing rules for values

3.3 Typing Rules for Values

The value typing judgment $\boxed{\Gamma; q \vdash v : \tau}$ states that under context Γ and requiring $q \in \mathbb{Q}_{\geq 0}$ additional potential, value v has annotated type τ. One way to understand these rules is that Γ and q provide the $\Phi(v : \tau)$ potential that v carries. Figure 5 contains the typing rules, in which the context Γ and the potential q are treated linearly. We describe the key rules:

T-Var All the potential is provided by the context $x : \tau$ and 0 additional potential is required.

T-Pair The context Γ_1, Γ_2 and potential $q_1 + q_2$ are split, and $\Gamma_i; q_i$ is used to type v_i.

T-Inl/T-Inr The potential $q_i + p$ is split. The context Γ and potential p are used to type v. q_i additional potential is required depending on which injection it is.

T-Cons The context Γ_1, Γ_2 and potential $q_1 + q_2 + p$ are split. Depending on i, $\Gamma_i; q_i$ types the head or tail of the list. p additional potential is required, as each list element must carry p potential.

T-Fun No potential is consumed when typing a function. $q = 0$, and the $\Gamma = |\Gamma|$ premise ensures that the context carries no potential. This guarantees that the only source of potential when evaluating the function body is the function argument. This way, functions are potential-free and can be invoked any number of times. The rule derives a result type $\mathcal{T}$ that contains possibly infinite arrow types $\langle \tau, q\rangle \to B \odot C$, where B is the annotated return type and C is the annotated exception type, explained further in the subsequent type rules for computations. Each of these arrow types is justified by typing the function body e given the argument $x : \tau$, accompanying constant potential q, the set of function types $\mathcal{T}$ for recursive calls, and the potential-free closure Γ.

The linear treatment of potential enables us to establish the following lemma for closed values, proved by induction on the value typing judgment.

Lemma 3.1. *If $\cdot; q \vdash v : \tau$ and $\cdot; q' \vdash v : \tau$ then $q = q'$.*

Since closed values must be typed with a unique q, we can define the potential function $\Phi(\_ : \tau)$ that maps values of type τ to non-negative rational numbers by setting $\Phi(v : \tau) = q$ if $\cdot; q \vdash v : \tau$.

$\boxed{\Gamma; q \vdash e : B \odot C}$ computation typing

$$\frac{\Gamma; q \vdash v : \tau}{\Gamma; q + p \vdash \mathsf{val}(v) : \langle \tau, p \rangle \odot C}\ \textsc{T-Val} \qquad \frac{\Gamma_1; q \vdash e_1 : \langle \tau, q' \rangle \odot C \qquad \Gamma_2, x : \tau; q' \vdash e_2 : B \odot C}{\Gamma_1, \Gamma_2; q \vdash \mathsf{let}(e_1; x.e_2) : B \odot C}\ \textsc{T-Let}$$

$$\frac{}{\cdot; q + p \vdash \mathsf{tick}\{q\} : \langle \mathsf{unit}, p \rangle \odot C}\ \textsc{T-Tick}^{+} \qquad \frac{}{\cdot; p \vdash \mathsf{tick}\{-q\} : \langle \mathsf{unit}, p + q \rangle \odot C}\ \textsc{T-Tick}^{-}$$

$$\frac{\Gamma_1; 0 \vdash v_1 : \mathcal{T} \qquad (\langle \tau, q_1 \rangle \to B \odot C) \in \mathcal{T} \qquad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{app}(v_1; v_2) : B \odot C}\ \textsc{T-App}$$

$$\textsc{T-Casevoid}\ \frac{\Gamma; q \vdash v : \mathsf{void}}{\Gamma; q \vdash \mathsf{casevoid}(v) : B \odot C} \qquad \textsc{T-Casepair}\ \frac{\Gamma_1; q_1 \vdash v : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2; q_2 \vdash e : B \odot C}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{casepair}(v; x_1.x_2.e) : B \odot C}$$

$$\textsc{T-Casesum}\ \frac{\Gamma_1; q_1 \vdash v : \langle \tau_1, p_1 \rangle + \langle \tau_2, p_2 \rangle \qquad \Gamma_2, x_1 : \tau_1; p_1 + q_2 \vdash e_1 : B \odot C \qquad \Gamma_2, x_2 : \tau_2; p_2 + q_2 \vdash e_2 : B \odot C}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{casesum}(v; x_1.e_1; x_2.e_2) : B \odot C}$$

$$\frac{\Gamma_1; q_1 \vdash v : \mathsf{L}(\langle \tau, p \rangle) \qquad \Gamma_2; q_2 \vdash e_1 : B \odot C \qquad \Gamma_2, x : \tau, y : \mathsf{L}(\langle \tau, p \rangle); q_2 + p \vdash e_2 : B \odot C}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{caselist}(v; e_1; x.y.e_2) : B \odot C}\ \textsc{T-Caselist}$$

$$\frac{\Gamma; q \vdash v : \sigma}{\Gamma; q + r \vdash \mathsf{raise}(v) : A \odot \langle \sigma, r \rangle}\ \textsc{T-Raise} \qquad \frac{\Gamma_1; q \vdash e_1 : A \odot \langle \sigma, r \rangle \qquad \Gamma_2, x : \sigma; r \vdash e_2 : A \odot C}{\Gamma_1, \Gamma_2; q \vdash \mathsf{try}(e_1; x.e_2) : A \odot C}\ \textsc{T-Try}$$

Fig. 6. Typing rules for computations

We call q the potential of v under τ. With Φ, we can state a useful lemma regarding the zeroing function $|\cdot|$, also proved by induction on the typing judgment. It states that $\tau = |\tau|$ ensures that values of type τ must carry no potential. We refer to such types as potential-free.

Lemma 3.2. *If $\tau = |\tau|$ and $\Phi(v : \tau) = q$, then $q = 0$.*

3.4 Typing Rules for Computations and Exceptions

The computation typing judgment $\boxed{\Gamma; q \vdash e : B \odot C}$ states that under context Γ and requiring $q \in \mathbb{Q}_{\geq 0}$ additional potential, computation e can either evaluate to a value and potential of annotated type B, or raise an exception value and potential of annotated type C. Figure 6 contains the syntax-directed computation typing rules, which all treat potential linearly. We describe some key rules:

T-Tick^{+} Ticks with a positive argument consume resources. The $q + p$ input potential covers the tick that consumes q potential, which gets typed as unit with p potential remaining. The annotated exception type C is arbitrary.

T-Tick^{-} Ticks with a negative argument return resources. The p input potential is arbitrary, and the tick is typed as unit with $p + q$ potential attached, since q potential is being returned. The annotated exception type C is arbitrary.

T-Val This rule lifts the typing judgment for value to a computation typing by threading through p additional potential alongside the value as the constant return and an arbitrary type C for the exception payload. This is justified since values cannot raise exceptions.

T-App The context Γ_1, Γ_2 and potential $q_1 + q_2$ are both split. Γ_1 is used to type v_1, which being a function, requires 0 additional potential. Γ_2 and q_2 are used to type v_2, the function argument.

$\boxed{\Gamma; q \vdash e : B \odot C}$ computation typing

$$\textsc{T-ContFun}\ \frac{\Gamma, x_1 : \mathcal{T}, x_2 : \mathcal{T}; q \vdash e : B \odot C}{\Gamma, x : \mathcal{T}; q \vdash [x, x/x_1, x_2]e : B \odot C} \qquad \textsc{T-WeakVar}\ \frac{\Gamma; q \vdash e : B \odot C}{\Gamma, x : \tau; q \vdash e : B \odot C} \qquad \textsc{T-WeakPot}\ \frac{\Gamma; q' \vdash e : B \odot C \qquad q \geq q'}{\Gamma; q \vdash e : B \odot C}$$

Fig. 7. Structural typing rules for computations

Finally, the resulting annotated result and exception types $B \odot C$ are valid if the arrow type $\langle \tau, q_1 \rangle \to B \odot C$ is in $\mathcal{T}$.

T-Let The context Γ_1, Γ_2 is split. $\Gamma_1; q$ types e_1. e_1's annotated type's potential q', along with Γ_2 augmented with $x : \tau$ are used to type e_2, whose type B is the overall result type. Most importantly for this paper, both e_1 and e_2 must agree on annotated exception type C so that raising an exception in either computation would redirect control flow the same way.

T-Caselist The context Γ_1, Γ_2 and potential $q_1 + q_2$ are both split. Γ_1 and q_1 are used to type v as a list $\mathsf{L}(\langle \tau, p \rangle)$. The nil branch e_1 is typed using Γ_2 and q_2. The cons branch e_2 also uses Γ_2 and q_2, but augmented with bound variables x for the head, typed τ, and y for the tail, typed the same as the original list, as well as p additional potential made available from the head. As in T-Casesum, the branches must agree on their final $B \odot C$.

T-Raise The potential $q + r$ is split. The context Γ along with potential q are used to type v, the exception payload, as some σ. The raise expression then has an annotated exception type of $\langle \sigma, r \rangle$. Its annotated result type A is arbitrary since it will never evaluate normally.

T-Try The context Γ_1, Γ_2 are split. $\Gamma_1; q$ types the client expression e_1. The annotated exception type $\langle \sigma, r \rangle$ of this typing is used when typing the "handler" expression e_2, which is typed using Γ_2 augmented with bound variable $x : \sigma$ and potential r. Importantly, the annotated result type A of both the main and handler expressions must agree. Furthermore, note that the handler expression itself may raise an exception too, and that its annotated exception type C is the overall exception type.

Observe the symmetry between T-Let and T-Try. In T-Let, normal return composes between e_1 and e_2, while exceptional flow $\odot C$ falls through between them. In T-Try, normal return falls through e_1 and e_2, while exceptional flow composes between. Furthermore, as alluded to in the overview section 2, there is no globally enforced annotated exception type. Instead, only local contracts are enforced between points where an exception is raised and its nearest enclosing handler (or function). This is crucial in improving the efficiency of the resource analysis; the same exception handled at two different places need not carry the same amount of potential.

3.5 Structural Typing Rules for Computations

Thus far, all the typing rules presented are syntax-directed and strictly linear. Such a type system would mostly behave as we would like, with two main issues to address. The obvious issue is that a fully linear system would require that all execution paths of a program consume the same amount of resources, no matter what branches are taken (in caselist, casesum, and try). This is in conflict with the way we decorate programs with tick when evaluating metrics such as number of function calls, which typically varies across different execution paths. Thus we make our type system *affine* instead of linear by adding the T-WeakVar and T-WeakPot rules, allowing variables and potential to be discarded at any point.

The subtler issue is that AARA functions are meant to be *structural.* Recall that in our discussion of the T-Fun rule, the premise $\Gamma = |\Gamma|$ is only necessary because we expect the function to be called multiple times, so it must not capture potential from its closure. We add the typing rule

T-ContFun to enable structural functions to be copied arbitrarily many times. This is also necessary for allowing a recursive function to make multiple recursive calls.

A notable and deliberate difference between our presentation of the structural rules and previous presentations of AARA is the lack of an explicit sharing judgment. In previous presentations of AARA, the sharing judgment $\tau \curlyvee (\tau_1, \tau_2)$ relates three annotated types which are structurally identical, differing only at their potential annotations. The potential carried by a value v of annotated type τ is the sum of the potentials carried by the same value v using annotations τ_1 and τ_2 instead. This judgment is then used to govern a contraction rule that preserves potential:

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

We argue that T-Cont is unnecessary in our type system. A well-known fact about linear type systems in general is that the contraction of purely positive types is admissible, i.e. types like $\mathsf{unit} + \mathsf{unit} \times \mathsf{unit}$ can be copied without modifying the typing rules. A similar fact applies to our type system[4] – purely positive types admit sharing. The only contraction rule we must add is T-ContFun, which is needed to copy functions. The admissibility of sharing is formally expressed in the following lemma.

Lemma 3.3. *If τ, τ_1, τ_2 are types such that the following statement holds:*

If $\Phi(v : \tau) = q$, then $q = \Phi(v : \tau_1) + \Phi(v : \tau_2)$.

Then then we can construct $\curlyvee_{\tau,\tau_1,\tau_2} : \tau \rightarrow (\tau_1 \times \tau_2) \odot \mathsf{void}$, which encodes sharing.

The lemma can be proven by induction on the type τ. The void exception type ensures that $\curlyvee_{\tau,\tau_1,\tau_2}$ does not "cheat" by raising an exception to achieve the desired result type – as there are no closed values of type void, it becomes impossible to raise an exception. Compiler passes can be easily implemented to translate a fully structural (contraction and weakening for all types) frontend language into our language, appropriately inserting calls to $\curlyvee_{\tau,\tau_1,\tau_2}$, T-WeakVar, and T-WeakPot. Together, this allows values to be be reused or discarded while potential is managed affinely.

3.6 Typing Rules for Computational Effects

It turns out that the typing rules for exceptions can be naturally extended to support effects and their handlers instead. We start by removing raise and try from the computation grammar, instead replacing them with the following:

$$e ::= \ldots \mid \mathsf{do}[\ell](v) \mid \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$$

The syntactic form $\mathsf{do}[\ell](v)$ performs effect ℓ with payload v. The syntactic form $\mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$ is an effect handler that handles any effect ℓ in effect signature Δ performed during the execution of computation e. Effect ℓ is handled by branch e_ℓ, which may use payload x as well as continuation c to resume the computation of e from the site where the effect was performed. If e evaluates to a pure value, this value is substituted for y in the return branch e_r. Formally, effect signatures Δ have the following grammar:

$$\Delta ::= \cdot \mid \Delta, \ell : A \Rightarrow B$$

Each effect ℓ has a type specification $A \Rightarrow B$, indicating that it expects a payload of annotated type A, and if resumed via the continuation available to the handler, it will be replaced by a value of annotated type B. For example, $\mathsf{inc} : \mathsf{int} \Rightarrow \mathsf{int}$ could indicate an effect that increments some counter by the provided payload, and if handled, return the updated value of the counter.

Furthermore, our typing judgments needs to be modified to accommodate effects instead of exceptions. The value typing judgment is unchanged, but our new judgment for computation typing

[4]This is in fact true of the previous AARA type systems too.

$\boxed{\Gamma; q \vdash e : B \odot \Delta}$ computation typing

$$\frac{\Gamma_1; p \vdash v_1 : \langle \tau, q_1 \rangle \multimap B \odot \Delta \qquad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; p + q_1 + q_2 \vdash \mathsf{linapp}(v_1; v_2) : B \odot \Delta}\ \textsc{T-Linapp}$$

$$\frac{\Gamma; p \vdash v : \tau_1 \qquad \ell : \langle \tau_1, q_1 \rangle \Rightarrow \langle \tau_2, q_2 \rangle \in \Delta}{\Gamma; p + q_1 \vdash \mathsf{do}[\ell](v) : \langle \tau_2, q_2 \rangle \odot \Delta}\ \textsc{T-Do}$$

$$\frac{\begin{array}{c}\Gamma_1; p \vdash e : \langle \tau, q \rangle \odot \Delta \qquad \Gamma_3, y : \tau; q \vdash e_r : C \odot \Delta' \\ \Gamma_2 = |\Gamma_2| \qquad \forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow B \in \Delta.\ (\Gamma_2, x : \tau_1, c : B \multimap C \odot \Delta'; q_1 \vdash e_\ell : C \odot \Delta')\end{array}}{\Gamma_1, \Gamma_2, \Gamma_3; p \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : C \odot \Delta'}\ \textsc{T-Handle}$$

$\boxed{\Gamma; q \vdash v : \tau}$ value typing

$$\frac{\Gamma, x : \tau; p + q \vdash e : B \odot \Delta}{\Gamma; p \vdash \mathsf{linfun}(x.e) : \langle \tau, q \rangle \multimap B \odot \Delta}\ \textsc{T-LinFun}$$

Fig. 8. New typing rules for effect handlers and linear functions

is $\boxed{\Gamma; q \vdash e : B \odot \Delta}$, where effect signature Δ contains all the effects that computation e can perform. Fortunately, the previous typing rules that do not concern effects can all be retained by α-renaming all C's to Δ's then reinterpreting all the annotated exception types as effect signatures instead. This change similarly impacts the typing of functions. Whereas the elements of the function set $\mathcal{T}$ had elements of the form $A \to B \odot C$, the elements now have the form $A \to B \odot \Delta$.

Linear Functions. To keep resource analysis tractable, we enforce that the continuations c in the effect handler $\mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$ are all one-shot by typing continuations as linear functions. This necessitates adding linear functions to the language:

$$\tau ::= \ldots \mid A \multimap B \odot \Delta \qquad v ::= \ldots \mid \mathsf{linfun}(x.e) \qquad e ::= \ldots \mid \mathsf{linapp}(v_1; v_2)$$

With the addition of linear function types, we extend the potential zeroing function $|\cdot|$ (defined in Figure 4). Interestingly, the right move is to make $|\cdot|$ partial, and reject linear function types as an input. This errs on the side of caution, ensuring that $A \multimap B \odot \Delta = |A \multimap B \odot \Delta|$ never holds, meaning linear functions can never get captured in the closures of structural functions, where they would be at risk of being invoked multiple times (nonlinearly).

Additionally, whereas our structural function types are resource polymorphic sets, we define linear function types to always be a single resource monomorphic type. This restriction simplifies type inference and suffices for most use cases, including all programs in our evaluation.

New Typing Rules. The new typing rules for linear functions and computational effects are presented in figure 8. We describe the key rules:

T-LinFun The linear function abstraction allows us to define functions that can invoke one-shot continuations available to effect handlers, as well as other linear functions, which is forbidden to structural functions $\mathsf{fun}(f.x.e)$. This is reflected in its typing rule, where the lack of constraints on Γ and p indicate that it may capture potential from its closure.

T-Do Much like T-Raise, the potential $p + q_1$ is split. The context Γ along with potential p are used to type v, the effect payload, as τ_1. $\langle \tau_1, q_1 \rangle$ is the annotated input type of this effect ℓ as specified in in the effect signature. The crucial difference is that computation can resume if an enclosing effect handler calls its continuation. Thus instead of having an arbitrary result type, it has result type $\langle \tau_2, q_2 \rangle$, which is the annotated output type of effect ℓ as specified in the effect signature.

T-Handle Context $\Gamma_1, \Gamma_2, \Gamma_3$ is split. Γ_1 and p are used to type e, the effectful computation being handled. It has an effect signature Δ, which matches the branches of the handler. As for the handler branches, there are several crucial differences from the T-Try rule.

First, Γ_2, which is used to type each effect handler branch, has the constraint that $\Gamma_2 = |\Gamma_2|$. This is because an effect handler branch can be executed an arbitrary number of times, just like the function body of a structural function, so we must forbid it from capturing potential or linear functions in Γ. T-Try lacked this restriction because exception handlers are executed at most once. Next, in addition to the effect's inputs payload $x : \tau_1$ and potential q_1, as specified by effect signature Δ, our handler branch is also able to use continuation c. This continuation has a linear function type to enforce the one-shot discipline. Its argument type matches the output type of effect ℓ, since the eventual output of a do is precisely the argument which we apply the continuation to. The continuation's result type C and its effect signature Δ' agree with the final result type, since the continuation reinstalls the effect handler. Given this context, each effect handler branch e_ℓ must have type C and also agree on an effect signature Δ'.
Finally, the return branch e_r is typed using Γ_3 along with the pure result of e bound to y. Its result type also agrees with the final result type C and effect signature Δ'.

Explicit Forwarding. Note that our typing rule for effect handlers, T-Handle, enforces explicit effect forwarding [68] by requiring that a handler handle all effects in Δ, the effect signature of the computation e being handled. The effects that are not morally handled still must be explicitly handled, but their handler branches e_l would forward the effect and thus expose it within Δ'. The result of enforcing this paradigm is that whenever an effect is performed, it always gets handled by the nearest enclosing handler (if one exists).

Concretely, a handler branch for an effect ℓ_f that is explicitly forwarded could be implemented as

$$\{\ell_f : x.c.\mathsf{let}(\mathsf{do}[\ell_f](x); y.\mathsf{linapp}(c; y))\}$$

The handler branch forwards the effect ℓ_f, then also forwards the result of the effect back to the computation being handled by applying the delimited continuation c to the effect result y.

Subsuming Exceptions. We certainly hope to be able to implement exceptions using effects. Well, this can be achieved by defining an exception effect whose input type is the exception type C and output type is void. Exception handlers by design cannot resume execution from where the exception was raised. Reusing the trick we used for lemma 3.3, since there are no closed values of type void, setting it to be the output type of the exception effect enforces this restriction, as it becomes impossible for the effect handler branch to construct an argument to call its continuation with. Formally, here is the translation of exception types and the raise and try computations into our new effect handler setting, where ℓ_e is the exception effect label:

$$C \rightsquigarrow \{\ell_e : C \Rightarrow \langle \mathsf{void}, 0\rangle\}$$

$$\mathsf{raise}(v) \rightsquigarrow \mathsf{let}(\mathsf{do}[\ell_e](v); x.\mathsf{casevoid}(x))$$

$$\mathsf{try}(e_1; x.e_2) \rightsquigarrow \mathsf{handler}(e_1; \{\ell_e : x.\_.e_2\}_{\ell_e \in \{\ell_e\}}; y.y)$$

Also, our translation of try into handler does not perfectly conserve potential, as try allows the handler to use linear variables from the context, while handler restricts the handler branches to structural variables. This restriction exists because our effect handlers are designed to handle resumable effects that can be handled multiple times, whereas exception handlers are only ever run once. To address this, we could add special cases for effect handler branches that never calls their continuations, but we omit that for brevity.

3.7 Type Inference

The new rules for exceptions and exceptions handlers are designed so that the existing type inference algorithm for AARA directly applies to our type system. The main idea of the type inference is to reduce the problem of finding resource annotations to off-the-shelf LP solving.

In this paper, we present the rules in a declarative way and to perform type inferences we have to first reformulate the rules in a more algorithmic way. This includes integrating the structural rules into the syntax directed ones and fixing finite representation for the sets of function types. The latter can be achieved by representing the types with a set of linear constraints. The new rules do not present additional challenges for designing the algorithmic type rules.

Using the algorithmic type rules, type inference for AARA consists of the following steps.

(1) Perform Hindley–Milner type inference for the structural types.
(2) Annotate the type derivations with yet unknown variables for the resource annotations.
(3) Generate linear constraints based on the algorithmic type rules.
(4) Find a solution for the constraints with an LP solver; minimizing the initial potential.

This strategy works for both functions and closed expressions. For functions, the initial potential is the potential of the argument. More details about the type inference are in the literature [55, 58, 60].

4 Soundness via K-Machine

This section contains the novel soundness proof that establishes the correctness of the derived bounds via progress and preservation using the K machine, a stack-based abstract machine.

4.1 K-Machine Cost Semantics

To define the cost semantics of our language, we introduce an abstract machine called the K-machine [49]. The K-machine includes an explicit control stack that records the work that remains to be done after a step is taken during the evaluation of a program.

A stack k is a list of stack frames, each of which is either a binder $x.e$ that sequences computation, or an effect handler awaiting the result of its computation $\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell\in\Delta}; y.e_r)$. Having only two kinds of frames is a benefit that stems from the modal separation of values and computations. Program evaluation is performed in a transition system with the following kinds of machine states:

- $k \triangleright e$. 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.
- $k \blacktriangleleft (\ell, v, k')$. This says an effect ℓ is being propagated to stack k, carrying payload v and accumulator for the delimited continuation k'.

The full grammar is given below.

Frame f	::=	$x.e$	sequence
	|	$\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell\in\Delta}; y.e_r)\Delta$	effect handler
Stack k	::=	ϵ	empty stack
	|	$k; f$	frame sequence
State s	::=	$k \triangleright e$	evaluate expression
	|	$k \triangleleft v$	return value
	|	$k \blacktriangleleft (\ell, v, k')$	propagate effect

We also introduce a new value for delimited continuations: $\mathsf{dcont}(k)$. It is a dynamic value, meaning it never appears in source programs of our language, but rather arises during the steps of our dynamics, to be defined below. It reifies a control stack k into a delimited continuation of a linear function type. When we apply it to some argument value v, we transition to $k \triangleleft v$, and return the final value this steps to.

We can now define the dynamics of our language via transitions between machine states. An initial state is one that evaluates an expression on an empty stack: $\epsilon \triangleright e; q$. A final state either returns a value to an empty stack $\epsilon \triangleleft v; q$ or propagates an unhandled effect to an empty stack

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

$$\frac{}{k \triangleright \mathsf{let}(e_1; x.e_2); q \mapsto k; x.e_2 \triangleright e_1; q}\ \textsc{D-let} \qquad \frac{}{k; x.e \triangleleft v; q \mapsto k \triangleright [v/x]e; q}\ \textsc{D-seq} \qquad \frac{p \geq q}{k \triangleright \mathsf{tick}\{q\}; p \mapsto k \triangleleft \langle\rangle\ ; p - q}\ \textsc{D-tick}$$

$$\frac{}{k \triangleright \mathsf{val}(v); q \mapsto k \triangleleft v; q}\ \textsc{D-Ret} \qquad \frac{}{k \triangleright \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q}\ \textsc{D-Cons}$$

$$\frac{}{k \triangleright \mathsf{linapp}(\mathsf{dcont}(k'); v); q \mapsto k \mathbin{@} k' \triangleleft v; q}\ \textsc{D-Dcont} \qquad \frac{}{k; x.e \blacktriangleleft (\ell, v, k'); q \mapsto k \blacktriangleleft (\ell, v, x.e \mathbin{@} k'); q}\ \textsc{D-Capture}$$

$$\frac{}{k \triangleright \mathsf{do}[\ell](v); q \mapsto k \blacktriangleleft (\ell, v, \epsilon); q}\ \textsc{D-Do} \qquad \frac{}{k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft v; q \mapsto k \triangleright [v/y]e_r; q}\ \textsc{D-normal}$$

$$\frac{\ell' \in \Delta \qquad d = \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k')}{k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)\Delta \blacktriangleleft (\ell', v, k'); q \mapsto k \triangleright [v, d/x, c]e_{\ell'}; q}\ \textsc{D-handle}$$

Fig. 9. Select State Transitions

$\epsilon \blacktriangleleft (\ell, v, k); q$. Figure 9 contains select rules (see Appendix B for complete set) for the transition judgment $\boxed{s; q \mapsto s'; q'}$, which states that machine state s given $q \in \mathbb{Q}_{\geq 0}$ resources transitions to machine state s' with $q' \in \mathbb{Q}_{\geq 0}$ resources remaining. It makes use of the @ operator that appends two stacks, which is defined inductively over the structure of the second stack as follows:

$$k \mathbin{@} \epsilon := k \qquad\qquad k \mathbin{@} k'; f := (k \mathbin{@} k'); f$$

The rules are mostly standard and the resource component q of the machines state is not modified except in the rule D-tick. We discuss that rule, as well as other key rules effects.

D-Tick requires that there are enough resources p supplied to cover the q resources in $\mathsf{tick}\{q\}$. In the resulting state, there are then $p - q$ resources.

D-Do performs an effect and is similar to the previously presented D-raise for exceptions. The key difference is that rather than just a payload value v, it also returns the effect label ℓ and an empty delimited continuation $\cdot$ that will capture more frames as the propagation proceeds.

D-Capture shows the propagate effect state $k \blacktriangleleft (\ell, v, k')$ captures sequence frames and accumulates them into k' as the effect propagates down the stack.

D-Handle handles an effect being propagated. Our type system ensures that the effect's label ℓ will be in the signature Δ of the handler frame. Handling the effect means we switch from propagating an effect to evaluating the handler body $e_{\ell'}$. During this, the effect payload v is substituted for x, and more importantly, a delimited continuation value is substituted for c. *Crucially, this delimited continuation is k', which consists of all the stack frames starting at and including the handler frame itself, all the way to the point where the effect was performed.* This "reinstallation" makes it clear that we are working with deep handlers.

D-Dcont states that when a delimited continuation $\mathsf{dcont}(k')$ is applied to a value v during linear function application, we simply append the stack within, k', to our current stack k, and return v to it. Ultimately, when the k' part of the stack finishes running, whatever value it returns will just be returned to the original stack k to proceed on.

$\boxed{A_1 \odot \Delta_1 \lhd\!: k \lhd\!: A_2 \odot \Delta_2}$ stack typing

$$\frac{}{B \odot \Delta \lhd\!: \epsilon \lhd\!: B \odot \Delta}\ \textsc{K-Emp} \qquad \frac{A \odot \Delta_1 \lhd\!: k \lhd\!: B \odot \Delta_2 \qquad x : \tau; q \vdash e : B \odot \Delta_2}{A \odot \Delta_1 \lhd\!: k; x.e \lhd\!: \langle \tau, q \rangle \odot \Delta_2}\ \textsc{K-Bnd}$$

$$\frac{\begin{array}{c} A \odot \Delta_1 \lhd\!: k \lhd\!: B \odot \Delta_3 \qquad y : \tau; q \vdash e_r : B \odot \Delta_3 \\ \forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow C \in \Delta_2.\ x : \tau_1, c : C \multimap B \odot \Delta_3; q_1 \vdash e_\ell : B \odot \Delta_3 \end{array}}{A \odot \Delta_1 \lhd\!: k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta_2}; y.e_r) \lhd\!: \langle \tau, q \rangle \odot \Delta_2}\ \textsc{K-Handler}$$

Fig. 10. Type rules for Stacks

4.2 Type Soundness

We now present the type soundness for our language via progress and preservation theorems regarding an abstract stack machine. This proof technique is novel and one of the main contributions of this paper. We start with several lemmas and additional judgments.

Substitution Lemma. A key ingredient to the type soundness is the following substitution lemma. It states that if we substitute a value v_1 for a variable $x_1 : \tau_1$ in a well-typed value (or computation) then we can type the resulting value (or computation) but have to account for the potential $q_1 = \Phi(v_1 : \tau_1)$ of the value.

Lemma 4.1 (Substitution). *If* $\cdot; q_1 \vdash v_1 : \tau_1$*:*

(1) If $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$*, then* $\Gamma; q_1 + q_2 \vdash [v_1/x_1]v : \tau$*.*
(2) If $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \odot \Delta$*, then* $\Gamma; q_1 + q_2 \vdash [v_1/x_1]e : B \odot \Delta$*.*

The lemma is proved by induction on the judgments $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$ and $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \odot \Delta$ respectively. The proof can be found in the Appendix C.

Type Judgments for Stacks and States. Figure 10 contains the rules for the stack typing judgment $\boxed{A_1 \odot \Delta_1 \lhd\!: k \lhd\!: A_2 \odot \Delta_2}$, which states that stack k is well-formed when it accepts either a value of annotated type A_2, or an effect in signature Δ_2, and returns either a value of annotated type A_1, or an effect in signature Δ_1. The stack typing judgment finally allows us to define a typing rule for $\mathsf{dcont}(k)$:

$$\frac{A_1 \odot \Delta_1 \lhd\!: k \lhd\!: A_2 \odot \Delta_2}{\Gamma; 0 \vdash \mathsf{dcont}(k) : A_2 \multimap A_1 \odot \Delta_1}\ \textsc{T-Dcont}$$

That is, a delimited continuation encoded by stack k is a linear function that takes type A_2 that k takes, an returns type A_1 with effect signature Δ_1 that k outputs.

Figure 11 contains the rules for judgment judgment $\boxed{q \vdash s}$, which specify that execution state s is well-formed given q potential.

The stack typing judgment allows us to state the following lemma, which says that two stacks can only be appended if the output type and effect signature of the second stack agrees with the input type and effect signature of the first stack. Following the definition of @ , the lemma can be proved by induction over the struture of the first stack.

Lemma 4.2 (Stack Append Typing). $B_2 \odot \Delta_2 \lhd\!: k_1 \mathbin{@} k_2 \lhd\!: B_1 \odot \Delta_1$ *if and only if there exists* B, Δ *such that* $B_2 \odot \Delta_2 \lhd\!: k_1 \lhd\!: B \odot \Delta$ *and* $B \odot \Delta \lhd\!: k_2 \lhd\!: B_1 \odot \Delta_1$

Type Soundness. With the well-formedness of execution states defined, we are finally able to state the theorems of soundness: preservation and progress.

$\boxed{q \vdash s}$ state typing

$$\frac{\cdot; q \vdash e : B \odot \Delta \qquad \_ \triangleleft\!: k \triangleleft\!: B \odot \Delta}{q \vdash k \triangleright e}\ \text{ST-Exp} \qquad \frac{\cdot; q_1 \vdash v : \tau \qquad \_ \triangleleft\!: k \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta}{q_1 + q_2 \vdash k \triangleleft v}\ \text{ST-Val}$$

$$\frac{\cdot; q_1 \vdash v : \tau \qquad \ell : \langle \tau, p \rangle \Rightarrow C \in \Delta' \qquad \_ \triangleleft\!: k \triangleleft\!: B \odot \Delta' \qquad B \odot \Delta' \triangleleft\!: k' \triangleleft\!: C \odot \Delta''}{q_1 + p \vdash k \blacktriangleleft (\ell, v, k')}\ \text{ST-Eff}$$

Fig. 11. Type rules for Execution States

Theorem 4.3 (Preservation). *If $q \vdash s$ and $s; q \mapsto s_0; q_0$ then $q_0 \vdash s_0$.*

Proof. Preservation is proved by case analysis on the state transition judgment $s; q \mapsto s_0; q_0$. We present one case illustrative of normal computation, and one case illustrative of effect handling. The complete proof can be found in the Appendix C.

D-Cons $k \triangleright \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q$

By the assumption, $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta$, $\cdot; q \vdash \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1) : B \odot \Delta$. It suffices to show $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \odot \Delta$. Induct on $\cdot; q \vdash \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1) : B \odot \Delta$. Only T-caselist and T-WeakPot are possible. We omit the latter.

By the premises of T-caselist, $x : \tau, y : \mathsf{L}(\langle \tau, p \rangle); q_2 + p \vdash e_0 : B \odot \Delta$, where $\cdot; q_1 \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(\langle \tau, p \rangle)$, $q = q_1 + q_2$. Invert $\cdot; q_1 \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(\langle \tau, p \rangle)$ to get $q_1 = q_1' + q_2' + p$, $\cdot; q_1' \vdash v_1 : \tau$, $\cdot; q_2' \vdash v_2 : \mathsf{L}(\langle \tau, p \rangle)$. Then apply the substitution lemma C.4, $\cdot; q_2 + p + q_1' + q_2' \vdash [v_1, v_2/x, y]e_1 : B \odot \Delta$, which is $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \odot \Delta$.

D-Handle $k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \blacktriangleleft (\ell', v, k'); q \mapsto\ k \triangleright [v, d/x, c]e_{\ell'}; q$

when $\ell \in \Delta$ and where $d = \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k')$.

By the premises of ST-Eff on the left, we have (1) $q = q_1 + p$, (2) $\cdot; q_1 \vdash v : \tau$, (3) $\ell' : \langle \tau, p \rangle \Rightarrow C \in \Delta$, (4) $\_ \triangleleft\!: k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft\!: B' \odot \Delta$, and (5) $B' \odot \Delta \triangleleft\!: k' \triangleleft\!: C \odot \Delta''$. Invert (4) using K-Handler to get (6) $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta_3$, (7) for all $\ell : \langle \tau_1, q_1 \rangle \Rightarrow C$ in Δ, $x : \tau_1, c : C \multimap B \odot \Delta_3; q_1 \vdash e_\ell : B \odot \Delta_3$, and (8) $y : \tau'; q' \vdash e_r : B \odot \Delta_3$ such that $B' = \langle \tau', q' \rangle$. Next, we use K-Emp, K-Handler, (7), (8), and lemma C.6 to construct the stack typing $B \odot \Delta_3 \triangleleft\!: \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k' \triangleleft\!: C \odot \Delta''$. Then by T-Dcont, we have (9) $\cdot; 0 \vdash d : C \multimap B \odot \Delta_3$.

Now, we can apply the substitution lemma for both x and c in e_ℓ. For $[v/x]$, we have (2). For $[\mathsf{dcont}(\ldots)/c]$, we have (9). For e_ℓ, we have (7). Putting it all together, we have

$$\cdot; p + q_1 + 0 \vdash [v, d/x, c]e' : B \odot \Delta_3$$

Finally, using $q = q_1 + p$ from (1), along with $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta_3$ from (6), we can apply ST-Exp to type the overall right hand side of the $\mapsto$. □

Theorem 4.4 (Progress). *If $q \vdash s$, then either s final or $s; q \mapsto s'; q'$.*

Proof. Progress is proved by case analysis on the state typing judgment $q \vdash s$. We present the case for the propagate effect state. The complete proof can be found in the Appendix C.

ST-Eff $q \vdash k \blacktriangleleft (\ell', v, k')$

Induct on structure of k.

- $k = \epsilon$: $k \blacktriangleleft (\ell', v, k')$ is final
- $k = k_1; x.e$: By D-Capture, $k_1; x.e \blacktriangleleft (\ell', v, k'); q \mapsto k_1 \blacktriangleleft (\ell', v, x.e \mathbin{@} k'); q$
- $k = k_1; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$: Our type system guarantees that $\ell' \in \Delta$. Then by D-handle, $k_1; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \blacktriangleleft (\ell', v, k'); q \mapsto k_1 \triangleright [v, d/x, c]e_{\ell'}; q$ where $d = \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k')$ □

5 Evaluation

In this section, we first describe our implementation of AARA with exceptions and effects. We then evaluate the accuracy and performance of the implementation with 21 representative benchmarks, while comparing it against previous implementations of AARA. Both the source code of our implementation and the 21 benchmarks are available in the associated artifact [25].

5.1 Implementation

Our implementation of AARA with exceptions and effects is built upon a novel framework that is modeled after RaML [53] and implemented in OCaml. We refer to it as RaML+ in this paper. RaML+ has an SML frontend and makes some command-line calls to the MLton SML compiler [105]. Therefore, all benchmarks and code examples in this paper are implemented in SML. As mentioned in Section 3, the analysis implemented in RaML+ is more complete than the formal development in this paper and supports many SML features, including products, sums, and recursive types. SML features like polymorphism, nested pattern matching, and modules are not supported by our type theory or RaML+; instead, we rely on the MLton compiler to do away with them, such as MLton's monomorphisation and pattern flattening passes.

The implementation of the new typing rules for exceptions and effects in RaML+ follows from the algorithmic versions (see Section 3.7) of the typing rules in Section 3. These typing rules generate linear constraints between potential annotations, thus reducing type inference to a linear optimization problem for the whole program. This optimization problem is then solved by the Coin-Or LP solver [99].

Exceptions Frontend. SML has native support for raising exceptions via **raise** ExName and handling them via e **handle** e'. We directly use these syntactic forms when writing programs for RaML+. Then to add exception support to RaML+, we simply translate SML exceptions and handlers into their corresponding AST variants in the IR.

Since MLton performs whole program compilation, we combine all exceptions into one large sum type, which serves as the payload of all exceptions. For example, our running example from Figure 1, which declares two list length mismatch exceptions, will use the **datatype** exn = Emis1 **of** unit | Emis2 **of** unit for execptions. As discussed in Section 3.6, another option is to translated exceptions into effects. However, we retain both syntactic forms due to the restriction on effect handler branches from using linear variables.

Effects frontend. Adding effect support was more challenging because SML does not have native support for effects and their handlers. Users define the effects they wish to perform in a datatype. They can then perform effects using an SML function we supply called e_do, and handle effects using e_handle. These functions are translated into explicit do and handler AST variants in the IR.

Cost Metric. As stated in the overview, the cost metric we chose in this paper is the number of function calls and handler evaluations. To streamline the evaluation, we added the command line flags -capps and -chandles to RaML+, which respectively insert (R.tick 1) next to function calls and handlers before running the main analysis pass. This enables us to omit R.tick expressions from our benchmark programs and minimize inconsistencies in how we evaluate benchmarks.

5.2 Goals and Methodology

The goal of our evaluation is to determine the efficiency of the new analysis, the quality of the derived bounds, as well as the overhead with respect to RaML, an AARA implementation that supports raising exceptions but not handling them.

To this end, we have constructed 21 benchmark programs which employ the raising and handling of exceptions and effects to induce nontrivial (non-local) control flow. Given these benchmarks, we

	RaML		RaML+ with Exceptions		
Function	Derived Bound	Time (s)	Derived Bound	Tight?	Time (s)
sqdist(v1 : L($\mathbb{R}$), v2 : L($\mathbb{R}$)) : $\mathbb{R}$	|v2|	0.05	|v1|	✓	0.007
distances_1(vs : L(L($\mathbb{R}$)), p : L($\mathbb{R}$)) : L($\mathbb{R}$)	$1 + 3\|vs\| + \sum_i \|v_i\|$	0.07	$1 + 3\|vs\| + \sum_i \|v_i\|$	✓	0.02
distances_2(vs : L(L($\mathbb{R}$)), p : L($\mathbb{R}$)) : L($\mathbb{R}$)	Unsolvable	–	$1 + 3\|vs\| + \sum_i \|v_i\|$	✓	0.03
nearest(vs : L(L($\mathbb{R}$)), p : L($\mathbb{R}$)) : ($\mathbb{R}$, L(L($\mathbb{R}$)))	Unsolvable	–	$1 + 3\|vs\| + \sum_i \|v_i\|$	✓	0.12
zip(l1 : L(a), l2 : L(b)) : L((a, b))	Unsolvable	–	$1 + 2\|l2\|$	✓	0.01
map2(f : (a, b) → c, l1 : L(a), l2 : L(b)) : c	Unsolvable	–	$1 + 4\|l2\|$	✓	0.01
eval_exp(e : exp) : $\mathbb{Z}$	Unsolvable	–	$1 + 2c + 2p + t$	✓	0.13
sort(l : L($\mathbb{Z}$)) : L($\mathbb{Z}$)	Unsolvable	–	$\|l\| + \binom{\|l\|}{2}$	✓[5]	0.07
zip'(l1 : L(a), l2 : L(b)) : L((a, b))	|l1|	0.03	|l1|	✓	0.01
map2'(f : (a, b) → c, l1 : L(a), l2 : L(b)) : c	2|l1|	0.03	2|l1|	✓	0.01
eval_exp'(e : exp) : $\mathbb{Z}$	$2p + 2t$	0.04	$c + p + t$	✓	0.04
sort'(l : L($\mathbb{Z}$)) : L($\mathbb{Z}$)	$\binom{\|l\|}{2}$	0.09	$\binom{\|l\|}{2}$	✓	0.06

Table 1. RaML+ with Exceptions Evaluation Results

automatically derive bounds using RaML+ with exceptions/effects and compare them to manually derived bounds to determine their tightness. For all but two benchmarks, the cost metric chosen measures the number of function calls and handler evaluations. Additionally, we also test our implementation on 6 examples without exception handlers to determine whether RaML+ has a significant performance overhead when analyzing the same programs as RaML. Since the extension of AARA to exceptions is conservative, we do not expect to derive bounds that differ from the RaML bounds for these benchmarks (and the experiments confirm this).

For each benchmark, we manually analyze the benchmarks by hand to determine whether the automatically derived bounds by RaML+ are tight. Furthermore, for the exception benchmarks, we also measure the time it takes for RaML+ to perform its analysis. We run 10 trials for each benchmark and report the average elapsed time across them. These experiments were run on a machine with an AMD Ryzen 9 7940HS processor and 64 GB of RAM.

5.3 Exception Benchmarks

Table 1 contains the results of the evaluation of exception benchmarks. The first group of functions we evaluate in are the running examples covered in the overview Section 2.2. The bounds for distances_1 and distances_2 differ from the bounds derived in the overview because we now count all function calls instead of calls to sqdist only. In the second group, we evaluate additional functions that contain both exceptions and handlers:

nearest The function nearest builds on the examples in Section 2.2 and returns the shortest distance between any vector in vs and vector p, as well as the subset of vs satisfying this distance. An interesting aspect of nearest is that it leverages exceptions with payloads to improve efficiency when the shortest distance is 0.

zip The zip function combines two lists. However, if the two lists are not the same length, it throws an exception UnequalLengths of ('a * 'b) list whose payload is the partialliy zipped result. The function needs to repeatedly handle and re-raise the exception with a larger payload.

map2 Similarly, the function map2 applies a function that takes two inputs across two lists, once again raising the same exception if they are not the same length.

eval_exp This function is an evaluator for a simple expression datatype.It uses exceptions and handlers to shortcut multiplication whenever there is a 0.

sort The function sort does a linear pass over the input integer list to check that it is sorted. If it is not, an exception is thrown, and the surrounding handler calls quicksort to sort the list.

[5]This bound is tight because we use quicksort, which is worst case quadratic time. RaML+ cannot derive logarithmic bounds.

	RaML+ with Effects	
Function	Derived Bound	Tight?
simple_IO(() : unit) : unit	11	✓
store_lists(vs : L(L($\mathbb{Z}$))) : unit	$6 + 9\|vs\| + \sum_i \|v_i\|$	✓
store_lists_q_naive(vs : L(L($\mathbb{Z}$))) : unit	Unsolvable	–
store_lists_q(vs : L(L($\mathbb{Z}$))) : unit	$8 + 13\|vs\| + \sum_i \|v_i\|$	✓
store_suffix_lists(vs : L($\mathbb{Z}$)) : unit	$14 + 10\|vs\| + \binom{\|vs\|}{2}$	✓
generator_to_list(n : nat) : L(nat)	$1 + 3\|n\|$	✓
generator_to_list_2(n : nat) : L(nat)	$1 + 5\|n\| + \binom{\|n\|}{2}$	✓
generator_counter(n : nat) : unit	$3 + 11\|n\| + \binom{\|n\|}{2}$	✓
prefix_sum_list(ns : L(nat)) : L(nat)	$2 + 12\|ns\| + \sum_i \|ns_i\|$	✓

Table 2. RaML+ with Effects Evaluation Results

Finally, the last group of benchmarks are just functions from the previous group with their exception handlers removed. For example, eval_exp’ no longer uses exceptions to shortcut multiplication with 0, and sort’ calls quicksort without first performing the sortedness check in sort.

5.4 Effect Benchmarks

The benchmarks for effect handlers are listed in Table 2. Since we are only evaluating RaML+ with Effects and not the original RaML (which cannot analyze programs with effects), we did not measure the analysis times for this benchmarks, as there would be nothing to compare against. The first benchmark, simple_IO, performs 3 IO effects. The second group contains functions that use the same Insert and Remove effects as our running example in the overview in Figure 3:

store_lists_q_naive is store_lists (the overview example) with a modification to the handler. While the effect handler in store_lists stores and removes elements as if it were a stack, store_lists_q_naive implements a queue instead. As naive in the name suggests, this queue is naively implemented as a single list that requires either Insert or Remove to traverse it, incurring cost proportional to the size of the store. This cost is impossible to capture in the effect signature, thus this example is intentionally impossible to analyze.

store_lists_q is an efficient implementation of store_lists_q_naive. It uses the standard functional queue implementation of two lists, which has amortized constant cost for both Insert and Remove operations, which is now possible to type.

store_suffix_lists is a twist on the original store_lists. Its argument vs is no longer a list of lists of integers, but rather just a list of integers. We store all the suffixes of this list into the store. We then remove all of them from the store and traverse each one, just as in store_lists.

Finally, we evaluate four benchmarks that use the Yield : nat $\Rightarrow$ unit effect that comes from generators in the literature. They all rely on datatype nat = Zero | Succ of nat. This representation allows us to store potential in a nat proportional to the number it represents.

generator_to_list takes a natural number *n*, creates a generator that performs Yield on every natural number from *n* to 0, then uses a handler to collect the yielded numbers into a list.

generator_to_list_2 is similar to generator_to_list but calls traverse (but for nats) on each element of the output list. For reasons similar to store_suffix_lists, it also requires quadratic potential.

generator_counter creates the same generator as the previous two benchmarks, then uses the Get and Put effects from state to sum up the yielded numbers. It shows that RaML+ with Effects can analyze code that composes effect handlers.

prefix_sum_list relies on a prefix_sum_generator, which uses the state effects Get and Put to keep track of the prefix sum of a list, and also performs the Yield effect at each prefix. On the other end, prefix_sum_list collects the yielded numbers back into a list. It shows that RaML+ with

Effects can analyze code that performs multiple effects together (the state and generator effects), as well as the explicit forwarding of the Yield effect to the outer handler.

5.5 Results

Our evaluation results are summarized in and Table 1 and Table 2. It contains, for each benchmark, the function name and its type, the bound derived by RaML+, and its tightness. For the exception benchmarks, we also provide the bounds derived by RaML, as well as analysis times for both RaML and RaML+ (in the *Time (s)* columns). As indicated by the checkmarks in the *Tight?* columns, all derived bounds are not only asymptotically tight but have optimal constant factors too.

There are a few trivial differences between the bounds derived by RaML and RaML+. For sqdist, RaML derives |v2| while RaML+ derives |v1|. These bounds differ, but as explained in section 2.1 of the overview, they are both valid and tight upper bounds of the true cost min(|v1|, |v2|) that our type system cannot directly encode. In the bounds for the functions eval_exp and eval_exp', c, p, and t respectively denote the number of Const, Plus, and Times nodes. For eval_exp', RaML derives $2p + 2t$ while RaML+ derives $c + p + t$. While these bounds appear quite different at a glance, they are nearly equal, since the number of leaves, c, in a binary tree, is always equal to the number of interior nodes, $p + t$, plus one. We are not entirely sure why these two tools report different bounds.

Our findings are that RaML+ with exceptions and effects performs better than RaML on our benchmarks. It can analyze a larger set of user functions that raise and handle exceptions and effects, while RaML can only analyze functions that raise exceptions. Furthermore, by observing the time it takes to analyze functions that both implementations can analyze, we see that despite the additional complexity, our implementation does not add overhead. In fact, our implementation is always faster, although that can be attributed to RaML tracking multivariate potential functions (such as |l1||l2|), which RaML+ does not support.

6 Related Work

Automatic resource bound analysis has been approached from many different angles, but works on non-local control transfer are rare.

Recurrence Relations. A traditional approach to automatic resource bound analysis, which was first studied by Wegbreit [106], is a two-step process that consists of extracting then solving recurrence relations. It has been studied for functional programs [28, 31, 42], logic programs [36, 73], and imperative languages and object-oriented languages in the COSTA project [2–4, 7, 8] and by using techniques such as abstract interpretation and symbolic analysis [39, 70, 71].

The only system that covers exception handlers is COSTA [4, 5]. The analysis operates on Java bytecode and can derive bounds for functions that have exceptions handlers which catch exceptions raised in the same function call. The idea is to turn code that raises an exception into a jump to the handler. In contrast, this paper focuses on non-local transfer of control where raising an exception can transfer control to a different function.

Term Rewriting. Most prominently, analyses for term rewriting systems focus on termination [14, 41]. However, resource bound analysis has also been studied. Two prominent implementations are AProVE [19, 40, 88] and TCT [12, 51]. It would be possible to model exceptions with non-determinism in term rewriting, but it is not clear if this abstraction would lead to satisfactory resource bounds. Using types (like in this work) is a natural approach to establish a modular interface between handling and raising exceptions.

Other Static Analyses. Other automatic resource analyses for imperative programs are based on loop analysis and instrumentation [16, 45–47], ranking functions [23, 108], and similar techniques [95]. In contrast to this paper, these works do not handle non-local transfer of control as needed for exception handlers.

Soundness via Abstract Machine. Whereas our work is the first to prove soundness of AARA on a language with effects by examining the semantics of a stack-based abstract machine, this approach is well-established in the wider effect handler community [82, 92]. Most closely related is Voigt et al's work [102] demonstrating an effect system that can ensure that resources (like file handles) are acquired and released safely.

Automatic Amortized Resource Analysis. Most closely related to this paper is work on AARA. The technique was introduced by Hofmann and Jost [60] to infer linear heap-space bounds for first-order functional programs. In the past two decades, AARA has been extended to many programming-language features [44, 57, 63, 80], to non-linear bounds [54, 61, 66], and to more advanced type systems that forgo full automation [75, 90]. Details can be found in a survey by Jost and Hoffmann [58]. Many works on AARA consider (deterministic) functional programs with strict evaluation. However, there are also works on bounds for parallel [59, 83] and lazy [65, 94] evaluation, lower bounds [85], and bounds on expected cost of probabilistic programs [103].

With the exception of Grosen et al. [44], prior work on AARA establishes the soundness of the derived bounds with respect to a big-step cost semantics. Grosen et al. present a semantic soundness prove using a small-step semantics and a logical relation. This paper presents the first syntactic soundness proof using a small-step cost semantics. It is also the first paper that establishes soundness for a stack-based abstract machine and expressions with a modal separation of computations and values.

Potential-based reasoning has also been integrated with (concurrent) separation logic [9, 22, 81], and Hoare logic [21] to reason about imperative programs. While the implementation of AARA in Resource Aware ML [56, 57] supports exceptions, no prior work on AARA considers either exception or effect handlers, or even non-local transfer of control, which is the focus of this paper.

Other Type-Based Systems. Other type-based approaches for resource bound analysis are based on sized types [10, 101], refinement types [26, 27, 48, 74, 75, 109], indexed types [104], linear dependent types [76–78], and structural dependent types [30, 43, 86, 87]. Avanzini et al. [11] studied code transformations to simplify bound inference for higher-order functions. Sekiyama et al. [69, 93] studied analysis of programs with algebraic effects and handlers using refinement types, as well as verification of programs with delimited continuations. These works do not include analyses that automatically derive bounds for programs with exception or effect handlers.

7 Conclusion and Future Work

This paper presents an extension of type-based AARA to exceptions, effects, and their handlers. The novel type system supports non-local transfer of control, and its soundness is established using progress and preservation theorems for an abstract machine with a control stack. This is a departure from the traditional soundness proof for AARA, which uses a big-step cost semantics. We expect that this alternative formulation of soundness will find further applications.

While it is possible to model exceptions and effects in a big-step semantics, control stacks enable a simpler and more local treatment. Similarly, it is possible to simulate exceptions and effects with sum types and higher order functions. However, this would require program transformations that propagate effects and continuations in deep subexpressions, which would encumber the analysis. Supporting general (callcc style) continuations using AARA seems to require a different approach, since continuations enable a program to use frames on a control stack multiple times. Other potential avenues for future work include supporting a richer effect type system, such as with row polymorphism, or allowing multi-shot continuations.

Data Availability Statement

In section 5, evaluation, we discuss how we implemented AARA with exceptions and effects as a tool named RaML+, and evaluated it against 21 benchmark programs. We have made the source code of RaML+ and the 21 benchmarks available as an artifact [25], which also contains a Dockerfile that automatically builds RaML+ into an executable. Additionally, the artifact contains an executable of RaML [53], a prior implementation of AARA that we are comparing RaML+ against. We have submitted the artifact for evaluation.

Acknowledgments

This material is based upon work supported by Jane Street Group, LLC and the National Science Foundation under grants no. 2311983 and 2525102. Any opinions, findings, and conclusions or recommendations in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Beniamino Accattoli. 2019. A Fresh Look at the lambda-Calculus. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:20. doi:10.4230/LIPIcs.FSCD.2019.1

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *15th Static Analysis Symp. (SAS'08)*. 221–237.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* (2011), 161–203.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2007. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*.

[5] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.* 413, 1 (2012), 142 – 159.

[6] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2021. *Don't run on fumes* - Parametric gas bounds for smart contracts. *J. Syst. Softw.* 176 (2021), 110923. doi:10.1016/J.JSS.2021.110923

[7] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*.

[8] Elvira Albert, Samir Genaim, and Abu Naser Masud. 2013. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic* 14, 3 (2013).

[9] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*.

[10] Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.* 1, ICFP, Article 43 (aug 2017), 29 pages. doi:10.1145/3110287

[11] Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2012. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

[12] Martin Avanzini and Georg Moser. 2013. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*.

[13] Martin Avanzini, Georg Moser, and Michael Schaper. 2020. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 172:1–172:30. doi:10.1145/3428240

[14] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

[15] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. doi:10.1016/J.JLAMP.2014.02.001

[16] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*.

[17] Guy E. Blelloch. 1995. *NESL: A Nested Data-Parallel Language (Version 3.1).* Technical Report CMU-CS-95-170. CMU.

[18] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *1st Int. Conf. on Funct. Prog. (ICFP'96)*.

[19] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *20th Int. Conf. on Tools and Alg. for the Constr. and Anal. of Systems (TACAS'14)*.

[20] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-End Verification of Stack-Space Bounds for C Programs. In *35th Conference on Programming Language Design and Implementation (PLDI'14)*. Artifact submitted and approved.

[21] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*. Artifact submitted and approved.

[22] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. doi:10.1007/s10817-017-9431-7

[23] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017. Non-polynomial Worst-Case Analysis of Recursive Programs. In *Computer Aided Verification - 29th Int. Conf. (CAV'17)*.

[24] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Tobias Meggendorfer, and undefinedorđe Žikelić. 2024. Quantitative Bounds on Resource Usage of Probabilistic Programs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 107 (apr 2024), 30 pages. doi:10.1145/3649824

[25] Ethan Chu, Jan Hoffmann, and Yiyang Guo. 2026. *Artifact for "Handling Exceptions and Effects with Automatic Resource Analysis"*. doi:10.5281/zenodo.18742777

[26] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*.

[27] Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*. 184–198.

[28] Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.* 4, ICFP (2020), 97:1–97:29. doi:10.1145/3408979

[29] Ugo Dal Lago and Simone Martini. 2006. An Invariant Cost Model for the Lambda Calculus. In *Logical Approaches to Computational Barriers*, Arnold Beckmann, Ulrich Berger, Benedikt Löwe, and John V. Tucker (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 105–114.

[30] Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*.

[31] Norman Danner and Daniel R. Licata. 2022. Denotational semantics as a foundation for cost recurrence extraction for functional languages. *J. Funct. Program.* 32 (2022), e8. doi:10.1017/S095679682200003X

[32] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *2021 IEEE Computer Security Foundations Symposium (CSF'21)*.

[33] Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. doi:10.1145/382780.382785

[34] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434314

[35] Paulo Emílio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 225–252. doi:10.1007/978-3-031-30044-8_9

[36] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. 1994. Estimating the Computational Cost of Logic Programs. In *Static Analysis, First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 28-30, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 864)*, Baudouin Le Charlier (Ed.). Springer, 255–265. doi:10.1007/3-540-58485-4_45

[37] Inc. Facebook. 2024. Infer Website - Cost: Runtime Complexity Analysis. https://fbinfer.com/docs/checker-cost.

[38] Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 180–190. doi:10.1145/73560.73576

[39] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*.

[40] Florian Frohn, M. Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. 2016. Lower Runtime Bounds for Integer Programs. In *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*.

[41] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2004. Automated Termination Proofs with AProVE. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3091)*, Vincent van Oostrom (Ed.). Springer, 210–220. doi:10.1007/978-3-540-25979-4_15

[42] Bernd Grobauer. 2001. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*. 253–264.

[43] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 8, POPL (2024), 273–301. doi:10.1145/3632852

[44] Jessie Grosen, David Kahn, , and Jan Hoffmann. 2023. Automatic Amortized Resource Analysis with Regular Recursive Types. In *38th ACM/IEEE Symposium on Logic in Computer Science (LICS' 23)*.

[45] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*.

[46] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*.

[47] Sumit Gulwani and Florian Zuleger. 2010. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*.

[48] Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate your assets: reasoning about resource usage in liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL, Article 24 (dec 2019), 27 pages. doi:10.1145/3371092

[49] Robert Harper. 2012. *Practical Foundations for Programming Languages.* Cambridge University Press.

[50] Daniel Hillerström, Sam Lindley, and John Longley. 2024. Asymptotic speedup via effect handlers. *Journal of Functional Programming* 34 (2024), e5. doi:10.1017/S0956796824000030

[51] Nao Hirokawa and Georg Moser. 2014. Automated Complexity Analysis Based on Context-Sensitive Rewriting. In *Rewriting and Typed Lambda Calculi (RTA-TLCA'14)*.

[52] Jan Hoffmann. 2011. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis.* Ph. D. Dissertation. Ludwig-Maximilians-Universität München.

[53] Jan Hoffmann. 2024. RAML Web Site. http://raml.co/.

[54] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*.

[55] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.* (2012).

[56] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*.

[57] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[58] Jan Hoffmann and Steffen Jost. 2022. Two Decades of Automatic Amortized Resource Analysis. *Math. Struct. Comput. Sci.* (2022).

[59] Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *24th European Symposium on Programming (ESOP'15)*.

[60] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*.

[61] Martin Hofmann, Lorenz Leutgeb, David Obwaller, Georg Moser, and Florian Zuleger. 2022. Type-based analysis of logarithmic amortised complexity. *Math. Struct. Comput. Sci.* 32, 6 (2022), 794–826. doi:10.1017/S0960129521000232

[62] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 567–584. https://www.usenix.org/conference/nsdi22/presentation/iyer

[63] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*.

[64] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*.

[65] Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *Journal of Automated Reasoning* 59 (2017), 87–120.

[66] David Kahn and Jan Hoffmann. 2020. Exponential Automatic Amortized Resource Analysis. In *23rd International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'20)*.

[67] David Kahn and Jan Hoffmann. 2021. Automatic Resource Analysis with the Quantum Physicist's Method. In *26th International Conference on Functional Programming (ICFP'21)*.

[68] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. doi:10.1145/2500365.2500590

[69] Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 8, POPL, Article 5 (Jan. 2024), 33 pages. doi:10.1145/3633280

[70] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional Recurrence Analysis Revisited. In *Conference on Programming Language Design and Implementation (PLDI'17)*.

[71] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-Linear Reasoning for Invariant Synthesis. *Proc. ACM Program. Lang.* 2, POPL, Article 54 (Dec. 2017), 33 pages. doi:10.1145/3158142

[72] Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4583)*, Simona Ronchi Della Rocca (Ed.). Springer, 223–239. doi:10.1007/978-3-540-73228-0_17

[73] Maximiliano Klemen, Pedro López-García, John P. Gallagher, José F. Morales, and Manuel V. Hermenegildo. 2019. A General Framework for Static Cost Analysis of Parallel Logic Programs. In *Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12042)*, Maurizio Gabbrielli (Ed.). Springer, 19–35. doi:10.1007/978-3-030-45260-5_2

[74] Tristan Knoth, Di Wang, Jan Hoffmann, and Nadia Polikarpova. 2019. Resource-Guided Program Synthesis. In *40th Conference on Programming Language Design and Implementation (PLDI'19)*.

[75] Tristan Knoth, Di Wang, Adam Reynolds, Nadia Polikarpova, and Jan Hoffmann. 2020. Liquid Resource Types. In *25th International Conference on Functional Programming (ICFP'20)*.

[76] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*.

[77] Ugo Dal Lago and Giulio Pellitta. 2013. Complexity Analysis in Presence of Control Operators and Higher-Order Functions. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8312)*, Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer, 258–273. doi:10.1007/978-3-642-45221-5_19

[78] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*.

[79] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9

[80] Benjamin Lichtman and Jan Hoffmann. 2017. Arrays and References in Resource Aware ML. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)*.

[81] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. doi:10.1007/978-3-030-17184-1_1

[82] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 255 (Oct. 2023), 30 pages. doi:10.1145/3622831

[83] Stefan Muller and Jan Hoffmann. 2024. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *ACM Transactions on Parallel Computing* (2024).

[84] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *39th Conference on Programming Language Design and Implementation (PLDI'18)*.

[85] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*.

[86] Yue Niu and Robert Harper. 2023. A Metalanguage for Cost-Aware Denotational Semantics. In *LICS*. 1–14. doi:10.1109/LICS56636.2023.10175777

[87] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498670

[88] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning* 51, 1 (2013), 27–56.

[89] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. doi:10.1007/978-3-642-00590-9_7

[90] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis. In *48th Symposium on Principles of Programming Languages (POPL'21)*.

[91] John C. Reynolds. 1993. The Discoveries of Continuations. *LISP Symb. Comput.* 6, 3-4 (1993), 233–248.

[92] Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A typed continuation-passing translation for lexical effect handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 566–579. doi:10.1145/3519939.3523710

[93] Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (Jan. 2023), 32 pages. doi:10.1145/3571264

[94] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12).*

[95] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14).*

[96] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021).* Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

[97] Wenhao Tang, Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, and Anton Lorenzen. 2024. Modal Effect Types. arXiv:2407.11816 [cs.PL] https://arxiv.org/abs/2407.11816

[98] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[99] The CLP Team. 2024. CLP. https://projects.coin-or.org/Clp.

[100] Orpheas van Rooij and Robbert Krebbers. 2025. Affect: An Affine Type and Effect System. *Proc. ACM Program. Lang.* 9, POPL (2025), 126–154. doi:10.1145/3704841

[101] Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15).*

[102] David Voigt, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2025. Dynamic Wind for Effect Handlers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 377 (Oct. 2025), 27 pages. doi:10.1145/3763155

[103] Di Wang, David M. Kahn, and Jan Hoffmann. 2020. Raising Expectations: Automating Expected Cost Analysis with Types. In *25th International Conference on Functional Programming (ICFP'20).*

[104] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (oct 2017), 26 pages. doi:10.1145/3133903

[105] Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). ACM, 1. doi:10.1145/1159876.1159877

[106] Ben Wegbreit. 1975. Mechanical Program Analysis. *Commun. ACM* 18, 9 (1975), 528–539.

[107] Reinhard Wilhelm et al. 2008. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008).

[108] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11).*

[109] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17).*

Appendices

A Type System for Linear-Bound AARA with Effects

A.1 Syntax Grammar

$$
\begin{array}{rcl}
v & ::= & x \\
 & & \mathsf{fun}(f.x.e) \\
 & & \langle\rangle \\
 & & \mathsf{pair}(v_1; v_2) \\
 & & \mathsf{inl}(v) \\
 & & \mathsf{inr}(v) \\
 & & \mathsf{nil} \\
 & & \mathsf{cons}(v_1; v_2) \\
 & & \mathsf{linfun}(x.e) \\
 & & \mathsf{dcont}(k)
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & \mathsf{val}(v) \\
 & & \mathsf{tick}\{q\} \\
 & & \mathsf{let}(e_1; x.e_2) \\
 & & \mathsf{app}(v_1; v_2) \\
 & & \mathsf{casepair}(v; x_1.x_2.e) \\
 & & \mathsf{casevoid}(v) \\
 & & \mathsf{casesum}(v; x_1.e_1; x_2.e_2) \\
 & & \mathsf{caselist}(v; e_1; x_1.x_2.e_2) \\
 & & \mathsf{do}[\ell](v) \\
 & & \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell\in\Delta}; y.e_r) \\
 & & \mathsf{linapp}(v_1; v_2)
\end{array}
$$

A.2 Resource Annotated Types and Effect Signatures

$$\tau ::= \mathcal{T} \mid \mathsf{unit} \mid \tau_1 \times \tau_2 \mid \mathsf{void} \mid A_1 + A_2 \mid \mathsf{L}(A) \mid A \multimap B \odot \Delta \qquad A, B, C ::= \langle\tau, q\rangle$$

$$\mathcal{T} ::= \{A \to B \odot \Delta \mid \Theta\} \qquad \Delta ::= \cdot \mid \Delta, \ell : A \Rightarrow B$$

A.3 Value Typing

$\boxed{\Gamma; q \vdash v : \tau}$

$$\text{T-Var}\ \frac{}{x : \tau; 0 \vdash x : \tau} \qquad \text{T-Fun}\ \frac{\Gamma = |\Gamma| \qquad \forall(\langle\tau, q\rangle \to B \odot \Delta) \in \mathcal{T}.\ \Gamma; f : \mathcal{T}, x : \tau; q \vdash e : B \odot \Delta}{\Gamma; 0 \vdash \mathsf{fun}(f.x.e) : \mathcal{T}}$$

$$\text{T-Unit}\ \frac{}{\cdot; 0 \vdash \langle\rangle : \mathsf{unit}} \qquad \text{T-Pair}\ \frac{\Gamma_1; q_1 \vdash v_1 : \tau_1 \qquad \Gamma_2; q_2 \vdash v_2 : \tau_2}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{pair}(v_1; v_2) : \tau_1 \times \tau_2}$$

$$\text{T-Inl}\ \frac{\Gamma; p \vdash v : \tau_1}{\Gamma; q_1 + p \vdash \mathsf{inl}(v) : \langle\tau_1, q_1\rangle + A_2} \qquad \text{T-Inr}\ \frac{\Gamma; p \vdash v : \tau_2}{\Gamma; q_2 + p \vdash \mathsf{inr}(v) : A_1 + \langle\tau_2, q_2\rangle}$$

$$\text{T-Nil}\ \frac{}{\cdot; 0 \vdash \mathsf{nil} : \mathsf{L}(A)} \qquad \text{T-cons}\ \frac{\Gamma_1; q_1 \vdash v_1 : \tau \qquad \Gamma_2; q_2 \vdash v_2 : \mathsf{L}(A) \qquad A = \langle\tau, p\rangle}{\Gamma_1, \Gamma_2; q_1 + q_2 + p \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(A)}$$

$$\text{T-LinFun}\ \frac{\Gamma, x : \tau; p + q \vdash e : B \odot \Delta}{\Gamma; p \vdash \mathsf{linfun}(x.e) : \langle\tau, q\rangle \multimap B \odot \Delta} \qquad \text{T-Dcont}\ \frac{A_1 \odot \Delta_1 \lhd\!: k \lhd\!: A_2 \odot \Delta_2}{\Gamma; 0 \vdash \mathsf{dcont}(k) : A_2 \multimap A_1 \odot \Delta_1}$$

A.4 Computation Typing

$\boxed{\Gamma; q \vdash e : B \odot \Delta}$

$$\frac{\Gamma; q \vdash v : \tau}{\Gamma; q + p \vdash \mathsf{val}(v) : \langle \tau, p \rangle \odot \Delta}\ \textsc{T-Val} \qquad \frac{\Gamma_1; q \vdash e_1 : \langle \tau, q' \rangle \odot C \qquad \Gamma_2, x : \tau; q' \vdash e_2 : B \odot C}{\Gamma_1, \Gamma_2; q \vdash \mathsf{let}(e_1; x.e_2) : B \odot \Delta}\ \textsc{T-Let}$$

$$\frac{}{\cdot; q + p \vdash \mathsf{tick}\{q\} : \langle \mathsf{unit}, p \rangle \odot \Delta}\ \textsc{T-Tick}^+ \qquad \frac{}{\cdot; p \vdash \mathsf{tick}\{-q\} : \langle \mathsf{unit}, p + q \rangle \odot \Delta}\ \textsc{T-Tick}^-$$

$$\textsc{T-App}\ \frac{\Gamma_1; 0 \vdash v_1 : \mathcal{T} \qquad (\langle \tau, q_1 \rangle \to B \odot \Delta) \in \mathcal{T} \qquad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{app}(v_1; v_2) : B \odot \Delta}$$

$$\textsc{T-Casepair}\ \frac{\Gamma_1; q_1 \vdash v : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2; q_2 \vdash e : B \odot \Delta}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{casepair}(v; x_1.x_2.e) : B \odot \Delta}$$

$$\textsc{T-Casevoid}\ \frac{\Gamma; q \vdash v : \mathsf{void}}{\Gamma; q \vdash \mathsf{casevoid}(v) : B \odot \Delta} \qquad \textsc{T-Casesum}\ \frac{\begin{array}{c}\Gamma_1; q_1 \vdash v : \langle \tau_1, p_1 \rangle + \langle \tau_2, p_2 \rangle \\ \Gamma_2, x_1 : \tau_1; p_1 + q_2 \vdash e_1 : B \odot \Delta \qquad \Gamma_2, x_2 : \tau_2; p_2 + q_2 \vdash e_2 : B \odot \Delta\end{array}}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{casesum}(v; x_1.e_1; x_2.e_2) : B \odot \Delta}$$

$$\textsc{T-Caselist}\ \frac{\Gamma_1; q_1 \vdash v : \mathsf{L}(\langle \tau, p \rangle) \qquad \Gamma_2; q_2 \vdash e_1 : B \odot \Delta \qquad \Gamma_2, x : \tau, y : \mathsf{L}(\langle \tau, p \rangle); q_2 + p \vdash e_2 : B \odot \Delta}{\Gamma_1, \Gamma_2; q_1 + q_2 \vdash \mathsf{caselist}(v; e_1; x.y.e_2) : B \odot \Delta}$$

$$\textsc{T-Linapp}\ \frac{\Gamma_1; p \vdash v_1 : \langle \tau, q_1 \rangle \multimap B \odot \Delta \qquad \Gamma_2; q_2 \vdash v_2 : \tau}{\Gamma_1, \Gamma_2; p + q_1 + q_2 \vdash \mathsf{linapp}(v_1; v_2) : B \odot \Delta} \qquad \textsc{T-Do}\ \frac{\Gamma; p \vdash v : \tau_1 \qquad \ell : \langle \tau_1, q_1 \rangle \Rightarrow \langle \tau_2, q_2 \rangle \in \Delta}{\Gamma; p + q_1 \vdash \mathsf{do}[\ell](v) : \langle \tau_2, q_2 \rangle \odot \Delta}$$

$$\textsc{T-Handle}\ \frac{\begin{array}{c}\Gamma_1; p \vdash e : \langle \tau, q \rangle \odot \Delta \qquad \Gamma_3, y : \tau; q \vdash e_r : C \odot \Delta' \\ \Gamma_2 = |\Gamma_2| \qquad \forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow B \in \Delta.\ (\Gamma_2, x : \tau_1, c : B \multimap C \odot \Delta'; q_1 \vdash e_\ell : C \odot \Delta')\end{array}}{\Gamma_1, \Gamma_2, \Gamma_3; p \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : C \odot \Delta'}$$

A.5 Structural Typing Rules

$\boxed{\Gamma; q \vdash e : B \odot \Delta}$

$$\textsc{T-ContFun}\ \frac{\Gamma, x_1 : \mathcal{T}, x_2 : \mathcal{T}; q \vdash e : B \odot \Delta}{\Gamma, x : \mathcal{T}; q \vdash [x, x/x_1, x_2]e : B \odot \Delta} \qquad \textsc{T-WeakFun}\ \frac{\Gamma; q \vdash e : B \odot \Delta}{\Gamma, x : \mathcal{T}; q \vdash e : B \odot \Delta} \qquad \textsc{T-WeakPot}\ \frac{\Gamma; q' \vdash e : B \odot \Delta \qquad q \geq q'}{\Gamma; q \vdash e : B \odot \Delta}$$

B Full Cost Semantics

B.1 State Transitions

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

D-TICK
$$\frac{p \geq q}{k \triangleright \mathsf{tick}\{q\}; p \mapsto k \triangleleft \langle\rangle\, ; p - q}$$

D-RET
$$\frac{}{k \triangleright \mathsf{val}(v); q \mapsto k \triangleleft v; q}$$

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

D-LET
$$\frac{}{k \triangleright \mathsf{let}(e_1; x.e_2); q \mapsto k; x.e_2 \triangleright e_1; q}$$

D-SEQ
$$\frac{}{k; x.e \triangleleft v; q \mapsto k \triangleright [v/x]e; q}$$

D-PAIR
$$\frac{}{k \triangleright \mathsf{casepair}(\mathsf{pair}(v_1; v_2); x_1.x_2.e); q \mapsto k \triangleright [v_1, v_2/x_1, x_2]e; q}$$

D-INL
$$\frac{}{k \triangleright \mathsf{casesum}(\mathsf{inl}(v); x_1.e_1; x_2.e_2); q \mapsto k \triangleright [v/x_1]e_1; q}$$

D-INR
$$\frac{}{k \triangleright \mathsf{casesum}(\mathsf{inr}(v); x_1.e_1; x_2.e_2); q \mapsto k \triangleright [v/x_2]e_2; q}$$

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

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

D-TRY
$$\frac{}{k \triangleright \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r); q \mapsto k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleright e; q}$$

D-DO
$$\frac{}{k \triangleright \mathsf{do}[\ell](v); q \mapsto k \blacktriangleleft (\ell, v, \epsilon); q}$$

D-CAPTURE
$$\frac{}{k; x.e \blacktriangleleft (\ell, v, k'); q \mapsto k \blacktriangleleft (\ell, v, x.e \mathbin{@} k'); q}$$

D-HANDLE
$$\frac{\ell' \in \Delta}{\begin{array}{c} k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \blacktriangleleft (\ell', v, k'); q \\ \mapsto k \triangleright [v, \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k')/x, c]e_{\ell'}; q \end{array}}$$

D-NORMAL
$$\frac{}{k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft v; q \mapsto k \triangleright [v/y]e_r; q}$$

D-DCONT
$$\frac{}{k \triangleright \mathsf{linapp}(\mathsf{dcont}(k'); v); q \mapsto k \mathbin{@} k' \triangleleft v; q}$$

D-LINFUN
$$\frac{}{k \triangleright \mathsf{linapp}(\mathsf{linfun}(x.e); v); q \mapsto k \triangleright [v/x]e; q}$$

B.2 Initial/Final States

$\boxed{s \quad \text{initial} \qquad s \quad \text{final}}$

$$\textsc{D-init}\ \frac{}{\epsilon \triangleright e; q \quad \text{initial}} \qquad \textsc{D-final}\ \frac{}{\epsilon \triangleleft v; q \quad \text{final}} \qquad \textsc{D-final-exn}\ \frac{}{\epsilon \blacktriangleleft (\ell, v, k); q \quad \text{final}}$$

B.3 Stack Typing

$\boxed{A_1 \odot \Delta_1 \triangleleft\!: k \triangleleft\!: A_2 \odot \Delta_2}$

$$\textsc{K-Emp}\ \frac{}{B \odot \Delta \triangleleft\!: \epsilon \triangleleft\!: B \odot \Delta} \qquad \textsc{K-Bnd}\ \frac{A \odot \Delta_1 \triangleleft\!: k \triangleleft\!: B \odot \Delta_2 \qquad x : \tau; q \vdash e : B \odot \Delta_2}{A \odot \Delta_1 \triangleleft\!: k; x.e \triangleleft\!: \langle \tau, q \rangle \odot \Delta_2}$$

$$\textsc{K-Handler}\ \frac{\begin{array}{c} A \odot \Delta_1 \triangleleft\!: k \triangleleft\!: B \odot \Delta_3 \\ \forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow C \in \Delta_2.\ x : \tau_1, c : C \multimap B \odot \Delta_3; q_1 \vdash e_\ell : B \odot \Delta_3 \qquad y : \tau; q \vdash e_r : B \odot \Delta_3 \end{array}}{A \odot \Delta_1 \triangleleft\!: k; \text{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta_2}; y.e_r) \triangleleft\!: \langle \tau, q \rangle \odot \Delta_2}$$

B.4 State Typing

$\boxed{q \vdash s}$

$$\textsc{ST-Exp}\ \frac{\cdot; q \vdash e : B \odot \Delta \qquad \_ \triangleleft\!: k \triangleleft\!: B \odot \Delta}{q \vdash k \triangleright e} \qquad \textsc{ST-Val}\ \frac{\cdot; q_1 \vdash v : \tau \qquad \_ \triangleleft\!: k \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta}{q_1 + q_2 \vdash k \triangleleft v}$$

$$\textsc{ST-Eff}\ \frac{\cdot; q_1 \vdash v : \tau \qquad \ell : \langle \tau, p \rangle \Rightarrow C \in \Delta' \qquad \_ \triangleleft\!: k \triangleleft\!: B \odot \Delta' \qquad B \odot \Delta' \triangleleft\!: k' \triangleleft\!: C \odot \Delta''}{q_1 + p \vdash k \blacktriangleleft (\ell, v, k')}$$

C Full Soundness Proof

Lemma C.1. *If $\tau = |\tau|$ and $\Phi(v : \tau) = q$, then $q = 0$.*

Lemma C.2 (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. By simultaneous induction on the types τ, τ_1, τ_2. □

Lemma C.3 (Potential Relax). *If $\Gamma, q \vdash e : \langle \tau, p \rangle \odot \Delta$, then $\Gamma, q + r \vdash e : \langle \tau, p + r \rangle \odot \Delta$.*

Proof. By induction on computation typing rules. □

Lemma C.4 (Substitution). *If $\cdot; q_1 \vdash v_1 : \tau_1$:*
(1) If $\Gamma, x_1 : \tau_1; q_2 \vdash v : \tau$, then $\Gamma; q_1 + q_2 \vdash [v_1/x_1]v : \tau$.
(2) If $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \odot \Delta$, then $\Gamma; q_1 + q_2 \vdash [v_1/x_1]e : B \odot \Delta$.

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

- T-var: /
- T-triv, T-fun, T-Pair, T-inl, T-inr, T-nil, T-cons, T-linfun, T-dcont:
  (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: $q_2 = 0, v = \mathsf{fun}(f.x.e)(x_1 \neq x), \tau = \mathcal{T}$.
    $[v_1/x_1]\mathsf{fun}(f.x.e) = \mathsf{fun}(f.x.[v_1/x_1]e)$.
    By the assumption, $\forall(\langle \tau_x, q' \rangle \to B \odot \Delta) \in \mathcal{T}$, we have $\Gamma, x_1 : \tau_1, f : \mathcal{T}, x : \tau_x; q' \vdash e : B \odot \Delta$ and $|\Gamma, x_1 : \tau_1| = \Gamma, x_1 : \tau_1$ (so $|\tau_1| = \tau_1$).
    Then by lemma C.1, $q_1 = 0$.
    Using IH(B), $\forall(\langle \tau_x, q' \rangle \to B \odot \Delta) \in \mathcal{T}$, we have $\Gamma, f : \mathcal{T}, x : \tau_x; q' + q_1 \vdash [v_1/x_1]e : B$, where $q_1 = 0$, $|\Gamma| = \Gamma$.
    By T-fun, $\Gamma; 0 \vdash \mathsf{fun}(f.x.[v_1/x_1]e) : \tau$.
  - T-Pair:

$$\text{T-Pair}\quad \frac{\Gamma_1; q_1' \vdash v_1' : \tau_1' \qquad \Gamma_2; q_2' \vdash v_2' : \tau_2'}{\underbrace{\Gamma_1, \Gamma_2}_{\Gamma, x_1:\tau_1}; \underbrace{q_1' + q_2'}_{q_2} \vdash \underbrace{\mathsf{pair}(v_1'; v_2')}_{v} : \underbrace{\tau_1' \times \tau_2'}_{\tau}}$$

    * If $x_1 \in \Gamma_1$, using IH(A) on the first premise, $\Gamma_1 \setminus x_1; q_1' + q_1 \vdash [v_1/x_1]v_1' : \tau_1'$.
      By T-Pair on this and the second premise, $\Gamma_1 \setminus x, \Gamma_2; q_1 + q_1' + q_2' \vdash \mathsf{pair}([v_1/x_1]v_1'; v_2') : \tau_1' \times \tau_2'$ as desired.
    * If $x_2 \in \Gamma_2$, a symmetric argument can be made, starting with IH(A) on the second premise.
  - T-inl:

$$\text{T-Inl}\quad \frac{\Gamma'; p \vdash v' : \tau_1'}{\underbrace{\Gamma'}_{\Gamma, x_1:\tau_1}; \underbrace{q_1' + p}_{q_2} \vdash \underbrace{\mathsf{inl}(v')}_{v} : \underbrace{\langle \tau_1', q_1' \rangle + A_2}_{\tau}}$$

    Applying IH(A) on the premise, $\Gamma; p + q_1 \vdash [v_1/x_1]v' : \tau_1$.
    By T-inl on this, $\Gamma; p + q_1 + q_1' \vdash \mathsf{inl}([v_1/x_1]v') : \langle \tau_1', q_1' \rangle + A_2$ as deisred.
  - T-inr: Symmetric to T-inl
  - T-nil: /

- T-cons: $v = \mathsf{cons}(v_1'; v_2')$, $\tau = \mathsf{L}(\langle \tau_0, p\rangle)$, $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$.
  By the assumption $q_2 = p + q_1' + q_2'$, $\Gamma_1; q_1' \vdash v_1' : \tau_0$, and $\Gamma_2; q_2' \vdash v_2' : \mathsf{L}(\langle \tau_0, p\rangle)$.
  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-cons, $\Gamma; q_2 + q_1 \vdash \mathsf{cons}([v_1/x_1]v_1'; v_2') : \mathsf{L}(\langle \tau_0, p\rangle)$.
  * Otherwise, $x_1 : \tau_1 \in \Gamma_2$, using IH(A), $\Gamma_2 \setminus (x_1 : \tau_1); q_2' + q_1 \vdash [v_1/x_1]v_2' : \mathsf{L}(\langle \tau_0, p\rangle)$.
    By T-cons, $\Gamma; q_2 + q_1 \vdash \mathsf{cons}(v_1'; [v_1/x_1]v_2') : \mathsf{L}(\langle \tau_0, p\rangle)$.
- T-linfun: $v = \mathsf{linfun}(x.e)(x \neq x_1)$ and $\tau = \langle \tau', p\rangle \multimap B \odot \Delta$.
  $[v_1/x_1]\mathsf{linfun}(x.e) = \mathsf{linfun}(x.[v_1/x_1]e)$.
  By the assumption. we have $\Gamma, x_1 : \tau_1, x : \tau'; q_2 + p \vdash e : B \odot \Delta$.
  Using IH(B), $\Gamma, x : \tau'; q_2 + p + q_1 \vdash [v_1/x_1]e : B \odot \Delta$.
  By T-linfun, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\mathsf{linfun}(x.e) : \langle \tau', p\rangle \multimap B \odot \Delta$.

- T-dcont: $v = \mathsf{dcont}(k)$ has no free variable, so vacuous.

(B): Induction on $\Gamma, x_1 : \tau_1; q_2 \vdash e : B \odot \Delta$
- T-val: $e = \mathsf{val}(v')$, $B = \langle \tau, p\rangle$.
  By the assumption, $\Gamma, x_1 : \tau_1; q_2 - p \vdash v' : \tau$.
  Using IH(A), $\Gamma; q_2 - p + q_1 \vdash [v_1/x_1]v' : \tau$.
  By T-val, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\mathsf{val}(v') : \langle \tau, p\rangle \odot \Delta$.
- T-let: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$ and $e = \mathsf{let}(e_1; x.e_2)$.
  By the assumption, $\Gamma_1; q_2 \vdash e_1 : \langle \tau, q_3\rangle \odot \Delta$ and $\Gamma_2, x : \tau_1; q_3 \vdash e_2 : B \odot \Delta (x \neq x_1)$
  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), $\Gamma_1 \setminus (x_1 : \tau_1); q_2 + q_1 \vdash [v_1/x_1]e_1 : \langle \tau, q_3\rangle \odot \Delta$.
    By T-let, $\Gamma; q_2 + q_1 \vdash \mathsf{let}([v_1/x_1]e_1; x.e_2) : B \odot \Delta$.
  * Otherwise $x_1 : \tau_1 \in \Gamma_2$, then by IH(B), $\Gamma_2 \setminus (x_1 : \tau_1), x : \tau; q_3 + q_1 \vdash [v_1/x_1]e_2 : B \odot \Delta$.
    By T-let, and potential relax lemma C.3, $\Gamma; q_2 + q_1 \vdash \mathsf{let}(e_1; x.[v_1/x_1]e_2) : B \odot \Delta$.
- T-app: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$ and $e = \mathsf{app}(v_1'; v_2')$.
  By the assumption, $q_2 = p_1 + p_2 + p$. $\Gamma_1; p \vdash v_1' : \mathcal{T}$, $\langle \tau, p_1\rangle \to B \odot \Delta \in \mathcal{T}$ 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), $\Gamma_1 \setminus (x_1 : \tau_1); q_1 + p \vdash [v_1/x_1]v_1' : \mathcal{T}$.
    By T-app, $\Gamma; q_1 + q_2 \vdash \mathsf{app}([v_1/x_1]v_1'; v_2') : B \odot \Delta$.
  * 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, $\Gamma; q_2 + q_1 \vdash \mathsf{app}(v_1'; [v_1/x_1]v_2') : B \odot \Delta$.
- T-tick: /
- T-casepair:

$$\textsc{T-Casepair}\quad \frac{\Gamma_1; q_1' \vdash v : \tau_1' \times \tau_2' \qquad \Gamma_2, x_1 : \tau_1', x_2 : \tau_2'; q_2' \vdash e' : B \odot \Delta}{\underbrace{\Gamma_1, \Gamma_2}_{\Gamma, x_1:\tau_1}; \underbrace{q_1' + q_2'}_{q_2} \vdash \underbrace{\mathsf{casepair}(v; x_1.x_2.e')}_{e} : B \odot \Delta}$$

  * If $x \in \Gamma_1$, by IH(A) on the first premise, $\Gamma_1 \setminus x; q_1' + q_1 \vdash [v_1/x_1]v : \tau_1' \times \tau_2'$.
    By T-casepair on this and the second premise, $\Gamma_1 \setminus x, \Gamma_2; q_1' + q_1 + q_2' \vdash \mathsf{casepair}([v_1/x_1]v; x_1.x_2.e') : B \odot \Delta$ as desired.
  * If $x \in \Gamma_2$, by IH(B) on the second premise, $\Gamma_2 \setminus x, x_1 : \tau_1', x_2 : \tau_2'; q_2' + q_1 \vdash [v_1/x_1]e' : B \odot \Delta$.
    By T-casepair on this and the first premise, $\Gamma_1 \setminus x, \Gamma_2; q_1' + q_1 + q_2' \vdash \mathsf{casepair}(v; x_1.x_2.[v_1/x_1]e') : B \odot \Delta$ as desired.

- T-casesum: Analogous to T-caseprod case.
- T-caselist: $\Gamma, x_1 : \tau_1 = \Gamma_1, \Gamma_2$ and $e = \mathsf{caselist}(v; e_0; x.y.e_1)$.
  By the assumption, $q_2 = q_1' + q_2'$, $\Gamma_1; q_1' \vdash v : \mathsf{L}(\langle \tau_0, r \rangle)$, $\Gamma_2; q_2' \vdash e_0 : B \odot \Delta$, $\Gamma_2, x : \tau_0, y : \mathsf{L}(\langle \tau_0, r \rangle); q_2' + r \vdash e_1 : B \odot \Delta$.
  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), $\Gamma_1 \setminus (x_1 : \tau_1); q_1 + q_1' \vdash [v_1/x_1]v : \mathsf{L}(\langle \tau_0, r \rangle)$.
    By T-caselist, $\Gamma; q_1 + q_2 \vdash \mathsf{caselist}([v_1/x_1]v; e_0; x.y.e_1) : B \odot \Delta$.
  * Otherwise $x_1 : \tau_1 \in \Gamma_2$, then by IH(B), $\Gamma_2 \setminus (x_1 : \tau_1); q_2' + q_1 \vdash [v_1/x_1]e_0 : B \odot \Delta$, and $\Gamma_2 \setminus (x_1 : \tau_1), x : \tau_0, y : \mathsf{L}(\langle \tau_0, r \rangle); q_2' + q_1 + r \vdash [v_1/x_1]e_1 : B \odot \Delta$.
    By T-caselist, $\Gamma; q_1 + q_2 \vdash \mathsf{caselist}(v; [v_1/x_1]e_0; x.y.[v_1/x_1]e_1) : B \odot \Delta$.
- T-do: By the premise, $\Gamma, x_1 : \tau_1; p \vdash v : \tau_0$ with $\ell : \langle \tau_0, q_0 \rangle \Rightarrow B \in \Delta$, $q_2 = p + q_0$.
  Using IH(A), $\Gamma; p + q_1 \vdash [v_1/x_1]v : \tau_0$.
  By T-do, $\Gamma; q_2 + q_1 \vdash [v_1/x_1]\mathsf{do}[\ell](v) : B \odot \Delta$.
- T-handle:

T-Handle

$$\frac{\Gamma_2 = |\Gamma_2| \qquad \begin{array}{c} \Gamma_1; q_2 \vdash e : \langle \tau, q \rangle \odot \Delta \\ \forall \ell : \langle \tau_1, q_l \rangle \Rightarrow B \in \Delta.\ (\Gamma_2, x : \tau_1, c : B \multimap C \odot \Delta'; q_l \vdash e_\ell : C \odot \Delta') \\ \Gamma_3, y : \tau; q \vdash e_r : C \odot \Delta' \end{array}}{\Gamma_1, \Gamma_2, \Gamma_3; q_2 \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : C \odot \Delta'}$$

  Let $\Gamma_1, \Gamma_2, \Gamma_3 = \Gamma, x_1 : \tau_1$. Since $\Gamma_2 = |\Gamma_2|$, only 2 cases are possible:
  * If $x_1 \in \Gamma_1$, by IH(B) on the first premise, $\Gamma_1 \setminus x_1; q_2 + q_1 \vdash [v_1/x_1]e : \langle \tau, q \rangle \odot \Delta$.
    By T-handle, we have the desired.
  * If $x_1 \in \Gamma_2$, by IH(B) on the last premise, $\Gamma_2 \setminus x_1, y : \tau; q + q_1 \vdash [v_1/x_1]e_r : C \odot \Delta'$.
    By T-handle and potential relax lemma C.3 on the first premis, we have the desired.
- T-ContFun, T-weekfun, T-weakpot: Follows from applying IH on the premise.

□

Lemma C.5 (State Relaxing). *If $q \vdash s$, then $q' \vdash s$ for $q' \geq q$.*

Proof. By induction on state and stack typing rules. □

Lemma C.6 (Stack Append Typing). *$B_2 \odot \Delta_2 \mathrel{\triangleleft{:}} k_1 \mathbin{@} k_2 \mathrel{\triangleleft{:}} B_1 \odot \Delta_1$ if and only if there exists B, Δ such that $B_2 \odot \Delta_2 \mathrel{\triangleleft{:}} k_1 \mathrel{\triangleleft{:}} B \odot \Delta$ and $B \odot \Delta \mathrel{\triangleleft{:}} k_2 \mathrel{\triangleleft{:}} B_1 \odot \Delta_1$*

Theorem C.7 (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: $k \triangleright \mathsf{tick}\{r\}; q \mapsto k \triangleleft \langle\rangle\, ; q_0$ and $q \geq r, q_0 = q - r$.
  $q \vdash s$, so $\cdot; q \vdash \mathsf{tick}\{r\} : B \odot \Delta$ and $\_ \mathrel{\triangleleft{:}} k \mathrel{\triangleleft{:}} B \odot \Delta$.
  We want to show $q - r \vdash k \triangleleft \langle\rangle$.
  Let $B = \langle \tau, x \rangle$. Induct on $\cdot; q \vdash \mathsf{tick}\{r\} : \langle \tau, x \rangle \odot \Delta$ to show $q - x \geq r$ and $\tau = \mathsf{unit}$. Only 2 cases are possible:
  - T-tick$^+$: Then $r \geq 0$, $q = r + x$, $\tau = \mathsf{unit}$, $x = 0$. Good.
  - T-tick$^-$: Then $r \leq 0$, $q = r + x$, $\tau = \mathsf{unit}$, $x = 0$. Good.
  - T-WeakPot: By the assumption, $\cdot; q' \vdash \mathsf{tick}\{r\} : \langle \tau, x \rangle \odot \Delta$, where $q \geq q'$.
    Using the inner IH $q' - x \geq r$, we have $q' - x \geq r$.
  By T-unit, $\cdot; 0 \vdash \langle\rangle : \mathsf{unit}$.
  We have $\_ \mathrel{\triangleleft{:}} k \mathrel{\triangleleft{:}} \langle \tau, x \rangle \odot \Delta$, then by ST-val, $x \vdash k \triangleleft \langle\rangle$.
  $x \leq q - r$. By state relaxing lemma C.5, $q - r \vdash k \triangleleft \langle\rangle$.

- D-FUN: $k \triangleright \mathsf{app}(\mathsf{fun}(f.x.e);v_2);q \mapsto k \triangleright [\mathsf{fun}(f.x.e),v_2/f,x]e;q$.
  $q \vdash s$, so $\_ \triangleleft: k \triangleleft: B \odot \Delta$, $\cdot;q \vdash \mathsf{app}(\mathsf{fun}(f.x.e);v_2) : B \odot \Delta$.
  It suffices to show $\cdot;q \vdash [\mathsf{fun}(f.x.e),v_2/f,x]e : B \odot \Delta$.
  Induct on $\cdot;q \vdash \mathsf{app}(\mathsf{fun}(f.x.e);v_2) : B \odot \Delta$. Only 2 cases are possible:
  - T-APP:
    By the assumption, $\cdot;0 \vdash \mathsf{fun}(f.x.e) : \mathcal{T}$, $\cdot;q_2 \vdash v_2 : \tau$, $q = q_1 + q_2$, $\langle \tau, q_1 \rangle \to B \odot \Delta \in \mathcal{T}$.
    Invert, get that for $\langle \tau, q_1 \rangle \to B \odot \Delta \in \mathcal{T}$, we have $f : \mathcal{T}, x : \tau; q_1 \vdash e : B \odot \Delta$.
    Using the substitution lemma C.4 twice on f, x, $\cdot;q_1 + q_2 \vdash [\mathsf{fun}(f.x.e),v_2/f,x]e : B \odot \Delta$.
  - T-WEAKPOT:
    By the assumption, $\cdot;q' \vdash \mathsf{app}(\mathsf{fun}(f.x.e);v_2) : B \odot \Delta$ and $q \geq q'$.
    Using the inner IH, $\cdot;q' \vdash [\mathsf{fun}(f.x.e),v_2/f,x]e : B \odot \Delta$.
    By T-WEAKPOT, $\cdot;q \vdash [\mathsf{fun}(f.x.e),v_2/f,x]e : B \odot \Delta$.
- D-RET: $k \triangleright \mathsf{val}(v);q \mapsto k \triangleleft v;q$.
  $q \vdash k \triangleright \mathsf{val}(v)$, so $\_ \triangleleft: k \triangleleft: B \odot \Delta$ and $\cdot;q \vdash \mathsf{val}(v) : B \odot \Delta$.
  Let $B = \langle \tau, r \rangle$. Induct on $\cdot;q \vdash \mathsf{val}(v) : \langle \tau, r \rangle \odot \Delta$ to show there exists q_1, where $\cdot;q_1 \vdash v : \tau$ and $q - r \geq q_1$.
  Only 2 cases are possible:
  - T-VAL: $B = \langle \tau, r \rangle$, and the assumption gives $\cdot;q_1 \vdash v : \tau$ where $q_1 + r = q$.
  - T-WEAKPOT:
    By the assumption, $\cdot;q \vdash \mathsf{val}(v) : B \odot \Delta$, and $q \geq q'$.
    Using the inner IH, we have $q - r \geq q' - r \geq q_1$, where $\cdot;q_1 \vdash v : \tau$.

  By ST-VAL, $q_1' + r \vdash k \triangleleft v$.
  $q_1' + r \leq q - r + r = q$, so $q \vdash k \triangleleft v$ by state relaxing lemma C.5.
- D-LET: $k \triangleright \mathsf{let}(e_1;x.e_2);q \mapsto k;x.e_2 \triangleright e_1;q$.
  By the assumption, $q \vdash s$, so $\cdot;q \vdash \mathsf{let}(e_1;x.e_2) : B \odot \Delta$, $\_ \triangleleft: k \triangleleft: B \odot \Delta$.
  Induct on $\cdot;q \vdash \mathsf{let}(e_1;x.e_2) : B \odot \Delta$ to show:
  for some τ_1, q_1, we have $\cdot;q \vdash e_1 : \langle \tau_1, q_1 \rangle \odot \Delta$ and $x : \tau_1; q_1 \vdash e_2 : B \odot \Delta$.
  Only 2 cases are possible:
  - T-LET: Immediate by the assumption.
  - T-WEAKPOT:.
    By the assumption, $\cdot;q' \vdash \mathsf{let}(e_1;x.e_2) : B \odot \Delta$, where $q \geq q'$.
    Using the inner IH, we have τ_1, q_1, such that $\cdot;q' \vdash e_1 : \langle \tau_1, q_1 \rangle \odot \Delta$, $x : \tau_1; q_1 \vdash e_2 : B \odot \Delta$.
    By T-WEAKPOT, $\cdot;q \vdash e_1 : \langle \tau_1, q_1 \rangle \odot \Delta$, $x : \tau_1; q_1 \vdash e_2 : B \odot \Delta$.

  By K-BND, $k;x.e_2 \triangleleft: \langle \tau_1, q_1 \rangle \odot \Delta$.
  We have $\cdot;q \vdash e_1 : \langle \tau_1, q_1 \rangle \odot \Delta$. By ST-EXP, $q \vdash k;x.e_2 \triangleright e_1$.
- D-SEQ: $k;x.e \triangleleft v;q \mapsto k \triangleright [v/x]e;q$.
  By the assumption, $q \vdash k;x.e \triangleleft v$. Invert, get $q = q_1 + q_2$, $\cdot;q_1 \vdash v : \tau$ and $\_ \triangleleft: k;x.e \triangleleft: \langle \tau, q_2 \rangle \odot \Delta$.
  Invert, $x : \tau; q_2 \vdash e : B \odot \Delta$ and $\_ \triangleleft: k \triangleleft: B \odot \Delta$.
  By the substitution lemma C.4, $\cdot;q_2 + q_1 \vdash [v/x]e : B \odot \Delta$.
  By ST-EXP, $q_2 + q_1 \vdash k \triangleright [v/x]e$ $(q = q_1 + q_2)$.
- D-NIL: $k \triangleright \mathsf{caselist}(\mathsf{nil};e_0;x.y.e_1);q \mapsto k \triangleright e_0;q$.
  By the assumption, $\_ \triangleleft: k \triangleleft: B \odot \Delta$, $\cdot;q \vdash \mathsf{caselist}(\mathsf{nil};e_0;x.y.e_1) : B \odot \Delta$.
  It suffices to show $\cdot;q \vdash e_0 : B \odot \Delta$.
  Induct on $\cdot;q \vdash \mathsf{caselist}(\mathsf{nil};e_0;x.y.e_1) : B \odot \Delta$. Only 2 cases are possible:
  - T-CASELIST: By the assumption, $\cdot;q_2 \vdash e_0 : B \odot \Delta$, $\cdot;q_1 \vdash \mathsf{nil} : \mathsf{L}(A)$ where $q = q_1 + q_2$.
    Then by T-WEAKPOT, $\cdot;q \vdash e_0 : B \odot \Delta$.
  - T-WEAKPOT: Analogous to other cases.

- D-cons: $k \triangleright \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q$.
  By the assumption, $\_ \triangleleft\colon k \triangleleft\colon B \odot \Delta$, $\cdot; q \vdash \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1) : B \odot \Delta$.
  It suffices to show $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \odot \Delta$.
  Induct on $\cdot; q \vdash \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1) : B \odot \Delta$. Only 2 cases are possible:
  - T-caselist:
    By the assumption, $x : \tau, y : \mathsf{L}(\langle \tau, p \rangle); q_2{+}p \vdash e_0 : B{\odot}\Delta$, where $\cdot; q_1 \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(\langle \tau, p \rangle)$, $q = q_1 + q_2$.
    Invert $\cdot; q_1 \vdash \mathsf{cons}(v_1; v_2) : \mathsf{L}(\langle \tau, p \rangle)$ to get $q_1 = q_1' + q_2' + p$, $\cdot; q_1' \vdash v_1 : \tau$, $\cdot; q_2' \vdash v_2 : \mathsf{L}(\langle \tau, p \rangle)$.
    Using substitution lemma C.4, $\cdot; q_2 + p + q_1' + q_2' \vdash [v_1, v_2/x, y]e_1 : B \odot p_0$, which is $\cdot; q \vdash [v_1, v_2/x, y]e_1 : B \odot \Delta$.
  - T-WeakPot: Analogous to other cases.
- D-pair: $k \triangleright \mathsf{casepair}(\mathsf{pair}(v_1; v_2); x_1.x_2.e); q \mapsto k \triangleright [v_1, v_2/x_1, x_2]e; q$.
  By the premise of ST-exp, $k \triangleleft\colon B \odot \Delta$, $\cdot; q \vdash \mathsf{casepair}(\mathsf{pair}(v_1; v_2); x_1.x_2.e) : B \odot \Delta$.
  It suffices to show $\cdot; q \vdash [v_1, v_2/x_1, x_2]e : B \odot \Delta$.
  Induct on $\cdot; q \vdash \mathsf{casepair}(\mathsf{pair}(v_1; v_2); x_1.x_2.e) : B \odot \Delta$. Only 2 cases are possible:
  - T-casepair:
    By the premises, $x_1 : \tau_1, x_2 : \tau_2; q_2 \vdash e : B \odot \Delta$, $\cdot; q_1 \vdash \mathsf{pair}(v_1; v_2) : \tau_1 \times \tau_2$, and $q = q_1 + q_2$.
    By inversion on $\cdot; q_1 \vdash \mathsf{pair}(v_1; v_2) : \tau_1 \times \tau_2$, we get $q_1 = q_1' + q_2'$, $\cdot; q_1' \vdash v_1 : \tau_1$, and $\cdot; q_2' \vdash v_2 : \tau_2$.
    Using substitution lemma C.4, $\cdot; q_2 + q_1' + q_2' \vdash [v_1, v_2/x_1, x_2]e : B \odot \Delta$ as desired, since $q = q_2 + q_1' + q_2'$.
  - T-WeakPot: Analogous to other cases.
- D-inl: $k \triangleright \mathsf{casesum}(\mathsf{inl}(v); x_1.e_1; x_2.e_2); q \mapsto k \triangleright [v/x_1]e_1; q$.
  By the premise of ST-exp, $k \triangleleft\colon B \odot \Delta$, $\cdot; q \vdash \mathsf{casesum}(\mathsf{inl}(v); x_1.e_1; x_2.e_2) : B \odot \Delta$.
  It suffices to show $\cdot; q \vdash [v/x_1]e_1 : B \odot \Delta$.
  Induct on $\cdot; q \vdash \mathsf{casesum}(\mathsf{inl}(v); x_1.e_1; x_2.e_2) : B \odot \Delta$. Only 2 cases are possible:
  - T-casesum:
    By the premises, $x_1 : \tau_1, ; p_1 + q_2 \vdash e_1 : B \odot \Delta$, $\cdot; q_1 \vdash \mathsf{inl}(v) : \langle \tau_1, p_1 \rangle + A_2$, and $q = q_1 + q_2$.
    By inversion on $\cdot; q_1 \vdash \mathsf{inl}(v) : \langle \tau_1, p_1 \rangle + A_2$, we get $q_1 = p_1 + q_1'$ and $\cdot; q_1' \vdash v : \tau_1$.
    Using substitution lemma C.4, $\cdot; p_1{+}q_2{+}q_1' \vdash [v/x_1]e_1 : B{\odot}\Delta$ as desired, since $q = p_1{+}q_2{+}q_1'$.
  - T-WeakPot: Analogous to other cases.
- D-inr: Symmetric to above
- D-do: $k \triangleright \mathsf{do}[\ell](v); q \mapsto k \blacktriangleleft (\ell, v, \epsilon); q$
  By the premises of ST-Exp on the left, $\cdot; q \vdash \mathsf{do}[\ell](v) : B \odot \Delta$ and $\_ \triangleleft\colon k \triangleleft\colon B \odot \Delta$.
  Induct on $\cdot; q \vdash \mathsf{do}[\ell](v) : B \odot \Delta$ to show that $\cdot; p \vdash v : \tau_1$ with $\ell : \langle \tau_1, q_1 \rangle \Rightarrow \langle \tau_2, q_2 \rangle \in \Delta$ and $p \leq q - q_1$. Only 2 cases are possible:
  - T-do: Follows directly from its premises, and $q = p + q_1$.
  - T-WeakPot: Analogous to other cases.
  Let $k' = \epsilon$. By K-Emp and ST-Eff, $q \vdash k \blacktriangleleft (l, v, \epsilon)$ as desired.
- D-Capture: $k; x.e \blacktriangleleft (\ell, v, k'); q \mapsto k \blacktriangleleft (\ell, v, x.e \mathbin{@} k'); q$
  By the premises of ST-Eff on the left, we have
  - $q = q_1 + p$
  - $\cdot; q_1 \vdash v : \tau$
  - $\Delta' = \Delta, \ell : \langle \tau, p \rangle \Rightarrow C$
  - $\_ \triangleleft\colon k; x.e \triangleleft\colon B \odot \Delta'$
  - $B \odot \Delta' \triangleleft\colon k' \triangleleft\colon C \odot \Delta''$

  The first 3 premises can be untouched. For the 4th premise, invert with K-Bnd to get $\_ \triangleleft\colon k \triangleleft\colon B_2 \odot \Delta'$ where $x : \tau_2; q_2 \vdash e : B_2 \odot \Delta'$ and $B = \langle \tau_2, q_2 \rangle$. For the 5th premise, we first use K-Emp

and K-Bnd to construct that the single-frame stack $x.e$ has typing $B_2 \odot \Delta' \triangleleft\!: x.e \triangleleft\!: B \odot \Delta'$, then apply lemma C.6 between stacks $x.e$ and k' to get that $B_2 \odot \Delta' \triangleleft\!: x.e\ @\ k' \triangleleft\!: C \odot \Delta''$.
- $q = q_1 + p$
- $\cdot; q_1 \vdash v : \tau$
- $\Delta' = \Delta, \ell : \langle \tau, p \rangle \Rightarrow C$
- $\_ \triangleleft\!: k \triangleleft\!: B_2 \odot \Delta'$
- $B_2 \odot \Delta' \triangleleft\!: x.e\ @\ k' \triangleleft\!: C \odot \Delta''$

Now with these 5 premises, we can apply ST-Eff to type the right side.

- D-Handle: $k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \blacktriangleleft (\ell', v, k'); q \mapsto\ k \triangleright [v, d/x, c]e_{\ell'}; q$ when $\ell \in \Delta$ and where $d = \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)\ @\ k')$.
  By the premises of ST-Eff on the left, we have (1) $q = q_1 + p$, (2) $\cdot; q_1 \vdash v : \tau$, (3) $\ell' : \langle \tau, p \rangle \Rightarrow C \in \Delta$, (4) $\_ \triangleleft\!: k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft\!: B' \odot \Delta$, and (5) $B' \odot \Delta \triangleleft\!: k' \triangleleft\!: C \odot \Delta''$. Invert (4) using K-Handler to get (6) $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta_3$, (7) for all $\ell : \langle \tau_1, q_1 \rangle \Rightarrow C$ in Δ, $x : \tau_1, c : C \multimap B \odot \Delta_3; q_1 \vdash e_\ell : B \odot \Delta_3$, and (8) $y : \tau'; q' \vdash e_r : B \odot \Delta_3$ such that $B' = \langle \tau', q' \rangle$. Next, we use K-Emp, K-Handler, (7), (8), and lemma C.6 to construct the stack typing $B \odot \Delta_3 \triangleleft\!: \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)\ @\ k' \triangleleft\!: C \odot \Delta''$. Then by T-Dcont, we have (9) $\cdot; 0 \vdash d : C \multimap B \odot \Delta_3$.
  Now, we can apply the substitution lemma for both x and c in e_ℓ. For $[v/x]$, we have (2). For $[\mathsf{dcont}(\ldots)/c]$, we have (9). For e_ℓ, we have (7). Putting it all together, we have
  $$\cdot; p + q_1 + 0 \vdash [v, d/x, c]e' : B \odot \Delta_3$$
  Finally, using $q = q_1 + p$ from (1), along with $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta_3$ from (6), we can apply ST-Exp to type the overall right hand side.
- D-normal: $k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft v; q \mapsto k \triangleright [v/y]e_r; q$.
  By the assumption, $q = q_1 + q_2$, $\cdot; q_1 \vdash v : \tau$, $\_ \triangleleft\!: k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta$.
  Invert, get $\_ \triangleleft\!: k \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta$.
  By ST-val, $q_1 + q_2 = q \vdash k \triangleleft v$.
- D-try: $k \triangleright \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r); q \mapsto k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleright e; q$.
  By the assumption, $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta$ and $\cdot; q \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : B \odot \Delta$.
  Induct on $\cdot; q \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : B \odot \Delta$ to show there exists Δ', q' such that $q \geq q'$, and $\cdot; q' \vdash e_1 : B \odot \Delta'$, and $\forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow \langle \tau_2, q_2 \rangle \in \Delta', x : \tau_1, c : \langle \tau_2, q_2 \rangle \multimap A \odot \Delta; q_1 \vdash e_\ell : A \odot \Delta$, and $y : \tau \vdash e_r : B \odot \Delta$
  Only 2 cases are possible:
  - T-Handle: Immediate by the assumption when context is empty, and choosing $q' = q$.
  - T-WeakPot: By the assumption, for some q''' such that $q \geq q'''$, we have $\cdot; q''' \vdash \mathsf{handler}(e; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) : B \odot \Delta$. Then apply the inner IH to obtain that there exists Δ', q'' such that $q''' \geq q''$, and $\cdot; q' \vdash e_1 : B \odot \Delta'$, and $\forall \ell : \langle \tau_1, q_1 \rangle \Rightarrow \langle \tau_2, q_2 \rangle \in \Delta', x : \tau_1, c : \langle \tau_2, q_2 \rangle \multimap A \odot \Delta; q_1 \vdash e_\ell : A \odot \Delta$, and $y : \tau \vdash e_r : B \odot \Delta$. Now choose $q' = q''$, then by transitivity $q \geq q'$. The remaining conditions immediately hold.

  Then by K-handler and ST-exp and T-WeakPot, $q \vdash k; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleright e$.
- D-Dcont: $k \triangleright \mathsf{app}(\mathsf{dcont}(k'); v); q \mapsto k\ @\ k' \triangleleft v; q$
  By the premises of ST-Exp on the left, $\cdot; q \vdash \mathsf{app}(\mathsf{dcont}(k'); v) : B \odot \Delta$ where $\_ \triangleleft\!: k \triangleleft\!: B \odot \Delta$.
  We want to show that $\cdot; q_1 \vdash v : \tau$ and $\_ \triangleleft\!: k\ @\ k' \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta'$ where $q = q_1 + q_2$.
  Induct on $\cdot; q \vdash \mathsf{app}(\mathsf{dcont}(k'); v) : B \odot \Delta$. Only 2 cases are possible:
  - T-linapp: By the premises, we have $\cdot; 0 \vdash \mathsf{dcont}(k') : \langle \tau, q_2 \rangle \multimap B \odot \Delta$ and $\cdot; q_1 \vdash v : \tau$ where $q = q_1 + q_2$. Invert on the first premise with T-Dcont to get that $B \odot \Delta \triangleleft\!: k' \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta'$. Then by lemma C.6 between the typings of k and k', we have that $\_ \triangleleft\!: k\ @\ k' \triangleleft\!: \langle \tau, q_2 \rangle \odot \Delta'$.
  - T-WeakPot: Analogous to other cases.

By ST-Val, we get the right side as desired.

□

Lemma C.8 (Canonical Forms). *If* $\cdot; q \vdash v : \tau$, *and*

- $\tau = \mathsf{L}(A)$, *then either* $v = \mathsf{nil}$ *or* $v = \mathsf{cons}(v_1; v_2)$.
- $\tau = \tau_1 \times \tau_2$, *then* $v = \mathsf{pair}(v_1; v_2)$
- $\tau = A_1 + A_2$, *then either* $v = \mathsf{inl}(v_1)$ *or* $v = \mathsf{inr}(v_2)$
- $\tau = \mathcal{T}$, *then* $v = \mathsf{fun}(f.x.e')$, *or* $\mathsf{linfun}(x.e)$, *or* $\mathsf{dcont}(k)$.

Corollary C.9. *If* $\cdot; q \vdash e : B \odot \Delta$, *and*

- $e = \mathsf{caselist}(v; e_0; x.y.e_1)$, *then either* $v = \mathsf{nil}$ *or* $v = \mathsf{cons}(v_1; v_2)$.
- $e = \mathsf{casepair}(v; x_1.x_2.e')$, *then* $v = \mathsf{pair}(v_1; v_2)$
- $e = \mathsf{casesum}(v; x_1.e_1; x_2.e_2)$, *then either* $v = \mathsf{inl}(v_1)$ *or* $v = \mathsf{inr}(v_2)$
- $e = \mathsf{app}(v_1; v_2)$, *then* $v_1 = \mathsf{fun}(f.x.e')$, *or* $\mathsf{linfun}(x.e')$, *or* $\mathsf{dcont}(k)$.

Theorem C.10 (Progress). *If* $q \vdash s$, *then either* s *final or* $s; q \mapsto s'; q'$.

Proof. Induct on $q \vdash s$.

- ST-val: $q \vdash k \triangleleft v$.
  Induct on structure of k.
  - $k = \epsilon$: $k \triangleleft v$ is final.
  - $k = k'; x.e$: By D-seq, $k'; x.e \triangleleft v; q \mapsto k' \triangleright [v/x]e; q$.
  - $k = k'; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$;: By D-normal, $k'; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \triangleleft v; q \mapsto k' \triangleright [v/y]e_r; q$.
- ST-Eff: $q \vdash k \blacktriangleleft (\ell', v, k')$
  Induct on structure of k.
  - $k = \epsilon$: $k \blacktriangleleft (\ell', v, k')$ is final
  - $k = k_1; x.e$: By D-Capture, $k_1; x.e \blacktriangleleft (\ell', v, k'); q \mapsto k_1 \blacktriangleleft (\ell', v, x.e \mathbin{@} k'); q$
  - $k = k_1; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r)$: Our type system guarantees that $\ell' \in \Delta$. Then by D-handle, $k_1; \mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \blacktriangleleft (\ell', v, k'); q$
    $\mapsto k_1 \triangleright [v, \mathsf{dcont}(\mathsf{handler}(\_; \{\ell : x.c.e_\ell\}_{\ell \in \Delta}; y.e_r) \mathbin{@} k')/x, c]e_{\ell'}; q$
- ST-exp: $q \vdash k \triangleright e$.
  By the premises, $\cdot; q \vdash e : B \odot \Delta$ and $\_ \triangleleft : k \triangleleft : B \odot \Delta$. Induct on the structure of e.
  - $e = \mathsf{caselist}(v; e_0; x.y.e_1)$
    By type inversion and the canonical forms lemma, either $v = \mathsf{nil}$ or $v = \mathsf{cons}(v_1; v_2)$.
    * $v = \mathsf{nil}$: by D-nil, $k \triangleright \mathsf{caselist}(\mathsf{nil}; e_0; x.y.e_1); q \mapsto k \triangleright e_0; q$
    * $v = \mathsf{cons}(v_1; v_2)$: by D-cons, $k \triangleright \mathsf{caselist}(\mathsf{cons}(v_1; v_2); e_0; x.y.e_1); q \mapsto k \triangleright [v_1, v_2/x, y]e_1; q$
  - e is some other non-$\mathsf{tick}\{r\}$ expression. These cases follow similarly to $e = \mathsf{caselist}(v; e_0; x.y.e_1)$.
    For each, first apply the canonical forms lemma/corollary, followed by one of the rules: D-ret, D-fun,D-let, D-seq, D-pair, D-inl, D-inr, D-do, D-try, D-linfun, D-Dcont. This will end up showing that $k \triangleright e; q \mapsto s'; q$ for some s'
  - $e = \mathsf{tick}\{r\}$.
    Induct on $\cdot; q \vdash \mathsf{tick}\{r\} : B \odot \Delta$ to show $q \geq r$. Only 2 cases are possible:
    * T-tick: $q = r$.
    * T-WeakPot: $q \geq r$ by the IH.
    $q \geq r$ then by D-tick, $k \triangleright \mathsf{tick}\{r\}; q \mapsto k \triangleleft \langle\rangle\, ; q - r$.

□

Received 2025-10-09; accepted 2026-02-17