Dependently-Typed AARA: A Non-Affine Approach for Resource Analysis of Higher-Order Programs

Han Xu[1][0000-0002-2548-6866] and Di Wang[2][0000-0002-2418-7987]⋆

[1] Princeton University, Princeton NJ 08544, USA
[2] Key Lab of HCST (PKU), MOE; SCS, Peking University, China

Abstract. Static resource analysis determines the resource consumption (e.g., time complexity) of a program without executing it. Among the numerous existing approaches for resource analysis, affine type systems have been one dominant approach. However, these affine type systems fall short of deriving precise resource behavior of higher-order programs, particularly in cases that involve partial applications.
This article presents $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, a non-affine AARA-style dependent type system for resource reasoning about higher-order functional programs. The key observation is that the main issue in previous approaches comes from (i) the close coupling of types and resources, and (ii) the conflict between affine and higher-order typing mechanisms. To derive precise resource behavior of higher-order functions, $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ decouples resources from types and follows a non-affine typing mechanism. The non-affine type system of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ achieves this by using dependent types, which allow expressing type-level potential functions separate from ordinary types. This article formalizes $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s syntax and semantics, and proves its soundness, which guarantees the correctness of resource bounds. Several challenging classic and higher-order examples are presented to demonstrate the expressiveness and compositionality of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s reasoning capability. This article also includes an algorithmic variant of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s type system and a discussion of the automation of type checking and inference for $\lambda^{\mathsf{na}}_{\mathsf{amor}}$.

Keywords: Type System · Static analysis · Resource Analysis.

1 Introduction

Resource Analysis Resource consumption (e.g., time, space, and complexity) of programs has always been at the heart of computer science, with programming language research being no exception. In this article, we focus on the static verification of upper bounds on the resource consumption of a higher-order functional program. For example, below shows the type and implementation of a *curried* `append` function for integer lists, where we wrap recursive calls in a special construct `tick 1` to indicate that we specify a resource model that counts the number of recursive calls.

⋆ Corresponding Author

```
append :: List(int) → List(int) → List(int)
append = λx. λy. case x of nil ⇒ y | cons(x0,x1) ⇒ cons(x0, tick
1 (append x1 y))
```

A resource analysis of the append function should yield that the number of recursive calls is upper-bounded by the length of the first argument. There have been tons of techniques for verifying or automatically inferring such an upper bound; in particular, there have been many *type systems* for resource analysis [23,37,36,20,18,17,31,32,8,9,38,11,15,35,57,51,13,33,34,14,10,67,2,19,22,48].

Among those type systems, *automatic amortized resource analysis* (AARA) has been a fundamental approach for deriving symbolic resource bounds, i.e., bounds that are given by a function of the program's input. AARA was initially studied by Hofmann and Jost [23], and later extended to various resource-analysis situations with a recent survey provided by Hoffmann and Jost [21]. The high-level idea of AARA is to augment the type system with *resource annotations*, which specify *potential functions* carried on data structures. For example, a value of type int^1 carries one unit of potential, which can be used to pay for one unit of resource consumption (i.e., `tick 1`). A more interesting example is the annotated list type $List(int^1)$, the value of which carries one unit of potential per list element, thus the whole potential equals exactly the list length. An AARA type system would derive an *uncurried* type annotation for append:

`append` :: $List(int^1) \times List(int^0) \to List(int^0)$

The type suggests that append consumes up to one unit of resource per element in the first list.

Closure and Resource With the *uncurried* annotated type shown above, can we say that the time complexity (in terms of the number of recursive calls) of a curried append is $O(n)$, if n is the length of the first argument? Unfortunately, the fact that append is *curried* makes things complicated, because it is obvious that partially applying append to *one* list, e.g., append ℓ_1, would consume no resource. Moreover, the partial application would create a closure that captures ℓ_1. When applied to another list ℓ_2, the closure's time complexity is linear—*not* in the length of the closure's argument ℓ_2—but in the length of the captured ℓ_1. In other words, precise resource analysis of higher-order functions requires that the type system track fine-grained information about closures, especially those that can capture data structures with potentials. We call such closures *potential-capturing closures*.

In this article, we focus on supporting AARA-style resource analysis with potential-carrying closures. To demonstrate the technical challenges, we need to first zoom in on how AARA works. Most AARA-style type systems rely on *linear* or *affine* typing [66], whose (informal) intension is to disallow multiple usages of the same program variable. This makes sense because the data structure referred to by a program variable may carry potentials, while duplicated uses of a variable cause duplications of its potentials. As a consequence, every function type in an affine type system would be annotated explicitly with a *multiplicity*, i.e., $A \to^m B$, where m is an upper bound on the times the function could be applied. For example, the *curried* `append` function is actually assigned with the following more verbose type annotation:

$$\texttt{append} :: \mathit{List}(\mathit{int}^1) \to^\infty \mathit{List}(\mathit{int}^0) \to^1 \mathit{List}(\mathit{int}^0)$$

The first arrow is annotated with the multiplicity ∞, which indicates that the `append` function can be applied an arbitrary number of times. One can justify this by observing that function `append` can generate arbitrarily many closures, as long as enough potentials are provided with its first argument ℓ_1. The second arrow is annotated with the multiplicity 1, which indicates that the closure obtained by `append` ℓ_1 can be applied at most once if ℓ_1 is of type $\mathit{List}(\mathit{int}^1)$. One can justify this by observing that the closure captures ℓ_1, whose potential equals the length of ℓ_1 (denoted by $|\ell_1|$), but applying the closure to ℓ_2 would consume $|\ell_1|$ units of resource and use up all the captured potential, thus the closure should not be applied more than once.

However, the verbose type annotation for `append` does not characterize its precise resource behavior, e.g., partially applying `append` to one list ℓ_1 does not consume any resources, but the type already requires $|\ell_1|$ units. In addition, the type system determines the number of uses by its type annotation instead of by its context. Thus, a re-analysis of typing is triggered every time we put `append` into a different program context, which is rather non-compositional. As an example, the verbose type annotation shown above is *not* suitable for checking

```
let app_par = append ℓ1 in (app_par ℓ2, app_par ℓ3)
```

because the program would use `append` ℓ_1 twice. We need a different annotation like

$$\texttt{append} :: \mathit{List}(\mathit{int}^2) \to^\infty \mathit{List}(\mathit{int}^0) \to^2 \mathit{List}(\mathit{int}^0)$$

to type-check the program.

This `append` problem is also pointed out by Scherer and Hoffmann [59] but it remains incompletely solved. In their work, they studied type-based analysis of closures and proposed *open closure types*, i.e., function types annotated with type contexts to track data-flow properties of captured variables. The `append` is then possible to track potentials with an annotated type like

$$\texttt{append} :: [](x : \mathit{List}(\mathit{int}^0)) \to^\infty [x : \mathit{List}(\mathit{int}^1)](y : \mathit{List}(\mathit{int}^0)) \to^1 \mathit{List}(\mathit{int}^0)$$

where $[\Gamma](x : A) \to^m B$ uses a resource-annotated type context Γ to describe the behavior of the closure application. However, even if we adapted open closure types to AARA, the resulting type system would still be affine and face the issue of having to determine multiplicities upon function definitions.

Challenge: Go beyond Affine Typing Our key observation is the following:

Existing AARA-style type systems require affine typing because they tightly couple datatypes with potentials.

For example, the type $\mathit{List}(\mathit{int}^1)$ shown above is a resource-annotated type that couples an ordinary list type $\mathit{List}(\mathit{int})$ with a potential function that equals the length of the list. More advanced examples include polynomial-potential annotations [20], e.g., $\mathit{List}^{(q_0,q_1,q_2)}(\mathit{int})$ couples $\mathit{List}(\mathit{int})$ with the potential function $\lambda\ell.\, q_0\binom{|\ell|}{0} + q_1\binom{|\ell|}{1} + q_2\binom{|\ell|}{2}$. Resource-annotated types play an important role in the *automation* part of AARA: a type system can design a specific numerical space of resource potentials (e.g., non-negative integers or rational numbers) and then reduce the type inference to solving a system of constraints by the annotations (e.g., linear programming).

In this article, we tackle the challenge of going beyond affine typing by completely *decoupling* potentials from datatypes. Intuitively, because all datatypes become potential-free, they do not need to be affine; thus, we fall back to a standard type-system design. On the other hand, we need a mechanism to *explicitly* represent potential functions at the type level. Taking inspiration from open closure types, we introduce a dependent-arrow-like notation $[f_1]_x T_1 \to [f_2]_y T_2$, where x and y bind the argument and result respectively, T_1 and T_2 are ordinary types, and f_1, f_2 are potential functions that can reference free variables in the type context of the function definition. For example, the `append` function has the following type in our type system:

$$\texttt{append} :: (x : \mathit{List}(\mathit{int}) \to (y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x)\to 0]})_{[0\to 0]}$$

Note that we hide unnecessary binders for brevity. The type above should be interpreted as follows: the `append` function takes an argument x and it does not require any potential to create a closure which captures x. The obtained closure then takes an argument y and it requires $\mathit{length}(x)$ units of potential to return a list without potential. This type annotation has two benefits: (i) it characterizes the precise resource behavior of `append`, including the behavior of its partial application, and (ii) it does not need to determine any multiplicity and it is not an affine type at all. As a tradeoff, our type system has to support type-level computations such as $\mathit{length}(x)$ that can use program variables. Thus, our type system is a dependently-typed variant of AARA.

Challenge: Support Lightweight Dependent Typing There have been several dependent-type-based approaches for resource analysis or verification. Wang et al. [67] developed a refinement type system for complexity analysis, focusing on constraint-based analysis instead of supporting potential-based analysis like AARA. Knoth et al.[36,37] developed liquid-style type systems that integrate refinement typing with AARA, while Rajani et al. [57] proposed a dependently-typed calculus for amortized resource analysis; however, those systems still require linear or affine typing for tracking potentials. Niu et al. [51] proposed a cost-aware logical framework that can be instantiated to build different resource analyses, including amortized analysis, but it emphasizes its logical framework instead of giving a concrete resource-annotated dependent type system.

In this article, we focus on designing a lightweight AARA-style dependent type system *without* affine typing. Our type system is lightweight in the sense that its dependent part describes *only* potential functions, e.g., numeric primitive recursions over inductive datatypes. One major challenge in designing the type system is to ensure *compositional* typing, especially for dependent function applications. For example, partially applying `append` to an expression e of type $\mathit{List}(\mathit{int})$ with $\mathit{length}(e)$ units of potential would yield the following typing:

$$\texttt{append}\ e\ ::\ \exists x : \mathit{List}(\mathit{int}).\, [\mathit{length}(x)](y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x)\to 0]}$$

where the existential indicates that `append` e constructs a closure that captures a list x with $\mathit{length}(x)$ units of potential. However, managing existentials would quickly complicate the typing, especially when there are higher-order functions with different levels of existential quantifiers. But fortunately, global existential quantifiers can satisfy our need for potential analysis.

We here propose a lightweight solution that keeps a *global* potential context Ω for those existentials along with the usual type context Γ. In some sense it is similar to the idea of prenex polymorphism, which places all universal quantifiers at the outermost position. In our setting, we consider placing all existential quantifiers, which account for potentials captured in a closure, at the outermost position. Our type judgements then take the form $\Omega \mid \Gamma \mid f_1 \vdash e : [f_2]_x T$, where Ω is our global potential context, Γ is the usual type context, f_1 and f_2 are potential functions, and x is the local binder on T which can be omitted for brevity. For example, below are possible type judgements for `append` and `append` e:

$$\cdot \mid \cdot \mid 0 \vdash \texttt{append} : (x : \mathit{List}(\mathit{int}) \to (y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x)\to 0]})_{[0\to 0]}$$
$$x : \mathit{List}(\mathit{int}) \mid \cdot \mid 0 \vdash \texttt{append}\ e : [\mathit{length}(x)](y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x)\to 0]}$$

In this way, our type system sidesteps the issue of possible "pollution" of existentials while still being effective enough for AARA-style reasoning.

Contributions This article makes the following three contributions:

- We propose a lightweight dependently-typed AARA for reasoning about the resource consumption of higher-order functional programs. Our type system is the first AARA-style type system that does not require linear/affine typing and can describe the precise behavior of curried functions.
- We formalize our dependent type system and prove its soundness with respect to a cost-aware operational semantics.
- We demonstrate the effectiveness of our type system on a suite of challenging examples. We also include a discussion towards automating our type system.

2 Overview

We call our new calculus $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, where "$\mathsf{amor}$" and "$\mathsf{na}$" are short for "amortized" and "non-affine" respectively. In this section, we sketch the general idea of how $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ works and use a few motivating examples to demonstrate its difference from prior type systems for AARA-style resource analysis and verification.

2.1 Prior Work: Automatic Amortized Resource Analysis

Automatic Amortized Resource Analysis (AARA) is a technique first introduced by Hofmann and Jost [23] as a type system for deriving linear worst-case bounds on the heap-space consumption of first-order functional programs with eager evaluation strategy. As surveyed by Hoffmann and Jost [21], dozens of works extended AARA to support different resource metrics, evaluation strategies, resource bounds, and language features, making AARA a state-of-the-art methodology for resource analysis and verification. At the core of AARA is the *potential method* for amortized complexity analysis, which was proposed by Tarjan [63] to manually derive an upper bound on the resource consumption of a sequence of operations. To apply the potential method to analyze general programs, one needs to specify potential functions that map program states to non-negative numbers, such that the potential at every possible program state is sufficient to pay for the cost of the next state transition as well as the potential of the next state. AARA introduces type annotations to encode such potential functions; thus, an AARA-style type system must statically verify the type annotations in a program to ensure the correct application of the potential method.

Typical AARA typing judgements take the form $\Gamma \mid_{q}^{p} e : A$ with $p, q \geq 0$, which reads as: under type context Γ and with p units of potential, the expression e has the type A with a return of q units of potential. As we discussed in §1, the type A and those types in Γ are *resource-annotated*, e.g., $\mathit{List}(\mathit{int}^1)$ which means a list that carries one unit of potential per list element. It has been shown that the close coupling of potential functions and data structures renders AARA effective and automatable [23,31,36]. Because types carry resources, it is natural for AARA type systems to adopt *linear* or *affine* typing. Below are some canonical typing rules for AARA, where $A \xrightarrow{p/q} B$ denotes a resource-annotated function type with p/q as the pre-/post-potentials of the function:

$$\text{(AARA:VAR)}\ \frac{}{x : A \mid_{0}^{0} x : A} \qquad \text{(AARA:NIL)}\ \frac{p \geq 0}{\cdot \mid_{0}^{0} \mathit{nil} : \mathit{List}(A^p)} \qquad \text{(AARA:CONS)}\ \frac{p \geq 0}{x_h : A, x_t : \mathit{List}(A^p) \mid_{0}^{p} \mathit{cons}(x_h, x_t) : \mathit{List}(A^p)}$$

$$\text{(AARA:LET)}\ \frac{\Gamma_1 \mid_{r}^{p} e_1 : A_1 \qquad \Gamma_2, x : A_1 \mid_{q}^{r} e_2 : A_2}{\Gamma_1, \Gamma_2 \mid_{q}^{p} \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : A_2} \qquad \text{(AARA:ABS)}\ \frac{\Gamma, x : A \mid_{q}^{p} e : B \quad \Gamma \text{ does not carry any potentials}}{\Gamma \mid_{0}^{0} \lambda x.\, e : A \xrightarrow{p/q} B}$$

Rules (AARA:NIL) and (AARA:CONS) describe how to store potentials in a list; for example, to build $\mathit{cons}(x_h, x_t)$ of type $\mathit{List}(\mathit{int}^p)$ carrying p units of potential per list element, one needs p units of potential (for x_h) and a x_t that has type $\mathit{List}(\mathit{int}^p)$. Rule (AARA:LET) indicates that the type system is linear: one needs two different contexts Γ_1 and Γ_2 to check e_1 and e_2, thus forbidding e_1 and e_2 to use the same variable.[3] Rule (AARA:ABS) is one of the most non-trivial rules: the side condition (marked in blue) is essential because AARA typically treats functions as *copyable* objects, i.e., a function can be applied an arbitrary number of times. If the side condition were discarded, the function body e would then be able to consume potentials stored in Γ to pay for the

[3] This is not a severe limitation, because AARA can use *sharing* to split a variable into two, with the understanding that the carried potentials are also split.

costs inside e; thus, multiple applications of the function would consume the same piece of potentials multiple times, resulting in unsoundness. Therefore, the rules above can *not* verify the resource bound of the *curried* `append` function shown in §1, because it would require a typing judgement like $x : \mathit{List}(\mathit{int}^1) \vdash \lambda y.\, e_{\mathsf{append}} : \mathit{List}(\mathit{int}^0) \xrightarrow{0/0} \mathit{List}(\mathit{int}^0)$, where the context *does* carry potentials.

Several efforts have been made to relax the limitation of (AARA:Abs) [18,36,57]. Hoffmann et al. [18] adapt a stack-based typing principle, which essentially uncurries function applications if needed. Below shows two typing rules of Hoffmann et al. [18]'s system for function definitions, where a stack Σ is used in the typing judgement $\Sigma; \Gamma \vdash e : T$ to record all the argument types:

$$\text{(AARA:Stack:AbsPop)}\ \frac{\Sigma; \Gamma \vdash \lambda x.\, e : T}{\cdot; \Gamma \vdash \lambda x.\, e : \Sigma \to T} \qquad \text{(AARA:Stack:AbsPush)}\ \frac{\Sigma; \Gamma, x : T_1 \vdash \lambda x.\, e : T}{T_1 :: \Sigma; \Gamma \vdash \lambda x.\, e : \Sigma \to T}$$

By the interaction of the (AARA:Stack:AbsPop) and (AARA:Stack:AbsPush), one becomes able to uncurry a higher-order type like $T_1 \to \cdots \to T_n \to T$ into a bracket function type $[T_1, \cdots, T_n] \to T$. Under this approach, the limitation of (AARA:Abs) is clearly handled by introducing all the n variables at the same time, thus curried functions like `append` can have a sound type annotation like $[\mathit{List}(\mathit{int}^1), \mathit{List}(\mathit{int}^0)] \to \mathit{List}(\mathit{int}^0)$, instead of the unsound annotation $\mathit{List}(\mathit{int}^1) \to \mathit{List}(\mathit{int}^0) \to \mathit{List}(\mathit{int}^0)$. However, this approach sets us aside from one of the most powerful tools—partial application—that higher-order functions can offer.

On the other hand, Knoth et al. [36] and Rajani et al. [57] adopt an affine type system with *multiplicities*, which bound the number a function can be applied. Take Knoth et al. [36]'s system as an example, function types in their works has the form similar to $m \cdot (A \xrightarrow{p/q} B)$, where m is a non-negative integer for the multiplicity. Note that we switch to a slightly different notation for multiplicities, compared with the example shown in §1. Intuitively, $\infty \cdot (A \xrightarrow{p/q} B)$ has the same meaning of the function type justified by (AARA:Abs), whereas $1 \cdot (A \xrightarrow{p/q} B)$ is similar to the function type in a usual linear type system, in the sense that such function can only be applied once. The typing rule for function definitions could become the one shown below, where $m \cdot \Gamma$ constructs a context with the same bindings as Γ, but with m times of the potentials in Γ:

$$\text{(AARA:Abs:Multi)}\ \frac{\Gamma, x : A \left|\frac{p}{q}\right. e : B}{m \cdot \Gamma \left|\frac{0}{0}\right. \lambda x.\, e : m \cdot (A \xrightarrow{p/q} B)}$$

With the rule above, it is possible to verify the curried `append` function has the type $\infty \cdot (\mathit{List}(\mathit{int}^1) \xrightarrow{0/0} 1 \cdot (\mathit{List}(\mathit{int}^0) \xrightarrow{0/0} \mathit{List}(\mathit{int}^0)))$, which we showed in §1.

However, as we discussed in §1, the linear or affine nature of AARA type systems makes it quite rigid to handle higher-order functions. (Recall the `app_par` example in §1 which partially applies `append` to obtain a closure and later uses the closure twice.) In §2.2, we show how our $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ tackles the problem.

2.2 This Work: Non-Affine Dependently-Typed AARA

The first observation on AARA-style type systems is that the affine typing is necessary for dealing with potential-carrying types. Most prior systems neglect the desire to separate potentials from types, despite that they actually feature such a separation in some parts. For example, in the typing judgement $\Gamma \,\Big|\frac{p}{q}\, e : T$, you get p and q as input/output *constant* potentials, which stand aside from context Γ and type T. However, such a separation is not enough because potentials associated with data structures like $\mathit{List}(\mathit{int}^1)$ can *not* be expressed by a constant. To decouple potentials completely from types, in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, we extend typing judgements of the form $\Gamma \,\Big|\frac{p}{q}\, e : T$ to the form

$$\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \left[f_2\right]_x T \;,$$

which replaces constant p, q with potential functions f_1, f_2 and reads as: under potential context Ω (which we will explain later in this section), type context Γ with f_1 units of input potential, the expression e has the type T, whose result is bound by a local binder x and carries an output potential of f_2 units. Thus in our design, f_1 and f_2 will carry all the potentials used and remaining, while the potential context Ω and type context Γ carry *no* potentials.

Then, a typical typing judgement in our system

$$\cdot \mid y : \mathit{List}(\mathit{int}) \mid \mathit{length}(y) \vdash \mathsf{id}\ y : [\mathit{length}(x)]_x \mathit{List}(\mathit{int})$$

may be read as: with a variable y of a list type in the context and $\mathit{length}(y)$ units of potential, the expression $\mathsf{id}\ y$ can return a result of a list type with $\mathit{length}(x)$ units of potential, where x binds the result of the expression. The type-level potential function $\mathit{length}(\cdot)$ serves as the resource annotation $\mathit{List}(\mathit{int}^1)$ in usual AARA systems. Since the type-level $\mathit{length}(\cdot)$ function can be applied to program variables, our $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ system is naturally *dependently* typed. Thus, one would define the type-level $\mathit{length}(\cdot)$ function as an ordinary function:

```
length = λx. case x of nil ⇒ 0 | cons(x0,x1) ⇒ 1 + length x1
```

It should be noted that type-level functions should guarantee *termination*, in order to render the type system sound. There have been many techniques in the community of dependent typing, such as measures [55], well-founded recursion [62], and primitive recursion on inductive datatypes [37]. In our design of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, we do not stick to any specific design of type-level functions; instead, we focus on the idea that *what capability our type system can offer if we can carry out arithmetic reasoning on type-level potential functions*. Thereby in the rest of the section, we assume we can perform arithmetic reasoning in the type system.

Because of the dependent nature of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, arrow types become $(x : A \to y : B)_{[f_1 \to f_2]}$ with binders x and y, which bind the argument and result and can be referred to in potential functions f_1 and f_2. It is now possible to describe the curried append's behavior in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ as follows, where $matd()$ stands for the case analysis for inductive datatypes adopted from Knoth et al. [37]'s work:

$$\begin{aligned} T_{\mathsf{append}} &\overset{\text{def}}{=} (x : List(int) \to ((y : List(int) \to List(int))_{[length(x) \to 0]}))_{[0 \to 0]} \\ \mathsf{append} &\overset{\text{def}}{=} \ fix\ \mathsf{append} : T_{\mathsf{append}}.\ \lambda x.\ \lambda y.\ matd(x, \{nil(x_0).\ y, cons(x_0, x_1). \\ &\qquad\qquad tick\ 1\ (cons(x_0, (\mathsf{append}\ x_1\ y)))\}) \end{aligned}$$

Notably, such a type precisely characterizes the cost behavior of the *curried* append: applying append with only one argument x would not consume any resources, and further applying the obtained closure with a second argument y would consume $length(x)$ units of resource, where x is the captured first argument. The key to this precise characterization is that we let the outermost binder in lambda abstraction capture all the potentials in the context:

$$\text{(TAbs)}\ \frac{\Omega \mid \Gamma, x : T_1 \mid f_1 \vdash e : [f_2]_y T_2 \qquad z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \mid \Gamma \mid 0 \vdash \lambda x : T_1 . e : [0]_z (x : T_1 \to y : T_2)_{[f_1 \to f_2]}}$$

While this rule may look similar to previously shown rules (AARA:Abs) or (AARA:Abs:Multi), the exception is our (TAbs) does not require potential-free context restrictions or explicit multiplicities. This is only viable in our $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ because all the contexts Ω and Γ are guaranteed to carry 0 potentials, and all the required potentials can be expressed and captured in the f_1 part. This feature enables us to do *compositional* reasoning about closures and partial applications, but yet a bit more elaboration on lambda applications is needed next.

The second observation on AARA-style type systems is that the *multiplicity* is a remedy for the imprecise characterization of function behaviors. Take the linear type $\infty \cdot (List(int^1) \to 1 \cdot (List(int^0) \to List(int^0)))$ for append as an example. The first argument requires $length(\cdot)$ units of potential in advance while the consumption actually happens after the second application. Thus a restriction of multiplicity 1 is imposed on the second arrow because the first argument only ask for potentials that is enough for 1 time of second application. In practice, one may *not* always be able to decide the number of usages in advance, thus multiplicity results in *non-compositionality*: you have to change the type annotation (especially the multiplicity) by the actual context involved.

In our approach, we remove multiplicities to achieve *compositionality*, but this removal does not come for free, especially in the case of lambda-applications. Before proceeding to the concrete typing rule, let's take an example to illustrate the challenge. Suppose we want to subject an expression e_1 of type $List(int)$ with $4 \times length(\ell_1)$ units of potential—where ℓ_1 binds the evaluation result of e_1—to the function append. We notice that in the first arrow of T_{append}, the type requires 0 units of potential. Thus, we need to keep the $4 \times length(\ell_1)$ units of potential for later use because there is no *multiplicity* for us to restrict the number of usage of the second arrow. In other words, append e_1 results in a *potential-capturing* closure:

$$\frac{\cdot \mid \cdot \mid 0 \vdash \mathsf{append} : [0] T_{\mathsf{append}} \qquad \cdot \mid \cdot \mid p_1 \vdash e_1 : [4 \times length(\ell_1)]_{\ell_1} List(int)}{\cdot \mid \cdot \mid p_1 \vdash \mathsf{append}\ e_1 : [4 \times length(\ell_1)] (y : List(int) \to List(int))_{[length(x) \to 0]}}\ \text{(TApp?)}$$

However, the typing judgement shown above is problematic: the variables ℓ_1 and x used in the result type are *not* bound. Intuitively, the application should be safe to proceed. One possible workaround—adapted by Knoth et al. [36,37]—is to require programs to be in *A-Normal-Form* (ANF); that is, the application arguments must be a variable or a value. For example, if e_1 is a variable z that can be located in the context, we can substitute both ℓ_1 and x with z to make the type valid. Another approach is to introduce *existential* types [55,4]; for example, even if e_1 is not a variable or a value, one can assign $\mathsf{append}\ e_1$ with the type $\exists z : \mathit{List}(\mathit{int}).\ \big[4 \times \mathit{length}(z)\big](y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(z) \to 0]}$, which uses an existential binder z for the evaluation result of e_1.

In this work, we aim to target a more flexible language, thus we do not want to restrict our language to allow only ANF programs. On the other hand, introducing existential binders would complicate the type system, which in itself is worth a separate study (as shown by Borkowski et al. [4]). As mentioned in §1, in our $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ system, we propose a lightweight approach: we introduce a *global potential context* Ω to keep track of those existentially-bound potential-carrying variables. Below shows a derivation for the partial application $\mathsf{append}\ e_1$ in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$:

$$\frac{\cdot \mid \cdot \mid 0 \vdash \mathsf{append} : \big[0\big]\, T_{\mathsf{append}} \qquad \cdot \mid \cdot \mid p_1 \vdash e_1 : \big[4 \times \mathit{length}(\ell_1)\big]_{\ell_1} \mathit{List}(\mathit{int})}{\begin{array}{c} x : \mathit{List}(\mathit{int}) \mid \cdot \mid p_1 \vdash \mathsf{append}\ e_1 : \\ {[4 \times \mathit{length}(x)]}(y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]} \end{array}}\ (\textsc{TApp})$$

As you can see, a substitution of ℓ_1 with x is triggered and the global potential context is extended with a binding of x. The typing judgement reads as follows: after the application of append on e_1 carrying $4 \times \mathit{length}(\ell_1)$ units of potential with ℓ_1 bound to e_1's result, ℓ_1 is then instantiated to the variable x while x is introduced as an existential variable to the global potential context. We can further apply the result of $\mathsf{append}\ e_1$ to another expression e_2 of type $\mathit{List}(\mathit{int})$ with no potentials; the typing derivation is shown below:

$$\cfrac{\cfrac{\begin{array}{l} x : \mathit{List}(\mathit{int}) \mid \cdot \mid p_1 \vdash \mathsf{append}\ e_1 : \\ \quad [4 \times \mathit{length}(x)](y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]} \\ \cdot \mid \cdot \mid p_2 \vdash e_2 : \big[0\big]\, \mathit{List}(\mathit{int}) \end{array}}{\begin{array}{l} x : \mathit{List}(\mathit{int}),\ y : \mathit{List}(\mathit{int}) \mid \cdot \mid p_1 + p_2 \vdash \mathsf{append}\ e_1\ e_2 : \\ \quad [3 \times \mathit{length}(x)]\mathit{List}(\mathit{int}) \end{array}}\ (\textsc{TApp})}{x : \mathit{List}(\mathit{int}) \mid \cdot \mid p_1 + p_2 \vdash \mathsf{append}\ e_1\ e_2 : \big[3 \times \mathit{length}(x)\big]\, \mathit{List}(\mathit{int})}\ (\textsc{TErase})$$

Similar to the previous use of the (TApp) rule, the application of $\mathsf{append}\ e_1$ to e_2 introduces another existential binding $y : \mathit{List}(\mathit{int})$. The difference is that now the application needs $\mathit{length}(x)$ units of potential to proceed. Because the closure returned by $\mathsf{append}\ e_1$ carries $4 \times \mathit{length}(x)$ units of potential, it is sound to make the application and leave $3 \times \mathit{length}(x)$ units of potential to the next. In the derivation, we also demonstrate the use of the (TErase) rule, which removes unnecessary existentially-bound variables from the global potential context. In this example, y is no longer needed so we can safely remove it.

We can now show $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s typing rule for lambda applications:

(TApp)
$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_1 : \left[f_2\right]_z((x : T_1 \to y : T_2)_{[f_3 \to f_4]}) \\ \Omega \mid \Gamma \mid f_5 \vdash e_2 : \left[f_6\right]_w T_1 \qquad \Omega, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \leq (f_2 + f_6)[w \mapsto x]\end{array}}{\Omega, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash e_1\ e_2 : \left[(f_2 + f_6 - f_3)[w \mapsto x] + f_4\right]_y T_2}$$

The first two premises include the typing judgements for e_1 and e_2, respectively. Then for the application, we need to instantiate all the potential associated with local binder w for e_2, to the local binder x for the argument type in e_1. The third premise checks whether the potentials carried by e_1 and e_2 are sufficient for the application or not. Finally, we store the unused potentials in $(f_2 + f_6 - f_3)[w \mapsto x] + f_4$, and add an existential binder x to the global potential context. In this way, we can derive $x : \mathit{List}(\mathit{int}), y : \mathit{List}(\mathit{int}) \mid \cdot \mid p_1 + p_2 \vdash \mathsf{append}\ e_1\ e_2 : \left[3 \times \mathit{length}(x)\right] \mathit{List}(\mathit{int})$ as shown earlier in this section.

With the lambda-abstraction rule and application rule provided, we can finally derive a typing judgement for the curried recursive function `append`. Let

$$\Gamma_0 \overset{\text{def}}{=} \mathsf{append} : T_{\mathsf{append}},\ x : \mathit{List}(\mathit{int}),\ y : \mathit{List}(\mathit{int})\,,$$
$$\Gamma_1 \overset{\text{def}}{=} \Gamma_0,\ x_0 : \mathit{int},\ x_1 : \mathit{List}(\mathit{int})\,.$$

We have the following derivation for the sub-expression that corresponds to the *cons* case. The notable thing here is how the potential is carried along with the sequential applications of x_1 and y:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\cdot \mid \Gamma_1 \mid 0 \vdash \mathsf{append} : \left[0\right] T_{\mathsf{append}} \\ \cdot \mid \Gamma_1 \mid \mathit{length}(x_1) \vdash x_1 : \left[\mathit{length}(x_1)\right]_{x_1} \mathit{List}(\mathit{int})\end{array}}{\begin{array}{c}x : \mathit{List}(\mathit{int}) \mid \Gamma_1 \mid \mathit{length}(x_1) \vdash \mathsf{append}\ x_1 \\ : \left[\mathit{length}(x)\right] (y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]}\end{array}}\ (\textsc{TApp})}{x : \mathit{List}(\mathit{int}), y : \mathit{List}(\mathit{int}) \mid \Gamma_1 \mid \mathit{length}(x_1) \vdash \mathsf{append}\ x_1\ y : [0] \mathit{List}(\mathit{int})}\ (\textsc{TApp})}{\cdot \mid \Gamma_1 \mid \mathit{length}(x_1) \vdash \mathsf{append}\ x_1\ y : \left[0\right] \mathit{List}(\mathit{int})}\ (\textsc{TErase})}{\cdot \mid \Gamma_1 \mid \mathit{length}(x_1) \vdash \mathit{cons}(x_0, (\mathsf{append}\ x_1\ y)) : \left[0\right] \mathit{List}(\mathit{int})}\ (\textsc{TCons})}{\cdot \mid \Gamma_1 \mid \mathit{length}(x_1) + 1 \vdash \mathit{tick}\ 1\ \mathit{cons}(x_0, (\mathsf{append}\ x_1\ y)) : \left[0\right] \mathit{List}(\mathit{int})}\ (\textsc{TTickP})$$

We then proceed to the case analysis as well as the fixed-point definition:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\cdots \qquad \cdot \mid \Gamma_0, x_0 : \mathit{unit} \mid 0 \vdash y : \left[0\right] \mathit{List}(\mathit{int}) \\ \cdot \mid \Gamma_0 \mid \mathit{length}(x) \vdash x : \left[\mathit{length}(x)\right]_x \mathit{List}(\mathit{int})\end{array}}{\cdot \mid \Gamma_0 \mid \mathit{length}(x) \vdash \mathit{matd}(\cdots) : \left[0\right] \mathit{List}(\mathit{int})}\ (\textsc{TDes})}{\begin{array}{c}\cdot \mid \mathsf{append} : T_{\mathsf{append}},\ x : \mathit{List}(\mathit{int}) \mid 0 \vdash \lambda y.\ \cdots : \\ [0]((y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]})\end{array}}\ (\textsc{TAbs})}{\cdot \mid \mathsf{append} : T_{\mathsf{append}} \mid 0 \vdash \lambda x.\,\lambda y.\ \cdots : \left[0\right] T_{\mathsf{append}}}\ (\textsc{TAbs})}{\cdot \mid \cdot \mid 0 \vdash \mathit{fix}\ \mathsf{append} : T_{\mathsf{append}}.\,\lambda x.\,\lambda y.\ \cdots : \left[0\right] T_{\mathsf{append}}}\ (\textsc{TFix})$$

Let us conclude using the higher-order example `app_par` introduced in §1. We can reimplement `app_par` as follows using lambda abstraction and application:

$$(\lambda z.\,(z\ \ell_2, z\ \ell_3))\ (\mathsf{append}\ \ell_1)$$

To show the capability of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ handling higher-order functions, we let ℓ_1 carrying $2 \times \mathit{length}(\ell_1)$ units of potential, while ℓ_2 and ℓ_3 carry no potentials. Let T_z be $(y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]}$, i.e., the type that the argument z is supposed to have. The typing derivation for the $\lambda z. \cdots$ part in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ is presented below:

$$\dfrac{\dfrac{\dfrac{x : \mathit{List}(\mathit{int}) \mid z : T_z \mid 0 \vdash z : \boxed{0}\, (y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]}}{x : \mathit{List}(\mathit{int}) \mid z : T_z \mid \mathit{length}(x) \vdash z\ \ell_2 : \boxed{0}\, \mathit{List}(\mathit{int})}\ (\text{TApp})}{x : \mathit{List}(\mathit{int}) \mid z : T_z \mid 2 \times \mathit{length}(x) \vdash (z\ \ell_2, z\ \ell_3) : \boxed{0}\, \mathit{List}(\mathit{int}) \times \mathit{List}(\mathit{int})}\ (\text{TPair})}{x : \mathit{List}(\mathit{int}) | \cdot | 0 \vdash \lambda z. (z\ \ell_2, z\ \ell_3) : [0](z : T_z \to \mathit{List}(\mathit{int}) \times \mathit{List}(\mathit{int}))_{[2 \times \mathit{length}(x) \to 0]}}\ (\text{TAbs})$$

Next, we derive a typing judgement for the partial application $\mathsf{append}\ \ell_1$:

$$\dfrac{\cdot \mid \cdot \mid p \vdash \ell_1 : \boxed{2 \times \mathit{length}(l)}_l\, \mathit{List}(\mathit{int})}{\begin{array}{c} x : \mathit{List}(\mathit{int}) \mid \cdot \mid p \vdash \mathsf{append}\ \ell_1 : \\ [2 \times \mathit{length}(x)](y : \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to 0]} \end{array}}\ (\text{TApp})$$

Finally, we can compositionally reason about the resource consumption of the `app_par` example:

$$\dfrac{\dfrac{\cdots \qquad \cdots}{x : \mathit{List}(\mathit{int}) | \cdot | p \vdash (\lambda z. (z\ \ell_2, z\ \ell_3))\ (\mathsf{append}\ \ell_1) : [0] \mathit{List}(\mathit{int}) \times \mathit{List}(\mathit{int})}\ (\text{TApp})}{\mid \cdot \mid p \vdash (\lambda z. (z\ \ell_2, z\ \ell_3))\ (\mathsf{append}\ \ell_1) : \boxed{0}\, \mathit{List}(\mathit{int}) \times \mathit{List}(\mathit{int})}\ (\text{TErase})$$

That is, p is an upper bound on the resource consumption if p units of potential are sufficient to build the list ℓ_1 with $2 \times \mathit{length}(\ell_1)$ units of potential.

3 Technical Details

In this section, we will give an overall description of the syntax and semantics of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, along with sketches of the type soundness proof under cost semantics. While examples and automation of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ will be later introduced at §4 and §5.

3.1 Syntax

$$\begin{array}{lrl}
\textsc{Types} & T ::= & \mathit{int} \mid T_1 \times T_2 \mid (x : T_1 \to y : T_2)_{[f_1 \to f_2]} \\
 & & \mid \forall x : T_1. T_2 \mid \mathit{ind}(\overrightarrow{C, (T, m)}) \\
\textsc{Expression} & e ::= & x \mid i \mid \mathbf{op_i}(\vec{e}) \mid \Lambda x : T. e \mid \lambda x : T. e \mid e_1\ e_2 \mid (e_1, e_2) \\
 & & \mid \pi_1\ e \mid \pi_2\ e \mid \mathit{fix}\ x : T. e \mid \mathit{tick}\ i\ e \mid\ \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \\
 & & \mid C_i(e_0, (e_1, ..., e_{m_i})) \mid \mathit{matd}(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e)}) \\
\textsc{Pre-Values} & \mathit{pv} ::= & x \mid i \mid \mathbf{op_i}(\vec{\mathit{pv}}) \mid \Lambda x : T. e \mid \lambda x : T. e \mid (\mathit{pv}_1, \mathit{pv}_2) \\
 & & \mid C_i(\mathit{pv}_0, (\mathit{pv}_1, ..., \mathit{pv}_{m_i})) \\
\textsc{Values} & v ::= & i \mid \Lambda x : T. e \mid \lambda x : T. e \mid (v_1, v_2) \mid C_i(v_0, (v_1, ..., v_{m_i})) \\
\textsc{Type Context} & \Gamma ::= & \cdot \mid \Gamma, x : T \\
\textsc{Potential Context} & \Omega ::= & \cdot \mid \Omega, x : T
\end{array}$$

Types The types and terms in our calculus are a combination of AARA [23] and Liquid Resource Type [37]. We choose *int* to serve as basic types to do arithmetic reasoning, while other resource types such as **R** work just as well in our system. $T_1 \times T_2$ and $\forall x : T_1 . T_2$ are standard product types and dependent types. Our arrow type $(x : T_1 \to y : T_2)_{[f_1 \to f_2]}$ is a generalization from AARA arrow type $A \xrightarrow{p/q} B$. f_1 and f_2 are the generalized potential resource input/output from p and q, while x and y specify potential variables bound on the input and output potential function f_1 and f_2. Here f_1, f_2, T_1 and T_2 can depend on x while only f_2 and T_2 can depend on y. $ind(\overrightarrow{C, (T, m)})$ is a simplified inductive data type adapted from prior work [37,18]. C stands for the name of the constructor, T is the type of content, and m is the number of copies of the inductive type itself. Such as $List(int)$ is represented as $List(int) = ind(\{nil(Unit, 0), cons(int, 1)\})$, and $Tree(int)$ is represented as $Tree(int) = ind(\{leaf(int, 0), node(Unit, 2)\})$.

(PConst)
$$\frac{\Omega \cap \Gamma = \emptyset}{\Omega \mid \Gamma \vdash c}$$

(PInt)
$$\frac{\Omega \cap \Gamma = \emptyset \qquad x : int \in \Omega \cup \Gamma}{\Omega \mid \Gamma \vdash x}$$

(POp-Linear)
$$\frac{\begin{array}{c}\textbf{op is linear}\\ \forall i,\ \Omega \mid \Gamma \vdash f_i\end{array}}{\Omega \mid \Gamma \vdash \mathbf{op}(\overrightarrow{f_i})}$$

(POp-Non-Linear)
$$\frac{\begin{array}{c}\textbf{op is non linear}\\ \forall i,\ \Omega_i \mid \Gamma_i \vdash f_i\\ \forall i, j,\ i \neq j \to \mathsf{dom}(\Omega_i) \cap \mathsf{dom}(\Gamma_j) = \mathsf{dom}(\Omega_i)\\ \cap \mathsf{dom}(\Omega_j) = \mathsf{dom}(\Gamma_i) \cap \mathsf{dom}(\Gamma_j) = \emptyset\end{array}}{\bigcup_i \Omega_i \mid \bigcup_i \Gamma_i \vdash \mathbf{op}(\overrightarrow{f_i})}$$

(PPair)
$$\frac{\begin{array}{c}\forall i \in [1,2], x_i \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)\\ x : T_1 \times T_2 \in \Omega \cup \Gamma, \qquad \forall i \in [1,2],\\ \Omega \backslash x,\ x_i : T_i \mid \Gamma \backslash x \vdash f_i[x \mapsto (x_1, x_2)]\end{array}}{\Omega \mid \Gamma \vdash \sum_i f_i}$$

(PCons)
$$\frac{\begin{array}{c}\forall i, \forall j \in [1, m_i], x_{ij} \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \qquad x : ind(\overrightarrow{C, (T, m)}) \in \Omega \cup \Gamma\\ \forall i, \forall j \in [1, m_i], \Omega \backslash x,\ x_{ij} : T_{ij},\ f : int \mid \Gamma \backslash x \vdash f_{ij}[x \mapsto C_i(x_0, (x_1, ..., x_{m_i}))]\end{array}}{\Omega \mid \Gamma \vdash fix\ f.\ matd(x, \overrightarrow{C(x_0, (x_1, ..., x_{m_i}).\ \sum_{j \in m_i} f_{ij}})}$$

Fig. 1: Potential Functions

Expressions Our expressions are based on the standard lambda calculus (including type abstraction, fixpoints, and product types), extended with cost semantics and recursive data structures. We annotate lambda abstractions $\lambda x : T.\, e$ with the argument type T to enable an algorithmic typing version, as discussed in §5. However, these annotations can be omitted in practice, as all typing, evaluation, and soundness results hold regardless of their presence. Indeed, we omit them in the presentation of examples for readability. The cost construct $tick\ i\ e$ expresses the consumption or release of i units of resource before evaluating the expression e—consuming resources when i is positive and releasing them when i is negative. A typical example of resource release is memory deallocation: once a program releases ownership of a block of memory, that portion of the resource becomes available for reuse, which corresponds to $tick\ -1$. For recursive datatypes, the term $C_i(e_0, (e_1, ..., e_{m_i}))$ denotes the i-th constructor, and $matd(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e})$ represents a case analysis over the constructors of datatype C applied to expression e. We also introduce the arithmetic operation $\mathbf{op_i}(\overrightarrow{e})$ to support the definition and computation of potential functions.

Values and Pre-Values Values are the normal forms that cannot be further reduced. Our values are standard, but we also include pre-values in our calculus. Pre-values are values with variables and arithmetic operations. We need these pre-values because we want to subject some potential functions using the potential abstraction $\Lambda X : T. e$. Dependent type systems with fix-points will not be decidable, so here we subject the restricted forms as pre-values into potential polymorphism $\Lambda X : T. e$, as shown in the typing rule (TPapp) rule in Fig. 2.

Type Context and Potential Context As shown in the §1 and §2, there are two kinds of contexts, type context Γ and potential context Ω. Type context Γ captures bound variables in the λ abstraction and fix-points, while potential context Ω captures existential global variables to bind potential functions. All the variables used in expressions or potentials function should appear in either type context or potential context, and common variables in type context and potential context should share the same type.

Potential functions Potential functions f are introduced to decouple potentials from types, especially from recursive data types. In our context, potential functions are just the primitive recursions that map pre-values to int. There is a set of rules associated with potential functions. $\Omega \mid \Gamma \vdash f$ is the well-formedness judgements, requiring all the variables in f appears in the context Ω or Γ. Besides, we have the inequality judgement $\Omega \mid \Gamma \vdash f_1 \leq f_2$ is the inequality judging, requires $f_1 \leq f_2$ under all instantiation of free variables by values v, to be solved by Linear Arithmetic Solver (with only linear operations), or other SMT solvers (with non-linear operations). The primitive recursion under our context only proceeds through product types $T_1 \times T_2$ and inductive types $ind(\overrightarrow{C, (T, m)})$, but have constant values on other types, similar to the *measures* used in some liquid type systems [37,55].

3.2 Typing Rules

Next we can introduce the type system of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$. The typing judgement $\Omega \mid \Gamma \mid f_1 \vdash e : [f_2]_x T$ reads as follows, under the potential context Ω, typing context Γ and with an input potential function f_1, we can have expression

e as type T, with an output potential function f_2. Besides, the output potential function f_2 has a local binder x on the type T, suggesting f_2 and T can depend on x. We sometimes omit the local binder x if x does not appear in f_2 and T, but here all the local binders are explicitly provided in the typing rules. Local binders can be variables that already appears in the context, as rule (TVar) shows, but for the other cases, we use a fresh variable to bind the result. The most important thing to notice in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s typing is the interaction between input/output potential functions, type structures and local binders.

$$\text{(TTickP)}\ \frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2]_x T \qquad p \geq 0}{\Omega \,|\, \Gamma \,|\, f_1 + p \vdash \mathit{tick}\ p\ e : [f_2]_x T} \qquad \text{(TTickN)}\ \frac{\Omega \,|\, \Gamma \,|\, f_1 - p \vdash e : [f_2]_x T \qquad p < 0}{\Omega \,|\, \Gamma \,|\, f_1 \vdash \mathit{tick}\ p\ e : [f_2]_x T}$$

$$\text{(TInt)}\ \frac{x \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \,|\, \Gamma \,|\, [0] \vdash i : [0]_x \mathit{int}} \qquad \text{(TDrop)}\ \frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2]_x T \qquad \Omega,\, x : T \,|\, \Gamma \vdash f_3 \leq f_2}{\Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_3]_x T} \qquad \text{(TFix)}\ \frac{z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \qquad x \notin \textbf{typefv}(e) \cup \textbf{fv}(T) \qquad \Omega \,|\, \Gamma,\, x : T \,|\, 0 \vdash e : [0]_y T}{\Omega \,|\, \Gamma \,|\, 0 \vdash \mathit{fix}\ x : T.\, e : [0]_z T}$$

$$\text{(TRename)}\ \frac{x, y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \qquad \Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2]_x T}{\Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2[x \mapsto y]]_y T[x \mapsto y]} \qquad \text{(TProj1)}\ \frac{y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \qquad \Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2]_x T_1 \times T_2 \qquad \Omega,\, x : T_1 \times T_2 \,|\, \Gamma \vdash f_3[y \mapsto \pi_1 x] \leq f_2}{\Omega \,|\, \Gamma \,|\, f_1 \vdash \pi_1 e : [f_3]_y T_1}$$

$$\text{(TRelax)}\ \frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash e : [f_2]_x T \qquad \Omega \,|\, \Gamma \vdash f_3 \geq 0}{\Omega \,|\, \Gamma \,|\, f_1 + f_3 \vdash e : [f_2 + f_3]_x T} \qquad \text{(TVar)}\ \frac{x : T \in \Gamma}{\Omega \,|\, \Gamma \,|\, 0 \vdash x : [0]_x T}$$

Fig. 2: Typing Rules (Selected, full in Appendix)

Most of rules like rule (TInt), rule (TPair), or rule (TPabs) are rather common since they do not involve potential changes. In Fig. 2, Rule (TTickp) and rule (TTickn) demonstrate the different behavior of consuming and releasing p units of resource into the context. Rule (TVar) allows you to use variables non-affinely but set a local binder that has the same name of the variable while potentials can be assigned later with the rule (TRelax) or rule (TDrop). Rule (TRelax) shows an increase on both the input and output resource by an equal amount is possible. Rule (TDrop) suggests the output resource can always be dropped without any side effects. Rule (TFix) shows a standard rule for fixpoints, except we prohibit appearance of x instead the types and potentials of e to avoid self pointing. Finally rule (TErase) helps erase unnecessary variables in potential contexts, while (TRename) helps rename local binder that does not appear in context.

(TAbs)

$$\frac{\Omega \mid \Gamma,\, x : T_1 \mid f_1 \vdash e : \left[f_2\right]_y T_2 \qquad z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \mid \Gamma \mid 0 \vdash \lambda x : T_1 .\, e : \left[0\right]_z ((x : T_1 \to y : T_2)_{[f_1 \to f_2]})}$$

(TApp)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_1 : \left[f_2\right]_z ((x : T_1 \to y : T_2)_{[f_3 \to f_4]}) \\ \Omega \mid \Gamma \mid f_5 \vdash e_2 : \left[f_6\right]_w T_1 \qquad \Omega,\, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \le (f_2 + f_6)[w \mapsto x]\end{array}}{\Omega,\, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash e_1\ e_2 : \left[(f_2 + f_6 - f_3)[w \mapsto x] + f_4\right]_y T_2}$$

(TLet)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_1 : \left[f_2\right]_z T_1 \\ \Omega \mid \Gamma,\, x : T_1 \mid f_3 \vdash e_2 : \left[f_4\right]_y T_2 \qquad \Omega \mid \Gamma,\, x : T_1 \vdash f_3 \le f_2[z \mapsto x]\end{array}}{\Omega,\, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash let\ x = e_1\ in\ e_2 : \left[f_2[z \mapsto x] - f_3 + f_4\right]_y T_2}$$

(TCons)

$$\frac{\begin{array}{c} y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash e_0 : \left[f_2\right]_{x_0} T_i \qquad \forall j \in [1, m_i], \Omega \mid \Gamma \mid f_{3_j} \vdash e_j : \left[f_{4_j}\right]_{x_j} ind\overrightarrow{(C, (T, m))} \\ \Omega,\, x_0 : T_i, ...,\, x_{m_i} : ind\overrightarrow{(C, (T, m))} \mid \Gamma \vdash f_5[y \mapsto C_i(x_0, (x_1, ..., x_{m_i}))] \le f_2 + \sum_{j=1}^{m_i} f_{4_j}\end{array}}{\Omega \mid \Gamma \mid f_1 + \sum_{j=1}^{m_i} f_{3_j} \vdash C_i(e_0, (e_1, ..., e_{m_i})) : \left[f_5\right]_y ind\overrightarrow{(C, (T, m))}}$$

(TDes)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_0 : \left[f_2\right]_x ind\overrightarrow{(C, (T, m))} \qquad \forall i, \forall j \in [1, m_i], x_j \notin \mathbf{fv}(f_3) \cup \mathbf{fv}(T_1) \\ \forall i, \Omega \mid \Gamma,\, x_0 : T_i, ...,\, x_{m_i} : ind\overrightarrow{(C, (T, m))} \mid f_2[x \mapsto C_i(x_0, (x_1, ..., x_{m_i}))] \vdash e_i : \left[f_3\right]_y T_1\end{array}}{\Omega \mid \Gamma \mid f_1 \vdash matd(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e)}) : \left[f_3\right]_y T_1}$$

Fig. 3: Typing Rules (Selected, full in Appendix)

Followed by the second set of typing rules in Fig. 3, the most notable one is rule (TAbs) and rule (TApp). For rule (TAbs), we pack all the potential function f_1 at this layer into the abstraction $(x : T_1 \to y : T_2)_{[f_1 \to f_2]}$. This design is only possible under the separation of types and potentials, and helps us gain *compositionality* in reasoning about higher-order functions. For Rule (TApp), there is an instantiation from w to x because e_2 will be the actual x in the context. When doing an application, we check whether the potential f_2 carried by the function itself and the potential f_6 carried by the parameter suffices or not, and then put the residual potential into the result. The rule (TLet) is analogous to a combination of (TAbs) and rule (TApp). The rule (TCons) and rule (TDes) work just as rules (TPair), (TProj1) and (TProj2). The immediate consequence of such typing rules is that the AARA type system is embedable in our system, where we delayed the proof in Appendix.

Theorem 1 (AARA embedding). *For $\Gamma \mid_{q}^{p} e : A$, with $Pure(\cdot)$ maps a AARA type context to a potential free $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ type context, $\Phi(\cdot)$ maps a AARA type context to the potential it carries, $h(\cdot)$ a non-trivial mapping from a AARA term to a $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ term, we have that there exists a potential function $\Phi'_x(A)$ such that $Pure(\Gamma) \mid \cdot \vdash q + \Phi_x(A) \leq \Phi'_x(A)$ and*

$$Pure(\Gamma) \mid \cdot \mid \Phi(\Gamma) + p \vdash h(e) : \left[\Phi'_x(A)\right]_x Type(A).$$

3.3 Operational Semantics

The evaluation rule $e_1 \mid p \hookrightarrow e_2 \mid q$ follows the line of resource-aware small-step semantics [36,37,14,67] without changing anything in substance, with full version in Appendix. The evaluation contexts contain constant potentials p and q, instead of using the potential function f_1 and f_2. The only restriction for potential p and q in the context is they have to be non-negative. Among all the terms, the only operation consumes potential here is the *tick* p e, while other operations just pass the potential to the next. With the potential p and q in the evaluation context, and input potential function f_1 in the typing judgement, now we can set up our soundness lemma.

3.4 Soundness

Next, we are going to prove the type soundness of the calculus. The proof for progress is standard since the only evaluation rule that consume potential is the *tick* i e, while other cases either do a substantial reduction like rule (EProj1) or rule (EProj2), or do a substructural reduction like rule (EPair1) and rule (EPair2). For substantial reductions other than *tick* i e, they do not consume potential thus can always proceed. For substructural reductions, everything needed is a weakening lemma. Here we leave the full proof in Appendix.

Lemma 1 (Progress Weakening). *If $e \mid p \hookrightarrow e' \mid q$, and $p \leq p'$, then there exists q' such that $e \mid p' \hookrightarrow e' \mid q'$*

Theorem 2 (Progress). *If $\Omega \mid \cdot \mid f \vdash e : \left[g\right]_x T$ and $\Omega \mid \cdot \vdash f \leq p$, then e is a value or exists e' q, $e \mid p \hookrightarrow e' \mid q$.*

The preservation proof is technically more involved than the relatively straightforward progress theorem. The difficulty arises from the fact that variables in the potential context Ω are *erased* as evaluation proceeds. As illustrated in the example in §2, suppose v_1 and v_2 are both values, and the function append captures a variable x. Then we may derive a typing judgment of the form:

$$x : List(int) \mid \cdot \mid p_1 + p_2 \vdash \mathsf{append}\ v_1\ v_2 : \left[3 \times length(x)\right] List(int)$$

However, after evaluation, the result $v_1 +\!\!+\ v_2$ no longer refers to x. We would instead expect the result typing to be:

$$\cdot \mid \cdot \mid p_1 + p_2 \vdash v_1 +\!\!+\ v_2 : [3 \times \mathit{length}(v_1)]\, \mathit{List}(\mathit{int})$$

To establish preservation between such pre- and post-evaluation typings, we must relate the two potential contexts. The insight is that evaluation continues to substitute program variables with values in function applications. In this example, the variable x captured by `append` is eventually instantiated with the value v_1. In our system, such substitution applies not only to term variables but also to existential variables inside potential functions. Preservation is established via continuous instantiation of potential variables. Concretely, substituting x with v_1 in the potential expression ensures a valid post-evaluation typing.

To formalize this idea, we introduce the substitution notation $\Omega[x \mapsto v]$, which denotes replacing all occurrences of x with v in Ω and removing the binding $x : T$. This substitution mechanism allows us to formally state and prove the preservation theorem, which can be found in Appendix. Please note that this preservation lemma also applies to the cases where no substitution happens, where we can use a fresh variable y so that $\Omega[y \mapsto v] = \Omega$, $g[y \mapsto v] = g$, $T[y \mapsto v] = T$ and $(q - f)[y \mapsto v] = q - f$.

Lemma 2 (Value Substitution). *If* $\Omega \mid \Gamma_1, x : T_0, \Gamma_2 \mid f_1 \vdash e : [f_2]_y T_1$, *and* $\Omega \mid \Gamma_1 \mid 0 \vdash v : [0]\, T_0$, *then* $\Omega[x \mapsto v] \mid \Gamma_1, \Gamma_2[x \mapsto v] \mid f_1[x \mapsto v] \vdash e[x \mapsto v] : [(f_2[x \mapsto v])]_y T_1[x \mapsto v]$.

Theorem 3 (Preservation). *If* $\Omega \mid \cdot \mid f \vdash e : [g]_x T$ *and* $e \mid q \hookrightarrow e' \mid q'$, *then there exists* $f'\ y\ v$, *such that* $\Omega[y \mapsto v] \mid \cdot \mid f' \vdash e : [g[y \mapsto v]]_x T[y \mapsto v]$, *and we have that* $\Omega[y \mapsto v] \mid \cdot \vdash (q - f)[y \mapsto v] \leq q' - f'$.

4 Case Studies

In this section, we make remarks on the generality of our approach, and show the inference process for several useful examples. We have shown the complete inference process for `append` and `app_par` in §2, and next are `traverse`, `curry`, `sort` and `map_append`. Due to the limit of space, we put `curry`, `sort` in the Appendix and show the inference process of `traverse` and `map_append` here.

4.1 List Traverse

We can first take a simple example: the traversal of $\mathit{List}(\mathit{int})$. The memory consumption of `traverse` accumulates as it proceeds through the list, and it will release the memory once the traversal is finished.

$$\begin{aligned}
\textsf{traverse} \quad &:: \quad (x : \mathit{List}(\mathit{int}) \to y : \mathit{List}(\mathit{int}))_{[\mathit{length}(x) \to \mathit{length}(y)]} \\
\textsf{traverse} \quad &\overset{\text{def}}{=} \quad \mathit{fix}\ \textsf{traverse} : x.\, \mathit{matd}(x, \{\mathit{nil}(x_0).\mathit{nil}(x_0), \\
&\qquad \mathit{cons}(x_0, x_1).\mathit{tick}\ 1\ (\mathit{lazy\_tick}\ (-1)\ \mathit{cons}(x_0, \textsf{traverse}\ x_1))\})
\end{aligned}$$

We apply an eager semantics for our $tick\ p\ e$, that is: consuming p units of resource before evaluating the expression e. However, for list traversal where we want to release the memory resource after the evaluation, we encode the semantics using the following semantics:

$$lazy\_tick\ p\ e = (\lambda x. (tick\ p\ x))\ e$$

In such a case, the inference process for $lazy\_tick\ \ -1\ e$ with the judgement $\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \big[f_2\big]_y List(int)$ where y is a local variable will be as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Omega \,|\, \Gamma, x : List(int) \,|\, 1 \vdash x : \big[1\big]_x List(int)}{\Omega \,|\, \Gamma, x : List(int) \,|\, 0 \vdash tick\ \ -1\ x : \big[1\big]_x List(int)}\,(\text{TTickN})}{\Omega \,|\, \Gamma \,|\, 0 \vdash \lambda x. \cdots : \big[0\big]\,(x : List(int) \to x : List(int))_{[0 \to 1]}}\,(\text{TAbs})}{\Omega, x : List(int) \,|\, \Gamma \,|\, f_1 \vdash \lambda x. \cdots\ e : \big[f_2[y \mapsto x] + 1\big]_x List(int)}\,(\text{TApp})}{\Omega \,|\, \Gamma \,|\, f_1 \vdash \lambda x. \cdots\ e : \big[f_2[y \mapsto x] + 1\big]_x List(int)}\,(\text{TErase})}{\Omega \,|\, \Gamma \,|\, f_1 \vdash \lambda x. \cdots\ e : \big[f_2 + 1\big]_y List(int)}\,(\text{TRename})$$

Next we want to type the `traverse`, let

$$T_{tvs} \overset{\text{def}}{=} (x : List(int) \to y : List(int))_{[length(x) \to length(y)]}$$
$$\Gamma_0 \overset{\text{def}}{=} \textsf{traverse} : T_{tvs},\ x : List(int)$$
$$\Gamma_1 \overset{\text{def}}{=} \Gamma_0,\ x_0 : int,\ x_1 : List(int)$$

Then the inference process shows as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot \,|\, \Gamma_1 \,|\, 0 \vdash \textsf{traverse} : \big[0\big] T_{tvs} \quad \cdot \,|\, \Gamma_1 \,|\, length(x_1) \vdash x_1 : \big[length(x_1)\big]_{x_1} List(int)}{x : List(int) \,|\, \Gamma \,|\, length(x_1) \vdash \textsf{traverse}\ x_1 : \big[length(y)\big]_y List(int)}\,(\text{TApp})}{\cdot \,|\, \Gamma_0 \,|\, length(x_1) \vdash cons(\cdots) : \big[length(y) - 1\big]_y List(int)}\,(\text{TCons})}{\cdot \,|\, \Gamma_0 \,|\, length(x_1) \vdash lazy\_tick\ (-1)\ \cdots : \big[length(y)\big]_y List(int)}\,(\text{LazyTick})}{\cdot \,|\, \Gamma_0 \,|\, length(x_1) + 1 \vdash tick\ 1\ \cdots : \big[length(y)\big]_y List(int)}\,(\text{TTick})}{\cdot \,|\, \Gamma_0 \,|\, length(x) \vdash matd(\cdots) : \big[length(y)\big]_y List(int)}\,(\text{TDes})}{\cdot \,|\, \cdot \,|\, 0 \vdash fix\ \textsf{traverse} : T_{tvs}. \cdots : \big[0\big] T_{tvs}}\,(\text{TFix})$$

One may notice a transition of output potential from $length(y)$ to $length(y) - 1$ at rule (TCons), and a change of input potential from $length(x_1)$ to $length(x)$ at rule (TDes). These transition are natural according to the increase or decrease of $length()$ by the constructors and destructors.

4.2 Map Append

Our last example is a higher-order usage of `append`. We want to use `map` to apply the partial application of `append` to another list of list ℓ_2

```
map_append = map (append ℓ1) ℓ2
```

Where `map` is implemented as follows:

$$\begin{aligned} &fix\ \mathsf{map} : T_{\mathsf{map}}.\ \lambda z.\ \lambda w.\ matd(w, \{nil(w_0).\ nil(w_0), \\ &\qquad cons(w_0, w_1).(cons(z\ w_0, \mathsf{map}\ z\ w_1))\}) \end{aligned}$$

Let the following be:

$$\begin{aligned} T_{map} &\overset{\text{def}}{=} (z : ((y : List(int) \to List(int))_{[length(x) \to 0]}) \to)_{[0 \to 0]} \\ &\quad ((w : List(List(int)) \to List(List(int)))_{[length(x) \times length(w) \to 0]}) \\ \Gamma_0 &\overset{\text{def}}{=} \mathsf{map} : T_{map},\ z : (y : List(int) \to List(int))_{[length(x) \to 0]}, \\ &\quad w : List(List(int)) \\ \Gamma_1 &\overset{\text{def}}{=} \Gamma_0, w_0 : List(int),\ w_1 : List(List(int)) \\ \Gamma_2 &\overset{\text{def}}{=} l_1 : List(int),\ l_2 : List(List(int)) \\ f(x,y) &\overset{\text{def}}{=} length(x) \times length(y) \end{aligned}$$

Then the inference process for `map` shows as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : List(int) \mid \Gamma_1 \mid f(x, w_1) \vdash w_1 : \big[f(x, w_1)\big]_{w_1} List(List(int))}{x : List(int) \mid \Gamma_1 \mid f(x, w_1) \vdash \mathsf{map}\ z\ w_1 : \big[0\big]\, List(List(int))}\ (\text{TApp})}{\begin{array}{l} x : List(int) \mid \Gamma_1 \mid length(x) \\ \times(length(w_1) + 1) \vdash (cons(z\ w_0, map\ z\ w_1)) : \big[0\big]\, List(List(int)) \end{array}}\ (\text{TCons})}{x : List(int) \mid \Gamma_0 \mid f(x, w) \vdash matd(w, \cdots) : \big[0\big]\, List(List(int))}\ (\text{TDes})}{x : List(int) \mid \mathsf{map} : T_{map} \mid 0 \vdash \lambda x.\ \lambda y.\ \cdots : \big[0\big]\, T_{map}}\ (\text{TAbs})}{x : List(int) \mid \cdot \mid 0 \vdash fix\ \mathsf{map} : T_{map} \cdot \cdots : \big[0\big]\, T_{map}}\ (\text{TFix})$$

Then we can type the `map_append`:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \mid \Gamma_2 \mid f(l_1, l_2) \vdash l_1 : \big[f(l_1, l_2)\big]_{l_1} List(List(int))}{\begin{array}{l} x : List(int) \mid \Gamma_2 \mid f(l_1, l_2) \vdash append\ l_1 : \\ \qquad [f(x, l_2)](y : List(int) \to List(int))_{[length(x) \to 0]} \end{array}}\ (\text{TApp})}{\begin{array}{l} x : List(int) \mid \Gamma_2 \mid f(l_1, l_2) \vdash \mathsf{map}\ (append\ l_1) \\ : \big[f(x, l_2)\big]\, (w : List(List(int)) \to List(List(int)))_{[f(x,w) \to 0]} \end{array}}\ (\text{TApp})}{\begin{array}{l} x : List(int) \mid \Gamma_2 \mid f(l_1, l_2) \vdash \mathsf{map}\ (append\ l_1)\ l_2 \\ : \big[0\big]\, List(List(int)) \end{array}}\ (\text{TApp})}{\begin{array}{l} \cdot \mid l_1 : List(int),\ l_2 : List(List(int)) \mid length(l_1) \\ \times length(l_2) \vdash \mathsf{map}\ (append\ l_1)\ l_2 : \big[0\big]\, List(List(int)) \end{array}}\ (\text{TErase})$$

Moveover, this program is not typable in AARA, with a proof delayed in Appendix.

Theorem 4. *The example Map Append is not typable in AARA type system.*

5 Discussion: Automated Type Checking and Inference

Our work focuses on formalizing a dependently-typed calculus with non-affine AARA for resource analysis; thus, developing an algorithm for automatic resource-bound inference is out of the scope of this article. Nevertheless, in this section, we discuss possible pathways towards automation.

$$\text{(AProj1)}\quad \dfrac{\begin{array}{c} y, z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T_1 \times T_2 \end{array}}{\Omega \mid \Gamma \mid f_1 \vdash_a \pi_1 e : \left[\mathbf{min}_{z:T_2}(f_2[x \mapsto (y,z)])\right]_y T_1} \qquad \text{(AFix)}\quad \dfrac{\begin{array}{c} z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma, x : T \mid 0 \vdash_a e : \left[f\right]_y T \end{array}}{\Omega \mid \Gamma \mid 0 \vdash_a \mathit{fix}\ x : T.\, e : \left[0\right]_z T}$$

$$\text{(AInt)}\quad \dfrac{}{\Omega \mid \Gamma \mid [0] \vdash_a i : \left[0\right]_x \mathit{int}} \qquad \text{(ATickN)}\quad \dfrac{\begin{array}{c} \Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T \\ p < 0 \end{array}}{\begin{array}{l} \Omega \mid \Gamma \mid \mathbf{max}(f_1 + p, 0) \vdash_a \\ \mathit{tick}\ p\ e : \left[f_2 - \mathbf{min}(f_1 + p, 0)\right]_x T \end{array}}$$

$$\text{(AAppP)}\quad \dfrac{\begin{array}{c} \Omega \mid \Gamma \mid f_1 \vdash_a e_1 : \left[f_2\right]_z ((x : T_1 \to y : T_2)_{[f_3 \to f_4]}) \\ \Omega \mid \Gamma \mid f_5 \vdash_a e_2 : \left[f_6\right]_w T_1 \qquad \Omega, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \leq (f_2 + f_6)[w \mapsto x] \end{array}}{\Omega, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash_a e_1\ e_2 : \left[(f_2 + f_6 - f_3)[w \mapsto x] + f_4\right]_y T_2}$$

$$\text{(AAppN)}\quad \dfrac{\begin{array}{ll} \Omega \mid \Gamma \mid f_1 \vdash_a e_1 : \left[f_2\right]_z ((x : T_1 \to y : T_2)_{[f_3 \to f_4]}) & \Omega \mid \Gamma \mid f_5 \vdash_a e_2 : \left[f_6\right]_w T_1 \\ \Omega, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] > (f_2 + f_6)[w \mapsto x] & \textbf{if } w \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma), \\ \textbf{then } f = \mathbf{min}_{x:T_1}((f_3 - f_2 - f_6)[w \mapsto x]) & \textbf{else } f = f_3 - f_2 - f_6 \end{array}}{\Omega, x : T_1 \mid \Gamma \mid f + f_1 + f_5 \vdash_a e_1\ e_2 : \left[(f + f_2 + f_6 - f_3)[w \mapsto x] + f_4\right]_y T_2}$$

Fig. 4: Selected Algorithmic Typing Rules

5.1 An Algorithmic System for Type Checking

As stated in §2 and §3, we assume we can perform arithmetic reasoning in the system, but still the type system is declarative and thus non-deterministic. That is to say, more than one rule can be applied even if we fix the context Ω, Γ and expression e. The non-determinism of typing rules is caused by the rule (TRelax), rule (TDrop), rule (TRename) and rule (TErase). While it is hard to set up a complete algorithmic version of the type system due to the *let* construct and underlying decision theory of potential functions, we can still set up a sound and determinstic algorithm using the parametrized **min**() and **max**() functions, which take an expression together with its parameter variables as input and return the minimum or maximum integer over all possible assignments to those variables (e.g., $\mathbf{min}_{x:\mathsf{Int}}(x^2) = 0$).

We here set up an algorithmic version $\Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T$, based on the observation that usages of rule (TRelax) and rule (TDrop) can be combined with usages of all the other rules. Thus, we let f_1 be the minimum input potential function and f_2 be the maximum output potential function under the minimum input in the algorithmic typing rules. If a rule like (AAppn) finds the potential provided by premise under the minimum input is not enough, it then use the rule (TRelax) to borrow the potentials, with the help of $\mathbf{min}()$ and $\mathbf{max}()$ operators. Determinism is then achieved by eliminating the rule (TRelax) and rule (TDrop), while arithmetic solver needs to deal with constraints with $\mathbf{min}(\cdot)$ and $\mathbf{max}(\cdot)$. As for the other two structural rules, the rule (TErase) can always be applied at the end of the inference while rule (TRename) is only needed for matching the local binder in the rule (TAbs). Thus by applying the strategy of not using (TErase) and (TRename) unless necessary, we get a deterministic result $\Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T$. Then compare the needed $\Omega \mid \Gamma \mid f_3 \vdash e : \left[f_4\right]_x T$ and check $\Omega \mid \Gamma \vdash f_3 \geq f_1$ and $\Omega, x : T \mid \Gamma \vdash f_2 + f_3 \geq f_1 + f_4$ we can get the result whether a typing is feasible or not. We have set up the following theorems for the soundness of algorithm, see full proof in Appendix.

Theorem 5 (Soundness). *If $\Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T$, then $\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T$.*

Theorem 6 (Determinisism). *If $\Omega \mid \Gamma \mid f_1 \vdash_a e : \left[f_2\right]_x T$, $\Omega \mid \Gamma \mid f_3 \vdash_a e : \left[f_4\right]_x T$, then $\Omega \mid \Gamma \vdash f_1 = f_3$, and $\Omega, x : T \mid \Gamma \vdash f_2 = f_4$.*

5.2 Towards Resource-Bound Synthesis for Type Inference

Although the type system presented in §5.1 is algorithmic, it still requires all functions, including both lambda abstractions and fixpoints, be annotated with their types. Type inference for dependently-typed systems is usually undecidable; in our system $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, even modulo constraint solving for arithmetics, we conjecture that type inference is undecidable, because it essentially needs to synthesize possibly-recursive potential functions that satisfy (first-order) arithmetic constraints. Thus, we discuss possible incomplete approaches for type inference.

Scenario I: all usable potential functions are provided. We first consider a simplified setting, where the user already provides essential potential functions defined over inductive datatypes, such as the $\mathit{length}(\cdot)$ function for lists or the $\mathit{depth}(\cdot)$ function for trees. The type-inference problem can then be reduced to infer arithmetic expressions over those potential functions as potential annotations used in the type system. For example, suppose we want to infer a type for append:

$$T_{\mathsf{append}} \overset{\text{def}}{=} [f_1]_x \mathit{List}(\mathit{int}) \to [f_2](\ [f_3]_y \mathit{List}(\mathit{int}) \to [f_4]_z \mathit{List}(\mathit{int})\ )$$

Using the algorithmic type system, we may collect some constraints:

$$\forall x\colon f_{1,x} \geq f_{2,x}, \quad \forall x,y\colon (x = \mathit{nil}) \implies f_{3,x,y} \geq f_{4,x,y,y},$$
$$\forall x,y,z,h,t\colon (x = \mathit{cons}(h,t)) \implies f_{3,x,y} \geq 1 + f_{1,t} \wedge f_{3,x,y} - 1 - f_{1,t} + f_{2,t}$$
$$\geq f_{3,t,y} \wedge f_{3,x,y} - 1 - f_{1,t} + f_{2,t} - f_{3,t,y} + f_{4,t,y,z} \geq f_{4,x,y,\mathit{cons}(h,z)}.$$

These constraints also indicate what set of variables f_1, f_2, f_3, f_4 can use, i.e., f_1, f_2 can use x, f_3 can use x, y, and f_4 can use x, y, z. It is straightforward to verify that $f_1 = f_2 = f_4 = 0$ and $f_3 = \mathit{length}(x)$ form a valid solution, which yields the type of `append` discussed in §2. You can also verify that $f_1 = f_2 = 0$, $f_3 = 2 \times \mathit{length}(x) + \mathit{length}(y)$, and $f_4 = \mathit{length}(z)$ give another valid solution.

There have been template-based approaches for synthesizing unknown expressions like f_1, f_2, f_3, f_4. For example, Avanzini et al. [3] studied modular cost analysis for imperative probabilistic programs, but their implementation features a constraint solver named *GUBS* for GUBS Upper Bound Solver, which tries to synthesize unknown arithmetic expressions (more precisely, polynomials with **max** operators) subject to a set of (in)equalities over arithmetic expressions. Such an approach is *template*-based in the sense that the solver pre-selects possible forms for the unknown expressions (polynomials in the GUBS case).

Scenario II: no potential functions are provided. We now consider the harder case: the type inference must also synthesize all the needed potential functions defined over inductive datatypes. Because our system $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ permits primitive recursion in the definition of potential functions, we can reduce the type-inference problem to synthesis of *recursive* functions, subject to (first-order) arithmetic constraints. Synthesis of recursive functional programs has been a popular research topic; for example, there are techniques that synthesize recursive programs from input-output examples [27,53,69,39], from logical specifications [55,45], or from reference implementations [29,30]. None of those approaches would be applicable to our setting: (i) the constraints for bound synthesis are logical, and (ii) the constraints do *not* directly specify recursive functions, but unknown arithmetic expressions over unknown recursive functions. However, existing techniques that synthesize recursive programs from logical specifications require that the constraints be declared directly on the unknown recursive function.

Nevertheless, recent work by Hong and Aiken [27] gives us some inspiration towards developing a type-inference algorithm for $\lambda^{\mathsf{na}}_{\mathsf{amor}}$. Hong and Aiken [27] propose to use *paramorphisms*—which generalize the usual `fold` combinator—to reduce synthesis of recursive programs to synthesis of non-recursive programs with primitive recursion achieved by paramorphisms. For example, the $\mathit{length}(\cdot)$ function on lists can be implemented with `fold` without recursion:

```
length = λx. fold x 0 (+1)
```

Therefore, the type T_{append} shown above for the curried `append` function can be defined as follows:

$$T_{\mathsf{append}} \overset{\text{def}}{=} [0]_x \mathit{List}(\mathit{int}) \to [0](\ [\texttt{fold}\ x\ 0\ (+1)]_y \mathit{List}(\mathit{int}) \to [0]_z \mathit{List}(\mathit{int})\ )$$

6 Related Work

Automatic Amortized Resource Analysis AARA has been extended to support various language features, such as higher-order functions [31], inductive datatypes [18,32], regular recursive types [14], mutable references [43,42], lazy evaluation [60,65], parallel computation [22], imperative programming [6,5,1] and object-oriented programming [24]. Researchers have proposed automated resource-bound inference algorithms for lower bounds [48], linear bounds [23], univariate/multivariate polynomial bounds [26,17,20,19], logarithmic bounds [25,41], and exponential bounds [33]. There is a line of studies on integrating amortized resource analysis into theorem provers [7,49,56,44]. These techniques rely on the principle of affine/linear typing and resource-annotated types to enable automatic bound inference, whereas our work builds a non-affine variant of AARA and focuses on verification of expressive resource bounds.

Some researchers proposed *cost-aware logical frameworks* [51,13,50], which can be instantiated to carry out amortized analysis, *without* requiring affine/-linear typing. Those studies focus on devising a general framework for various methodologies of resource analysis, but it is unclear how they can make better use of existing AARA-style type systems. As an analogy, they propose logical frameworks like LF [16], and both AARA and our work are concrete logics, such as first-order or higher-order logic. In this sense, our work is orthogonal and focuses primarily on AARA itself, building a dependently typed AARA.

Dependent Type Systems for Resource Analysis Knoth et al. [36,37] combined liquid types [58] with AARA and proposed liquid resource types (LRT), which support user-defined non-linear potential functions on inductive datatypes, while still keeping the capability of automated type checking. LRT relies on the principle of affine typing for the resource-analysis part; in particular, LRT requires tightly-coupled resource-annotated inductive types. There have been dependent type systems utilizing linear typing to perform resource-bound verification, though not directly connected to AARA. Lago and Gaboardi [38] presented dℓPCF, which uses linear dependent types—where the dependency is achieved by an index language layer—to reason about the resource consumption of PCF expressions. Orchard et al. [52] extended dℓPCF with graded modal typing and proposed Granule, which has many applications (e.g., reasoning about privacy), despite ones of amortized resource analysis. Rajani et al. [57] proposed λ-amor as a unifying type theory for amortized resource analysis: their system has a similar design to dℓPCF using dependent affine typing, but introduces an indexed cost monad to enable amortization. Compared to our work, aforementioned approaches still require linear or affine typing.

There have also been some non-affine dependent type systems for resource analysis, though not directly supporting amortized analysis. Wang et al. [67] developed TiML, which extends Dependent ML [68] with type-level *indices* that describe data-structure sizes and time complexities, thus enabling complexity analysis. *Sized types* [28,64,2] form another line of work that also uses dependent types to keep track of sizes of data structures. Danielsson [10] described a lightweight approach to implement a dependent cost monad in the Agda theorem prover and reduce resource-bound verification to reasoning about the monad in Agda. Handley et al. [15] devised a different way from Re^2 [36] and LRT [37] to extend liquid types to perform resource analysis. These techniques diverge from AARA and it is unclear how they would support amortized analysis naturally.

Closure Types and Contextual Types As we discussed in §1, our work takes motivation from Scherer and Hoffmann [59]'s work, which mentioned the desire to precisely characterize the resource behavior of curried higher-order functions. Scherer and Hoffmann proposed open closure types that can carry a set of tagged captured variables to track data-flow. Such an idea is not new, e.g., Leroy [40] used closure types to track type variables that cannot be further generalized during type checking. In those type systems, arrow types are annotated with contexts; such a notion resembles the contextual arrow types from *contextual type theory* [47,54,61]. The difference is that, the context attached to a closure type essentially keeps track of the bindings in the typing context of the closure, whereas a contextual arrow type—e.g., of the form $[\Psi](A \to B)$—uses the context Ψ to record meta-variables that can be used to build inhabitants. Our potential-carrying arrow types $[f_1]_x T_1 \to [f_2]_y T_2$ are more similar to the closure types, in the sense that the potential functions f_1, f_2 should also be well-typed under the typing context of the closure.

Recall that our typing rule for lambda applications essentially introduces existential binders. This looks similar to a standard approach to understand the capturing behavior of closures, e.g., Minamide et al. [46] presented *typed closure conversion* that uses existential types to represent the captured data. However, our setting is different because our binders bind *values*, whereas typed closure conversion's existential binders bind *types*; for example, Minamide et al.'s approach would assign $\mathsf{append}\ \ell_1$ with a type $\exists T_{env}. T_{env} \times (T_{env} \to \mathit{List}(\mathit{int}) \to \mathit{List}(\mathit{int}))$, i.e., the type does *not* reflect that the closure captures a list. In contrast, our mechanism of introducing existential binders is similar to existing techniques in refinement type systems [55,4], which propose complex mechanisms to handle general existential types resulted from function applications. In our system, we devise a lightweight approach using a global potential context to track those existential binders, striking a balance between expressibility and convenience.

There is another work worth mentioning by Kahn and Hoffmann [34], who augmented AARA-style affine typing with *remainder contexts* and *resource tunneling*, in order to express that a function may return potentials that depend on the input and/or even the typing context. For example, the resource-annotated type $\mathit{List}(\mathit{int}^2) \to \mathit{int} \circ \mathit{List}(\mathit{int}^1)$ describes a function that takes a list ℓ with $2 \times |\ell|$ units of potential as input, returns an integer as its result, and there still remains $|\ell|$ units of potential associated with the input list ℓ. In our system, we can simply express such behavior by $[2 \times \mathit{length}(x)]_x \mathit{List}(\mathit{int}) \to [\mathit{length}(x)]\mathit{int}$ without affine typing.

7 Conclusion

In this article, we introduced $\lambda^{\mathsf{na}}_{\mathsf{amor}}$, a non-affine AARA-style dependent type system for resource reasoning about higher-order programs. We formalized our $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ in syntax and semantics, and provided a soundness proof based on type-level potential functions and a cost-aware semantics. Besides the theoretical aspects, we demonstrated the expressiveness and compositionality of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s reasoning process through several challenging examples that involve higher-order functions. Additionally, we developed an algorithmic variant of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$'s type system, which transforms typing derivation into constraint solving. Future work includes extending $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ with refinement types [12] or liquid types [58], as well as building up program-synthesis techniques for automatic resource-bound inference.

Data-Availability Statement The Appendix is available at `https://arxiv.org/abs/2601.12943`.

Acknowledgements. This work was sponsored by National Key R&D Program of China, Grant No. 2024YFE0204100.

References

1. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: European Symp. on Programming. pp. 85–103. ESOP'10 (2010). https://doi.org/10.1007/978-3-642-11957-6_6
2. Avanzini, M., Lago, U.D.: Automating Sized-type Inference for Complexity Analysis. Proc. ACM Program. Lang. **1**(43), 43:1–43:29 (August 2017). https://doi.org/10.1145/3110287
3. Avanzini, M., Moser, G., Schaper, M.: A Modular Cost Analysis for Probabilistic Programs. Proc. ACM Program. Lang. **4**(172) (November 2020). https://doi.org/10.1145/3428240
4. Borkowski, M.H., Vazou, N., Jhala, R.: Mechanizing Refinement Types. Proc. ACM Program. Lang. **8**(70), 2099–2188 (January 2024). https://doi.org/10.1145/3632912
5. Carbonneaux, Q., Hoffmann, J., Reps, T., Shao, Z.: Automated Resource Analysis with Coq Proof Objects. In: Computer Aided Verif. CAV'17 (2017). https://doi.org/10.1007/978-3-319-63390-9_4
6. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional Certified Resource Bounds. In: Prog. Lang. Design and Impl. PLDI'15 (2015). https://doi.org/10.1145/2737924.2737955
7. Charguéraud, A., Pottier, F.: Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In: Interactive Theorem Proving. pp. 137–153. ITP'15 (2015). https://doi.org/10.1007/978-3-319-22102-1_9
8. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational Cost Analysis. In: Princ. of Prog. Lang. pp. 316–329. POPL'17 (2017). https://doi.org/10.1145/3009837.3009858

9. Crary, K., Weirich, S.: Resource Bound Certification. In: Princ. of Prog. Lang. pp. 184–198. POPL'00 (2000). https://doi.org/10.1145/325694.325716
10. Danielsson, N.A.: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In: Princ. of Prog. Lang. pp. 133–144. POPL'08 (2008). https://doi.org/10.1145/1328438.1328457
11. Danner, N., Licata, D.R., Ramyaa, R.: Denotational Cost Semantics for Functional Languages with Inductive Types. In: Int. Conf. on Functional Programming. pp. 140–151. ICFP'15 (2015). https://doi.org/10.1145/2784731.2784749
12. Freeman, T., Pfenning, F.: Refinement Types for ML. In: Prog. Lang. Design and Impl. PLDI'91 (1991). https://doi.org/10.1145/113446.113468
13. Grodin, H., Niu, Y., Sterling, J., Harper, R.: Decalf: A Directed, Effectful Cost-Aware Logical Framework. Proc. ACM Program. Lang. **8**(10), 273–301 (January 2024). https://doi.org/10.1145/3632852
14. Grosen, J., Kahn, D.M., Hoffmann, J.: Automatic Amortized Resource Analysis with Regular Recursive Types. In: Logic in Computer Science. pp. 1–14. LICS'23 (2023). https://doi.org/10.1109/LICS56636.2023.10175720
15. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. Proc. ACM Program. Lang. **4**(24), 24:1–24:27 (December 2019). https://doi.org/10.1145/3371092
16. Harper, R., Honsell, F., Plotkin, G.D.: A Framework for Defining Logics. J. ACM **40**, 143–184 (January 1993). https://doi.org/10.1145/138027.138060
17. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: Princ. of Prog. Lang. pp. 357–370. POPL'11 (2011). https://doi.org/10.1145/1926385.1926427
18. Hoffmann, J., Das, A., Weng, S.C.: Towards Automatic Resource Bound Analysis for OCaml. In: Princ. of Prog. Lang. pp. 359–373. POPL'17 (2017). https://doi.org/10.1145/3009837.3009842
19. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Asian Symp. on Prog. Lang. and Systems. APLAS'10 (2010). https://doi.org/10.1007/978-3-642-17164-2_13
20. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: European Symp. on Programming. ESOP'10 (2010). https://doi.org/10.1007/978-3-642-11957-6_16
21. Hoffmann, J., Jost, S.: Two Decades of Automatic Amortized Resource Analysis. Math. Struct. Comp. Sci. **32**, 729–759 (June 2022). https://doi.org/10.1017/S0960129521000487
22. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. In: European Symp. on Programming. ESOP'15 (2015). https://doi.org/10.1007/978-3-662-46669-8_6
23. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 185–197. POPL '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604131.604148, https://doi.org/10.1145/604131.604148
24. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: European Symp. on Programming. ESOP'06 (2006). https://doi.org/10.1007/11693024_3
25. Hofmann, M., Leutgeb, L., Obwaller, D., Moser, G., Zuleger, F.: Type-based analysis of logarithmic amortised complexity. Math. Struct. Comp. Sci. **32**, 794–826 (June 2022). https://doi.org/10.1017/S0960129521000232

26. Hofmann, M., Moser, G.: Multivariate Amortised Resource Analysis for Term Rewrite Systems. In: Int. Conf. on Typed Lambda Calculi and Applications. TLCA'15 (2015). https://doi.org/10.4230/LIPIcs.TLCA.2015.241
27. Hong, Q., Aiken, A.: Recursive Program Synthesis using Paramorphisms. Proc. ACM Program. Lang. **8**(151), 102–125 (June 2024). https://doi.org/10.1145/3656381
28. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Princ. of Prog. Lang. pp. 410–423. POPL'96 (1996). https://doi.org/10.1145/237721.240882
29. Ji, R., Zhao, Y., Polikarpova, N., Xiong, Y., Hu, Z.: Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis. Proc. ACM Program. Lang. **8**(185), 939–964 (June 2024). https://doi.org/10.1145/3656415
30. Ji, R., Zhao, Y., Xiong, Y., Wang, D., Zhang, L., Hu, Z.: Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms. Trans. on Prog. Lang. and Syst. **46**(8), 8:1–8:59 (June 2024). https://doi.org/10.1145/3648440
31. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: Princ. of Prog. Lang. pp. 223–236. POPL'10 (2010). https://doi.org/10.1145/1706299.1706327
32. Jost, S., Loidl, H.W., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In: Formal Methods. pp. 354–369. FM'09 (2009). https://doi.org/10.1007/978-3-642-05089-3_23
33. Kahn, D.M., Hoffmann, J.: Exponential Automatic Amortized Resource Analysis. In: Foundations of Software Science and Computation Structures. FoSSaCS'20 (2020). https://doi.org/10.1007/978-3-030-45231-5_19
34. Kahn, D.M., Hoffmann, J.: Automatic Amortized Resource Analysis with the Quantum Physicist's Method. Proc. ACM Program. Lang. **5**(76) (August 2021). https://doi.org/10.1145/3473581
35. Kavvos, G.A., Morehouse, E., Licata, D.R., Danner, N.: Recurrence Extraction for Functional Programs through Call-by-Push-Value. Proc. ACM Program. Lang. **4**(15), 15:1–15:31 (December 2019). https://doi.org/10.1145/3371083
36. Knoth, T., Wang, D., Polikarpova, N., Hoffmann, J.: Resource-Guided Program Synthesis. In: Prog. Lang. Design and Impl. PLDI'19 (2019). https://doi.org/10.1145/3314221.3314602
37. Knoth, T., Wang, D., Reynolds, A., Hoffmann, J., Polikarpova, N.: Liquid resource types. Proc. ACM Program. Lang. **4**(ICFP) (aug 2020). https://doi.org/10.1145/3408988, https://doi.org/10.1145/3408988
38. Lago, U.D., Gaboardi, M.: Linear Dependent Types and Relative Completeness. In: Logic in Computer Science. pp. 133–142. LICS'11 (2011). https://doi.org/10.1109/LICS.2011.22
39. Lee, W., Cho, H.: Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. Proc. ACM Program. Lang. **7**(70), 2048–2078 (January 2023). https://doi.org/10.1145/3571263
40. Leroy, X.: Polymorphic Typing of an Algorithmic Language. Tech. rep., INRIA (1992)
41. Leutgeb, L., Moser, G., Zuleger, F.: ATLAS: Automated Amortised Complexity Analysis of Self-adjusting Data Structures. In: Computer Aided Verif. pp. 99–122. CAV'21 (2021). https://doi.org/10.1007/978-3-030-81688-9_5
42. Lian, Q., Wang, D.: Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. Proc. ACM Program. Lang. **9**(130), 1406–1433 (April 2025). https://doi.org/10.1145/3720492

43. Lichtman, B., Hoffmann, J.: Arrays and References in Resource Aware ML. In: Formal Struct. for Comput. and Deduction. pp. 26:1–26:20. FSCD'17 (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.26
44. Mével, G., Jourdan, J.H., Pottier, F.: Time Credits and Time Receipts in Iris. In: European Symp. on Programming. pp. 3–29. ESOP'19 (2019). https://doi.org/10.1007/978-3-030-17184-1_1
45. Miltner, A., Nuñez, A.T., Brendel, A., Chaudhuri, S., Dillig, I.: Bottom-Up Synthesis of Recursive Functional Programs using Angelic Execution. Proc. ACM Program. Lang. **6**(21), 21:1–21:29 (January 2022). https://doi.org/10.1145/3498682
46. Minamide, Y., Morrisett, G., Harper, R.: Typed Closure Conversion. In: Princ. of Prog. Lang. pp. 271–283. POPL'96 (1996). https://doi.org/10.1145/237721.237791
47. Nanevski, A., Pfenning, F., Pientka, B.: Contextual Modal Type Theory. Trans. on Comp. Logic **9**(23), 23:1–23:49 (June 2008). https://doi.org/10.1145/1352582.1352591
48. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Verifying and Synthesizing Constant-Resource Implementations with Types. In: Symp. on Sec. and Privacy. pp. 710–728. SP'17 (2017). https://doi.org/10.1109/SP.2017.53
49. Nipkow, T.: Amortized Complexity Verified. In: Interactive Theorem Proving. ITP'15 (2015). https://doi.org/10.1007/978-3-319-22102-1_21
50. Niu, Y., Harper, R.: A Metalanguage for Cost-Aware Denotational Semantics. In: Logic in Computer Science. pp. 1–14. LICS'23 (2023). https://doi.org/10.1109/LICS56636.2023.10175777
51. Niu, Y., Sterling, J., Grodin, H., Harper, R.: A Cost-Aware Logical Framework. Proc. ACM Program. Lang. **6**(9), 9:1–9:31 (January 2022). https://doi.org/10.1145/3498670
52. Orchard, D., Liepelt, V.B., Eades III, H.: Quantitative Program Reasoning with Graded Modal Types. Proc. ACM Program. Lang. **3**(110), 110:1–110:30 (July 2019). https://doi.org/10.1145/3341714
53. Osera, P.M., Zdancewic, S.: Type-and-Example-Directed Program Synthesis. In: Prog. Lang. Design and Impl. PLDI'15 (2015). https://doi.org/10.1145/2737924.2738007
54. Pientka, B., Dunfield, J.: Programming with Proofs and Explicit Contexts. In: Int. Conf. on Principles and Practice of Declarative Programming. pp. 163–173. PPDP'08 (2008). https://doi.org/10.1145/1389449.1389469
55. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program Synthesis from Polymorphic Refinement Types. In: Prog. Lang. Design and Impl. PLDI'16 (2016). https://doi.org/10.1145/2908080.2908093
56. Pottier, F., Guéneau, A., Jourdan, J.H., Mével, G.: Thunks and Debits in Separation Logic with Time Credits. Proc. ACM Program. Lang. **8**(50), 1482–1508 (January 2024). https://doi.org/10.1145/3632892
57. Rajani, V., Gaboardi, M., Garg, D., Hoffmann, J.: A unifying type-theory for higher-order (amortized) cost analysis. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434308, https://doi.org/10.1145/3434308
58. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid Types. In: Prog. Lang. Design and Impl. PLDI'08 (2008). https://doi.org/10.1145/1379022.1375602
59. Scherer, G., Hoffmann, J.: Tracking Data-Flow with Open Closure Types. In: Logic for Programming Artificial Intelligence and Reasoning. LPAR'13 (2013). https://doi.org/10.1007/978-3-642-45221-5_47

60. Simões, H., Vasconcelos, P.B., Florido, M., Jost, S., Hammond, K.: Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In: Int. Conf. on Functional Programming. pp. 165–176. ICFP'12 (2012). https://doi.org/10.1145/2364527.2364575
61. Stampoulis, A., Shao, Z.: Static and User-Extensible Proof Checking. In: Princ. of Prog. Lang. pp. 273–284. POPL'12 (2012). https://doi.org/10.1145/2103656.2103690
62. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent Types and Multi-Monadic Effects in F*. In: Princ. of Prog. Lang. pp. 256–270. POPL'16 (2016). https://doi.org/10.1145/2837614.2837655
63. Tarjan, R.E.: Amortized Computational Complexity. SIAM J. Algebraic Discrete Methods **6** (August 1985). https://doi.org/10.1137/0606031
64. Vasconcelos, P.B.: Space Cost Analysis Using Sized Types. Ph.D. thesis, University of St Andrews (2008)
65. Vasconcelos, P.B., Jost, S., Florido, M., Hammond, K.: Type-Based Allocation Analysis for Co-Recursion in Lazy Functional Languages. In: European Symp. on Programming. ESOP'15 (2015). https://doi.org/10.1007/978-3-662-46669-8_32
66. Walker, D.: Substructural Type Systems. In: Advanced Topics in Types and Programming Languages. MIT Press (2002)
67. Wang, P., Wang, D., Chlipala, A.: TiML: A Functional Language for Practical Complexity Analysis with Invariants. Proc. ACM Program. Lang. **1**(79), 79:1–79:26 (October 2017). https://doi.org/10.1145/3133903
68. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: Princ. of Prog. Lang. POPL'99 (1999). https://doi.org/10.1145/292540.292560
69. Yuan, Y., Radhakrishna, A., Samanta, R.: Trace-Guided Inductive Synthesis of Recursive Functional Programs. Proc. ACM Program. Lang. **7**(141), 860–883 (June 2023). https://doi.org/10.1145/3591255

A Appendix

A.1 Definitions

Free Variables All the free variables in the type and potential functions are bound by the type context and potential context. And the free variable $\mathbf{fv}(f)$ and $\mathbf{fv}(T)$ are defined as followed:

- $\mathbf{fv}(f)$ is all the variables in the f.
- $\mathbf{fv}(int) = \emptyset$.
- $\mathbf{fv}(T_1 \times T_2) = \mathbf{fv}(T_1) \cup \mathbf{fv}(T_2)$.
- $\mathbf{fv}((x : T_1 \to y : T_2)_{[f_1 \to f_2]}) = (\mathbf{fv}(f_1) \cup \mathbf{fv}(T_1) \cup \mathbf{fv}(f_2) \backslash y \cup \mathbf{fv}(T_2)) \backslash x$.
- $\mathbf{fv}(ind\overrightarrow{(C, (T, m))}) = \bigcup \mathbf{fv}(T_i)$.

Substitution Next we are going to define the substitution. The substitution happens when a application $e_1\ e_2$ reduces, the variable in the λ, fixpoint or Λ then disappear, so we need to erase them in the Type System correspondingly. The substitution actually substitution the variables in potential context into an existing value. If a variable x is substituted by a value v, then the substitution $T[x \mapsto v]$ defined inductively as followed:

- $int[x \mapsto v] = int$.
- $(T_1 \times T_2)[x \mapsto v] = T_1[x \mapsto v] \times T_2[x \mapsto v]$.
- if $z \neq x$, then $(y : T_1 \to z : T_2)_{[f_1 \to f_2]}[x \mapsto v] = \prod\limits_{y:[(f_1[x \mapsto v])]T_1[x \mapsto v]} \big[(f_2[x \mapsto v])\big]_z T_2[x \mapsto v]$ Otherwise: $(y : T_1 \to z : T_2)_{[f_1 \to f_2]}[x \mapsto v] = \prod\limits_{y:[(f_1[x \mapsto v])]T_1[x \mapsto v]} \big[f_2\big]_z T_2[x \mapsto v]$ (For substitution we use, it is guaranteed that $y \neq x$)
- $\forall y : T_1. T_2[x \mapsto v] = \forall y : T_1[x \mapsto v]. T_2[x \mapsto v]$ (For substitution we use, it is guaranteed that $y \neq x$)
- $ind\overrightarrow{(C, (T, m))}[x \mapsto v] = ind\overrightarrow{(C, (T[x \mapsto v], m))}$.

Based on the substitution $T[x \mapsto v]$, we can define the substitution on type contexts $\Gamma[x \mapsto v]$ and potential context

- For $y \neq x$, we have $(\Gamma, y : T)[x \mapsto v] = \Gamma[x \mapsto v], y : T[x \mapsto v]$.

- $(\mathit{\Gamma}, x : T)[x \mapsto v] = \mathit{\Gamma}[x \mapsto v]$.
- For $y \neq x$, we have $(\mathit{\Omega}, y : T)[x \mapsto v] = \mathit{\Omega}[x \mapsto v], y : T[x \mapsto v]$ and.
- $(\mathit{\Omega}, x : T)[x \mapsto v] = \mathit{\Omega}[x \mapsto v]$.
- For both type context and potential context, we have $\cdot[x \mapsto v] = \cdot$.

As for substitution on terms, we need to be careful about the variables in type annotations, so the substitution would be

- $x[x \mapsto v] = v$
- $y[x \mapsto v] = v$ for $y \neq x$
- $i[x \mapsto v] = i$
- $\lambda y : T.\, e[x \mapsto v] = \lambda y : T[x \mapsto v].\, e[x \mapsto v]$
- $e_1\ e_2[x \mapsto v] = e_1[x \mapsto v]\ e_2[x \mapsto v]$
- $(e_1, e_2)[x \mapsto v] = (e_1[x \mapsto v], e_2[x \mapsto v])$
- $(\pi_1 e)[x \mapsto v] = \pi_1 e[x \mapsto v]$
- $(\pi_2 e)[x \mapsto v] = \pi_2 e[x \mapsto v]$
- $\mathit{fix}\ y : T.\, e[x \mapsto v] = \mathit{fix}\ y : T[x \mapsto v].\, e[x \mapsto v]$ (For the substitution we use, it is guaranteed that $y \neq x$).
- $\mathit{tick}\ i\ e[x \mapsto v] = \mathit{tick}\ i\ e[x \mapsto v]$
- $\mathbf{op_i}(\overrightarrow{e})[x \mapsto v] = \mathbf{op_i}(\overrightarrow{e[x \mapsto v]})$
- $C_i(e_0, (e_1, ..., e_{m_i}))[x \mapsto v] = C_i(e_0[x \mapsto v], (e_1[x \mapsto v], ..., e_{m_i}[x \mapsto v]))$
- $\mathit{matd}(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e})[x \mapsto v] = \mathit{matd}(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e[x \mapsto v]})$

A.2 Curry and Uncurry Functions

An interesting higher-order example involves typing the *curry* and *uncurry* functions. These functions convert between curried and uncurried forms of higher-order functions. They can be defined as follows:

$$\begin{aligned}
&\mathsf{curry} && :: && \big(((z : T_1 \times T_2 \to w : T_3)_{[f_1 \to f_2]}\big) \to \big((x : T_1 \to \big((y : T_2 \to w : T_3)_{[f_1[z \mapsto (x,y)] \to f_2]}\big))_{[0 \to 0]}\big))_{[0 \to 0]} \\
&\mathsf{curry} && \overset{\text{def}}{=} && \lambda f.\, \lambda x.\, \lambda y.\, f\ (x, y) \\
&\mathsf{uncurry} && :: && \big(((x : T_1 \to \big((y : T_2 \to w : T_3)_{[f_2 \to f_3]}\big))_{[f_1 \to 0]}\big) \to \\
& && && \big((z : T_1 \times T_2 \to w : T_3)_{[f_1[x \mapsto \pi_1 z] + f_2[x \mapsto \pi_2 z] \to f_3]}\big))_{[0 \to 0]} \\
&\mathsf{uncurry} && \overset{\text{def}}{=} && \lambda f.\, \lambda z.\, f\ (\pi_1 z)\ (\pi_2 z)
\end{aligned}$$

The type of curry reads as follows: If the input function consumes f_1 units of resource and produces f_2 units of potential, then the curried version initially consumes 0 resources when given the first argument x, but will require f_1 units of potential upon receiving the second argument y. The return value carries f_2 units of potential.

Let:

$$T_{\mathsf{fun0}} \overset{\text{def}}{=} (z : T_1 \times T_2 \to w : T_3)_{[f_1 \to f_2]}$$
$$\mathit{\Gamma}_0 \overset{\text{def}}{=} f : T_{\mathsf{fun0}},\, x : T_1,\, y : T_2$$

The type inference proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \mid \mathit{\Gamma}_0 \mid 0 \vdash f : \left[0\right] T_{\mathsf{fun0}} \qquad \cdot \mid \mathit{\Gamma}_0 \mid f_1[z \mapsto (x,y)] \vdash (x,y) : \left[f_1\right]_z T_3}{z : T_1 \times T_2 \mid \mathit{\Gamma}_0 \mid f_1[z \mapsto (x,y)] \vdash f\ (x,y) : \left[f_2\right]_w T_3}\,(\textsc{TApp})}{z : T_1 \times T_2 \mid f : T_{\mathsf{fun0}},\, x : T_1 \mid 0 \vdash \lambda y.\, f\ (x,y) : \left[0\right] (y : T_2 \to w : T_3)_{[f_1[z \mapsto (x,y)] \to f_2]}}\,(\textsc{TAbs})}{z : T_1 \times T_2 \mid f : T_{\mathsf{fun0}} \mid 0 \vdash \lambda x.\, \lambda y.\, f\ (x,y) : \left[0\right] (x : T_1 \to (y : T_2 \to w : T_3)_{[f_1[z \mapsto (x,y)] \to f_2]})_{[0 \to 0]}}\,(\textsc{TAbs})}{z : T_1 \times T_2 \mid \cdot \mid 0 \vdash \mathsf{curry} : \left[0\right] (T_{\mathsf{fun0}} \to (x : T_1 \to (y : T_2 \to w : T_3)_{[f_1[z \mapsto (x,y)] \to f_2]})_{[0 \to 0]})_{[0 \to 0]}}\,(\textsc{TAbs})$$

Note that the final typing judgment still contains an existential variable z in the potential function. This is intentional: since the potential function f_2 may refer to the function argument z, we preserve this binding. However, if f_2 does not actually depend on z, we may erase it using rule (TErase).

We now perform a similar type inference for the `uncurry` function. Let:

$$T_{\mathsf{fun1}} \overset{\text{def}}{=} (x : T_1 \to \big((y : T_2 \to w : T_3)_{[f_2 \to f_3]}\big))_{[f_1 \to 0]}$$
$$\mathit{\Gamma}_1 \overset{\text{def}}{=} f : T_{\mathsf{fun1}},\, z : T_1 \times T_2$$

We derive its type as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot \mid \mathit{\Gamma}_0 \mid 0 \vdash f : \left[0\right] T_{\mathsf{fun1}} \qquad \cdot \mid \mathit{\Gamma}_0 \mid f_1[x \mapsto \pi_1 z] \vdash \pi_1 z : \left[f_1\right]_x T_1}{x : T_1 \mid \mathit{\Gamma}_0 \mid f_1[x \mapsto \pi_1 z] \vdash f\ (\pi_1 z) : \left[f_1\right] (y : T_2 \to w : T_3)_{[f_2 \to f_3]} \qquad \cdots}\,(\textsc{TApp})}{x : T_1,\, y : T_2 \mid \mathit{\Gamma}_0 \mid f_1[x \mapsto \pi_1 z] + f_2[x \mapsto \pi_2 z] \vdash \lambda z.\, f\ (\pi_1 z)\ (\pi_2 z) : \left[0\right]_w T_3}\,(\textsc{TApp})}{x : T_1,\, y : T_2 \mid f : T_{\mathsf{fun1}} \mid 0 \vdash f\ (\pi_1 z)\ (\pi_2 z) : \left[0\right] (z : T_1 \times T_2 \to w : T_3)_{[f_1[x \mapsto \pi_1 z] + f_2[x \mapsto \pi_2 z] \to f_3]}}\,(\textsc{TAbs})}{x : T_1,\, y : T_2 \mid \cdot \mid 0 \vdash \mathsf{uncurry} : \left[0\right] (T_{\mathsf{fun1}} \to ((z : T_1 \times T_2 \to w : T_3)_{[f_1[x \mapsto \pi_1 z] + f_2[x \mapsto \pi_2 z] \to f_3]}))_{[0 \to 0]}}\,(\textsc{TAbs})$$

Just like in the curry case, the final type for uncurry contains dependencies on potential variables x, y, and z. However, if the final potential function f_3 does not actually depend on any of these variables, we may safely erase them using the rule (TErase). This results in a cleaner and more general type.

A.3 Insertion Sort

Next, we can take a more complicated example for $\lambda^{\mathsf{na}}_{\mathsf{amor}}$. In this example, we will type the classic insertion sort. The result type in our implementation is:

$$T_{srt} \overset{\text{def}}{=} \forall i : int.\, (z : List(int) \to l : List(int))_{[\frac{1}{2}(length(z)^2 + (3+2i) \times length(z)) \to i \times length(l)]}$$

One can see the power as well as the limitations of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$. The separation of type and potentials enables us to precisely describe the cost behavior of `sort`.

However, we lose track of the information of $length(z) = length(l)$, which suggests the sorting does not change the length of a list. Such invariant tracking is only possible with approaches like refinement types, which will be addressed in our future work. In the current case, a polymorphism i is added as a remedy to exchange potentials of $length(z)$ to $length(l)$, so that one can later instantiate i with 0 to get the classic sorting bound as:

$$T_{srt0} \overset{\text{def}}{=} (z : List(int) \to l : List(int))_{[\frac{1}{2}(length(z)^2 + 3 \times length(z)) \to 0]}$$

Back to our example, we take the sorting on $List(int)$ as the case. We want to first show that how to type the `insert`, and then go to `sort`. We implement `insert` and `sort` as follows:

$$\begin{aligned} fix\ \mathsf{insert} : T_{ins}.\, \Lambda i : int.\, \lambda x.\, \lambda y.\, tick\ 1\ matd(y, \{nil(x_0).cons(x, x_0), cons(x_0, x_1).(matd(x < x_0),& \\ \{false(x_0').cons(x_0, \mathsf{insert}\ i\ x\ x_1), true(x_0').cons(x, (cons(x_0, x_1)))\}\})& \\ fix\ \mathsf{sort} : T_{srt}.\, \Lambda i : int.\, \lambda z.\, matd(z, \{nil(z_0).y, cons(z_0, z_1).tick\ 1\ \mathsf{insert}\ i\ z_0\ (\mathsf{sort}\ (i+1)\ z_1)\}) \end{aligned}$$

Where $<$ is function of the type $(int \to ((int \to \mathsf{Bool})_{[0 \to 0]}))_{[0 \to 0]}$. Here we encode type $Bool$ as an inductive type containing only two constructors $false$ and $true$. then we let the following be:

$$\begin{aligned} T_{ins} &\overset{\text{def}}{=} \forall i : int.\, (x : int \to ((y : List(int) \to l : List(int))_{[(i+1) \times (length(y)+1) \to i \times length(l)]}))_{[0 \to 0]} \\ \Gamma_0 &\overset{\text{def}}{=} \mathsf{insert} : T_{srt},\ i : int,\ x : int,\ y : List(int) \\ \Gamma_1 &\overset{\text{def}}{=} \Gamma_0,\ x_0 : int,\ x_1 : List(int),\ x_0' : Bool \\ \Gamma_1' &\overset{\text{def}}{=} \Gamma_0,\ x_0 : int,\ x_1 : List(int) \end{aligned}$$

Thus we get the inductive branch of $false(x_0').cons(x_0, \mathsf{insert}\ i\ x\ x_1)$ types as follows:

$$\dfrac{\dfrac{\dfrac{\cdot \mid \Gamma_1 \mid 0 \vdash \mathsf{insert} :\ [0]\, T_{ins} \qquad \cdot \mid \Gamma_1 \mid 0 \vdash i :\ [0]\, int \qquad \cdot \mid \Gamma_1 \mid 0 \vdash x :\ [0]\, int}{\cdot \mid \Gamma_1 \mid 0 \vdash \mathsf{insert}\ i\ x :\ [0]\, (y : List(int) \to l : List(int))_{[(i+1) \times (length(y)+1) \to i \times length(y)]}}\ (\textsc{TApp})}{\cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash \mathsf{insert}\ i\ x\ x_1 :\ [i \times length(l)]_l\, List(int)}\ (\textsc{TApp})}{\cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash cons(x_0, \mathsf{insert}\ i\ x\ x_1) :\ [i \times (length(l) - 1)]_l\, List(int)}\ (\textsc{TCons})$$

For the other branch $true(x_0').cons(x, (cons(x_0, x_1)))$ we have:

$$\dfrac{\dfrac{\dfrac{\begin{array}{l} \cdot \mid \Gamma_1 \mid 0 \vdash x_0 :\ [0]\, int \\ \cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash x_1 :\ [(i+1) \times (length(x_1)+1)]_{x_1}\, int \end{array}}{\cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash cons(x_0, x_1) :\ [(i+1) \times length(l)]_l\, int}\ (\textsc{TCons})}{\cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash cons(x, (cons(x_0, x_1))) :\ [(i+1) \times (length(l) - 1)]_l\, int}\ (\textsc{TCons})}{\cdot \mid \Gamma_1 \mid (i+1) \times (length(x_1)+1) \vdash cons(x, (cons(x_0, x_1))) :\ [i \times (length(l) - 1)]_l\, List(int)}\ (\textsc{TDrop})$$

Combined together we can have:

$$\dfrac{\cdots \qquad \cdots}{\cdot \mid \Gamma_1' \mid (i+1) \times (length(x_1)+1) \vdash matd(x < x_0, \cdots) :\ [i \times (length(l) - 1)]_l\, List(int)}\ (\textsc{TDes})$$

Then on the upper level branch $nil(x_0).cons(x, x_0)$ we have:

$$\dfrac{\dfrac{\cdot \mid \Gamma_0,\, x_0 : nil \mid (i+1) \times length(x_0) \vdash x_0 : \big[(i+1) \times length(x_0)\big]_{x_0} List(int)}{\cdot \mid \Gamma_0,\, x_0 : nil \mid (i+1) \times length(x_0) \vdash cons(x, x_0) : \big[(i+1) \times (length(l)-1)\big]_{l} List(int)}\;(\text{TCons})}{\cdot \mid \Gamma_0,\, x_0 : nil \mid (i+1) \times length(x_0) \vdash cons(x, x_0) : \big[i \times (length(l)-1)\big]_{l} List(int)}\;(\text{TDrop})$$

Thus we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots \qquad \cdots}{\cdot \mid \Gamma_0 \mid (i+1) \times length(y) \vdash matd(y, \cdots) : \big[i \times (length(l)-1)\big]_{l} List(int)}\;(\text{TDes})}{\cdot \mid \Gamma_0 \mid (i+1) \times length(y) + 1 \vdash tick\ 1\ matd(y, \cdots) : \big[i \times (length(l)-1)\big]_{l} List(int)}\;(\text{TTickP})}{\cdot \mid \Gamma_0 \mid (i+1) \times (length(y) + 1) \vdash tick\ 1\ matd(y, \cdots) : \big[i \times length(l)\big]_{l} List(int)}\;(\text{TRelax})}{\cdot \mid \mathsf{insert} : T_{ins} \mid 0 \vdash \Lambda x : int.\, \lambda x.\, \lambda y.\, \cdots : \big[0\big]\, T_{ins}}\;(\text{TAbs+TPAbs})}{\cdot \mid \cdot \mid 0 \vdash fix\ \mathsf{insert} : T_{ins}.\, \cdots : \big[0\big]\, T_{ins}}\;(\text{TFix})$$

Then we can go the `sort`. Let the following be:

$$T_{srt} \overset{\text{def}}{=} \forall i : int.\, (z : List(int) \to l : List(int))_{[\frac{1}{2}(length(z)^2 + (3+2i) \times length(z)) \to i \times length(l)]}$$

$$\Gamma_2 \overset{\text{def}}{=} sort : T_{srt}, i : int,\ z : List(int), z_0,\ int,\ z_1 : List(int)$$

$$f(z) \overset{\text{def}}{=} \frac{1}{2}(length(z)^2 + (5 + 2i) \times length(z) + 2i + 2)$$

$$g(z) \overset{\text{def}}{=} \frac{1}{2}(length(z)^2 + (3 + 2i) \times length(z))$$

Then the inference process will be

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix} \cdot \mid \Gamma_2 \mid 0 \vdash sort\ (i+1) : \big[0\big]\, (z : List(int) \to l : List(int))_{[f(z) - i - 1 \to (i+1) \times length(l)]} \\ \cdot \mid \Gamma_2 \mid f(z_1) \vdash z_1 : \big[f(z_1)\big]_{z_1} List(int) \end{matrix}}{\cdot \mid \Gamma_2 \mid f(z_1) \vdash sort\ (i+1)\ z_1 : \big[(i+1) \times (length(l) + 1)\big]_{l} List(int)}\;(\text{TApp})}{\cdot \mid \Gamma_2 \mid f(z_1) \vdash insert\ i\ z_0\ sort\ (i+1)\ z_1 : \big[i \times length(l)\big]_{l} List(int)}\;(\text{TApp})}{\cdot \mid \Gamma_2 \mid f(z_1) + 1 \vdash tick\ 1\ \cdots : \big[i \times (length(l)\big]_{l} List(int)}\;(\text{TTickP})}{\cdot \mid sort : T_{srt}, i : int,\ z : List(int) \mid g(z) \vdash matd(z, \cdots) : \big[i \times (length(l))\big]_{l} List(int)}\;(\text{TDes})}{\cdot \mid sort : T_{srt}, i : int \mid 0 \vdash \lambda z.\, \cdots : \big[0\big]\, (z : List(int) \to l : List(int))_{[g(z) \to i \times length(l)]}}\;(\text{TAbs})}{\cdot \mid sort : T_{srt},\ \mid 0 \vdash \Lambda i : \cdots .\, int : \big[0\big]\, T_{srt}}\;(\text{TPAbs})}{\cdot \mid \cdot \mid 0 \vdash fix\ sort : T_{srt}.\, \cdots : \big[0\big]\, T_{srt}}\;(\text{TFix})}{}$$

With an instantiation of `sort` by 0, we can get our `sort` as $(z : List(int) \to l : List(int))_{[\frac{1}{2}(length(z)^2 + 3 \times length(z)) \to 0]}$.

A.4 Progress

Lemma 3 (Progress Weakening). *If $e \mid p \hookrightarrow e' \mid q$, and $p \le p'$, then there exists q' such that $e \mid p' \hookrightarrow e' \mid q'$*

Proof. By direct induction on $e \mid p \hookrightarrow e' \mid q$, the only case we need to care about is the tick. But it is again easy to prove by the linearity of addition.

Then we can prove the progress lemma.

Theorem 7 (Progress). *If $\Omega \mid \cdot \mid f \vdash e : \left[g\right]_x T$ and $\Omega \mid \cdot \vdash f \leq p$, then e is a value or exists e' q, $e \mid p \hookrightarrow e' \mid q$.*

Proof. By induction on the inductive hypothesis $\Omega \mid \cdot \mid f \vdash e : \left[g\right]_x T$.

- Case TAbs: e is already a value.
- Case TPAbs: e is already a value.
- Case TApp: By induction on $\Omega \mid \cdot \mid f \vdash e_1\, e_2 : \left[g\right]_y T$, suppose the induction hypothesis breaks into the following:

  1. H_1: $\Omega' \mid \Gamma \mid f_1 \vdash e_1 : \left[f_2\right]_z (x : T_1 \to y : T_2)_{[f_3 \to f_4]}$
  2. H_2: $\Omega' \mid \Gamma \mid f_5 \vdash e_2 : \left[f_6\right]_w T_1$
  3. H_3: $\Omega', x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \leq (f_2 + f_6)[w \mapsto x]$
  4. $Constraint_1$: $\Omega = \Omega', x : T_1$
  5. $Constraint_2$: $\Gamma = \cdot$
  6. $Constraint_3$: $f_1 + f_5 = f$
  7. $Constraint_4$: $(f_2 + f_6 - f_3)[w \mapsto x] + f_4 = g$
  8. $Constraint_5$: $T_2 = T$

  First by applying the inductive hypothesis on H_1, we have either e_1 is a value, or for all p_1 such that $\Omega \mid \cdot \vdash f_1 \leq p_1$, there exists e_1' and q_1 such that $e_1 \mid p_1 \hookrightarrow e_1' \mid q_1$.

  If e_1 is nnot a value, then since we have $\Omega \mid \cdot \vdash f \leq p$, thus we have $\Omega \mid \cdot \vdash f_1 \leq p$. By the reduction rule EAppl, we prove the case.

  Using the same technique we can also prove the case for e_1 is a value and e_2 is not. Finally is the case e_1 and e_2 are all values, because e_1 is of type $(x : T_1 \to y : T_2)_{[f_3 \to f_4]}$, so it has to be in the form of $\lambda x.\, e : (x : T_1 \to y : T_2)_{[f_3 \to f_4]}$. Then by the reduction rule EApp, we prove the case.
- Case TPApp: Same as TApp.
- Case TLet: Same as TApp.
- Case TVar: type context is empty, so it is not well-typed.

– Case TPAIR: Same as case TAPP1, by induction on $\Omega \mid \cdot \mid f \vdash (e_1, e_2) : [g]_x T$, and discuss the three cases where e_1 is not a value, e_1 is a value while e_2 is not, and both e_1 and e_2 are both values.

– Case TINT: e is already a value.

– Case TRELAX: By induction on $\Omega \mid \cdot \mid f \vdash e : [g]_x T$, suppose the induction hypothesis breaks into the following:

1. H_1: $\Omega \mid \Gamma \mid f_1 \vdash e : [f_2]_x T$
2. H_2: $\Omega \mid \Gamma \vdash f_3 \geq 0$
3. $Constraint_1$: $\Omega = \Omega$
4. $Constraint_2$: $\Gamma = \cdot$
5. $Constraint_3$: $f_1 + f_3 = f$
6. $Constraint_4$: $f_2 + f_3 = g$

By the inductive hypothesis on H_1, we have either e is a value, or for all p_1 such that $\Omega \mid \cdot \vdash f_1 \leq p_1$, there exists e' and q_1 such that $e \mid p_1 \hookrightarrow e' \mid q_1$.

If e is a value, then we prove the case. If e is not a value, then since we have $\Omega \mid \cdot \vdash f \leq p$, thus we have $\Omega \mid \cdot \vdash f_1 \leq p$, therefore $e \mid p_1 \hookrightarrow e' \mid q_1$ thus we prove the case.

– Case TProj_1 and TProj_2: By induction on $\Omega \mid \cdot \mid f \vdash \pi_1 e : [g]_x T$ or $\Omega \mid \cdot \mid f \vdash \pi_2 e : [g]_x T$, we have the induction hypothesis that either e is a value or not. If e is a not value, then apply the same technique in Case TApp we get the proof. If e is a value, then e has to be in the form of a pair (v_1, v_2), then by the reduction rule EPROJ1 and EPROJ2 we obtain the proof.

– Case TTICKN and TTICKP: By induction on $\Omega \mid \cdot \mid f \vdash tick\ q\ e : [g]_y T$, we get for TTICKN, it always get stepped because $q < 0$. For TTICKP, it is guaranteed that $\Omega \mid \cdot \vdash f \geq q$. Thus it can be safely stepped.

– Case TFIX: Fix-points can always be reduced without consuming potentials.

– Case TCONS: $\Omega \mid \cdot \mid f \vdash C_i(e_0, (e_1, ..., e_{m_i})) : [g]_x T$ by induction we obtain the cases where for $j \in [0, m_i]$, and for all $j' \in [0, j-1]$, $e_{j'}$ is a value while e_j is not, or the case that forall $j'' \in [0, m_i]$, $e_{j''}$ is a value. For the former cases, we can apply the same technique in Case TAPP and then we get the proof. For the later case, it is already a value.

– Case TDES: By direct induction on the judgement, we have the induction hypothesis that either e_0 is a value or not. If e_0 is a not value, then apply the same technique in case TAPP we get the proof. If e_0 is a value, then it has to be in the form of $C_i(v_0,(v_1,...,v_{m_i}))$, then by the reduction rule ECase we obtain the proof.

- Case TOp: Same as TCons.
- Case TDrop, TErase and TRename: We directly prove the case.

Thus we prove the progress.

A.5 Preservation

Next we can prove the preservation lemma, we first need to prove some auxiliary lemmas.

Lemma 4 (Value Strengthening). *If* $\Omega \,|\, \Gamma \,|\, f_1 \vdash v : [f_2]_x T$, *then* $\Omega \,|\, \Gamma \,|\, 0 \vdash v : [0]_x T$.

Proof. By direct induction on $\Omega \,|\, \Gamma \,|\, f_1 \vdash v : [f_2]_x T$ we obtain the proof.

Lemma 5 (Potentials of Values). *If* $\Omega \,|\, \Gamma \,|\, f_1 \vdash v : [f_2]_x T$, *then* $\Omega \,|\, \Gamma \vdash f_2[x \mapsto v] \leq f_1$.

Proof. By direct induction on the inductive hypothesis $\Omega \,|\, \Gamma \,|\, f_1 \vdash v : [f_2]_x T$, we prove the Rule (TAbs), (TPAbs), and (TInt) cases because here $f_1 = f_2 = 0$. For Rule (TErase), rule (TRelax), rule (TRename) and (TDrop), it is trivial. For Rule (TPair) and (TCons), since by induction we have for all the subcases that $\Omega \,|\, \Gamma \vdash f_2'[x' \mapsto v'] \leq f_1'$. Finally add up these inequations we prove the case.

Lemma 6 (Value Relaxing). *If* $\Omega \,|\, \Gamma \vdash f$ *and* $x \notin \Omega \cup \Gamma$, *then we have* $\Omega \,|\, \Gamma \,|\, f[x \mapsto v] \vdash v : [f]_x T$.

Proof. By direct induction on v, we have the following cases:

- Case v is the value don't have structures to decompose, where v is i, $\lambda x. e : (x : T_1 \to y : T_2)_{[f_1 \to f_2]}$,or $\Lambda X : e. T$.

  We notice that for every value here, we have $\Omega \,|\, \Gamma \,|\, 0 \vdash v : [0]_x T$, then by TRelax we have $\Omega \,|\, \Gamma \,|\, f[x \mapsto v] \vdash v : [f[x \mapsto v]]_x T$. Since potential function can not cotain structures for such T type, thus $\Omega,\, x : T \,|\, \Gamma \vdash f[x \mapsto v] = f$ because the actual value of x doesn't matter here. Then by the rule TDrop we prove the case.

- Case (v_1, v_2) or $C_i(v_0,(v_1,...,v_{m_i}))$. They are technically the same, we go with the proof for (v_1, v_2). Notice that for the final function $f_2[x_1 \mapsto \pi_1 x] + f_4[x_2 \mapsto \pi_2 x]$ in the type judgement $\Omega \,|\, \Gamma \,|\, f_1 + f_3 \vdash (e_1, e_2) : [f_2[x_1 \mapsto \pi_1 x] + f_4[x_2 \mapsto \pi_2 x]]_x T_1 \times T_2$, we can always separate the variable $\pi_1 X$ from $\pi_2 X$, thus by induction we prove the case.

Lemma 7 (Value Substitution on one variable). *If $\Omega \mid \Gamma_1,\, x : T_0,\, \Gamma_2 \mid f_1 \vdash e : \left[f_2\right]_y T_1$, and $\Omega \mid \Gamma_1 \mid 0 \vdash v : \left[0\right] T_0$, then $\Omega[x \mapsto v] \mid \Gamma_1, \Gamma_2[x \mapsto v] \mid f_1[x \mapsto v] \vdash e[x \mapsto v] : \left[(f_2[x \mapsto v])\right]_y T_1[x \mapsto v]$.*

Proof. By direct induction on $\Omega_1 \mid \Gamma_1,\, x : T_0,\, \Gamma_2 \mid f_1 \vdash e : \left[f_2\right]_y T_1$, we have the following Cases:

- Case TPAbs, TAbs, and TDes: Since all newly introduced variables are appended to the tail of Γ_2, Thus we can apply the induction hypothesis and obtain the proof (Also note that the type annotation changed accordingly).
- Case TApp, TPair, TRename, TErase, TProj1, TProj2, TTickn, TTickp, TOp, TLet, TPApp, and TCons. Since the substitution keeps the equality and linear order by definition, thus we prove the case.
- Case TRelax, the only minor case we need to care is that the local binder x is exactly the x we are substituting. But by lemma 6 we can prove the case.
- Case TInt: We directly obtain the proof.
- Case TVar: by lemma 4 we prove the case.

Lemma 8 (Potential instantiation on one variable). *If $\Omega \mid \cdot \mid f_1 \vdash e : \left[f_2\right]_x T,\ y : T' \in \Omega$ and $\Omega \mid \cdot \mid 0 \vdash v : \left[0\right]_x T$, then $\Omega[y \mapsto v] \mid \cdot \mid f_1[y \mapsto v] \vdash e[y \mapsto v] : \left[f_2[y \mapsto v]\right]_x T[y \mapsto v]$.*

Proof. First we add the $y : T$ to the context we get $\Omega \mid y : T \mid f_1 \vdash e : \left[f_2\right]_x T$. This is safe for that y shares the same type in both potential context and type context. Then by lemma 7 we prove the case.

Then we can go to the preservation.

Theorem 8 (Preservation). *If $\Omega \mid \cdot \mid f \vdash e : \left[g\right]_x T$ and $e \mid q \hookrightarrow e' \mid q'$, then there exists $f'\ y\ v$, such that $\Omega[y \mapsto v] \mid \cdot \mid f' \vdash e : \left[g[y \mapsto v]\right]_x T[y \mapsto v]$, and we have that $\Omega[y \mapsto v] \mid \cdot \vdash (q - f)[y \mapsto v] \leq q' - f'$.*

Proof. By direct induction on $\Omega \mid \cdot \mid f \vdash e : \left[g\right]_x T$, then we have

- Case TAbs, TInt, TPAbs, TFix: e is already a value, thus it can't be further reduced, so $e \mid q \hookrightarrow e' \mid q'$ causes a contradiction.
- Case TApp: By induction on $\Omega \mid \cdot \mid f \vdash e_1\ e_2 : \left[g\right]_y T$, we suppose the induction hypothesis breaks into the following:
  1. H_1: $\Omega' \mid \cdot \mid f_1 \vdash e_1 : \left[f_2\right]_z (x : T_1 \to y : T)_{[f_3 \to f_4]}$
  2. H_2: $\Omega' \mid \cdot \mid f_5 \vdash e_2 : \left[f_6\right]_w T_1$
  3. H_3: $\Omega',\, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \leq (f_2 + f_6)[w \mapsto x]$
  4. $Constraint_1$: $\Omega = \Omega',\, x : T_1$
  5. $Constraint_2$: $f_1 + f_5 = f$

6. $Constraint_3$: $(f_2 + f_6 - f_3)[w \mapsto x] + f_4 = g$

We first get either e_1 is a value, or e_1 is not a value. If e_1 is not a value, then the reduction of e_1 e_2 must take form of $e_1\ e_2 \mid q \hookrightarrow e_1'\ e_2 \mid q'$, which means $e_1 \mid q \hookrightarrow e_1' \mid q'$. Then by the induction hypothesis on H_1, we have that there exists f_1', y' and v such that:

1. $\Omega'[y' \mapsto v] \mid \cdot \mid f_1' \vdash e_1' : \left[f_2[y' \mapsto v]\right]_z ((x : T_1 \to y : T)_{[f_3 \to f_4]})[y' \mapsto v]$
2. $\Omega'[y' \mapsto v] \mid \cdot \vdash (q - f_1)[y' \mapsto v] \leq q' - f_1'$

Then applying 8 on H_2 we get

1. $\Omega'[y' \mapsto v] \mid \cdot \mid f_5[y' \mapsto v] \vdash e_2[y' \mapsto v] : \left[f_6[y' \mapsto v]\right]_w T_1[y' \mapsto v]$

Thus we take $f_1' + f_5[y' \mapsto v]$ as the existential f', then we have

1. $\Omega[y' \mapsto v] \mid \cdot \mid f_1' + f_5[y' \mapsto v] \vdash e_1'\ e_2 : \left[f[y' \mapsto v]\right]_y T[y' \mapsto v]$
2. $\Omega'[y' \mapsto v] \mid \cdot \vdash (q - f_1)[y' \mapsto v] - f_5[y' \mapsto v] \leq q' - f_1' - f_5[y' \mapsto v]$

Notice that $\Omega = \Omega', x : T_1$, thus the inequality (2) still holds after we add x to the Ω'. Thus we prove the case where e_1 is not a value. Similarly, we can prove the case where e_1 is a value and e_2 is not a value by the same proof strategy.

If e_1 and e_2 are both values, then the reduction will be $e_1\ e_2 \mid q \hookrightarrow e' \mid q$ where e' is the application result of e_1 on e_2 by rule EApp. We first apply the lemma 4 to H_1 and H_2 and get:

1. $\Omega' \mid \cdot \mid 0 \vdash e_1 : \left[0\right]_z (x : T_1 \to y : T)_{[f_3 \to f_4]}$
2. $\Omega' \mid \cdot \mid 0 \vdash e_2 : \left[0\right]_w T_1$

Then apply the substitution lemma 7 to (1) and (2), we get the application result will have:

1. $\Omega'[x \mapsto e_2] \mid \cdot \mid f_3[x \mapsto e_2] \vdash e' : \left[(f_4[x \mapsto e_2])\right]_y T[x \mapsto e_2]$

Next we find e_2 is a value, thus the local binder w here will not appear in type context. Therefore, f_2 and f_3 will contain no free occurrence of w, otherwise it is not well-typed. Then we can rewrite the H_3 to:

1. $\Omega'[x \mapsto e_2] \mid \cdot \vdash (f_2 + (f_6[w \mapsto x]) - f_3)[x \mapsto e_2] \geq 0$

We can now use TRelax to add both sides of typing on e' by $(f_2 + (f_6[w \mapsto x]) - f_3)[x \mapsto e_2]$.

1. $\Omega'[x \mapsto e_2] \mid \cdot \mid (f_2 + (f_6[w \mapsto x]))[x \mapsto e_2] \vdash e' : \left[g[x \mapsto e_2]\right]_y T[x \mapsto e_2]$

We then apply the lemma 5 to H_1 and H_2 to get:

1. $\Omega' \mid \cdot \vdash f_2[z \mapsto e_1] \leq f_1$
2. $\Omega' \mid \cdot \vdash f_6[w \mapsto e_2] \leq f_5$

Since z binds on an arrow type, while potential functions do not proceed over arrow type, thus we can rewrite the constraint as:

1. $\Omega'[x \mapsto e_2] \mid \cdot \vdash f_2[x \mapsto e_2] \leq f_1[x \mapsto e_2]$
2. $\Omega'[x \mapsto e_2] \mid \cdot \vdash f_6[w \mapsto x][x \mapsto e_2] \leq f_5[x \mapsto e_2]$

Then by $Constraint_2$ we have:

1. $\Omega'[x \mapsto e_2] \mid \cdot \vdash (f_2 + (f_6[w \mapsto x]))[x \mapsto e_2] \leq f[x \mapsto e_2]$

Therefore we can use a TRelax rule to add both sides of e' typing by $(f - ((f_2 + (f_6[w \mapsto x])))[x \mapsto e_2])$, and then followed by a TDrop rule that drops the right side by $(f - ((f_2 + (f_6[w \mapsto x])))[x \mapsto e_2])$ then we got:

1. $\Omega'[x \mapsto e_2] \mid \cdot \mid f[x \mapsto e_2] \vdash e' : \left[g[x \mapsto e_2]\right]_y T[x \mapsto e_2]$

Notice that $\Omega[x \mapsto e_2] = (\Omega', x : T_1)[x \mapsto e_2] = \Omega'[x \mapsto e_2]$, thus we prove the case since $\Omega[x \mapsto e_2] \,|\, \cdot \vdash (q - f)[x \mapsto e_2] = q - f[x \mapsto e_2]$ as the evaluation process is $e_1\ e_2 \mid q \hookrightarrow e' \mid q$.

– Case TPApp: Same as TApp but simpler, we can directly get the proof by applying the theorem 7.

– Case TLet: Same as TApp.

– Case TVar: it is not well-typed since type context is empty.

– Case TPair: By induction we can handle the case where e_1 is not a value or e_1 is a value while e_2 is not just as the case TApp. As for the case where e_1 and e_2 are both values, then it is a value it slfe.

– Case TFix: By induction it is easy to show if $\Omega \,|\, \Gamma, x : T \,|\, f_1 \vdash e[x \mapsto fix\ x : T.\, e] : \lfloor f_2 \rfloor_y T$ and $x \notin \mathbf{typefv}(e) \cup \mathbf{fv}(T)$, then $\Omega, x : T \,|\, \Gamma \,|\, f_1 \vdash e[x \mapsto fix\ x : T.\, e] : \lfloor f_2 \rfloor_y T$. By premise we have $\Omega \,|\, \Gamma, x : T \,|\, 0 \vdash e : \lfloor 0 \rfloor_y T$. Thus $\Omega, x : T \,|\, \Gamma \,|\, 0 \vdash e[x \mapsto fix\ x : T.\, e] : \lfloor 0 \rfloor_y T$, then by rule TErase we prove the case.

– Case TRelax: By induction on $\Omega \,|\, \cdot \,|\, f \vdash e : \lfloor g \rfloor_x T$, suppose the induction hypothesis breaks into the following:

1. H_1: $\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \lfloor f_2 \rfloor_x T$
2. H_2: $\Omega \,|\, \Gamma \vdash f_3$
3. $Constraint_1$: $\Omega = \Omega$
4. $Constraint_2$: $\Gamma = \cdot$
5. $Constraint_3$: $f_1 + f_3 = f$
6. $Constraint_4$: $f_2 + f_3 = g$

If e is a value, then it can not further step. If e is not a value, then by the inductive hypothesis on H_1, for $e \mid q \hookrightarrow e' \mid q'$, we have that $\Omega[y \mapsto v] \,|\, \cdot \,|\, f_1' \vdash e' : \lfloor f_2[y \mapsto v] \rfloor_x T$ and $\Omega[y \mapsto v] \,|\, \cdot \vdash q - f_1[y \mapsto v] \le q' - f_1'$. Thus we have $\Omega[y \mapsto v] \,|\, \cdot \,|\, f_1' + f_3[y \mapsto v] \vdash e' : \lfloor f_2[y \mapsto v] + f_3[y \mapsto v] \rfloor_x T[y \mapsto v]$ by TRelax and lemma 8, and that $\Omega[y \mapsto v] \,|\, \cdot \vdash q - f_1[y \mapsto v] - f_3[y \mapsto v] \le q' - f_1' - f_3[y \mapsto v]$. Thus we prove the case.

– Case TProj1 and TProj2: By induction on $\Omega \,|\, \cdot \,|\, f \vdash \pi_1 e : \lfloor g \rfloor_y T$ or $\Omega \,|\, \cdot \,|\, f \vdash \pi_2 e : \lfloor g \rfloor_y T$, we have the induction hypothesis that either e is a value or not. If e is a not value, then apply the same technique in case TApp we get the proof. If e is a value, then e has to be in the form of a pair (v_1, v_2), for the case TProj_1. We may get the rule $\Omega \,|\, \cdot \,|\, f_1 + f_3 \vdash (e_1, e_2) : \lfloor f_2[x_1 \mapsto \pi_1 y] + f_4[x_2 \mapsto \pi_2 y] \rfloor_y T_1 \times T_2$ from

TRelax, TDrop, TErase, TRename, TPair.

Rule TDrop, TErase, TRename has no effect on our proof since they don't change the input potenital f. For rule TRelax, we can always combine it with the previous inference rule. If the rule before TRelax is TRelax, then we can combine this two into one TRelax. If the rule before TRelax is TPair, then we can lift the relaxation to its first projection. Thus we only need to consider the case for TPair.

Suppose it is typed as followed with $\Omega, y : T_1 \times T_2 \,|\, \cdot \vdash f_5[x_1 \mapsto \pi_1 y] \leq f_2[x_1 \mapsto \pi_1 y] + f_4[x_2 \mapsto \pi_2 y]$:

$$\dfrac{\dfrac{\Omega \,|\, \cdot \,|\, f_1 \vdash v_1 : \left[f_2\right]_{x_1} T_1 \qquad \Omega \,|\, \cdot \,|\, f_3 \vdash v_2 : \left[f_4\right]_{x_2} T_2}{\Omega \,|\, \cdot \,|\, f_1 + f_3 \vdash (e_1, e_2) : \left[f_2[x_1 \mapsto \pi_1 y] + f_4[x_2 \mapsto \pi_2 y]\right]_y T_1 \times T_2}\,(\text{TPair})}{\Omega \,|\, \cdot \,|\, f_1 + f_3 \vdash \pi_1(v_1, v_2) : \left[f_5\right] T_1}\,(\text{TProj1})$$

Notice that $\Omega, x_1 : T_1 \,|\, \cdot \vdash (f_5[x_1 \mapsto \pi_1 y])[y \mapsto (x_1, v_2)] \leq (f_2[x_1 \mapsto \pi_1 y] + f_4[x_2 \mapsto \pi_2 y])[y \mapsto (x_1, v_2)]$, thus $\Omega, x_1 : T_1 \,|\, \cdot \vdash f_5 \leq f_2 + f_4[x_2 \mapsto v_2]$.

By lemma 5 we have $\Omega \,|\, \cdot \vdash f_4[x_2 \mapsto v_2] \leq f_3$. Thus by the TRelax we have $\Omega \,|\, \Gamma \,|\, f_1 + f_3 \vdash v_1 : \left[f_2 + f_4[x_2 \mapsto v_2]\right]_{x_1} T_1$. Then by TDrop we obtain the proof for the case TProj_1. Similar we can prove the case TProj_2.

– Case TTickN and TTickP: By induction on $\Omega \,|\, \cdot \,|\, f \vdash tick\ q\ e : \left[g\right]_x T$ we can easily get that $\Omega \,|\, \cdot \,|\, f' \vdash tick\ q\ e : \left[g\right]_x T$ and $\Omega \,|\, \cdot \vdash q - f = q' - f'$. Take y as a fresh variable we prove the case.

– Case TCons: $\Omega \,|\, \cdot \,|\, f \vdash C_i(e_0, (e_1, ..., e_{m_i})) : \left[g\right]_y ind\overrightarrow{(C, (T, m))}$ by induction we obtain the cases where for $j \in [0, m_i]$, and for all $j' \in [0, j-1]$, $e_{j'}$ is a value while e_j is not, or the case that forall $j'' \in [0, m_i]$, $e_{j''}$ is a value. For the former cases, we can apply the same technique in case TApp and then we get the proof. For the later case, it cannot be further reduced.

– Case TOp: Same as TCons.

– Case TDes: By direct induction on the judgement, we have the induction hypothesis that either e_0 is a value or not. If e_0 is a not value, then apply the same technique in Case TApp we get the proof. If e_0 is a value, then it has to be in the form of $C_i(v_0, (v_1, ..., v_{m_i}))$, then by apply the substitution lemma 7 in the premise for multiple times get the proof.

– Case TRename, TDrop, TErase: We directly prove the case.

Theorem 9 (Soundness). *If $\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T$, then $\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \left[f_2\right]_x T$.*

Proof. By direct induction of $\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : [f_2]_x T$, The only difference is in Aproj1, Aproj2, ATickn, AAppn, ACons, ADes. All of them can be achieved by adding the TRelax and TDrop, thus we prove the case.

Theorem 10 (Determinisism). *If $\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : [f_2]_x T$, $\Omega \,|\, \Gamma \,|\, f_3 \vdash_a e : [f_4]_x T$, then $\Omega \,|\, \Gamma \vdash f_1 = f_3$, and $\Omega, x : T \,|\, \Gamma \vdash f_2 = f_4$.*

Proof. It is easy to prove for all $\Omega, \Gamma, f_1, e, f_2, T, x$, if $\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : [f_2]_x T$ and $y \notin \Omega \cup \Gamma$, then $\Omega, y : T_1 \,|\, \Gamma \,|\, f_1 \vdash_a e : [f_2]_x T$. Notice that for fixed Ω, Γ, e, x, we can only get it from either TErase, TRename or another concrete rule. It is easy to show that TErase commutes with all other rule, and TRename has no effect on all of the subsequent typing judgement Ω, Γ, f_1, while f_2 is equivalent up to variable renaming. Since $\Omega, \Gamma, f_1, e, T, x$ are all fixed, thus $f_3 = f_4$.

A.6 Full Algorithm

The full rules for algorithm shows here:

A.7 Embedding

We choose the AARA variant presented in [21], which serves as a comprehensive summary of two decades of development in automatic amortized resource analysis.

The syntax is given as follows:

$$\begin{aligned}
e ::=\ & x \\
& \mid \langle\rangle \\
& \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
& \mid \langle e_1, e_2 \rangle \\
& \mid \textbf{letp}\ \langle x_1, x_2 \rangle = e_1\ \textbf{in}\ e_2 \\
& \mid \textbf{left}(e) \mid \textbf{right}(e) \\
& \mid \textbf{case}\ e\{\, \textbf{left}(x_1) \mapsto e_1 \mid \textbf{right}(x_2) \mapsto e_2 \,\} \\
& \mid \textbf{nil} \mid \textbf{cons}(x_1, x_2) \\
& \mid \textbf{case}\ x\{\, \textbf{nil} \mapsto e_0 \mid \textbf{cons}(x_1, x_2) \mapsto e_1 \,\} \\
& \mid x_1(x_2) \\
& \mid \textbf{fun}\ f\ x = e \\
& \mid \textbf{tick}\ q \\
& \mid \textbf{share}\ x\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e
\end{aligned}$$

In AARA, both construction and deconstruction operations incur explicit resource costs, such as c_{Let_1} or c_{app}. All such costs can be uniformly encoded using the cost construct *tick c e*.

Notably, the report [21] does not include typing rules for the pair constructor $\langle e_1, e_2\rangle$ or the destructor $\textbf{letp}\ \langle x_1, x_2\rangle = e_1\ \textbf{in}\ e_2$. However, pairs can be straightforwardly encoded in AARA using the **cons** constructor. Concretely,

$$\langle e_1, e_2\rangle \ \equiv\ \textit{let}\ x_1 = e_1\ \textit{in let}\ x_2 = e_2\ \textit{in let}\ x_3 = \textbf{cons}(x_2, \textbf{nil})\ \textit{in}\ \textbf{cons}(x_1, x_3).$$

Similarly,

$$\textbf{letp}\ \langle x_1, x_2\rangle = e_1\ \textbf{in}\ e_2 \ \equiv\ \textit{let}\, x = e_1\ \textit{in}\, \textbf{case}\ x\{\textbf{nil} \mapsto \cdots\ \text{(unused branch)} \mid \textbf{cons}(x_1, x_2) \mapsto e_2\}.$$

Therefore, we omit further discussion of pair translation in what follows.

We use the following encodings of standard types:

$$\begin{aligned}
\mathsf{Unit} &= \mathsf{ind}(\{\}),\\
\mathsf{List}(T) &= \mathsf{ind}(\{\textbf{nil}(\mathsf{Unit}, 0), \textbf{cons}(T, 1)\}),\\
T_1 + T_2 &= \mathsf{ind}(\{\textbf{left}(T_1, 0), \textbf{right}(T_2, 0)\}).
\end{aligned}$$

Our translation function h is defined as follows:

$$\begin{aligned}
h(x) &= \textit{tick}\ c_{\mathsf{var}}\ x\\
h(\langle\rangle) &= \textit{tick}\ c_{\mathsf{Unit}}\ \mathsf{Unit}\\
h(\textit{let}\ x = e_1\ \textit{in}\ e_2) &= \textit{tick}\ c_{\mathsf{Let}_3}\ \textit{let}\ x = \textit{tick}\ c_{\mathsf{Let}_1}\ h(e_1)\ \textit{in tick}\ c_{\mathsf{Let}_2}\ h(e_2)\\
h(\textbf{left}(e)) &= \textit{tick}\ c_{\mathsf{left}}\ \textbf{left}(h(e))\\
h(\textbf{right}(e)) &= \textit{tick}\ c_{\mathsf{right}}\ \textbf{right}(h(e))\\
h(\textbf{case}\ x\{\textbf{left}(x_1) \mapsto e_1 \mid \textbf{right}(x_2) \mapsto e_2\}) &= \mathsf{matd}\big(x, \{\textbf{left}(x_1).\textit{tick}\ c_{\mathsf{CaseLeft}}\ h(e_1),\\
&\qquad \textbf{right}(x_2).\textit{tick}\ c_{\mathsf{CaseRight}}\ h(e_2)\}\big)\\
h(\textbf{nil}) &= \textit{tick}\ c_{\mathsf{nil}}\ \mathsf{Nil}()\\
h(\textbf{cons}(x_1, x_2)) &= \textit{tick}\ c_{\mathsf{cons}}\ \mathsf{Cons}(x_1, x_2)\\
h(\textbf{case}\ x\{\textbf{nil} \mapsto e_0 \mid \textbf{cons}(x_1, x_2) \mapsto e_1\}) &= \mathsf{matd}\big(x, \{\textbf{nil}.\textit{tick}\ c_{\mathsf{CaseNil}}\ h(e_0),\\
&\qquad \textbf{cons}(x_1, x_2).\textit{tick}\ c_{\mathsf{CaseCons}}\ h(e_1)\}\big)\\
h(x_1\, x_2) &= \textit{tick}\ c_{\mathsf{app}}\ x_1\ x_2\\
h(\textbf{fun}\ f\, x = e) &= \textit{tick}\ c_{\mathsf{fun}}\ (\textbf{fix}\ f.\lambda x.\, h(e))\\
h(\textit{tick}\ q\ ) &= \textit{tick}\ q\ \mathsf{Unit}\\
h(\textbf{share}\ x\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e) &= h(e[x_1 \mapsto x,\ x_2 \mapsto x])
\end{aligned}$$

Most of the translation are straightforwards to show as the direction translation, except for the $\textbf{share}\ x\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e$. Since in our type system, the type no longer carries potentials, thus we don't need to split the type.

Now we do some definitions that will be used in our proof.

Definition 1 (Potentials and Types). *We first define the potentials $\Phi_x(A)$ of a type A so that it returns the potential function of a program variable x of* *the type A.*

$$\begin{aligned}
&\Phi_x(1) &&= Unit\\
&\Phi_x(L^p(A)) &&= fix\ f\ matd(x, \{nil(x).0, cons(x_1, x_2).(\Phi_{x_1}(x_1) + p + f\ x_2)\}\\
&\Phi_x(A^p + B^r) &&= fix\ f\ matd(x, \{left(x).(\Phi_x(A) + p), right(x).(\Phi_x(B) + r)\}\\
&\Phi_x(A \times B) &&= \Phi_x(A)[x \mapsto \pi_1 x] + \Phi_x(B)[x \mapsto \pi_2 x]\\
&\Phi_x(A \xrightarrow{p/q} B) = 0
\end{aligned}$$

We also define types $Type(\cdot)$ of a AARA types as follows:

$$\begin{aligned}
&Type(1) &&= Unit\\
&Type(L^p(A)) &&= List(Type(A))\\
&Type(A^p + B^r) &&= Type(A) + Type(B)\\
&Type(A \times B) &&= Type(A) \times Type(B)\\
&Type(A \xrightarrow{p/q} B) &&= (x : Type(A) \to y : Type(B))_{[p+\Phi_x(A) \to q+\Phi_y(B)]}
\end{aligned}$$

Thus the potential function of a context Γ in AARA system is defined as follows:

$$\begin{aligned}
&\Phi(x,\ A : \Gamma) &&= \Phi_x(A) + \Phi(\Gamma)\\
&\Phi(\cdot) &&= 0
\end{aligned}$$

We can also define the type context translation as follows:

$$\begin{aligned}
&Pure(x,\ A : \Gamma) &&= x, Type(A) : Pure(\Gamma)\\
&\Phi(\cdot) &&= \cdot
\end{aligned}$$

Next, we prove the embedding. Before presenting the main proof, we first establish two useful lemmas.

Definition 2 (Context Substitution). *For an AARA typing context Γ, we define context substitution $[x \mapsto y](\Gamma)$ as follows:*

– *If $\Gamma = \Gamma_1,\ x{:}A_1,\ \Gamma_2,\ y{:}A_2,\ \Gamma_3$ and $A_3 \curlyvee (A_1, A_2)$, then*

$$[x \mapsto y](\Gamma) = \Gamma_1,\ \Gamma_2,\ y{:}A_3,\ \Gamma_3.$$

– *If $\Gamma = \Gamma_1,\ y{:}A_1,\ \Gamma_2,\ x{:}A_2,\ \Gamma_3$ and $A_3 \curlyvee (A_1, A_2)$, then*

$$[x \mapsto y](\Gamma) = \Gamma_1,\ y{:}A_3,\ \Gamma_2,\ \Gamma_3.$$

– *If $\Gamma = \Gamma_1,\ x{:}A_1,\ \Gamma_2$ and $y \notin Var(\Gamma)$, then*

$$[x \mapsto y](\Gamma) = \Gamma_1,\ y{:}A_1,\ \Gamma_2.$$

– *If $x \notin Var(\Gamma)$, then*

$$[x \mapsto y](\Gamma) = \Gamma.$$

– *Otherwise,* $[x \mapsto y](\Gamma)$ *is undefined.*

Note that for any A_1 *and* A_2 *such that* $Type(A_1) = Type(A_2)$*, a type* A_3 *satisfying* $A_3 \curlyvee (A_1, A_2)$ *always exists.*

We now state a lemma that holds in the AARA system and a lemma that holds in $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ that will be used in the embedding proof.

Theorem 11 (AARA Substitution). *If* $\Gamma \mid\!\frac{p}{q}\; e : A$*, then for all variables* x *and* y*, if either (i) both* x *and* y *appear in* Γ *with* $x : A_1$ *and* $y : A_2$ *and* $Type(A_1) = Type(A_2)$*, or (ii) at most one of* x *and* y *appears in* Γ*, then*

$$([x \mapsto y]\Gamma) \mid\!\frac{p}{q}\; ([x \mapsto y]e) : A.$$

Proof. The proof proceeds by direct induction on the AARA typing derivation and is straightforward.

Theorem 12 (Weakening). *If* $\Gamma \mid \cdot \mid f_1 \vdash e : \left[f_2\right]_x A$ *and* y *is a fresh variable with respect to* Γ*, then*

$$\Gamma, y : B \mid \cdot \mid f_1 \vdash e : \left[f_2\right]_x A.$$

Proof. The proof proceeds by direct induction on the typing derivation of $\lambda^{\mathsf{na}}_{\mathsf{amor}}$ and is straightforward.

We now proceed to the main embedding proof.

Theorem 13 (AARA embedding). *For* $\Gamma \mid\!\frac{p}{q}\; e : A$*, there exists a potential function* $\Phi'_x(A)$ *such that* $Pure(\Gamma) \mid \cdot \vdash q + \Phi_x(A) \leq \Phi'_x(A)$ *and*

$$Pure(\Gamma) \mid \cdot \mid \Phi(\Gamma) + p \vdash h(e) : \left[\Phi'_x(A)\right]_x Type(A).$$

Proof. We prove the theorem by induction on the AARA typing derivation $\Gamma \mid\!\frac{p}{q}$ $e : A$.

– **Case L:Var.** The AARA typing rule is:

$$\text{(L:V\small{AR})}\quad \frac{q \geq q' + c_{Var}}{x : A \mid\!\frac{q}{q'}\; x : A}$$

By translation, we must show that there exists $\Phi'_x(A)$ such that $x : Type(A) \mid \cdot \vdash q + \Phi_x(A) \leq \Phi'_x(A)$ and

$$x : Type(A) \mid \cdot \mid \Phi_x(A) + q \vdash tick\ c_{Var}\ x : \left[\Phi'_x(A)\right]_x Type(A).$$

Using rules TTick, TVar, and TRelax, we derive:

$$x : Type(A) \mid \cdot \mid \Phi_x(A) + q \vdash tick\ c_{Var}\ x : \left[\Phi_x(A) + q - c_{Var}\right]_x Type(A).$$

Taking $\Phi'_x(A) = \Phi_x(A) + q - c_{Var}$ completes this case.

- **Case L:Unit, L:Let, L:Left, L:Right, L:MatchSum, L:Nil, L:Cons, L:MatchList.** These cases follow analogously to Case L:Var.
- **Case L:App.**

$$\text{(L:App)}\quad \frac{q \geq p + c_{App} \qquad q - q' \geq p - p' + c_{App}}{x_1 : A \xrightarrow{p/p'} B, x_2 : A \,\Big|_{q'}^{q}\, x_1\ x_2 : B}$$

By translation, we must show

$x_1 : (x : Type(A) \rightarrow y : Type(B))_{[p+\Phi_x(A) \rightarrow q + \Phi_y(B)]},\ x_2 : Type(A) \mid \cdot \mid q + \Phi_{x_2}(A) \vdash$ $tick\ c_{app}\ (x_1\ x_2) : [q' + \Phi_y(B)]_y Type(B)$.

By rule TApp, we obtain

$x_1 : (x : Type(A) \rightarrow y : Type(B))_{[p+\Phi_x(A) \rightarrow q + \Phi_y(B)]},\ x_2 : Type(A) \mid \cdot \mid q + \Phi_{x_2}(A) \vdash$ $tick\ c_{app}\ (x_1\ x_2) : \big[q - p - c_{App} + \Phi_y(B)\big]_y Type(B)$.

Since $q \geq p + c_{App}$, this judgment is well-typed. Moreover, from $q - q' \geq p - p' + c_{App}$, taking $\Phi'_y(B) = q - p - c_{App} + \Phi_y(B)$ completes the proof of this case.

- **Case L:fun.**

$$\text{(L:fun)}\quad \frac{\Gamma \curlyvee (\Gamma, \Gamma) \qquad \Gamma, f : A \xrightarrow{p/p'} B, x : A \,\Big|_{p'}^{p}\, e : B \qquad q \geq q' + c_{fun}}{\Gamma \,\Big|_{q'}^{q}\, \textbf{fun}\ f\, x = e : B}$$

Since $\Gamma \curlyvee (\Gamma, \Gamma)$, we have $\Phi(\Gamma) = 0$. By translation, it suffices to show

$Pure(\Gamma),\ f : (x : Type(A) \rightarrow y : Type(B))_{[p+\Phi_x(A) \rightarrow q + \Phi_y(B)]}, x : Type(A) \mid \cdot \mid q + \Phi_{x_2}(A) \vdash$ $tick\ c_{fun}\ (fix\ f.\lambda x.h(e)) : [q' + \Phi_y(B)]_y Type(B)$.

By the induction hypothesis, there exists $\Phi'_y(B)$ such that

$$Pure(\Gamma, f : A \xrightarrow{p/p'} B, x : A) \mid \cdot \vdash p' + \Phi_y(B) \leq \Phi'_y(B)$$

and

$$Pure(\Gamma, f : A \xrightarrow{p/p'} B, x : A) \mid \cdot \mid p + \Phi_x(A) \vdash e : \big[\Phi'_y(B)\big]_y Type(B).$$

Consequently,

$Pure(\Gamma),\ f : (x : Type(A) \rightarrow y : Type(B))_{[p+\Phi_x(A) \rightarrow q + \Phi_y(B)]}, x : Type(A) \mid \cdot \mid c_{fun} + \Phi_{x_2}(A) \vdash$ $tick\ c_{fun}\ (fix\ f.\lambda x.h(e)) : \big[\Phi_y(B)\big]_y Type(B)$.

Applying rule TRelax yields the desired judgment.

- **Case L:Relax.**

$$\text{(L:Relax)}\quad \frac{\Gamma \left|\frac{p}{p'}\right. e : A \qquad q \geq p \qquad q - q' \leq p - p'}{\Gamma \left|\frac{q}{q'}\right. e : A}$$

  By induction, there exists $\Phi'_x(A)$ such that

$$Pure(\Gamma) \mid \cdot \vdash p' + \Phi_x(A) \leq \Phi'_x(A)$$

  and

$$Pure(\Gamma) \mid \cdot \mid \Phi(\Gamma) + p \vdash h(e) : \left[\Phi'_x(A)\right]_x Type(A).$$

  Let $\Phi''_x(A) = \Phi'_x(A) + q - p$. Applying rule TRelax yields

$$Pure(\Gamma) \mid \cdot \vdash q - p + p' + \Phi_x(A) \leq \Phi''_x(A)$$

  and

$$Pure(\Gamma) \mid \cdot \mid \Phi(\Gamma) + q \vdash h(e) : \left[\Phi''_x(A)\right]_x Type(A).$$

  This completes the case.
- **Case L:Weak.** This case follows directly from Theorem 12, and then reduces to Case L:Relax.
- **Case L:share.**

$$\text{(L:share)}\quad \frac{A \curlyvee (A_1, A_2) \qquad \Gamma,\ x_1 : A_1,\ x_2 : A_2 \left|\frac{q}{q'}\right. e : B}{\Gamma, x : A \left|\frac{q}{q'}\right. e : B}$$

  By Theorem 11, we have

$$\Gamma, x : A \left|\frac{q}{q'}\right. [x_1 \mapsto x, x_2 \mapsto x]e : B.$$

  By induction (note that $\Gamma, x : A$ contains strictly fewer variables than $\Gamma, x_1 : A_1, x_2 : A_2$), there exists $\Phi'_x(A)$ such that

$$Pure(\Gamma) \mid \cdot \vdash q' + \Phi_x(A) \leq \Phi'_x(A)$$

  and

$$Pure(\Gamma) \mid \cdot \mid \Phi(\Gamma) + q \vdash h(e) : \left[\Phi'_x(A)\right]_x Type(A).$$

  This corresponds exactly to the translation

$$h(\textbf{share } x \textbf{ as } x_1, x_2 \textbf{ in } e) = h(e[x_1 \mapsto x, x_2 \mapsto x]),$$

  and therefore the case is proved.

A.8 Untypability of AARA

In this section, we formally show that AARA is unable to type the *map-append* example as well as the *append* example introduced earlier.

To this end, it suffices to show that the term *append* e_1 e_2 is not typable in AARA, and that the intermediate closure *append* e_1 already leads to a problematic typing. In AARA, the closure *append* e_1 must have a type of the form

$$List^{p_2}(\mathsf{int}) \xrightarrow{p_3/q_3} List^{q_2}(\mathsf{int}),$$

and therefore the only admissible typing judgment for *append* e_1 is

$$\cdot \,\Big|^{p_1}_{q_1}\, append\ e_1 : List^{p_2}(\mathsf{int}) \xrightarrow{p_3/q_3} List^{q_2}(\mathsf{int}).$$

Consequently, for the application *append* e_1 e_2, the function consumes $p_2 \cdot \mathsf{length}(e_2)$ units of resource and produces $q_2 \cdot \mathsf{length}(e_1 + e_2)$ units of resource. Since p_1, q_1, p_2, q_2, p_3, and q_3 are all constants that do not depend on e_1, we may choose e_1 such that

$$\mathsf{length}(e_1) > \frac{p_3 + p_1}{q_2}.$$

Next, let e_2 be the empty list nil. In this case, the output potential $q_2 \cdot \mathsf{length}(e_1 + \mathsf{nil}) = q_2 \cdot \mathsf{length}(e_1)$ is strictly greater than the total input potential available.

This contradicts the soundness of AARA, which guarantees that all typable terms can be evaluated safely without exceeding available resources. Hence, the term *append* e_1 e_2 cannot be typable in AARA.

Therefore, *append* e_1 e_2—and consequently higher-order examples such as *map-append*—are not typable in AARA.

(AProj1)

$$\frac{y, z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T_1 \times T_2}{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a \pi_1 e : \left[\mathbf{min}_{z:T_2}(f_2[x \mapsto (y,z)])\right]_y T_1}$$

(AVar)

$$\frac{x : T \in \Gamma}{\Omega \,|\, \Gamma \,|\, 0 \vdash_a x : \left[0\right]_x T}$$

(AProj2)

$$\frac{y, z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T_1 \times T_2}{\Omega \,|\, \Gamma \,|\, f_1 \vdash \pi_2 e : \left[\mathbf{min}_{y:T_1}(f_2[x \mapsto (y,z)])\right]_z T_2}$$

(AInt)

$$\frac{}{\Omega \,|\, \Gamma \,|\, \left[0\right] \vdash_a i : \left[0\right]_x int}$$

(AErase)

$$\frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_y T \quad x : T' \in \Omega \quad x \notin \mathbf{fv}(\Omega) \cup \mathbf{fv}(\Gamma) \cup \mathbf{fv}(f_1) \cup \mathbf{fv}(f_2) \cup \mathbf{fv}(T)}{\Omega \backslash x \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_y T}$$

(ATickP)

$$\frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T \quad p \geq 0}{\Omega \,|\, \Gamma \,|\, f_1 + p \vdash_a tick\ p\ e : \left[f_2\right]_x T}$$

(ATickN)

$$\frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T \quad p < 0}{\Omega \,|\, \Gamma \,|\, \mathbf{max}(f_1 + p, 0) \vdash_a tick\ p\ e : \left[f_2 - \mathbf{min}(f_1 + p, 0)\right]_x T}$$

(ARename)

$$\frac{x, y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_x T}{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2[x \mapsto y]\right]_y T}$$

(AOp)

$$\frac{y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \forall j \in [1, m_i], \Omega \,|\, \Gamma \,|\, f_{1_j} \vdash_a e_j : \left[f_{2_j}\right]_{x_j} int}{\Omega \,|\, \Gamma \,|\, \sum_{j=1}^{m_i} f_{1_j} \vdash_a \mathbf{op_i}(\overrightarrow{e}) : \left[\sum_{j=1}^{m_i} f_{2_j}\right]_y int}$$

(AFix)

$$\frac{z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma, x : T \,|\, 0 \vdash_a e : \left[f\right]_y T}{\Omega \,|\, \Gamma \,|\, 0 \vdash_a fix\ x : T.\, e : \left[0\right]_z T}$$

(APAbs)

$$\frac{z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma, x : T_1 \,|\, 0 \vdash_a e : \left[f\right]_y T_2}{\Omega \,|\, \Gamma \,|\, 0 \vdash_a \Lambda x : T_1.\, e : \left[0\right]_z \forall x : T_1.\, T_2}$$

(APApp)

$$\frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e : \left[f_2\right]_y \forall x : T_1.\, T_2 \quad \Omega \,|\, \Gamma \,|\, 0 \vdash_a pv : \left[0\right]_z T_1 \quad w \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e\ pv : \left[f_2\right]_w T_2[x \mapsto pv]}$$

(APair)

$$\frac{\Omega \,|\, \Gamma \,|\, f_1 \vdash_a e_1 : \left[f_2\right]_{x_1} T_1 \quad \Omega \,|\, \Gamma \,|\, f_3 \vdash_a e_2 : \left[f_4\right]_{x_2} T_2 \quad x \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \,|\, \Gamma \,|\, f_1 + f_3 \vdash_a (e_1, e_2) : \left[f_2[x_2 \mapsto \pi_2 x] + f_4[x_1 \mapsto \pi_1 x]\right]_x T_1 \times T_2}$$

(AAbs)

$$\frac{\Omega \,|\, \Gamma, x : T_1 \,|\, f_3 \vdash_a e : \left[f_4\right]_y T_2 \quad z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \quad \Omega \,|\, \Gamma, x : T_1 \vdash f_3 \leq f_1 \quad \Omega, y : T_2 \,|\, \Gamma, x : T_1 \vdash f_2 + f_3 \leq f_4 + f_1}{\Omega \,|\, \Gamma \,|\, 0 \vdash_a \lambda x : T_1.\, e : \left[0\right]_z ((x : T_1 \to y : T_2)_{[f_1 \to f_2]})}$$

Fig. 5: Algorithmic Typing Rules

(AAppp)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash_a e_1 : \left[f_2\right]_z((x:T_1 \to y:T_2)_{[f_3 \to f_4]}) \\ \Omega \mid \Gamma \mid f_5 \vdash_a e_2 : \left[f_6\right]_w T_1 \qquad \Omega,\, x:T_1 \mid \Gamma \vdash f_3[w \mapsto x] \le (f_2+f_6)[w \mapsto x]\end{array}}{\Omega,\, x:T_1 \mid \Gamma \mid f_1+f_5 \vdash_a e_1\ e_2 : \left[(f_2+f_6-f_3)[w\mapsto x]+f_4\right]_y T_2}$$

(AAppn)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash_a e_1 : \left[f_2\right]_z((x:T_1 \to y:T_2)_{[f_3 \to f_4]}) \qquad \Omega \mid \Gamma \mid f_5 \vdash_a e_2 : \left[f_6\right]_w T_1 \\ \Omega,\, x:T_1 \mid \Gamma \vdash f_3[w\mapsto x] > (f_2+f_6)[w\mapsto x] \qquad \textbf{if } w \notin \mathsf{dom}(\Omega)\cup\mathsf{dom}(\Gamma), \\ \textbf{then } f = \mathbf{min}_{x:T_1}((f_3-f_2-f_6)[w\mapsto x]) \qquad \textbf{else } f = f_3-f_2-f_6\end{array}}{\Omega,\, x:T_1 \mid \Gamma \mid f+f_1+f_5 \vdash_a e_1\ e_2 : \left[(f+f_2+f_6-f_3)[w\mapsto x]+f_4\right]_y T_2}$$

(ALet)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash_a e_1 : \left[f_2\right]_z T_1 \\ \Omega \mid \Gamma,\, x:T_1 \mid f_3 \vdash_a e_2 : \left[f_4\right]_y T_2 \qquad \Omega \mid \Gamma,\, x:T_1 \vdash f_3 \le f_2\end{array}}{\Omega,\, x:T_1 \mid \Gamma \mid f_1+f_5 \vdash_a \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \left[f_2-f_3+f_4\right]_y T_2}$$

(ACons)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash_a e_0 : \left[f_2\right]_{x_0} T_i \qquad \forall j \in [1,m_i], \Omega \mid \Gamma \mid f_{3_j} \vdash_a e_j : \left[f_{4_j}\right]_{x_j} \mathit{ind}\overrightarrow{(C,(T,m))} \\ f_5 = \mathit{matd}(y, C_i(x_0,(x_1,...,x_{m_i})).(f_2+\sum_{j=1}^{m_i} f_{4_j}), \overrightarrow{C_{j(i\neq j)}(x_0,(x_1,...,x_{m_j})).+\infty})\end{array}}{\Omega \mid \Gamma \mid f_1 + \sum_{j=1}^{m_i} f_{3_j} \vdash_a C_i(e_0,(e_1,...,e_{m_i})) : \left[f_5\right]_y \mathit{ind}\overrightarrow{(C,(T,m))}}$$

(ADes)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash_a e_0 : \left[f_2\right]_x \mathit{ind}\overrightarrow{(C,(T,m))} \\ \forall i, \Omega \mid \Gamma,\, x_0:T_i, ...,\, x_{m_i} : \mathit{ind}\overrightarrow{(C,(T,m))} \mid f_{3i} \vdash_a e_i : \left[f_{4i}\right]_y T_1 \\ f = \mathbf{max}(\mathbf{max}_i(\mathbf{max}_{x_0:T_i,...,\, x_{m_i}:\mathit{ind}\overrightarrow{(C,(T,m))}}(f_{3i})), f_2) \\ \textbf{if } x \notin \Omega\cup\Gamma, \textbf{ then } f_5 = \mathbf{min}_{x:\mathit{ind}\overrightarrow{(C,(T,m))}}(f-f_2) \qquad \textbf{else } f_5 = f-f_2\end{array}}{\Omega \mid \Gamma \mid f_1+f_5 \vdash_a \mathit{matd}(e_0, \overrightarrow{C(x_0,(x_1,...,x_m).e)}) : \left[\mathbf{min}_i(\mathbf{min}_{x_0:T_i,...,\, x_{m_i}:\mathit{ind}\overrightarrow{(C,(T,m))}}(f_5+f_2-f_{3i}+f_{4i}))\right]_y T_1}$$

Fig. 6: Algorithmic Typing Rules

(TTickP)
$$\frac{\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T \qquad p \geq 0}{\Omega \mid \Gamma \mid f_1 + p \vdash \mathit{tick}\ p\ e : \left[f_2\right]_x T}$$

(TTickN)
$$\frac{\Omega \mid \Gamma \mid f_1 - p \vdash e : \left[f_2\right]_x T \qquad p < 0}{\Omega \mid \Gamma \mid f_1 \vdash \mathit{tick}\ p\ e : \left[f_2\right]_x T}$$

(TDrop)
$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T \\ \Omega, x : T \mid \Gamma \vdash f_3 \leq f_2\end{array}}{\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_3\right]_x T}$$

(TInt)
$$\frac{x \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \mid \Gamma \mid \left[0\right] \vdash i : \left[0\right]_x \mathit{int}}$$

(TFix)
$$\frac{\begin{array}{c}z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ x \notin \mathbf{typefv}(e) \cup \mathbf{fv}(T) \\ \Omega \mid \Gamma, x : T \mid 0 \vdash e : \left[0\right]_y T\end{array}}{\Omega \mid \Gamma \mid 0 \vdash \mathit{fix}\ x : T.\, e : \left[0\right]_z T}$$

(TRename)
$$\frac{\begin{array}{c}x, y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T\end{array}}{\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2[x \mapsto y]\right]_y T[x \mapsto y]}$$

(TProj1)
$$\frac{\begin{array}{c}y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T_1 \times T_2 \\ \Omega, x : T_1 \times T_2 \mid \Gamma \vdash f_3[y \mapsto \pi_1 x] \leq f_2\end{array}}{\Omega \mid \Gamma \mid f_1 \vdash \pi_1 e : \left[f_3\right]_y T_1}$$

(TProj2)
$$\frac{\begin{array}{c}y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T_1 \times T_2 \\ \Omega, x : T_1 \times T_2 \mid \Gamma \vdash f_3[y \mapsto \pi_2 x] \leq f_2\end{array}}{\Omega \mid \Gamma \mid f_1 \vdash \pi_2 e : \left[f_3\right]_y T_2}$$

(TVar)
$$\frac{x : T \in \Gamma}{\Omega \mid \Gamma \mid 0 \vdash x : \left[0\right]_x T}$$

(TRelax)
$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e : \left[f_2\right]_x T \\ \Omega \mid \Gamma \vdash f_3 \geq 0\end{array}}{\Omega \mid \Gamma \mid f_1 + f_3 \vdash e : \left[f_2 + f_3\right]_x T}$$

Fig. 7: Typing Rules

(TPair)

$$\frac{\begin{array}{c}\Omega \,|\, \Gamma \,|\, f_1 \vdash e_1 : \left[f_2\right]_{x_1} T_1 \\ \Omega \,|\, \Gamma \,|\, f_3 \vdash e_2 : \left[f_4\right]_{x_2} T_2 \qquad x \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)\end{array}}{\Omega \,|\, \Gamma \,|\, f_1 + f_3 \vdash (e_1, e_2) : \left[f_2[x_1 \mapsto \pi_1 x] + f_4[x_2 \mapsto \pi_2 x]\right]_x T_1 \times T_2}$$

(TErase)

$$\frac{\begin{array}{c}\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \left[f_2\right]_y T \qquad x : T' \in \Omega \\ x \notin \mathbf{fv}(\Omega \backslash x) \cup \mathbf{fv}(\Gamma) \cup \mathbf{fv}(f_1) \cup (\mathbf{fv}(f_2) \backslash y) \cup \mathbf{fv}(T \backslash y)\end{array}}{\Omega \backslash x \,|\, \Gamma \,|\, f_1 \vdash e : \left[f_2\right]_y T}$$

(TPAbs)

$$\frac{\begin{array}{c}z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \,|\, \Gamma,\, x : T_1 \,|\, 0 \vdash e : \left[0\right]_y T_2\end{array}}{\Omega \,|\, \Gamma \,|\, 0 \vdash \Lambda x : T_1 .\, e : \left[0\right]_z \forall x : T_1 .\, T_2}$$

(TPApp)

$$\frac{\begin{array}{c}\Omega \,|\, \Gamma \,|\, f_1 \vdash e : \left[f_2\right]_y \forall x : T_1 .\, T_2 \\ \Omega \,|\, \Gamma \,|\, 0 \vdash pv : \left[0\right]_z T_1 \qquad w \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)\end{array}}{\Omega \,|\, \Gamma \,|\, f_1 \vdash e\ pv : \left[f_2\right]_w T_2[x \mapsto pv]}$$

(TOp)

$$\frac{\begin{array}{c}y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \forall j \in [1, m_i], \Omega \,|\, \Gamma \,|\, f_{1_j} \vdash e_j : \left[f_{2_j}\right]_{x_j} int\end{array}}{\Omega \,|\, \Gamma \,|\, \sum_{j=1}^{m_i} f_{1_j} \vdash \mathbf{op_i}(\overrightarrow{e}) : \left[\sum_{j=1}^{m_i} f_{2_j}\right]_y int}$$

Fig. 8: Typing Rules

(TAbs)

$$\frac{\Omega \mid \Gamma, x : T_1 \mid f_1 \vdash e : [f_2]_y T_2 \qquad z \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma)}{\Omega \mid \Gamma \mid 0 \vdash \lambda x : T_1.\, e : [0]_z ((x : T_1 \to y : T_2)_{[f_1 \to f_2]})}$$

(TApp)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_1 : [f_2]_z ((x : T_1 \to y : T_2)_{[f_3 \to f_4]}) \\ \Omega \mid \Gamma \mid f_5 \vdash e_2 : [f_6]_w T_1 \qquad \Omega, x : T_1 \mid \Gamma \vdash f_3[w \mapsto x] \leq (f_2 + f_6)[w \mapsto x]\end{array}}{\Omega, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash e_1\ e_2 : [(f_2 + f_6 - f_3)[w \mapsto x] + f_4]_y T_2}$$

(TLet)

$$\frac{\begin{array}{c}\Omega \mid \Gamma \mid f_1 \vdash e_1 : [f_2]_z T_1 \\ \Omega \mid \Gamma, x : T_1 \mid f_3 \vdash e_2 : [f_4]_y T_2 \qquad \Omega \mid \Gamma, x : T_1 \vdash f_3 \leq f_2[z \mapsto x]\end{array}}{\Omega, x : T_1 \mid \Gamma \mid f_1 + f_5 \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : [f_2[z \mapsto x] - f_3 + f_4]_y T_2}$$

(TCons)

$$\frac{\begin{array}{c} y \notin \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Gamma) \\ \Omega \mid \Gamma \mid f_1 \vdash e_0 : [f_2]_{x_0} T_i \qquad \forall j \in [1, m_i], \Omega \mid \Gamma \mid f_{3_j} \vdash e_j : [f_{4_j}]_{x_j} \mathit{ind}\overrightarrow{(C, (T, m))} \\ \Omega, x_0 : T_i, ..., x_{m_i} : \mathit{ind}\overrightarrow{(C, (T, m))} \mid \Gamma \vdash f_5[y \mapsto C_i(x_0, (x_1, ..., x_{m_i}))] \leq f_2 + \sum_{j=1}^{m_i} f_{4_j} \end{array}}{\Omega \mid \Gamma \mid f_1 + \sum_{j=1}^{m_i} f_{3_j} \vdash C_i(e_0, (e_1, ..., e_{m_i})) : [f_5]_y \mathit{ind}\overrightarrow{(C, (T, m))}}$$

(TDes)

$$\frac{\begin{array}{c} \Omega \mid \Gamma \mid f_1 \vdash e_0 : [f_2]_x \mathit{ind}\overrightarrow{(C, (T, m))} \qquad \forall i, \forall j \in [1, m_i], x_j \notin \mathbf{fv}(f_3) \cup \mathbf{fv}(T_1) \\ \forall i, \Omega \mid \Gamma, x_0 : T_i, ..., x_{m_i} : \mathit{ind}\overrightarrow{(C, (T, m))} \mid f_2[x \mapsto C_i(x_0, (x_1, ..., x_{m_i}))] \vdash e_i : [f_3]_y T_1 \end{array}}{\Omega \mid \Gamma \mid f_1 \vdash \mathit{matd}(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e)}) : [f_3]_y T_1}$$

Fig. 9: Typing Rules

(EAppl)
$$\frac{e_1 \mid p \hookrightarrow e_1' \mid q}{e_1\ e_2 \mid p \hookrightarrow e_1'\ e_2 \mid q}$$

(EAppr)
$$\frac{e_2 \mid p \hookrightarrow e_2' \mid q}{v_1\ e_2 \mid p \hookrightarrow v_1\ e_2' \mid q}$$

(EProj1)
$$\frac{}{\pi_1(v_1, v_2) \mid p \hookrightarrow v_1 \mid p}$$

(EProj2)
$$\frac{}{\pi_2(v_1, v_2) \mid p \hookrightarrow v_2 \mid p}$$

(EPair1)
$$\frac{e_1 \mid p \hookrightarrow e_1' \mid q}{(e_1, e_2) \mid p \hookrightarrow (e_1', e_2) \mid q}$$

(EPair2)
$$\frac{e_2 \mid p \hookrightarrow e_2' \mid q}{(v_1, e_2) \mid p \hookrightarrow (v_1, e_2') \mid q}$$

(ETick)
$$\frac{q - p \geq 0}{\mathit{tick}\ p\ e \mid q \hookrightarrow e \mid q - p}$$

(EOp)
$$\frac{\mathbf{Eval}(\mathbf{op_i}, \overrightarrow{v}) = v'}{\mathbf{op_i}(\overrightarrow{v}) \mid p \hookrightarrow v' \mid p}$$

(EFix)
$$\frac{}{\mathit{fix}\ x : T.\, e \mid p \hookrightarrow e[x \mapsto \mathit{fix}\ x : T.\, e] \mid p}$$

(EPApp)
$$\frac{}{\Lambda x : T.\, e\ v \mid p \hookrightarrow e[x \mapsto v] \mid p}$$

(EApp)
$$\frac{}{\lambda x : T_1.\, e_1\ v_2 \mid p \hookrightarrow e_1[x \mapsto v_2] \mid p}$$

(ECon0)
$$\frac{e_0 \mid p \hookrightarrow e_0' \mid q}{C_j(e_0, (e_1, ..., e_{m_j})) \mid p \hookrightarrow C_j(e_0', (e_1, ..., e_{m_j})) \mid q}$$

(ELetl)
$$\frac{e_1 \mid p \hookrightarrow e_1' \mid q}{\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid p \hookrightarrow \mathit{let}\ x = e_1'\ \mathit{in}\ e_2 \mid q}$$

(ELet)
$$\frac{}{\mathit{let}\ x = v_1\ \mathit{in}\ e_2 \mid p \hookrightarrow e_2[x \mapsto v_1] \mid q}$$

(EConl)
$$\frac{e_i \mid p \hookrightarrow e_i' \mid q \qquad i \geq 1}{C_j(v_0, (v_1, ..., v_{i-1}, e_i, e_{i+1}, ..., e_{m_j})) \mid p \hookrightarrow C_j(v_0, (v_1, ..., v_{i-1}, e_i', e_{i+1}, ..., e_{m_j})) \mid q}$$

(EOpi)
$$\frac{e_i \mid p \hookrightarrow e_i' \mid q}{\mathbf{op_j}(v_0, (v_1, ..., v_{i-1}, e_i, e_{i+1}, ..., e_{m_j})) \mid p \hookrightarrow \mathbf{op_j}(v_0, (v_1, ..., v_{i-1}, e_i', e_{i+1}, ..., e_{m_j})) \mid q}$$

(ECasel)
$$\frac{e_0 \mid p \hookrightarrow e_0' \mid q}{\mathit{matd}(e_0, \overrightarrow{C(x_0, (x_1, ..., x_m).e)}) \mid p \hookrightarrow \mathit{matd}(e_0', \overrightarrow{C(x_0, (x_1, ..., x_m).e)}) \mid q}$$

(ECase)
$$\frac{}{\mathit{matd}(C_j(v_0, (v_1, ..., v_{m_j})), \overrightarrow{C_j(x_0, (x_1, ..., x_{m_j}).e_j)}) \mid p \hookrightarrow e_j[x_0, ..., x_{m_j} \mapsto v_0, ..., v_{m_j}] \mid p}$$

Fig. 10: Evalutaion Rules