# A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis

VINEET RAJANI*, Max Planck Institute for Security and Privacy, Germany
MARCO GABOARDI, Boston University, USA
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
JAN HOFFMANN, Carnegie Mellon University, USA

This paper presents $\lambda$-amor, a new type-theoretic framework for amortized cost analysis of higher-order functional programs and shows that existing type systems for cost analysis can be embedded in it. $\lambda$-amor introduces a new modal type for representing *potentials* – costs that have been accounted for, but not yet incurred, which are central to amortized analysis. Additionally, $\lambda$-amor relies on standard type-theoretic concepts like affineness, refinement types and an indexed cost monad. $\lambda$-amor is proved sound using a rather simple logical relation. We embed two existing type systems for cost analysis in $\lambda$-amor showing that, despite its simplicity, $\lambda$-amor can simulate cost analysis for different evaluation strategies (call-by-name and call-by-value), in different styles (effect-based and coeffect-based), and with or without amortization. One of the embeddings also implies that $\lambda$-amor is relatively complete for all terminating PCF programs.



CCS Concepts: • **Theory of computation** → **Type theory**; **Linear logic**; *Operational semantics.*

Additional Key Words and Phrases: amortized cost analysis, type theory, relative completeness

**ACM Reference Format:**
Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis. *Proc. ACM Program. Lang.* 5, POPL, Article 27 (January 2021), 28 pages. https://doi.org/10.1145/3434308

## 1 INTRODUCTION

Cost analysis is the static verification of an upper bound on the evaluation cost of a program, measured in some abstract unit such as the number of reduction steps, the number of function calls or the number of case analyses during the execution of a program.[1] Cost analysis is a well-studied topic. Of interest to us in this paper are *type systems* for cost analysis, many of which exist [Avanzini and Dal Lago 2017; Çiçek et al. 2017; Crary and Weirich 2000; Dal Lago and Gaboardi 2011; Dal Lago and Petit 2012; Danner et al. 2015; Girard et al. 1992; Handley et al. 2020; Hoffman 2011; Hoffmann et al. 2011, 2017; Hoffmann and Hofmann 2010; Hofmann and Jost 2003; Jost et al. 2010, 2009, 2017; Kavvos et al. 2020; Knoth et al. 2019].

*Vineet Rajani is now a post-doctoral researcher at the Max Planck Institute for Security and Privacy but this work was mostly done while he was a graduate student at the Max Planck Institute for Software Systems and Saarland University.
[1]Cost analysis can also be used to establish lower bounds but, here, as in most prior work, the focus is on upper bounds.

Authors' addresses: Vineet Rajani, Max Planck Institute for Security and Privacy, Germany, vineet.rajani@csp.mpg.de; Marco Gaboardi, Boston University, USA, gaboardi@bu.edu; Deepak Garg, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Jan Hoffmann, Carnegie Mellon University, USA, jhoffmann@cmu.edu.


2475-1421/2021/1-ART27
https://doi.org/10.1145/3434308

Despite the common theme of cost analysis, these type systems vary along a number of axes. First, they cover different evaluation strategies like call-by-value and call-by-name, which result in very different cost models. Second, they vary in *how* the type system tracks costs. Most commonly and perhaps most intuitively, cost is tracked as an *effect* in the type system [Avanzini and Dal Lago 2017; Çiçek et al. 2017; Crary and Weirich 2000; Danner et al. 2015; Handley et al. 2020; Hoffman 2011; Hoffmann et al. 2011, 2017; Hoffmann and Hofmann 2010; Hofmann and Jost 2003; Jost et al. 2010, 2009, 2017; Kavvos et al. 2020; Knoth et al. 2019]. To a first approximation, a cost effect can be thought of as an additional output of the program – a quantitative value representing the cost of computing the usual output. Interestingly, other work [Dal Lago and Gaboardi 2011; Dal Lago and Petit 2012; Girard et al. 1992] treats cost as a *coeffect*. To the best of our knowledge, coeffect-based tracking of cost was introduced in Bounded Linear Logic (BLL) [Girard et al. 1992]. The key idea of coeffect-based cost analysis is to represent cost as a requirement on the context (a coeffect) instead of an additional output (an effect). BLL computes cost by tracking the number of times each variable is used (a coeffect) in an affine type system and is limited to polynomial costs. Subsequent work generalizes the idea of coeffect-based cost analysis to not just non-polynomial costs but also to other quantitative measures besides cost [Brunel et al. 2014; Dal Lago and Gaboardi 2011; Dal Lago and Petit 2012; Gaboardi et al. 2016; McDermott and Mycroft 2018; Petricek et al. 2014, 2013].

Third, prior work varies in whether it supports only standard (worst-case) cost analysis or also *amortized* cost analysis. Amortized cost analysis is useful for stateful data structures where certain operation invocations pay a huge cost to change the internal state in a way that reduces the cost of subsequent invocations [Tarjan 1985]. In these cases, it is not very useful to compute the worst-case cost of an individual operation; instead, one wants to compute an upper bound on the cost of a sequence of $n$ operations for large $n$. This is called amortized cost analysis. Standard textbook examples that rely on amortized cost analysis are a FIFO queue implemented using a pair of functional (LIFO) lists, Fibonacci heaps and the union-find data structure with path compression.

*Research question.* Given this diversity in type systems for cost analysis, a natural question is whether one can build a *unifying framework* that can *embed* type systems covering different evaluation strategies, effect- and coeffect-based cost tracking, and amortized cost analysis. Note that this is unlikely to be a trivial exercise particularly because the effect- and coeffect-based styles of cost tracking are quite different.

In this paper, we answer this question in the affirmative. We build a new type theory, called $\lambda$-amor, that is somewhat minimal but can embed type systems covering call-by-value and call-by-name evaluation, effect- and coeffect-based cost tracking, and amortized costs. Hence, $\lambda$-amor is a unifying framework in the sense described above.

*Overview of $\lambda$-amor.* To motivate $\lambda$-amor's design, we start by describing the typical structure of amortized cost analysis through the so-called method of potentials [Tarjan 1985]. Consider a data structure with internal state and suppose that the cost of an operation on the data structure depends on this state, so it varies. We say that the amortized cost of an operation is $c$ if the cost of $n$ consecutive operations is upper-bounded by $n \cdot c$. To prove this, we find a function $\phi$ that maps the state $s$ of the data structure to a non-negative number, called a *potential*, and show (using a type-theory like $\lambda$-amor) that an invocation of the operation that changes the data structure from state $s_i$ to $s_{i+1}$ has a cost upper-bounded by $\phi(s_i) - \phi(s_{i+1}) + c$. It immediately follows that a sequence of $n$ operations starting in state $s_0$ with $\phi(s_0) = 0$ has a total cost upper-bounded by $(\phi(s_0) - \phi(s_1) + c) + \ldots + (\phi(s_{n-1}) - \phi(s_n) + c)$. This is a telescopic series that equals $\phi(s_0) - \phi(s_n) + n \cdot c$, which in turn is upper-bounded by $n \cdot c$ since $\phi(s_0) = 0$ and $\phi(s_n)$ is non-negative. Hence, the cost of the $n$ operations is no more than $c$, as required. The value $\phi(s)$ is called the potential associated with the state $s$. This potential is needed for verification only, i.e., it is ghost state and it does not

exist at run time. The type theory is used to prove only that the cost of an individual operation is upper-bounded by $\phi(s_i) - \phi(s_{i+1}) + c$ (the rest is trivial).

Based on this intuition, we describe the requirements for a type theory to support amortized cost analysis, and how $\lambda$-amor satisfies these requirements.

R1) The type theory must include some construct to associate the ghost potential to the type of a data structure. To this end, $\lambda$-amor introduces a new type constructor written $[p]\ \tau$, which ascribes a value of type $\tau$ with associated potential $p$. Here, $p$ is a non-negative number.

R2) Since the potential $p$ is related to the state $s$ ($p$ equals $\phi(s)$), the type $\tau$ of the data structure must reflect its state to sufficient precision, to allow relating $p$ and $\tau$ meaningfully. $\lambda$-amor uses standard *refinement types* [Xi 2007] for this. For instance, $L^n\ \tau$ is the type of lists of length $n$, and $[2n]\ (L^n\ \tau)$ is the type of lists of length $n$ with associated potential $2n$. Note how the refinement $n$ relates an aspect of the state, the length of the list, to the potential associated with the list.

R3) The type theory must be able to represent execution costs since we need to establish upper bounds on them. As mentioned above, we can choose to represent costs as an effect or as a coeffect. Upfront, $\lambda$-amor represents cost as an effect: It includes an indexed monad $\mathbb{M}\,\kappa\,\tau$ to represent a computation of type $\tau$ and cost $\kappa$. However, we show that coeffect-based cost analysis can be simulated in $\lambda$-amor using a combination of effects and potentials. To the best of our knowledge, this simulation of coeffect-based analysis in effect-based analysis using potentials is a completely new result.

R4) The type theory must prevent the duplication of any type that has a potential associated with it, otherwise the type theory would not be sound. For example, if a typing derivation duplicates the potential $\phi(s_i)$ even once, then the operation's real cost may be up to $2 \cdot \phi(s_i) - \phi(s_{i+1}) + c$, and the amortized analysis described earlier breaks completely. Hence, all potential-carrying types must be treated *affinely* in the type theory. Accordingly, $\lambda$-amor is an affine type theory with the usual operators of affine logic like $\otimes$, & and $\oplus$. Duplicable resources are explicitly represented using the standard exponential ! of affine logic. To improve expressiveness, we allow the exponential ! to be indexed, following the dependent sub-exponential of Bounded Linear Logic [Girard et al. 1992].

Overall, $\lambda$-amor can be seen as a computational $\lambda$-calculus (in the sense of Moggi [Moggi 1991]) equipped with a type system that has the four features mentioned above – the construct $[p]\ \tau$ to associate potential to a type, type refinements, the indexed cost monad $\mathbb{M}\,\kappa\,\tau$, and affineness with an indexed sub-exponential !.[2] We give the pure (non-monadic) part of $\lambda$-amor a *call-by-name* semantics with eager evaluation for all pairs and sums. However, as shown by Moggi [1991], simulating call-by-value semantics in a monadic setting is not difficult, a fact we exploit for our embedding of call-by-value cost analysis in $\lambda$-amor later.

$\lambda$-amor is conceptually very simple. We prove it sound using an elementary logical relation that extends Pym's semantics of BI [Pym et al. 2004]. The key novelty in building this relation is the treatment of potentials, and their interaction with the cost monad (available potential can offset the cost in the monad).

*$\lambda$-amor as a unifying framework.* We show that $\lambda$-amor is unifying in the sense described earlier by embedding two state-of-the-art frameworks for (amortized) cost analysis faithfully in $\lambda$-amor. Our first embedding is that of a core calculus for Resource-aware ML or RAML [Hoffman 2011; Hoffmann and Hofmann 2010], an implemented, effect-based framework for amortized cost analysis

[2]The name "$\lambda$-amor" refers to both the calculus and its type system. The intended use can be disambiguated from the context.

of ML programs. RAML is call-by-value, so this embedding shows how (amortized) call-by-value cost analysis can be simulated in $\lambda$-amor.

Our second embedding is that of $d\ell$PCF [Dal Lago and Gaboardi 2011], a coeffect-based type system for non-amortized cost analysis of call-by-name PCF programs. Unlike the first embedding (of RAML), which is relatively easy, this embedding is surprising and difficult, as it simulates $d\ell$PCF's coeffects using $\lambda$-amor's potentials and effects.

Together, the two embeddings cover call-by-value and call-by-name evaluation, effect- and coeffect-based cost tracking, and amortized cost analysis, thus fulfilling our goal of "unifying".

The embedding of $d\ell$PCF also shows that $\lambda$-amor is very expressive. Dal Lago and Gaboardi [2011] showed that $d\ell$PCF is relatively complete for PCF programs, meaning that every PCF program can be typed with precise cost in it.[3] Hence, our embedding of $d\ell$PCF in $\lambda$-amor shows that $\lambda$-amor is also relatively complete for PCF programs.

*Added expressiveness relative to prior work on amortized analysis.* Besides being a unifying framework for cost analysis as just described, $\lambda$-amor also improves the expressiveness of prior type systems for amortized cost analysis with the method of potentials. The main line of work here is RAML and its variants [Hoffmann et al. 2011, 2017; Hoffmann and Hofmann 2010; Hofmann and Jost 2003; Jost et al. 2010]. However, this line of work fulfills requirements R1 and R4 only partially – potential can be associated only with first-order types and functions are non-affine. As a result, RAML has difficulty in handling the interaction between higher-order values and potential. To understand the issue, consider a curried function of two arguments, of which the first argument carries potential that offsets the cost of executing the function. Suppose that this function is applied partially. The resulting closure must not be duplicable because it captures the potential from the first argument. However, since RAML (and its many extensions) do not treat functions affinely, they cannot type check such programs. For instance, RAML extensions like [Hoffmann et al. 2017] completely exclude the handling of curried functions while others like [Jost et al. 2010] can type check only those curried functions where potential is limited to the last argument. In contrast, $\lambda$-amor, being fully affine, can handle such examples trivially.

Additionally, RAML and its variants do not support polymorphism over costs in full generality, which limits their expressiveness. For example, RAML cannot represent the cost of a standard list fold function precisely when the cost of the binary operator passed as an argument to fold varies depending on its arguments. In contrast, $\lambda$-amor has full cost polymorphism and can type such a fold function, as we show in Section 3.3.

*Organization.* To simplify the presentation, we describe $\lambda$-amor in two stages. First, we describe $\lambda$-amor without indexing on exponentials (Section 2). This suffices for most examples (Section 3) and the embedding of RAML (Section 4), but not the embedding of $d\ell$PCF. Then, we introduce the full $\lambda$-amor by adding indexed exponentials (Section 5) and show how to embed $d\ell$PCF (Section 6). We compare to related work in Section 7. The full technical development with proofs of all theorems and further examples can be found in a technical appendix available from author's homepages.

*Limitations and scope.* Our focus in this paper is on the foundations of a unifying type theory for (amortized) cost analysis. An implementation of the type theory is beyond the scope of this paper, even though in restricted settings like polynomial-time analysis, one could use ideas from prior work like RAML to implement the type theory efficiently (we are working on such an implementation). Further, we focus only on the cost of non-reusable resources like time. The cost of reusable resources like heap space is not within the scope of this paper. Finally, we do not consider call-by-need evaluation (lazy evaluation with sharing), as in the work of Danielsson

[3]The adjective "relative" means relative to having a refinement domain that is sufficiently expressive.

| | | | |
|---|---|---|---|
| Types | $\tau$ | ::= | $\mathbf{1} \mid \mathsf{b} \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \tau_1 \oplus \tau_2 \mid !\tau \mid [p]\,\tau \mid \mathbb{M}\,\kappa\,\tau \mid L^n\,\tau$<br>$\alpha \mid \forall \alpha : \mathsf{K}.\tau \mid \forall i : \mathsf{S}.\tau \mid \lambda_t i : \mathsf{S}.\tau \mid \tau\ I \mid \exists i : \mathsf{S}.\tau \mid C \Rightarrow \tau \mid C \& \tau$ |
| Expressions | $e$ | ::= | $v \mid x \mid e_1\ e_2 \mid \langle\langle e_1, e_2 \rangle\rangle \mid \mathsf{let}\langle\langle x, y\rangle\rangle = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\, x.e \mid$<br>$\langle e, e\rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}\ e, x.e, y.e \mid$<br>$\mathsf{let}\,!\,x = e_1\ \mathsf{in}\ e_2 \mid e :: e \mid (\mathsf{match}\ e\ \mathsf{with} \mid nil \mapsto e_1 \mid h :: t \mapsto e_2) \mid e\ [] \mid$<br>$\mathsf{xlet}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{clet}\ x = e_1\ \mathsf{in}\ e_2$ |
| Values | $v$ | ::= | $() \mid c \mid \lambda x.e \mid \langle\langle v_1, v_2\rangle\rangle \mid \langle v, v\rangle \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid\ !\,e \mid nil \mid$<br>$\Lambda.e \mid \mathsf{ret}\ e \mid \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 \mid \uparrow^\kappa \mid \mathsf{release}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{store}\ e$ |
| Indices | $I, J, \kappa, p, n$ | ::= | $i \mid N \mid R^+ \mid I + I \mid I - I \mid \sum_{a<J} I \mid \lambda_s i : \mathsf{S}.I \mid I\ I$ |
| Constraints | $C$ | ::= | $I = I \mid I < I \mid C \wedge C$ |
| Sorts | S | ::= | $\mathbb{N} \mid \mathbb{R}^+ \mid \mathsf{S} \rightarrow \mathsf{S}$ |
| Kinds | K | ::= | $Type \mid \mathsf{S} \rightarrow \mathsf{K}$ |

Fig. 1. $\lambda$-amor$^-$'s syntax

[2008]; Jost et al. [2017]; Okasaki [1996]. Call-by-need is very hard to simulate without imperative state (which we do not include in $\lambda$-amor) and, additionally, amortized analysis for call-by-need does *not* require affineness. Consequently, unifying work on cost analysis for call-by-need with that for call-by-value or call-by-name will require a significant amount of further work. We also do not consider call-by-push-value (CBPV) evaluation, although we believe that CBPV is actually within reach – we would need to give $\lambda$-amor a CBPV semantics instead of its current monadic semantics.

*Other related work.* We wish to note that, besides type systems for cost analysis, there is also work on program logics for (amortized) cost analysis [Carbonneaux et al. 2015; Charguéraud and Pottier 2019; Mével et al. 2019]. Some of this work, e.g., [Mével et al. 2019] could be made more expressive than our type theory (by adding an expressive language of costs), but this line of work does not attempt to unify existing frameworks for cost analysis, which is our primary objective here. We could have chosen Mével et al. [2019] as the base of our work and used that in place of $\lambda$-amor as the target of our embeddings, but we chose to design $\lambda$-amor because we believe that our design is considerably less complex. $\lambda$-amor focuses only on cost analysis. In contrast, Mével et al. [2019] add cost analysis to a framework for full functional verification, which is really orthogonal to our goal here.

## 2 $\lambda$-amor$^-$ (NO SUB-EXPONENTIALS)

To simplify the presentation, we first describe $\lambda$-amor$^-$, the subset of $\lambda$-amor that only considers the standard exponential ! from affine logic, without any indexing (that $\lambda$-amor supports).

### 2.1 Syntax and Semantics

The syntax of $\lambda$-amor$^-$ is shown in Fig. 1. We describe the various syntactic categories below.

*Indices, sorts, kinds and constraints.* $\lambda$-amor$^-$ is a refinement type system. (Static) indices, à la DML [Xi 2007], are used to track information like list lengths, computation costs and potentials. These indices can be natural or positive real numbers, with support for addition and subtraction. There is also a function to obtain a bounded sum ($\sum_{a<J} I$) over indices. It basically describes summation of $I$ with $a$ ranging from 0 to $J-1$ inclusive, i.e., $I[0/a] + \ldots + I[J-1/a]$. Besides this, we also have index-level functions and index-level function application. List lengths are represented using natural numbers (sort $\mathbb{N}$). Potentials and costs are both represented using non-negative real numbers (sort $\mathbb{R}^+$). $\lambda$-amor$^-$ also features kinds, denoted by K. $Type$ is the kind of standard affine types and $\mathsf{S} \rightarrow \mathsf{K}$ represents a kind family indexed by the sort S. Finally, constraints (denoted by $C$) are predicates ($=, <, \wedge$) over indices.

Forcing reduction, $e \Downarrow^{\kappa} v$

$$\frac{e \Downarrow v}{\mathsf{ret}\, e \Downarrow^{0} v}\ \text{E-return} \qquad \frac{e_1 \Downarrow v_1 \qquad v_1 \Downarrow^{\kappa_1} v_1' \qquad e_2[v_1'/x] \Downarrow v_2 \qquad v_2 \Downarrow^{\kappa_2} v_2'}{\mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\kappa_1+\kappa_2} v_2'}\ \text{E-bind}$$

$$\frac{}{\uparrow^{\kappa} \Downarrow^{\kappa} ()}\ \text{E-tick} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v_2 \qquad v_2 \Downarrow^{\kappa} v_2'}{\mathsf{release}\, x = e_1\ \mathsf{in}\ e_2 \Downarrow^{\kappa} v_2'}\ \text{E-release} \qquad \frac{e \Downarrow v}{\mathsf{store}\, e \Downarrow^{0} v}\ \text{E-store}$$

Fig. 2. Selected evaluation rules

*Types.* $\lambda$-amor$^{-}$ is an affine type system. The most important type is the modal type $[p]\,\tau$, which ascribes values of type $\tau$ that have potential $p$ associated with them (as a ghost). We have the multiplicative unit type (denoted $\mathbf{1}$) and an abstract base type (denoted b) to represent types like integers or booleans. Then, there are standard affine types – affine function spaces ($\multimap$), sums ($\oplus$), pairs (both the multiplicative $\otimes$ and the additive &) and the exponential (!), which ascribes expressions that can be duplicated. We include only one representative data type – the length-refined list type $L^n\tau$, where the length $n$ is drawn from the language of indices (described earlier). Other data types can be added if needed.

$\lambda$-amor$^{-}$ also has universal quantification over types and indices denoted by $\forall\alpha : \mathsf{K}.\tau$ and $\forall i : \mathsf{S}.\tau$, respectively, and existential quantification over indices, denoted $\exists i : \mathsf{S}.\tau$. Quantification over indices comes in quite handy for representing variable cost of function arguments (as we exemplify in our encoding of Church numerals in Section 3.2). The constraint type $C \Rightarrow \tau$ means that *if* constraint $C$ holds *then* the underlying term has the type $\tau$. The dual type $C\&\tau$ means that the constraint $C$ holds *and* the type of the underlying term is $\tau$. For instance, the type of non-empty lists can be written as $\exists n.\,(n > 0)\&(L^n\tau)$. We also have sort-indexed type families, which are type-level functions from sorts to kinds.

Finally, $\lambda$-amor$^{-}$ has the monadic type $\mathbb{M}\,\kappa\,\tau$, which represents computations of cost at most $\kappa$. Technically, $\mathbb{M}\,\kappa\,\tau$ is a graded or indexed monad [Gaboardi et al. 2016]. A non-zero cost can be incurred only by an expression of the monadic type. Following standard convention we call such expressions impure, while expressions of all other types are called pure.

*Expressions and values.* There are term-level constructors for all types (in the kind $Type$) except for the modal type ($[p]\,\tau$). The inhabitants of type $[p]\,\tau$ are exactly those of type $\tau$ since the potential is ghost state without a runtime manifestation.

We describe the expression and value forms for some of the types. The term-level constructors for the constraint type ($C \Rightarrow \tau$), type and index-level quantification ($\forall\alpha : \mathsf{K}.\tau$, $\forall i : \mathsf{S}.\tau$) are all denoted $\Lambda.e$. (Note that indices, types and constraints do not occur in terms.) We also have a fixpoint operator (fix) which is used to encode recursion.

The monadic type $\mathbb{M}\,\kappa\,\tau$ has several term constructors, including the standard monadic unit ($\mathsf{ret}\, e$) and bind ($\mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2$). The construct $\mathsf{store}\, e$ stores potential with a term and is the introduction form of the type $[p]\,\tau$. Dually, $\mathsf{release}\, x = e_1\ \mathsf{in}\ e_2$ releases potential stored with $e_1$ and makes it available to offset the cost of $e_2$. Note that $\mathsf{store}\, e$ and $\mathsf{release}\, x = e_1\ \mathsf{in}\ e_2$ are useful only for the type system: they indicate ghost operations, i.e., where potentials should be stored and released, respectively. Operationally, they are uninteresting: $\mathsf{store}\, e$ evaluates exactly like $\mathsf{ret}\, e$, while $\mathsf{release}\, x = e_1\ \mathsf{in}\ e_2$ evaluates exactly like $\mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2$. Finally, we have a construct for incurring non-zero cost – the "tick" construct denoted $\uparrow^{\kappa}$. This construct indicates that cost $\kappa$ is incurred where it is placed. Programmers place the construct at appropriate points in a program to model costs incurred during execution, as in prior work [Danielsson 2008].

Typing judgment: $\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau$

$$\frac{\Psi; \Theta; \Delta; \Omega; \Gamma_1 \vdash e_1 : \tau_1 \qquad \Psi; \Theta; \Delta; \Omega; \Gamma_2 \vdash e_2 : \tau_1}{\Psi; \Theta; \Delta; \Omega; \Gamma_1 + \Gamma_2 \vdash \langle\langle e_1, e_2 \rangle\rangle : (\tau_1 \otimes \tau_2)} \text{ T-tensorI}$$

$$\frac{\Psi; \Theta; \Delta; \Omega; \Gamma_1 \vdash e : (\tau_1 \otimes \tau_2) \qquad \Psi; \Theta; \Delta; \Omega; \Gamma_2, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Psi; \Theta; \Delta; \Omega; \Gamma_1 + \Gamma_2 \vdash \text{let} \langle\langle x, y \rangle\rangle = e \text{ in } e' : \tau} \text{ T-tensorE}$$

$$\frac{\Psi; \Theta; \Delta; \Omega; . \vdash e : \tau}{\Psi; \Theta; \Delta; \Omega; . \vdash\ !e : !\tau} \text{ T-ExpI} \qquad \frac{\Psi; \Theta; \Delta; \Omega; \Gamma_1 \vdash e : !\tau \qquad \Psi; \Theta; \Delta; \Omega, x : \tau; \Gamma_2 \vdash e' : \tau'}{\Psi; \Theta; \Delta; \Omega; \Gamma_1 + \Gamma_2 \vdash \text{let}\, !\, x = e \text{ in } e' : \tau'} \text{ T-ExpE}$$

$$\frac{\Psi; \Theta; \Delta; \Omega, x : \tau; . \vdash e : \tau}{\Psi; \Theta; \Delta; \Omega; . \vdash \text{fix} x.e : \tau} \text{ T-fix}$$

$$\frac{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \qquad \Psi; \Theta; \Delta \vdash \Gamma' \sqsubseteq \Gamma \qquad \Psi; \Theta; \Delta \vdash \Omega' \sqsubseteq \Omega \qquad \Psi; \Theta; \Delta \vdash \tau <: \tau'}{\Psi; \Theta; \Delta; \Omega'; \Gamma' \vdash e : \tau'} \text{ T-weaken}$$

$$\frac{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau}{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash \text{ret}\, e : \mathbb{M}\, 0\, \tau} \text{ T-ret} \qquad \frac{\begin{array}{c}\Psi; \Theta; \Delta; \Omega; \Gamma_1 \vdash e_1 : \mathbb{M}\, \kappa_1\, \tau_1 \\ \Psi; \Theta; \Delta; \Omega; \Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}\, \kappa_2\, \tau_2 \\ \Theta \vdash \kappa_1 : \mathbb{R}^+ \qquad \Theta \vdash \kappa_2 : \mathbb{R}^+\end{array}}{\Psi; \Theta; \Delta; \Omega; \Gamma_1 + \Gamma_2 \vdash \text{bind}\, x = e_1 \text{ in } e_2 : \mathbb{M}(\kappa_1 + \kappa_2)\, \tau_2} \text{ T-bind}$$

$$\frac{\Theta \vdash \kappa : \mathbb{R}^+}{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash \uparrow^\kappa : \mathbb{M}\, \kappa\, \mathbf{1}} \text{ T-tick} \qquad \frac{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \qquad \Theta \vdash p : \mathbb{R}^+}{\Psi; \Theta; \Delta; \Omega; \Gamma \vdash \text{store}\, e : \mathbb{M}\, p\, ([p]\, \tau)} \text{ T-store}$$

$$\frac{\begin{array}{c}\Psi; \Theta; \Delta; \Omega; \Gamma_1 \vdash e_1 : [p_1]\, \tau_1 \qquad \Psi; \Theta; \Delta; \Omega; \Gamma_2, x : \tau_1 \vdash e_2 : \mathbb{M}(p_1 + p_2)\, \tau_2 \\ \Theta \vdash p_1 : \mathbb{R}^+ \qquad \Theta \vdash p_2 : \mathbb{R}^+\end{array}}{\Psi; \Theta; \Delta; \Omega; \Gamma_1 + \Gamma_2 \vdash \text{release}\, x = e_1 \text{ in } e_2 : \mathbb{M}\, p_2\, \tau_2} \text{ T-release}$$

Fig. 3. Selected typing rules for $\lambda$-amor$^-$

*Operational semantics.* $\lambda$-amor$^-$ is a call-by-name calculus with eager evaluation.[4] We use two evaluation judgments – pure and forcing. The pure evaluation judgment ($e \Downarrow v$) relates an expression $e$ to the value $v$ it evaluates to. All monadic expressions are treated as values in the pure evaluation. The rules for pure evaluation are standard so we defer them to the technical appendix. The forcing evaluation judgment $e \Downarrow^\kappa v$ is a relation between terms of type $\mathbb{M}\, \kappa\, \tau$ and values of type $\tau$. $\kappa$ is the cost incurred in executing (forcing) $e$. The rules of this judgment are shown in Fig. 2. E-return states that if $e$ reduces to $v$ in the pure reduction, then ret $e$ forces to $v$ with 0 cost. E-store is exactly like E-return, emphasizing the ghost nature of potential annotations in types. E-bind is the standard monadic composition of $e_1$ with $e_2$. The effect (cost) of bind is the sum of the costs of forcing $e_1$ and $e_2$. E-release is similar. $\uparrow^\kappa$ is the only cost-consuming construct in the language. E-tick says that $\uparrow^\kappa$ forces to () and it incurs cost $\kappa$.

## 2.2 Type system

The typing judgment of $\lambda$-amor$^-$ is written $\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau$. Here, $\Psi$ is a context mapping type-level variables to their kinds, $\Theta$ is a context mapping index-level variables to their sorts, $\Delta$ is a context of constraints on the index variables, and $\Omega$ and $\Gamma$ are the non-affine and affine typing

[4]Perhaps somewhat surprisingly, even additive (&) pairs are evaluated eagerly. However, since all effects are confined to a monad, this choice does not matter. ! is lazy as in a standard affine $\lambda$-calculus.

$$\frac{\Psi;\Theta;\Delta \vdash \tau <: \tau' \qquad \Psi;\Theta;\Delta \vdash p' \leq p}{\Psi;\Theta;\Delta \vdash [p]\,\tau <: [p']\,\tau'}\ \text{sub-potential} \qquad \frac{\Psi;\Theta;\Delta \vdash \tau <: \tau' \qquad \Psi;\Theta;\Delta \vdash \kappa \leq \kappa'}{\Psi;\Theta;\Delta \vdash \mathbb{M}\,\kappa\,\tau <: \mathbb{M}\,\kappa'\,\tau'}\ \text{sub-monad}$$

$$\frac{\Theta \vdash p : \mathbb{R}^+ \qquad \Theta \vdash p' : \mathbb{R}^+}{\Psi;\Theta;\Delta \vdash [p](\tau_1 \multimap \tau_2) <: ([p']\,\tau_1 \multimap [p'+p]\,\tau_2)}\ \text{sub-potArrow} \qquad \frac{}{\Psi;\Theta;\Delta \vdash \tau <: [0]\,\tau}\ \text{sub-potZero}$$

$$\frac{\Psi;\Theta, i : \mathrm{S};\Delta \vdash \tau <: \tau'}{\Psi;\Theta;\Delta \vdash \lambda_t i : \mathrm{S}.\tau <: \lambda_t i : \mathrm{S}.\tau'}\ \text{sub-familyAbs} \qquad \frac{\Theta \vdash I : \mathrm{S}}{\Psi;\Theta;\Delta \vdash (\lambda_t i : \mathrm{S}.\tau)\ I <: \tau[I/i]}\ \text{sub-familyApp1}$$

$$\frac{\Theta \vdash I : \mathrm{S}}{\Psi;\Theta;\Delta \vdash \tau[I/i] <: (\lambda_t i : \mathrm{S}.\tau)\ I}\ \text{sub-familyApp2}$$

Fig. 4. Selected subtyping rules

contexts respectively, both mapping term-level variables to their types. We use the notation $\Gamma_1 + \Gamma_2$ to describe the disjoint union of the affine contexts $\Gamma_1$ and $\Gamma_2$. Selected typing rules are listed in Fig. 3, and the full set of rules can be found in the technical appendix.

Rules for the affine type constructs of $\lambda$-amor are standard. T-tensorI is the type rule for introducing the tensor pair $\langle\langle e_1, e_2\rangle\rangle$ – if $e_1$ and $e_2$ are typed $\tau_1$ and $\tau_2$ under affine contexts $\Gamma_1$ and $\Gamma_2$, respectively, then $\langle\langle e_1, e_2\rangle\rangle$ is typed $(\tau_1 \otimes \tau_2)$ under the context $(\Gamma_1 + \Gamma_2)$. Dually, T-tensorE is the type rule for eliminating the tensor pair – if expression $e$ has type $(\tau_1 \otimes \tau_2)$ in the context $\Gamma_1$ and a continuation $e'$ is of type $\tau'$ in the context $\Gamma_2$ plus both elements of the tensor pair (named $x$ and $y$ here), then the expression $\mathsf{let}\langle\langle x, y\rangle\rangle = e\ \mathsf{in}\ e'$ is of type $\tau'$ under the context $(\Gamma_1 + \Gamma_2)$. T-expI ascribes $!e$ the type $!\tau$ if $e$ can be ascribed the type $\tau$ under an empty affine context. The subtyping relation (<:) used in the rule T-weaken is described below, but we skip describing the standard details of the auxiliary relation $\sqsubseteq$, which is described in the technical appendix. T-expE is the rule for the elimination form of $!\tau$. The important thing here is that the continuation $e'$ has unbounded access to $e$ via the *non-affine* variable $x$.

Rules for monadic types are interesting. T-ret types the return of the monad. In the operational semantics, ret $e$ takes a well-typed expression and returns it with 0 cost. Hence, its type $\mathbb{M}\,0\,\tau$ also includes 0 cost. T-bind types the monadic bind, which basically sequences two computations. The cost in the type of the bind is the sum of the costs of the two computations, again mirroring the operational semantics. T-tick type checks $\uparrow^\kappa$ at type $\mathbb{M}\,\kappa\,\mathbf{1}$ – a monad of unit type with cost $\kappa$.

T-store types store $e$, which is used to associate potential with the expression $e$. If $e$ has type $\tau$, the rule gives store $e$ the type $\mathbb{M}\,\kappa\,([\kappa]\,\tau)$. Intuitively, if $\kappa$ units of potential are attached to $e$, then the cost of doing so is $\kappa$ units. Finally, T-release is dual to T-store: It uses the potential $p_1$ stored with the first expression $e_1$ to reduce the cost of the continuation by the same amount.

*Subtyping.* Selected subtyping rules are shown in Fig. 4. As mentioned earlier, $\lambda$-amor$^-$ also has type-level functions and applications. Accordingly, we have subtyping rules to convert the type-level application form $((\lambda_t i : \mathrm{S}.\tau)\ I)$ to the substitution form $(\tau[I/i])$ and vice versa. Rule sub-potArrow distributes potential on a function type over the argument and the return value. sub-potZero allows silently casting an expression of type $\tau$ to type $[0]\,\tau$. This reinforces the ghost nature of potential. The subtyping of the modal type $[p]\,\tau$ is contra-variant in the potential because it is sound to throw away potential. The subtyping for the monadic type is covariant in both the type and the cost (because the cost in the monadic type is an upper bound). There are additional, standard typing rules for sorts and kinds, which we defer to the technical appendix.

$$
\begin{array}{lcl}
[\![\mathbf{1}]\!] & \triangleq & \{(p,T,())\} \\
[\![\mathrm{b}]\!] & \triangleq & \{(p,T,v) \mid v \in [\![\mathrm{b}]\!]\} \\
[\![L^0\tau]\!] & \triangleq & \{(p,T,nil)\} \\
[\![L^{s+1}\tau]\!] & \triangleq & \{(p,T,v::l) | \exists p_1,p_2.p_1+p_2 \leq p \wedge (p_1,T,v) \in [\![\tau]\!] \wedge (p_2,T,l) \in [\![L^s\tau]\!]\} \\
[\![\tau_1 \otimes \tau_2]\!] & \triangleq & \{(p,T,\langle\!\langle v_1,v_2 \rangle\!\rangle) \mid \exists p_1,p_2.p_1+p_2 \leq p \wedge (p_1,T,v_1) \in [\![\tau_1]\!] \wedge (p_2,T,v_2) \in [\![\tau_2]\!]\} \\
[\![\tau_1 \,\&\, \tau_2]\!] & \triangleq & \{(p,T,\langle v_1,v_2 \rangle) \mid (p,T,v_1) \in [\![\tau_1]\!] \wedge (p,T,v_2) \in [\![\tau_2]\!]\} \\
[\![\tau_1 \oplus \tau_2]\!] & \triangleq & \{(p,T,\mathsf{inl}(v)) \mid (p,T,v) \in [\![\tau_1]\!]\} \cup \{(p,T,\mathsf{inr}(v)) \mid (p,T,v) \in [\![\tau_2]\!]\} \\
[\![!\tau]\!] & \triangleq & \{(p,T,!e) \mid (0,T,e) \in [\![\tau]\!]_{\mathcal{E}}\} \\
[\![\tau_1 \multimap \tau_2]\!] & \triangleq & \{(p,T,\lambda x.e) \mid \forall p',e',T'{<}T\,.(p',T',e') \in [\![\tau_1]\!]_{\mathcal{E}} \implies (p+p',T',e[e'/x]) \in [\![\tau_2]\!]_{\mathcal{E}}\} \\
[\![[n]\,\tau]\!] & \triangleq & \{(p,T,v) \mid \exists p'.p'+n \leq p \wedge (p',T,v) \in [\![\tau]\!]\}\} \\
[\![\mathbb{M}\,\kappa\,\tau]\!] & \triangleq & \{(p,T,v) \mid \forall \kappa',v',T'{<}T\,.v \Downarrow^{\kappa'}_{T'} v' \implies \exists p'.\kappa'+p' \leq p+\kappa \wedge (p',T-T',v') \in [\![\tau]\!]\} \\
[\![\forall\alpha.\tau]\!] & \triangleq & \{(p,T,\Lambda.e) \mid \forall\tau',T'{<}T\,.(p,T',e) \in [\![\tau[\tau'/\alpha]]\!]_{\mathcal{E}}\} \\
[\![\forall i.\tau]\!] & \triangleq & \{(p,T,\Lambda.e) \mid \forall I,T'{<}T\,.(p,T',e) \in [\![\tau[I/i]]\!]_{\mathcal{E}}\} \\
[\![C \Rightarrow \tau]\!] & \triangleq & \{(p,T,\Lambda.e) \mid . \models C \implies (p,T,e) \in [\![\tau]\!]_{\mathcal{E}}\} \\
[\![C\&\tau]\!] & \triangleq & \{(p,T,v) \mid . \models C \wedge (p,T,v) \in [\![\tau]\!]\} \\
[\![\exists s.\tau]\!] & \triangleq & \{(p,T,v) \mid \exists s'.(p,T,v) \in [\![\tau[s'/s]]\!]\} \\
[\![\lambda_t i.\tau]\!] & \triangleq & f \text{ where } \forall I.\, f\ I = [\![\tau[I/i]]\!] \\
[\![\tau\ I]\!] & \triangleq & [\![\tau]\!]\ I \\
\\
[\![\tau]\!]_{\mathcal{E}} & \triangleq & \{(p,T,e) \mid \forall\, T'{<}T, v.e \Downarrow_{T'} v \implies (p,T-T',v) \in [\![\tau]\!]\} \\
\\
[\![\Gamma]\!]_{\mathcal{E}} & \triangleq & \{(p,T,\gamma) \mid \exists f : \mathcal{V}ars \to \mathcal{P}ots. \\
 & & (\forall x \in dom(\Gamma).\,(f(x),T,\gamma(x)) \in [\![\Gamma(x)]\!]_{\mathcal{E}}) \wedge (\textstyle\sum_{x \in dom(\Gamma)} f(x) \leq p)\} \\
[\![\Omega]\!]_{\mathcal{E}} & \triangleq & \{(0,T,\delta) \mid (\forall x \in dom(\Omega).(0,T,\delta(x)) \in [\![\tau]\!]_{\mathcal{E}})\}
\end{array}
$$

Fig. 5. Model of $\lambda$-amor$^-$ types

Theorem 1 is the soundness of $\lambda$-amor$^-$: If $e$ is a closed term which has a statically approximated cost of $\kappa$ units (as specified in the monadic type $\mathbb{M}\,\kappa\,\tau$) and forcing $e$ actually consumes $\kappa'$ units of cost, then $\kappa' \leq \kappa$. We prove this theorem using a logical relation in Section 2.3.

**Theorem 1** (Soundness). $\forall e,v,\kappa,\kappa',\tau \in Type.\vdash e : \mathbb{M}\,\kappa\,\tau \wedge e \Downarrow^{\kappa'} v \implies \kappa' \leq \kappa$

### 2.3 Model of types and soundness

To prove the soundness of $\lambda$-amor$^-$, we develop a logical-relation model of its types. The model is an extension of Pym's semantics of BI [Pym et al. 2004] with potentials, the cost monad, and type refinements. We also step-index the model [Ahmed 2004] to break a circularity in its definition, arising from impredicative quantification over types, as in the work of Neis et al. [2011]. Because we use step-indices, we also have augmented operational semantics that count the number of rules (denoted $T$) used during evaluation. The revised judgments are written $e \Downarrow_T v$ (pure) and $e \Downarrow^{\kappa}_T v$ (forcing). The expected details are in the technical appendix. Note that there is no connection between $T$ and $\kappa$ in the forcing judgment – the former is purely an artifact of our metatheoretic proofs, while the latter is induced by $\uparrow$ constructs in the program. Our use of step-indices, also written $T$, is standard and readers not familiar with them may simply ignore them. The model (Fig. 5) is defined using four relations: a value relation, an expression relation and substitution relations for the affine and non-affine contexts. The first two are mutually recursive, well-founded in the lexicographic order $\langle$step index ($T$), type ($\tau$), value $<$ expression$\rangle$.

*Value relation.* The value relation (denoted by $[\![.]\!]$) gives an interpretation to $\lambda$-amor$^-$ types (of kind $Type$) as sets of triples of the form $(p,T,v)$. Importantly, the potential $p$ is an upper-bound on the ambient potential *required* to construct the value $v$. It must include potential associated with the (types of) subexpressions of $v$.

We describe interesting cases of this relation. The interpretation for the list type is defined by a further induction on list size. For a list of size 0 the value relation contains a *nil* value with any potential (since *nil* captures no potential). For a list of size $s+1$, the value relation is defined inductively on $s$, similar to the tensor pair, which we describe next. For a tensor ($\otimes$) pair, both components can be used. Therefore, the potential required to construct a tensor pair is at least the sum of the potentials needed to construct the two components. On the other hand, for a with ($\&$) pair, either but not both of the components can be used by the context. So the potential needed for a $\&$ pair should be sufficient for each component separately. The type $!\tau$ contains $!e$ when $e$ is in $\tau$. The important aspect here is that the potential associated with $e$ must be 0, otherwise we would have immediate unsoundness due to replication of potential, as described in Section 1.

Next, we explain the interpretation of the arrow type $\tau_1 \multimap \tau_2$: $\lambda x.e$ is in this type with potential $p$ if for any substitution $e'$ (of type $\tau_1$) that comes with potential $p'$, the total potential $p + p'$ is sufficient for the body $e[e'/x]$ (of type $\tau_2$).

The step indices $T$ play an important role only in the interpretation of the polymorphic type $\forall\alpha.\tau$. Since the type-level parameter $\alpha$ may be substituted by any type, potentially one even larger than $\tau$, the relation would not be well-founded by induction on types alone. Here, we rely on step-indices, noting that substituting $\alpha$ with a type consumes at least one step in our operational semantics, so the relation for $\tau$ (with the substitution) needs to be defined only at a smaller step index. This follows prior work [Neis et al. 2011].

Next, we come to the new, interesting types for potential and the cost monad. The potential type $[n]\,\tau$ contains $v$ with required potential $p$ if $p$ is sufficient to account for $n$ *and* the potential required for $v$. (Note that the same value $v$ is in the interpretation of both $\tau$ and $[n]\,\tau$.) The graded monadic type $\mathbb{M}\,\kappa\,\tau$ contains the (impure) value $v$ with required potential $p$ if $p$ and $\kappa$ *together* suffice to cover the cost $\kappa'$ of actually forcing $v$ and the potential $p'$ required for the resulting value, i.e., if $p + \kappa \geq \kappa' + p'$. The ambient potential $p$ and the cost $\kappa$ on the monad appear together in a sum, which explains why the typing rule T-release can *offset* cost on the monad using potential.

The interpretation of a type family $\lambda_t i.\tau$ is a type-level function, as expected. The interpretation of type-level application is an application of such a function. The remaining cases of the value relation of Fig. 5 should be self-explanatory.

*Expression relation.* The expression relation, denoted $[\![.]\!]_{\mathcal{E}}$, maps a type to a set of triples of the form $(p, T, e)$. Its definition is fairly simple and standard: we simply check if the value $v$ obtained by pure evaluation of $e$ is in the value relation of the same type. The potential does not change during pure evaluation, but we adjust the step index correctly.

*Substitution relations.* Finally, we describe the substitution relations for the affine context ($\Gamma$) and the non-affine context ($\Omega$). Each relation maps the context to a set of valid substitutions for the context, paired with a step index and a potential. The two key points about the interpretation of $\Gamma$ are: 1) The substitution $\gamma$ should map each variable to a value of the correct type (semantically), and 2) The potential $p$ for the context should be more than the sum of the potentials required for the substitutions of each of the variables. The interpretation of the non-affine context $\Omega$ is simpler. It only demands that the substituted value is in the interpretation of the correct type with 0 potential.

*Soundness.* As is standard for logical-relations models, the main meta-theoretic property is the "fundamental theorem" (Theorem 2). The theorem basically says that if $e$ is well-typed in some contexts at type $\tau$, then the application of any substitutions in the semantic interpretation of the contexts map $e$ into the semantic interpretation of $\tau$. The important, interesting aspect of the theorem in $\lambda$-amor$^{-}$ is that the potential needed for $e$ (after substitution) equals the potential coming from the context, $p_l$. This the crux of the soundness of (amortized) cost analysis in $\lambda$-amor$^{-}$.

**Theorem 2** (Fundamental theorem for $\lambda$-amor$^{-}$)**.** $\forall\Theta, \Omega, \Gamma, e, \tau, T, p_l, \gamma, \delta, \sigma, \iota.$

$\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \wedge (p_l, T, \gamma) \in [\![\Gamma\ \sigma\iota]\!]_{\mathcal{E}} \wedge (0, T, \delta) \in [\![\Omega\ \sigma\iota]\!]_{\mathcal{E}} \wedge . \models \Delta\ \iota \implies$
$(p_l, T, e\ \gamma\delta) \in [\![\tau\ \sigma\iota]\!]_{\mathcal{E}}$.

Here, $\iota$, $\sigma$, $\delta$ and $\gamma$ denote substitutions for the index context $\Theta$, the type context $\Psi$, the non-affine context $\Omega$ and the affine context $\Gamma$, respectively. This theorem is proved by induction on the given typing judgment with a subinduction on the step-index for the case of fix. The technical appendix has the entire proof.

Theorem 1 is a direct corollary of this fundamental theorem. We can derive several additional corollaries about execution cost directly from this fundamental theorem. For instance, for open terms which only partially use the input potential and save the rest with the result, we can derive Corollary 3. Here, $e$ is a thunk that expects as input a unit argument, but with some associated potential $q$. When applied, $e$ returns a computation (of 0 cost) that forces to a value with a residual potential $q'$. The corollary says that if the context $\Gamma$ provides a potential $p_l$, then forcing $e$ (with the substitution $\gamma$) incurs a cost $J$ and produces a value $v$ that requires potential $p_v$ such that $J \leq (q + p_l) - (q' + p_v)$. This expression may look complex, but it is simply a difference of the incoming potentials of $e$ ($q$ and $p_l$) and the outgoing potentials of $e$ ($q'$ and $p_v$). In Section 4, we show an interesting use of this corollary for deriving an alternate proof of soundness of univariate RAML via its embedding in $\lambda$-amor$^{-}$.

**Corollary 3.** $\forall \Gamma, e, q, q', \tau, p_l, \gamma, J, v_t, v.$
$.; .; .; .; \Gamma \vdash e : [q]\ \mathbf{1} \multimap \mathbb{M}\,0\,([q']\ \tau) \wedge (p_l, \_, \gamma) \in [\![\Gamma]\!]_{\mathcal{E}} \wedge e\ ()\ \gamma \Downarrow v_t \Downarrow^J v \implies$
$\exists p_v.\ (p_v, \_, v) \in [\![\tau]\!] \wedge J \leq (q + p_l) - (q' + p_v)$.

## 3 EXAMPLES

Next, we show three nontrivial examples of cost analysis in $\lambda$-amor$^{-}$. Complete type derivations for the examples can be found in the technical appendix. The technical appendix also has additional examples that we omit here: Okasaki's implicit queue in the call-by-name setting, and the standard list map and append functions.

### 3.1 Functional queue

Eager functional FIFO queues are often implemented using two LIFO stacks represented as standard functional lists, say $l_1$ and $l_2$. Enqueue is implemented as a push (cons) on $l_1$. Dequeue is implemented as a pop (head) from $l_2$ if it is non-empty. However, if $l_2$ is empty, then the contents of $l_1$ are transferred, one at a time, to $l_2$ and the new $l_2$ is popped. The transfer from $l_1$ to $l_2$ reverses $l_1$, thus changing the stack's LIFO semantics to a queue's FIFO semantics. We describe the amortized analysis of this eager queue in $\lambda$-amor$^{-}$. Our cost model is that every list cons operation incurs a unit cost and no other operation incurs any cost.

Note that the worst-case cost of dequeue is linear in the size of $l_1$. However, the *amortized* cost of dequeue is actually constant. This is proved by counting the cost of transferring an element from $l_1$ to $l_2$ ahead of time – when that element is enqueued in $l_1$. This works because an enqueued element can be transferred from $l_1$ to $l_2$ at most once. Concretely, the enqueue operation has a cost (it requires a potential) of 3 units, of which 1 is used by the enqueue operation itself and the remaining 2 are stored as potential with the element in the list $l_1$, to be used later in the dequeue operation if required. This is reflected in the type of enqueue below. The program code for enqueue is straightforward, so we skip it here.

$enq : \forall m, n.\ [3]\ \mathbf{1} \multimap \tau \multimap L^n([2]\ \tau) \multimap L^m\tau \multimap \mathbb{M}\,0\,(L^{n+1}([2]\ \tau) \otimes L^m\tau)$

Observe how each element of the first list $l_1$ in both the input and the output has a potential 2 associated with it. The dequeue operation (denoted by $dq$ below) is a bit more involved. The

constraints in the type of dequeue reflect that: a) dequeue can only be performed on a non-empty queue, i.e., if $m + n > 0$ and b) the sum of the lengths of the resulting list is only 1 less than the length of the input lists, i.e., $\exists m', n'.((m' + n' + 1) = (m + n))$. The full type and the term for the dequeue operation are described in Listing 1. Dequeue uses a function *move*, which moves elements from the first list to the second. We skip the description of *move*. Type-checked terms for *enq*, *dq* and *move* are in the technical appendix.

$$dq : \forall m, n.(m + n > 0) \Rightarrow L^m([2]\,\tau) \multimap L^n\tau \multimap \mathbb{M}\,0\,(\exists m', n'.((m' + n' + 1) = (m + n)) \& (L^{m'}[2]\,\tau \otimes L^{n'}\tau))$$

$$\begin{array}{l} dq \triangleq\ \Lambda.\Lambda.\Lambda.\lambda\ l_1\ l_2.\ \mathsf{match}\ l_2\ \mathsf{with} \\ \quad |nil \mapsto \mathsf{bind}\, l_r = move\ []\,[]\ l_1\ nil\ \mathsf{in} \\ \qquad \mathsf{match}\ l_r\ \mathsf{with} \\ \qquad\quad |nil \mapsto \mathsf{fix}x.x \\ \qquad\quad |h_r :: l_r' \mapsto \mathsf{ret}\,\Lambda.\langle\langle nil, l_r'\rangle\rangle \\ \quad |h_2 :: l_2' \mapsto \mathsf{ret}\,\Lambda.\langle\langle l_1, l_2'\rangle\rangle \end{array}$$

Listing 1. Dequeue operation for eager functional queue in $\lambda$-amor$^-$

### 3.2 Church encoding

Unlike our previous example where we had a fixed cost for both enqueue and dequeue, now we describe a function whose cost actually varies each time it is applied. To illustrate this point, we encode Church numerals in $\lambda$-amor$^-$. Church numerals are a classic example of higher-order terms where an argument function is applied multiple times to encode various natural numbers. This means that if we want to assign a variable cost (say $C\ i$ for the $i$th instance) to the argument function, we really need to make the type polymorphic in $i$. This gives rise to a type which uses *second-rank index polymorphism*.

To recap, Church numerals encode natural numbers as function applications. For example, a Church zero is defined as $\lambda f.\lambda x.x$ (with zero applications), a Church one as $\lambda f.\lambda x.f\ x$ (with one application), a Church two as $\lambda f.\lambda x.f\ (f\ x)$ (with two applications) and so on. To type a Church numeral, we must specify a type for $f$. We assume that we have an $\mathbb{N}$-indexed family of types $\alpha$, and that $f$ maps $\alpha\ i$ to $\alpha\ (i+1)$ for every $i$. Then, the $n$th Church numeral, given such a function $f$, maps $\alpha\ 0$ to $\alpha\ n$.

Next, we consider costs. Just for illustration, suppose that we count a unit cost for every *function application*. We want to encode the precise costs of operations like addition, multiplication and exponentiation in types. Classically, these operations are defined compositionally. For example, addition of $m$ and $n$ is defined by applying $m$ to the successor function and $n$. This iterates the successor function $m$ times over $n$. To type this, the type of $f$ in the Church nats must be general enough. For this, we use a cost family $C$ from $\mathbb{N}$ to $\mathbb{R}^+$. The cost of applying $f$ depends on the index of the argument. Then, given such a $f$, the $n$th Church numeral maps $\alpha\ 0$ to $\alpha\ n$ with cost $((\sum_{i<n} C\ i) + n)$, where each $C\ i$ is the cost of using $f$ the $i$th time, and the last $n$ is the cost of the $n$ applications in the definition of the $n$th Church numeral. Our type for Church numerals, called Nat below, captures exactly this intuition.

$$\begin{array}{l} \mathsf{Nat} = \lambda_t n.\forall\alpha : \mathbb{N} \to Type.\forall C : \mathbb{N} \to \mathbb{R}^+. \\ \qquad !(\forall j.((\alpha\ j \otimes [C\ j]\,\mathbf{1}) \multimap \mathbb{M}\,0\,(\alpha\ (j+1)))) \multimap \\ \qquad \mathbb{M}\,0\,((\alpha\ 0 \otimes [(\textstyle\sum_{i<n} C\ i) + n]\,\mathbf{1}) \multimap \mathbb{M}\,0\,(\alpha\ n)) \end{array}$$

Below, we describe a term for the Church one, denoted $\overline{1}$, that has type Nat 1. For notational simplification, we define $e_1\ \uparrow^1 e_2 \triangleq (\mathsf{bind} - = \uparrow^1\ \mathsf{in}\ \mathsf{ret}(e_1\ e_2))$, which applies $e_1$ to $e_2$ and additionally incurs a cost of 1 unit.

$$\overline{1} : \mathsf{Nat}\ 1$$

$$\overline{1} \triangleq \Lambda.\Lambda.\lambda f.\,\mathsf{ret}\ (\lambda x.\,\mathsf{let}\,!f_u = f\ \mathsf{in}\ \mathsf{let}\,\langle\langle y_1, y_2\rangle\rangle = x\ \mathsf{in}\ \mathsf{release} - = y_2\ \mathsf{in}\ \mathsf{bind}\, a = \mathsf{store}()\ \mathsf{in}\ f_u\ []\ \uparrow^1\langle\langle y_1, a\rangle\rangle)$$

The term $\overline{1}$ takes the input pair $x$ of type $\alpha\ 0 \otimes [((C\ 0) + 1)]\ \mathbf{1}$ and binds its two components to $y_1$ and $y_2$. It then releases the potential $((C\ 0) + 1)$ in $y_2$, stores $C\ 0$ of the released potential in $a$, and applies the input function $f_u$ to $\langle\langle y_1, a\rangle\rangle$, incurring a cost of 1 unit. This incurred cost models the cost of the application (which we want to count). It is offset by the remaining 1 potential that was released from $y_2$.

Next, we show the encoding for Church addition. Church addition is defined using a successor function (*succ*), which is also defined and type-checked in $\lambda$-amor$^{-}$, but whose details we elide here. It is just enough to know that the cost of *succ* under our cost model is two units.

$$succ : \forall n.\ [2]\ \mathbf{1} \multimap \mathbb{M}\,0\ (\mathsf{Nat}[n] \multimap \mathbb{M}\,0\ (\mathsf{Nat}[n+1]))$$

An encoding of Church addition (*add*) in $\lambda$-amor$^{-}$ is shown below. The type of *add* takes the required potential ($4 * n_1 + 2$ here) along with two Church naturals (Nat $n_1$ and Nat $n_2$) as arguments and computes their sum. The potential of $(4 * n_1 + 2)$ units corresponds to the precise cost of performing the Church addition in our cost model. The whole type is parameterized on $n_1$ and $n_2$. Ignoring the decorations for monadic operations, *add* simply applies $\overline{N_1}$ to *succ* and $\overline{N_2}$, as expected.

$$\begin{aligned}
&add : \forall n_1, n_2.\ [(4 * n_1 + 2)]\ \mathbf{1} \multimap \mathbb{M}\,0\ (\mathsf{Nat}\ n_1 \multimap \mathbb{M}\,0\ (\mathsf{Nat}\ n_2 \multimap \mathbb{M}\,0\ (\mathsf{Nat}\ (n_1 + n_2))))\\
&add \triangleq\ \Lambda.\Lambda.\lambda p.\ \mathsf{ret}(\lambda\overline{N_1}.\ \mathsf{ret}(\lambda\overline{N_2}.\ \ \mathsf{release}\ -\ =\ p\ \mathsf{in}\ \ \mathsf{bind}\ a = E_1\ \mathsf{in}\ E_2))\\
&E_1 \triangleq\ \overline{N_1}\ \ []\ \ []\ \ \uparrow^1!(\Lambda.\lambda t.\ \mathsf{let}\ \langle\langle y_1, y_2\rangle\rangle = t\ \mathsf{in}\ \ \mathsf{release}\ -\ =\ y_2\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{bind}\ b_1 = (\mathsf{bind}\ b_2 = \mathsf{store}()\ \mathsf{in}\ (succ\ []\ b_2))\ \mathsf{in}\ \ b_1\ \uparrow^1 y_1)\\
&E_2 \triangleq\ \mathsf{bind}\ b = \mathsf{store}()\ \mathsf{in}\ a\ \uparrow^1\langle\langle\overline{N_2}, b\rangle\rangle
\end{aligned}$$

Listing 2. Encoding of the Church addition in $\lambda$-amor$^{-}$

We have similarly encoded Church multiplication and exponentiation in $\lambda$-amor$^{-}$. We are unaware of such a general encoding of Church numerals in a monadic cost framework without the use of potentials.

### 3.3 List fold

As our final example, we describe the cost analysis of a program that uses higher-order functions with variable cost and recursion, a combination that prior work like RAML/AARA [Hoffman 2011; Hoffmann et al. 2017] cannot handle. Our example is a slightly modified right-fold function for lists.

Conceptually, the right fold function aggregates (folds) a list by applying an aggregation function (given as an argument) to the elements of a list (starting from the rightmost element) along with a starting value (also given as an argument). We are interested in an encoding where the cost of applying the aggregation function varies in each recursion step. To model this kind of variable cost we make use of a cost family $C : \mathbb{N} \to \mathbb{R}^+$. The intent is that every time the aggregation function is applied, its cost could be different. In particular, the cost of the $i$th application is $C\ (n - i)$ units, where $n$ is the length of the list.

$$\begin{aligned}
&foldr : \forall \alpha, \beta, n, C : \mathbb{N} \to \mathbb{R}^+.\\
&!(\forall i.\ [C\ \ i]\ \mathbf{1} \multimap \mathsf{Nat}(i) \multimap \alpha \multimap \beta \multimap \mathbb{M}\,0\,\beta) \multimap\ !\mathsf{Nat}(n) \multimap \beta \multimap L^n\alpha \multimap [\textstyle\sum_{i<n} C\ i]\ \mathbf{1} \multimap \mathbb{M}\,0\,\beta
\end{aligned}$$

$$\begin{aligned}
&foldr \triangleq \mathsf{fix}f'.\Lambda.\Lambda.\Lambda.\Lambda.\lambda f\ \ c\ \ s\ \ ls\ \ p.\\
&\mathsf{let}\ \ !f_u = f\ \ \mathsf{in}\\
&\ \ \mathsf{let}\ \ !c_u = c\ \ \mathsf{in}\\
&\ \ \ \ \mathsf{match}\ \ ls\ \ \mathsf{with}\\
&\ \ \ \ \ \ |nil \mapsto \mathsf{ret}\ s\\
&\ \ \ \ \ \ |h :: t \mapsto \mathsf{release}\ _ = p\ \mathsf{in}\\
&\qquad\qquad\quad \mathsf{bind}\ p' = \mathsf{store}()\ \mathsf{in}
\end{aligned}$$

```
bind p'' = store() in
  bind tr = f' [][][][] !f_u !(c_u − 1) s t p'' in
    (f_u [] p' (c_u − 1) h tr)
```

Listing 3. foldr function

The type and the term for $foldr$ are described in Listing 3. The type of $foldr$ is parametric in $\alpha, \beta, n$ and $C$, where $\alpha$ is the type of the elements of the list, $\beta$ is the type of the result, $n$ is the length of the input list and $C$, as mentioned earlier, is the cost family used to model the varying cost of the aggregation function. To obtain a variable cost for each application of the aggregating function, we make use of *second-rank index polymorphism* like we did for Church numerals. This means that we parameterize the aggregating function with an index which can be instantiated appropriately to obtain the desired cost. We specify the cost of each application as a potential of $(C\ i)$ units in a negative position of the aggregating function. To obtain a function whose cost truly depends on $i$, we add a parameter of type $\mathrm{Nat}(i)$ (which denotes a singleton type over naturals) to the aggregating function (the first argument of $foldr$).[5] The overall type of the aggregating function has an exponential because we use it multiple times, once in every recursive call. Finally, the total cost (obviously $\sum_{i<n} C\ i$ units) must be provided as an input potential to $foldr$.

## 4 EMBEDDING UNIVARIATE RAML

In this section we describe an embedding of Resource Aware ML (RAML) [Hoffman 2011; Hoffmann and Hofmann 2010] into $\lambda$-amor$^{-}$. RAML is an *effect-based* type system for amortized analysis of OCaml programs using the method of potentials [Cormen et al. 2009; Tarjan 1985]. The main motivation for this embedding is to show that: 1) $\lambda$-amor$^{-}$ can also perform *effect-based* cost analysis like RAML and thus can be used to analyze all examples that have been tried on RAML, 2) $\lambda$-amor$^{-}$, despite being *call-by-name* in the pure part, can embed RAML which is a *call-by-value* framework.

We describe an embedding of Univariate RAML [Hoffman 2011; Hoffmann and Hofmann 2010] (which subsumes Linear RAML [Hofmann and Jost 2003]) into $\lambda$-amor$^{-}$. We leave embedding multivariate RAML [Hoffmann et al. 2011] to future work but anticipate no fundamental difficulties in doing so.

### 4.1 A brief primer on Univariate RAML

We give a brief primer on Univariate RAML [Hoffman 2011; Hoffmann and Hofmann 2010] here. The key feature of Univariate RAML is its ability to encode univariate polynomials in the size of the input as potential functions. Such functions are expressed as non-negative linear combinations of binomial coefficients $\binom{n}{k}$, where $n$ is the size of the input data structure and $k$ is some natural number. Vector annotations on the list type $L^{\vec{q}}\tau$, for instance, are used as a representation of such univariate polynomials. The underlying potential on a list of size $n$ and type $L^{\vec{q}}\tau$ can then be described as $\phi(\vec{q}, n) \triangleq \sum_{1\le i\le k} \binom{n}{i} q_i$ where $\vec{q} = \{q_1 \dots q_k\}$. The authors of RAML show using the properties of binomial coefficients, that such a representation is amenable to an inductive characterization of polynomials which plays a crucial role in setting up the typing rules of their system. If $\vec{q} = \{q_1 \dots q_k\}$ is the potential vector associated with a list then $\triangleleft(\vec{q}) = \{q_1 + q_2, q_2 + q_3, \dots q_{k-1} + q_k, q_k\}$ is the potential vector associated with the tail of that list. Trees follow a treatment similar to lists. Base types (unit, bools, ints) have zero potential and the potential of a pair is just the sum of the potentials of the components. A snippet of the definition of the potential function $\Phi(a : A)$ (from Hoffman [2011]) is described below.

A type system is built around this basic idea with a typing judgment of the form $\Sigma; \Gamma \vdash^{q}_{q'} e_r : \tau$ where $\Gamma$ is a typing context mapping free variables to their types, $\Sigma$ is a context for function

[5] One could choose to represent $\mathrm{Nat}(i)$ as $L^{i}\mathbf{1}$ in $\lambda$-amor. We use singleton types just for clarity here.

$$\Phi(a:A) = 0 \quad \text{where } A \in \{unit, int, bool\} \qquad \Phi([]:L^{\vec{q}}A) = 0$$
$$\Phi((a_1,a_2):(A_1,A_2)) = \Phi(a_1:A_1)+\Phi(a_2:A_2) \qquad \Phi((a::\ell):L^{\vec{q}}A) = q_1+\Phi(a:A)+\Phi(\ell:L^{\lhd\vec{q}}A) \text{ where } \vec{q}=\{q_1\ldots q_k\}$$

$$\frac{\tau_1 \overset{q/q'}{\to} \tau_2 \in \Sigma(f)}{\Sigma; x:\tau_1 \vdash^{q+K_1^{app}}_{q'-K_2^{app}} f\ x : \tau_2}\ \text{app} \qquad \frac{\vec{p}=(p_1,\ldots,p_k)}{\Sigma; x_h:\tau, x_t : L^{(\lhd\vec{p})}\tau \vdash^{q+p_1+K^{cons}}_{q} cons(x_h,x_t) : L^{\vec{p}}\tau}\ \text{cons}$$

Fig. 6. Selected type rules of Univariate RAML from Hoffman [2011]

signatures mapping a function name to a type (this is separate from the typing context because RAML only has first-order functions that are declared at the top-level), $q$ and $q'$ denote the statically approximated available and remaining potential before and after the execution of $e_r$, respectively, and $\tau$ is the zero-order type of $e_r$. Vector annotations are specified on list and tree types (as mentioned above).

Types of first-order functions follow an intuition similar to the typing judgment above. $\tau_1 \overset{q/q'}{\to} \tau_2$ denotes the type of a first-order RAML function which takes an argument of type $\tau_1$ and returns a value of type $\tau_2$. $q$ units of potential are needed before this function can be applied and $q'$ units of potential are left after this function has been applied. Intuitively, the cost of the function is upper-bounded by ($q$+potential of the input) - ($q'$+potential of the result). Fig. 6 describe typing rules for function application and list cons. The *app* rule type-checks the function application with an input and remaining potential of $(q+K_1^{app})$ and $(q'-K_2^{app})$[6] units, respectively. RAML divides the cost of application into $K_1^{app}$ and $K_2^{app}$ units. Of the available $q+K_1^{app}$ units, $q$ units are required by the function itself and $K_1^{app}$ units are consumed before the application is performed. Likewise, of the remaining $q'-K_2^{app}$ units, $q'$ units are made available from the function and $K_2^{app}$ units are consumed after the application is performed. The *cons* rule requires an input potential of $q+p_1+K^{cons}$ units of which $p_1$ units are added to the potential of the resulting list and $K^{cons}$ units are consumed as the cost of performing this operation.

Soundness of the type system is defined by Theorem 4. Soundness is defined for *top-level* RAML programs (formalized later in Definition 6), which basically consist of first-order function definitions (denoted by $F$) and the "main" expression $e$, where execution starts. Stack (denoted by $V$) and heap (denoted by $H$) are used to provide bindings for free variables and locations in $e$.

**Theorem 4** (Univariate RAML's soundness)**.** $\forall H, H', V, \Gamma, \Sigma, e, \tau, {}^s v, p, p', q, q', t.$
$P = F, e$ *is a RAML top-level program and*
$H \models V : \Gamma \wedge \Sigma, \Gamma \vdash^{q}_{q'} e : \tau \wedge V, H \vdash^{p}_{p'} e \Downarrow_t {}^s v, H' \implies p - p' \leq (\Phi_{H,V}(\Gamma)+q) - (q' + \Phi_H({}^s v:\tau))$

### 4.2 Type-directed translation of Univariate RAML into $\lambda$-amor$^-$

As mentioned above, types in Univariate RAML include unit, booleans, integers, lists, trees, pairs and first-order functions. Without loss of generality we introduce two simplifications: a) we abstract RAML's bool and int types into an arbitrary base type denoted by b and b) we just choose to work with the list type only ignoring trees. These simplifications only make the development more concise as we do not have to deal with the redundancy of treating similar types again and again.

The translation from Univariate RAML to $\lambda$-amor$^-$ is type-directed. We describe the type translation function (denoted by $(\!|.|\!)$) from RAML types to $\lambda$-amor$^-$ types in Fig. 7.

Since RAML allows for full replication of unit and base types, we translate RAML's base type, b, into !b of $\lambda$-amor$^-$. But translation of the unit type does not need a !, as **1** and !**1** are isomorphic in $\lambda$-amor$^-$. Unlike the unit and base type of RAML, the list type does have some potential associated with it, represented by $\vec{q}$. Therefore, we translate RAML's list type into a pair type composed of

[6]Every time a subtraction like $(I-J)$ appears, RAML implicitly assumes that there is a side condition $(I-J) \geq 0$.

$$
\begin{array}{lcl|lcl}
(\!|unit|\!) & = & \mathbf{1} & (\!|(\tau_1,\tau_2)|\!) & = & ((\!|\tau_1|\!) \otimes (\!|\tau_2|\!)) \\
(\!|\mathsf{b}|\!) & = & !\mathsf{b} & (\!|\tau_1 \overset{q/q'}{\to} \tau_2|\!) & = & [q]\,\mathbf{1} \multimap (\!|\tau_1|\!) \multimap \mathbb{M}\,0\,([q']\,(\!|\tau_2|\!)) \\
(\!|L^{\vec{q}}\ \tau|\!) & = & \exists s.([\phi(\vec{q},s)]\,\mathbf{1} \otimes L^s(\!|\tau|\!)) & & &
\end{array}
$$

Fig. 7. Type translation of Univariate RAML

$$
\frac{\tau_1 \overset{q/q'}{\to} \tau_2 \in \Sigma(f)}{\Sigma; x:\tau_1 \vdash^{q+K_1^{app}}_{q'-K_2^{app}} f\ x : \tau_2 \rightsquigarrow \lambda u.\mathsf{release}\ \_ = u\ \mathsf{in}\ \ \mathsf{bind}\ \_ = \uparrow^{K_1^{app}}\ \mathsf{in}\ \mathsf{bind}\ P = \mathsf{store}()\ \mathsf{in}\ E_1}\ \text{app}
$$

$$
E_1 = \mathsf{bind}\ f_1 = (f\ P\ x)\ \mathsf{in}\ \mathsf{release}\ f_2 = f_1\ \mathsf{in}\ \ \mathsf{bind}\ \_ = \uparrow^{K_2^{app}}\ \mathsf{in}\ \mathsf{bind}\ f_3 = \mathsf{store}\ f_2\ \mathsf{in}\ \mathsf{ret}\ f_3
$$

Fig. 8. Expression translation for the app case: Univariate RAML to $\lambda$-amor$^-$

a modal unit type carrying the required potential and a $\lambda$-amor$^-$ list type. Since the list type in $\lambda$-amor$^-$ is refined with size, we add an existential on the pair to quantify the size of the list. The potential captured by the unit type must equal the potential associated with the RAML list (this is represented by the function $\phi(\vec{q},s)$). The function $\phi(\vec{q},s)$ corresponds to the one that RAML uses to compute the total potential associated with a list of $s$ elements, which we described above. Note the difference in how potentials are managed in RAML vs how they are managed in the translation. In RAML, the potential for an element gets added to the potential of the tail with every cons operation and, dually only the potential of the head element is consumed in the match operation. The translation, however, does not assign potential on a per-element basis. Instead, the total potential of the entire list is captured using the $\phi$ function and the translations of the cons and the match expressions work by adding or removing potential from this total. We believe a translation which works with per element potential is also feasible but we would need an additional index to identify the elements of the list in the list data type.

We translate a RAML pair type into a tensor ($\otimes$) pair. This is in line with how pairs are treated in RAML (both elements of the pair are available on elimination). Finally, a function type $\tau_1 \overset{q/q'}{\to} \tau_2$ in RAML is translated into the function type $[q]\,\mathbf{1} \multimap (\!|\tau_1|\!) \multimap \mathbb{M}\,0\,([q']\,(\!|\tau_2|\!))$. As in RAML, the translated function type also requires a potential of $q$ units for application and a potential of $q'$ units remains after the application. The monadic type is required because we cannot release/store potential without going into the monad. The translation of typing contexts is defined pointwise using the type translation function.

We use this type translation function to produce a translation for Univariate RAML expressions by induction on RAML's typing judgment. The translation judgment is $\Sigma;\Gamma \vdash^{q}_{q'} e_r : \tau \rightsquigarrow e_a$. It basically means that a well-typed RAML expression $e_r$ is translated into a $\lambda$-amor$^-$ expression $e_a$. The translated expression is of the type $[q]\,\mathbf{1} \multimap \mathbb{M}\,0\,([q'](\!|\tau|\!))$. We only describe the app rule here (Fig. 8). Since we know that the desired term must have the type $[q+K_1^{app}]\,\mathbf{1} \multimap \mathbb{M}\,0\,([q'-K_2^{app}](\!|\tau|\!))$, the translated term is a function which takes an argument, $u$, of the desired modal type and releases the potential to make it available for consumption. The continuation then consumes $K_1^{app}$ potential that leaves $q$ potential remaining for $\mathsf{bind}\ P = \mathsf{store}()\ \mathsf{in}\ E_1$. We then store $q$ units of potential with the unit and use it to perform a function application. We get a result of type $\mathbb{M}\,0\,([q']\,(\!|\tau_2|\!))$. We release these $q'$ units of potential and consume $K_2^{app}$ units from it. This leaves us with a remaining potential of $q'-K_2^{app}$ units. We store this remaining potential with $f_2$ and wrap it up in a monad to get the desired type. Translations of other RAML terms (which we do not describe here) follow a similar approach. The entire translation is intuitive and relies extensively on the ghost operations store and release at appropriate places.

We show that the translation is type-preserving by proving that the obtained $\lambda$-amor$^-$ terms are well-typed (Theorem 5). The proof of this theorem works by induction on RAML's type derivation.

**Theorem 5** (Type preservation: Univariate RAML to $\lambda$-amor$^{-}$). *If* $\Sigma;\Gamma \vdash^{q}_{q'} e : \tau$ *in Univariate RAML then there exists* $e'$ *such that* $\Sigma;\Gamma \vdash^{q}_{q'} e : \tau \rightsquigarrow e'$ *and there is a derivation of* $.;.;.;(\!|\Sigma|\!), (\!|\Gamma|\!) \vdash e' : [q]\,\mathbf{1} \multimap \mathbb{M}\,0\,([q'](\!|\tau|\!))$ *in* $\lambda$-*amor*$^{-}$.

As mentioned earlier, RAML only has first-order functions which are defined at the top-level. So, we need to lift this translation to the top-level. Definition 6 defines the top-level RAML program along with the translation.

**Definition 6** (Top level RAML program translation). *Assume a top-level RAML program*
$P \triangleq F, e_{main}$ *where* $F \triangleq f1(x) = e_{f1}, \ldots, fn(x) = e_{fn}$ *s.t.*
$\Sigma, x : \tau_{f1} \vdash^{q_1}_{q'_1} e_{f1} : \tau'_{f1} \ \ldots\ \Sigma, x : \tau_{fn} \vdash^{q_n}_{q'_n} e_{fn} : \tau'_{fn}$ *and* $\Sigma, \Gamma \vdash^{q}_{q'} e_{main} : \tau$
*where* $\Sigma = f1 : \tau_{f1} \overset{q_1/q'_1}{\rightarrow} \tau'_{f1}, \ldots, fn : \tau_{fn} \overset{q_n/q'_n}{\rightarrow} \tau'_{fn}$.
*Then, the translation of* $P$, *denoted by* $\overline{P}$, *is defined as* $(\overline{F}, e_t)$ *where*
$\overline{F} = \mathsf{fix} f_1.\lambda u.\lambda x.e_{t1}, \ldots, \mathsf{fix} f_n.\lambda u.\lambda x.e_{tn}$ *s.t.*
$\Sigma, x : \tau_{f1} \vdash^{q_1}_{q'_1} e_{f1} : \tau'_{f1} \rightsquigarrow e_{t1} \ \ldots\ \Sigma, x : \tau_{fn} \vdash^{q_n}_{q'_n} e_{fn} : \tau'_{fn} \rightsquigarrow e_{tn}$ *and*
$\Sigma, \Gamma \vdash^{q}_{q'} e_{main} : \tau \rightsquigarrow e_t$.

### 4.3 Semantic properties of the translation

Besides type-preservation, we additionally: 1) prove that our translation preserves semantics and cost of the source RAML term and 2) re-derive RAML's soundness result using $\lambda$-amor$^{-}$'s fundamental theorem (Theorem 2) and properties of the translation. This is a sanity check to ensure that our type translation preserves cost meaningfully (otherwise, we would not be able to recover RAML's soundness theorem in this way).

Semantics and cost preservation is formally stated in Theorem 7, which can be read as follows: if $e_s$ is a closed source (RAML) term which translates to a target ($\lambda$-amor$^{-}$) term $e_t$ and if the source expression evaluates to a value (and a heap $H$, because RAML uses imperative boxed data structures), then the target term after applying to a unit (because the translation is always a function) can be evaluated to a value ${}^{t}v_f$ via pure ($\Downarrow$) and forcing ($\Downarrow^{J}$) relations such that the source and the target values are the same and the cost of evaluation in the target is at least as much as the cost of evaluation in the source.

**Theorem 7** (Semantics and cost preservation). $\forall H, e, {}^{s}v, p, p', q, q'.$
$.;. \vdash^{q}_{q'} e_s : \mathrm{b} \rightsquigarrow e_t \wedge .,. \vdash^{p}_{p'} e \Downarrow {}^{s}v, H \implies$
$\exists {}^{t}v_f, J. e_t() \Downarrow \_ \Downarrow^{J} {}^{t}v_f \wedge {}^{s}v = {}^{t}v_f \wedge p - p' \leq J$

The proof of Theorem 7 is via a cross-language relation between RAML and $\lambda$-amor$^{-}$ terms. The relation (described in the technical appendix) is complex because it has to relate RAML's imperative data structures (like list which is represented as a chain of pointers in the heap) with $\lambda$-amor$^{-}$'s purely functional datastructures. The fundamental theorem of this relation allows us to establish that the source expression and its translation are related, which implies semantics and cost preservation as required by Theorem 7.

Finally, we re-derive RAML's soundness (Theorem 4) in $\lambda$-amor$^{-}$ using $\lambda$-amor$^{-}$'s fundamental theorem and the properties of the translation. To prove this theorem, we obtain a translated term corresponding to the term $e$ (of Theorem 4) via our translation. Then, using Theorem 7, we show that the cost of forcing the unit application of the translated term is lower-bounded by $p - p'$. After that, we use Corollary 3 to obtain the upper-bound on $p - p'$ as required in the statement of Theorem 4.

$$
\begin{array}{lll}
\text{Indices} & I, J, K & ::= \quad \ldots \mid \triangle_a^{J,K} I \mid \ldots \\
\text{Types} & \tau & ::= \quad \ldots \mid !_{a<I}\tau \mid \ldots \\
\text{Non-affine context for term variables} & \Omega & ::= \quad . \mid \Omega, x :_{a<I} \tau
\end{array}
$$

$$
\Omega_1 + \Omega_2 \triangleq \begin{cases} \Omega_2 & \Omega_1 = . \\ (\Omega_1' + \Omega_2/x), x :_{c<I+J} \tau & \Omega_1 = \Omega_1', x :_{a<I} \tau[a/c] \wedge (x :_{b<J} \tau[I+b/c]) \in \Omega_2 \\ (\Omega_1' + \Omega_2), x :_{a<I} \tau & \Omega_1 = \Omega_1', x :_{a<I} \tau \wedge (x :_{\_} -) \notin \Omega_2 \end{cases}
$$

$$
\textstyle\sum_{a<I} \Omega \triangleq \begin{cases} . & \Omega = . \\ (\sum_{a<I} \Omega), x :_{c<\sum_{a<I} J} \sigma & \Omega = \Omega', x :_{b<J} \sigma[(\sum_{d<a} J[d/a] + b)/c] \end{cases}
$$

Fig. 9. Changes to the type system syntax to obtain $\lambda$-amor from $\lambda$-amor$^-$

## 5 $\lambda$-amor (WITH SUB-EXPONENTIALS)

$\lambda$-amor$^-$ is quite expressive, but it can only represent one or an unbounded number of copies of a term. This was evident in the encoding of Church numerals (Section 3.2) and $foldr$ (Section 3.3). In the Church numeral $n$, the argument function can only be used $n$ times, yet the type requires an unbounded number of copies of the function, since we cannot express "$n$ copies" using just !. A similar situation reappeared in $foldr$ where the aggregating function can only be used $n$ times (once in every recursion step), but the type requires an unbounded number of copies of this function.

To overcome this limitation and improve expressiveness, we refine the exponential type $!\tau$ of $\lambda$-amor$^-$ to a dependent sub-exponential $!_{i<n}\tau$, which is morally equivalent to the iterated tensor $\tau[0/i] \otimes \tau[1/i] \ldots \otimes \tau[(n-1)/i]$. Thus, $!_{i<n}\tau$ not only specifies a finite bound $n$ on the number of copies of the underlying term, but also provides the ability to give each of them a different type (by varying the substitution for $i$). To the best of our knowledge, this dependent sub-exponential was first introduced in Bounded Linear Logic [Girard et al. 1992]. Subsequent work on $d\ell$PCF [Dal Lago and Gaboardi 2011] and its variant [Dal Lago and Petit 2012] showed how the dependent sub-exponential can be used as part of a *coeffect*-based cost analysis for PCF programs.

In the rest of this section, we extend $\lambda$-amor$^-$ with the dependent sub-exponential to obtain the full system $\lambda$-amor. This allows the representation of bounded replication as explained above. However, unlike the work on $d\ell$PCF, we do not adopt the coeffect-style of cost analysis in $\lambda$-amor. Costs are still represented via the cost monad of $\lambda$-amor$^-$. We later show in Section 6 how $d\ell$PCF's coeffect-based cost analysis can be simulated in $\lambda$-amor.

### 5.1 Changes to the types and type system

*Syntax.* We take the same language that was described in Section 2 but replace the exponential type with an indexed sub-exponential type. There are no changes to the term syntax or the semantics of the language. However, we extend the index term language with additional counting constructs that are described below. The changes to the types and indices are summarized in Fig. 9.

In particular, we include the *forest cardinality* operator $\triangle_a^{I,J} K$ in the index language. This operator, inspired from $d\ell$PCF, counts the number of nodes in a forest of trees. Specifically, suppose we have a forest of trees whose nodes are numbered in a pre-order depth-first traversal starting from the roots of the trees (where the trees have been totally ordered in some way). Assume that the number of children of node $a$ is $K(a)$ ($K$ has $a$ free in it). Then, $\triangle_a^{I,J} K$ is the total number of nodes in the trees rooted at the node $I$ and its next $J-1$ siblings. The formal definition of the forest cardinality operator is shown in Fig. 10. The forest cardinality operator is used to represent the cardinality of arbitrary recursion trees (note that $\lambda$-amor inherits the general fixpoint operator from $\lambda$-amor$^-$).

$$\begin{aligned} \triangle_a^{I,0} K &= 0 \\ \triangle_a^{I,J+1} K &= \triangle_a^{I,J} K + (\triangle_a^{I+1+\triangle_a^{I,J}K, K[I+\triangle_a^{I,J}K/a]} K) \end{aligned}$$

Fig. 10. Formal definition of forest cardinality from [Dal Lago and Gaboardi 2011]

*Typing judgment.* The typing judgment of $\lambda$-amor is the same as that of $\lambda$-amor$^-$: $\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau$. However, the definition of $\Omega$ is now different. The non-affine context $\Omega$ now carries elements of the form $x :_{a<I} \tau$ (Fig. 9). An element of this form means that $I$ copies of $x$ are available with types $\tau[0/i] \ldots \tau[(I-1)/i]$, mirroring the dependent sub-exponential. The non-affine context also differs in the definition of the operator +, which was formerly just a disjoint union. Now, the contexts being added may have common variables and we have to add their multiplicities. The revised + operation on non-affine contexts is defined in Fig. 9. The figure also defines an iterated sum operation on contexts, $\sum_{a<I} \Omega$.

*Typing rules.* We only describe the type rules for the sub-exponential, the fixpoint and the use of non-affine assumptions as these are the only rules of $\lambda$-amor$^-$ that change. See Fig. 11. T-subExpI is the rule for the introduction form of the sub-exponential. It says that if an expression $e$ has type $\tau$ in the non-affine context $\Omega$ and the constraint $a < I$ for a fresh parameter $a$, and $e$ does not use any any affine resources (indicated by an empty $\Gamma$) then $!e$ has type $!_{a<I}\tau$ in the non-affine context $\sum_{a<I} \Omega$. Observe how the multiplicity $I$ of resources in the introduced type $!_{a<I}\tau$ matches the multiplicity $I$ in the concluding context $\sum_{a<I} \Omega$. The dual rule T-subExpE is straightforward: Eliminating $!_{a<I}\tau$ yields the assumption $x :_{a<I} \tau$ in the non-affine context for the continuation. The rule T-var2 specifies when such a hypothesis from the non-affine context can be used. The key requirement is that $I \geq 1$, i.e., the multiplicity of the hypothesis being used should be at least 1.

The fixpoint expression (fix$x.e$) encodes recursion by allowing $e$ to refer to fix$x.e$ via $x$. T-fix is the typing rule for this fixpoint construct. It is a refinement of the corresponding rule in Fig. 3. The refinements serve two purposes: 1) they make the total number of recursive calls of the fixpoint explicit (this number is denoted $L$) and 2) they introduce a parameter $b$ that ranges over the different recursive calls, enumerated in a pre-order traversal of the recursion tree. The root of the tree (the top-level call) corresponds to $b = 0$. The type $\tau$ is parameterized by $b$. The term $I(b)$ (the term $I$ has $b$ free in it) is the number of children of the $b$th node in the recursion tree. The rule can be understood as follows. The first premise of the rule types an *arbitrary* recursive call corresponding to the node $b$ of the recursion tree. To type the fixpoint body $e$ for this call, the first premise allows $I(b)$ copies of the parameter $x$ with appropriate types. These copies correspond to the results of the recursive calls below node $b$ (note that $(b + 1 + \triangle_b^{b+1,a} I)$ is the index of the $a$th child of node $b$ in the recursion tree). The second premise merely says that $L$ must really be an upper bound on the cardinality of a tree in which the $b$th node has $I(b)$ children. The conclusion of the rule says that the entire fixpoint can be typed with $L$ copies of $\Omega$, and the final type is $\tau[0/b]$.

*Subtyping.* We also introduce a new subtyping rule, sub-bSum, which moves potential outside a sub-exponential to inside. The rule, shown below, is sound because it does not change the total potential. Potentials are anyway ghost, so moving them from one place to another is allowed in our semantic model. Formally, the soundness of this as well as other subtyping rules is captured by Lemma 8. $\sigma$ and $\iota$ represent the substitutions for the type and index variables respectively.

$$\frac{}{\Psi; \Theta; \Delta \vdash ([\textstyle\sum_{a<I} K]\ !_{a<I}\tau) <: (!_{a<I}\ [K]\ \tau)}\ \text{sub-bSum}$$

**Lemma 8** (Value subtyping lemma). $\forall \Psi, \Theta, \Delta, \tau \in \mathit{Type}, \tau', \sigma, \iota.$
$\Psi; \Theta; \Delta \vdash \tau <: \tau' \wedge . \models \Delta\iota \implies [\![\tau\ \sigma\iota]\!] \subseteq [\![\tau'\ \sigma\iota]\!]$

$$\frac{\Psi; \Theta, a; \Delta, a < I; \Omega; . \vdash e : \tau}{\Psi; \Theta; \Delta; \sum_{a<I} \Omega; . \vdash !e :!_{a<I}\tau} \text{ T-subExpI}$$

$$\frac{\Psi; \Theta; \Delta; \Omega_1; \Gamma_1 \vdash e : (!_{a<I}\tau) \qquad \Psi; \Theta; \Delta; \Omega_2, x :_{a<I} \tau; \Gamma_2 \vdash e' : \tau'}{\Psi; \Theta; \Delta; \Omega_1 + \Omega_2; \Gamma_1 + \Gamma_2 \vdash \text{let}\, !\, x = e \text{ in } e' : \tau'} \text{ T-subExpE}$$

$$\frac{\Theta, \Delta \models I \geq 1}{\Psi; \Theta; \Delta; \Omega, x :_{a<I} \tau; \Gamma \vdash x : \tau[0/a]} \text{ T-var2}$$

$$\frac{\Psi; \Theta, b; \Delta, b < L; \Omega, x :_{a<I} \tau[(b + 1 + \triangle_b^{b+1,a} I)/b]; . \vdash e : \tau \qquad L \geq \triangle_b^{0,1} I}{\Psi; \Theta; \Delta; \sum_{b<L} \Omega; . \vdash \text{fix } x.e : \tau[0/b]} \text{ T-fix}$$

Fig. 11. Changes to the type rules

It is noteworthy that sub-bSum is the *only* subtyping rule in $\lambda$-amor that specifies how two modalities interact. In particular, we do not have a rule to make the sub-exponential and the monad interact (i.e., we do not have what is often called a "distributive law" [Gaboardi et al. 2016]). We have not encountered the need for such an interaction. However, studying such an interaction could be an interesting direction for future work.

### 5.2 Model of types and soundness

We only describe the value relation for the sub-exponential here as the remaining cases of the value relation are exactly the same as before. $(p, T, !e)$ is in the value interpretation at type $!_{a<I}\tau$ iff the potential $p$ suffices for all $I$ copies of $e$ at the instantiated types $\tau[i/a]$ for $0 \leq i < I$. The other change to the model is in the interpretation of $\Omega$. This time we have $(p, \delta)$ instead of $(0, \delta)$ in the interpretation of $\Omega$ such that $p$ is sufficient for all copies of all variables in the context. Both changes to the model are described below.

$$\begin{array}{lcl} [\![!_{a<I}\tau]\!] & \triangleq & \{(p, !e) \mid \exists p_0, \ldots, p_{I-1}. p_0 + \ldots + p_{I-1} \leq p \wedge \forall 0 \leq i < I. (p_i, e) \in [\![\tau[i/a]]\!]_{\mathcal{E}}\} \\ [\![\Omega]\!]_{\mathcal{E}} & = & \{(p, \delta) \mid \exists f : \mathcal{V}ars \to \mathcal{I}ndices \to \mathcal{P}ots. \\ & & (\forall (x :_{a<I} \tau) \in \Omega. \forall 0 \leq i < I. (f\ x\ i, \delta(x)) \in [\![\tau[i/a]]\!]_{\mathcal{E}}) \wedge \\ & & (\sum_{x:_{a<I}\tau \in \Omega} \sum_{0 \leq i < I} f\ x\ i) \leq p\} \end{array}$$

We formalize the soundness of $\lambda$-amor's type system as the following fundamental theorem. Compared to the fundamental theorem of $\lambda$-amor$^-$ (Theorem 2), this fundamental theorem has an additional potential $p_m$ that comes from the interpretation of $\Omega$ (this potential was 0 in $\lambda$-amor$^-$).

**Theorem 9** (Fundamental theorem). $\forall \Psi, \Theta, \Delta, \Omega, \Gamma, e, \tau \in Type, p_l, p_m, \gamma, \delta, \sigma, \iota.$
$\Psi; \Theta; \Delta; \Omega; \Gamma \vdash e : \tau \wedge (p_l, \gamma) \in [\![\Gamma\ \sigma\iota]\!]_{\mathcal{E}} \wedge (p_m, \delta) \in [\![\Omega\ \sigma\iota]\!]_{\mathcal{E}} \wedge . \models \Delta\ \iota \implies$
$(p_l + p_m, e\ \gamma\delta) \in [\![\tau\ \sigma\iota]\!]_{\mathcal{E}}$

The proof of the theorem proceeds in a manner similar to that of Theorem 2, i.e., by induction on the typing derivation. Now, in the fix case, we additionally induct on the recursion tree. This also requires generalizing the induction hypothesis to account for the potential of the children of a node in the recursion tree. The technical appendix has the entire proof.

## 6 EMBEDDING $d\ell$PCF

In this section we describe an embedding of $d\ell$PCF [Dal Lago and Gaboardi 2011] into $\lambda$-amor. $d\ell$PCF is a *coeffect*-based cost-analysis type system (contrast this with RAML which is an *effect*-based type system) which has been shown to be relatively complete for the cost analysis of PCF programs.

$$\begin{array}{llll} d\ell\text{PCF terms} & t & ::= & n \mid s(t) \mid p(t) \mid ifz\ t\ then\ u\ else\ v \mid \lambda x.t \mid tu \mid \text{fix}\ x.t \\ d\ell\text{PCF types} & \sigma, \tau & ::= & Nat[I, J] \mid A \multimap \sigma \\ & A & ::= & [a < I]\sigma \end{array}$$

Fig. 12. $d\ell$PCF's syntax of terms and types from [Dal Lago and Gaboardi 2011]

$$\frac{\Theta; \Delta \models J \geq 0 \qquad \Theta; \Delta \models I \geq 1 \qquad \Theta; \Delta \vdash \sigma[0/a] <: \tau}{\Theta; \Delta; \Gamma, x : [a < I]\sigma \vdash_J x : \tau}\ \text{var} \qquad \frac{\Theta; \Delta; \Gamma, x : [a < I]\tau_1 \vdash_J e : \tau_2}{\Theta; \Delta; \Gamma \vdash_J \lambda x.e : ([a < I].\tau_1) \multimap \tau_2}\ \text{lam}$$

$$\frac{\Theta; \Delta; \Gamma \vdash_J e_1 : ([a < I]\tau_1) \multimap \tau_2 \qquad \Theta, a; \Delta, a < I; \Delta \vdash_K e_2 : \tau_1 \qquad \Gamma' \sqsupseteq \Gamma \oplus \sum_{a<I} \Delta \qquad H \geq I + J + \sum_{a<I} K}{\Theta; \Delta; \Gamma' \vdash_H e_1\ e_2 : \tau_2}\ \text{app}$$

Fig. 13. Selected typing rules of $d\ell$PCF from [Dal Lago and Gaboardi 2011]

The objective of this embedding is twofold: It shows that (a) $\lambda$-amor can simulate *coeffect*-based cost analysis like $d\ell$PCF's and (b) $\lambda$-amor is also relatively complete for PCF.

### 6.1 A brief primer on $d\ell$PCF

$d\ell$PCF [Dal Lago and Gaboardi 2011] is a call-by-name PCF with an affine, refinement type system for cost analysis.

*Syntax.* Terms and types of $d\ell$PCF are described in Fig. 12. $d\ell$PCF uses standard PCF terms but refines the standard types of PCF for cost analysis. The type of natural numbers is refined with two indices, yielding $Nat[I, J]$, the type of natural numbers in the range $[I, J]$ both inclusive. The function type has the form $[a < I]\tau_1 \multimap \tau_2$ or, more precisely, $([a < I]\tau_1) \multimap \tau_2$. Here, $[a < I]\tau_1$ is just $d\ell$PCF's notation for the dependent sub-exponential $!_{a<I}\tau$ that we introduced in Section 5. Hence, $[a < I]\tau_1 \approx \tau_1[0/a] \otimes \ldots \otimes \tau_1[(I-1)/a]$ and the function type $[a < I]\tau_1 \multimap \tau_2$ is morally equivalent to $(\tau_1[0/a] \otimes \ldots \otimes \tau_1[(I-1)/a]) \multimap \tau_2$. This way, the function type explicitly states the number of times $I$ the argument can be used by the function. This is important for cost analysis in the type system (explained below). Note that, in $d\ell$PCF, the sub-exponential can only appear to the left of an arrow, and in hypothesis.

*Type system.* The typing judgment of $d\ell$PCF is $\Theta; \Delta; \Gamma \vdash_C e_d : \tau$. Here, $\Theta$ is a context of index variables (as in $\lambda$-amor), $\Delta$ is a set of assumed constraints on index variables (as in $\lambda$-amor), $\Gamma$ is a context of term variables, and $C$ is an upper bound on the cost of evaluation of $e_d$. The important parts here are $\Gamma$ and $C$. $\Gamma$ is actually analogous to the non-affine context $\Omega$ of $\lambda$-amor: It contains variables with multiplicities and dependencies. An entry in $\Omega$ has the form $x : [a < I]\tau$, which is exactly the same as an entry $x :_{a<I} \tau$ in $\lambda$-amor and means that $I$ copies of $x$ are available with types $\tau[0/a], \ldots, \tau[(I-1)/a]$. The cost $C$ comes from the language of index terms. Note that in $d\ell$PCF, costs exist only on typing derivations; they are never internalized into types.

We show the important typing rules of $d\ell$PCF in Fig. 13. While reading these rules, the reader should keep $d\ell$PCF's cost model in mind: A *unit* cost is counted for every *use* of a variable, which corresponds to a unit cost for every variable lookup during execution on a Krivine stack machine for PCF [Krivine 2007]. Rule (var) allows the use of a hypothesis from $\Gamma$. The variable's multiplicity $I$ must be at least 1. Perhaps surprisingly, the cost of this rule $J$ can be 0, even though this rule corresponds to the use of a variable. The reason is that the cost for all the uses of a variable is counted together when the variable is bound (i.e., during function application), as we explain later.

The rule for function abstraction (lam) is perhaps the most interesting and the defining rule of $d\ell$PCF's cost tracking. Here, the cost $J$ of $\lambda x.e$ is taken to be the same as the cost of the body $e$. This may sound overly conservative: $\lambda x.e$ is a value and it evaluates with zero cost, so why count

the cost of the body $e$ in its cost? The reason is that there is no other place to put the cost of the body. If this were an effect system (which it is not), the body's cost would be internalized into the function's *type*, and be counted when the function is applied. However, since this is a coeffect system, we don't have this luxury, so the cost of the body is *pre-counted* when the abstraction is created. This is sound because the type system is affine: We cannot apply the abstraction more than once, so the amount we pre-count here is certainly an upper-bound on the cost that could arise from this function's use in the future. It is this unusual way of counting costs that makes $d\ell$PCF substantially different from effect-based systems, and also why potentials are needed for embedding into $\lambda$-amor (this is explained in Section 6.2).

The (app) rule for function application is also interesting. Here, $e_1$ is a term of type $([a < I]\tau_1) \multimap \tau_2$, so the argument $e_2$ must be typed parametrically in $a$ for all $a < I$. The total cost $H$ includes the cost $J$ of evaluating $e_1$, the cost evaluating $e_2$ $I$-times (written $\sum_{a<I} K$), and, importantly, an additional cost $I$ for the up-to $I$ uses of the bound variable in the body of the function after application. This is where the cost for variable use is actually counted. Note how information about the multiplicity $I$ of a variable – the coeffect – contributes directly to the cost analysis. This justifies why $d\ell$PCF can be classified as a coeffect-based cost tracking system.

*Operational semantics.* $d\ell$PCF uses a Krivine machine for PCF, called $K_{PCF}$ [Krivine 2007]. States of $K_{PCF}$ are triples of the form $(t, \rho, \theta)$ where $t$ is a $d\ell$PCF term, $\rho$ is an environment with variable bindings (it maps variables to terms) and $\theta$ is stack of closures. A closure (denoted by C) is simply a pair consisting of a term and an environment. The left side of Fig. 14 lists some evaluation rules of $K_{PCF}$ from [Dal Lago and Gaboardi 2011]. For instance, the application triple $(e_1\ e_2, \rho, \theta)$ reduces in one step to $e_1$; the argument $e_2$, along with the current environment, is pushed to the top of the stack for later evaluation. This is exactly how one would expect an evaluation to happen in a call-by-name scheme.

$$
\begin{array}{lcl}
(e_1\ e_2, \rho, \theta) & \to & (e_1, \rho, (e_2, \rho).\theta) \\
(\lambda x.e, \rho, \mathsf{C}.\theta) & \to & (e_1, \mathsf{C}.\rho, \theta) \\
(x, (t_0, \rho_0) \ldots (t_n, \rho_n), \theta) & \to & (t_x, \rho_x, \theta) \\
(\mathrm{fix} x.e, \rho, \theta) & \to & (e, (x, (\mathrm{fix} x.e, \mathsf{C}).\rho, \theta)
\end{array}
\qquad
\begin{array}{lcl}
|x| & = & 1 \\
|c| & = & 1 \\
|\lambda x.e| & = & |e| + 1 \\
|e_1\ e_2| & = & |e_1| + |e_2| + 1 \\
|\mathrm{fix} x.e| & = & |e| + 1
\end{array}
$$

Fig. 14. $K_{PCF}$ reduction rules (left) and size function (right) from [Dal Lago and Gaboardi 2011]

*Soundness.* $d\ell$PCF's cost analysis is sound for reductions on the Krivine machine, up to constant factors. This is formalized in Theorem 10, which says that if the $d\ell$PCF type system provides a cost bound $I$ on term $t$ (of type $Nat$) and $t$ reduces for $n$ steps on the Krivine machine, then $n \leq |t| * (I + 1)$, where $|t|$ is the size of the term $t$, defined on the right side of Fig. 14. The factor $|t|$ arises because $d\ell$PCF counts only variable uses, while the Krivine machine has other reductions as well. However, one variable lookup is forced every $|t|$ reductions. (In the statement of the theorem, $\Downarrow^n$ is just shorthand for $n$-step Krivine reduction starting from the initial state $(t, \epsilon, \epsilon)$.)

**Theorem 10** ($d\ell$PCF's soundness from [Dal Lago and Gaboardi 2011]). $\forall t, I, J, K.$
$\vdash_I t : Nat[J, K] \wedge t \Downarrow^n m \implies n \leq |t| * (I + 1)$

### 6.2 Type-directed translation of $d\ell$PCF into $\lambda$-amor

We now describe our embedding of $d\ell$PCF in $\lambda$-amor. The translation of types, denoted $(\!|\bullet|\!)$, is shown in the left side of Fig. 15. We abstract the type of naturals and treat them as a general abstract base type b. In the translation, the annotation $[a < I]$ simply changes to $!_{a<I}$ as the two have the same meaning. Further, following Moggi's embedding of call-by-name in the computational $\lambda$-calculus [Moggi 1991], the basic skeleton of the translation of a function type $A \multimap B$ is $(\mathbb{M}\,0\,(\!|A|\!)) \multimap (\mathbb{M}\,0\,(\!|B|\!))$. The actual translation of the function type, shown in Fig. 15, additionally

requires as argument a potential of $I$ units (type $[I]\,\mathbf{1}$), where $I$ is the multiplicity of the argument. This pays for the cost of using the argument up to $I$ times in the body of the function and corresponds to the additional $I$ cost added to the total in the application rule (app) of $d\ell$PCF (Section 6.1).

$$\begin{array}{lcl|lcl} (\!|b|\!) & = & b & (\!|.|\!) & = & . \\ (\!|[a<I]\tau_1 \multimap \tau_2|\!) & = & !_{a<I}\,\mathbb{M}\,0\,(\!|\tau_1|\!) \multimap [I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!) & (\!|\Gamma, x : [a<I]\tau|\!) & = & (\!|\Gamma|\!), x :_{a<I} \mathbb{M}\,0\,(\!|\tau|\!) \end{array}$$

Fig. 15. Type and context translation for $d\ell$PCF

The translation of $d\ell$PCF contexts, shown on the right side of Fig. 15, maps a $d\ell$PCF context to a non-affine context ($\Omega$) in $\lambda$-amor. Again, a variable of type $\tau$ is mapped to variable of type $\mathbb{M}\,0\,(\!|\tau|\!)$ with the same multiplicity.

*Term translation.* To explain the translation of terms, we need an auxiliary function on $d\ell$PCF contexts, which we write $count(\Gamma)$. This function simply adds the multiplicities of all variables in $\Gamma$.

$$count(.) = 0 \qquad\qquad count(\Gamma, x : [a < I]\tau) = count(\Gamma) + I$$

The translation of $d\ell$PCF terms is type-derivation directed. The translation judgment is of the form $\Theta; \Delta; \Gamma \vdash_I e_d : \tau \rightsquigarrow e_a$ where $\Theta; \Delta; \Gamma \vdash_I e_d : \tau$ is a valid $d\ell$PCF typing judgment and $e_a$ denotes the $\lambda$-amor translation of the $d\ell$PCF expression $e_d$. Importantly, the expected type of $e_a$ is $[I + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$. This type means that $e_a$ yields something of type $\mathbb{M}\,0\,(\!|\tau|\!)$ (the expected type) after it has been provided (applied to) enough potential. Here, "enough" is $I + count(\Gamma) - I$ pays for the cost of evaluation of $e_a$, as manifest in the given $d\ell$PCF typing judgment, while $count(\Gamma)$ pays for the cost of using each variable from the context its multiplicity number of times.

With this basic structure of the translation in mind, we present the important term translation rules in Fig. 16. The translation of a variable $x$ of type $\tau$, rule (var), is a term of type $[J + count(\Gamma) + I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$. This term releases the potential $J + count(\Gamma) + I$ it receives as argument, *then incurs a unit cost* ($\uparrow^1$), and then returns the variable $x$. The unit cost is needed to faithfully represent $d\ell$PCF's cost model, which counts a unit cost for every use of a variable. Note that the net remaining cost is 0 because $I \geq 1$ (premise of the rule), so the released potential $J + count(\Gamma) + I$ is at least as much as the incurred unit cost.

The rule for translating functions, rule (lam), handles $d\ell$PCF's unusual way of counting costs of functions. Here, the $d\ell$PCF context in the premise is $\Gamma, x : [a < I]\tau_1$, so the type of $e_t$ is $[J + count(\Gamma) + I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!)$ in a context that includes $x : !_{a<I}\,\mathbb{M}\,0\,(\!|\tau_1|\!)$. We wish to construct a term of type $[J + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(!_{a<I}\,\mathbb{M}\,0\,(\!|\tau_1|\!) \multimap [I]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau_2|\!))$. Observe how the types of $e_t$ and the term we want are very similar: They both have the same total potential $J + count(\Gamma) + I$ in negative positions. As a result, the required term can be constructed quite easily. The exact term is shown in the conclusion of rule (lam). The rule for function application, rule (app), can be understood similarly.

**Theorem 11** (Type preservation: $d\ell$PCF to $\lambda$-amor)**.** *If $\Theta; \Delta; \Gamma \vdash_I e : \tau$ in $d\ell$PCF then there exists $e'$ such that $\Theta; \Delta; \Gamma \vdash_I e : \tau \rightsquigarrow e'$ and there is a derivation of $\bullet; \Theta; \Delta; (\!|\Gamma|\!); \bullet \vdash e' : [I + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$ in $\lambda$-amor.*

*Remark about the importance of potentials.* The astute reader has probably noticed that, in this translation, we represent costs using potentials in negative positions, not monad indices (the monad index is 0 everywhere in the translation!). Could we have used the monad index instead, and not needed potentials at all? The answer is no: Trying to embed $d\ell$PCF using the monad alone breaks the well-typedness of the translation of functions in the (lam) rule of Fig. 16. In the following, we explain this further.

Suppose we were to do away with potentials. Then, in the term translation, a $d\ell$PCF term $e_d$ typed as $\Theta; \Delta; \Gamma \vdash_I e_d : \tau$ would no longer translate to a term of type $[I + count(\Gamma)]\,\mathbf{1} \multimap \mathbb{M}\,0\,(\!|\tau|\!)$, but

$$\frac{\Theta;\Delta \models J \geq 0 \qquad \Theta;\Delta \models I \geq 1 \qquad \Theta;\Delta \vdash \sigma[0/a] <: \tau}{\Theta;\Delta;\Gamma, x : [a < I]\sigma \vdash_J x : \tau \rightsquigarrow \lambda p.\mathsf{release} - = p \text{ in } \mathsf{bind} - = \uparrow^1 \text{ in } x}\ \text{var}$$

$$\frac{\Theta;\Delta;\Gamma, x : [a < I]\tau_1 \vdash_J e : \tau_2 \rightsquigarrow e_t}{\Theta;\Delta;\Gamma \vdash_J \lambda x.e : ([a < I].\tau_1) \multimap \tau_2 \rightsquigarrow \lambda p_1.\, \mathsf{ret}\, \lambda y.\lambda p_2.\, \mathsf{let!}\, x = y \text{ in } \mathsf{release} - = p_1 \text{ in } \mathsf{release} - = p_2 \text{ in } \mathsf{bind}\, a = \mathsf{store}() \text{ in } e_t\ a}\ \text{lam}$$

$$\frac{\Theta;\Delta;\Gamma \vdash_J e_1 : ([a < I]\tau_1) \multimap \tau_2 \rightsquigarrow e_{t1} \qquad \Theta, a;\Delta, a < I;\Delta \vdash_K e_2 : \tau_1 \rightsquigarrow e_{t2} \qquad \Gamma' \sqsupseteq \Gamma \oplus \textstyle\sum_{a<I} \Delta \qquad H \geq J + I + \sum_{a<I} K}{\Theta;\Delta;\Gamma' \vdash_H e_1\ e_2 : \tau_2 \rightsquigarrow \lambda p.E_0}\ \text{app}$$

$$E_0 \triangleq \mathsf{release} - = p \text{ in } E_1,\ E_1 \triangleq \mathsf{bind}\, a = \mathsf{store}() \text{ in } E_2$$
$$E_2 \triangleq \mathsf{bind}\, b = e_{t1}\ a \text{ in } E_3,\ E_3 \triangleq \mathsf{bind}\, c = \mathsf{store!}() \text{ in } E_4$$
$$E_4 \triangleq \mathsf{bind}\, d = \mathsf{store}() \text{ in } E_5,\ E_5 \triangleq b\ (\mathit{coerce}\ !e_{t2}\ c)\ d$$

$$\boxed{\begin{array}{l}\mathit{coerce} : !_{a<I}(\tau_1 \multimap \tau_2) \multimap !_{a<I}\tau_1 \multimap !_{a<I}\tau_2 \\ \mathit{coerce}\ F\ X \triangleq \mathsf{let!}\, f = F \text{ in } \mathsf{let!}\, x = X \text{ in } !(f\ x)\end{array}}$$

Fig. 16. Expression translation: $d\ell$PCF to $\lambda$-amor

would instead translate to a term of the "equivalent" monadic type $\mathbb{M}(I + \mathit{count}(\Gamma))\ (\!|\tau|\!)$. Further, the $d\ell$PCF function type $[a < I]\tau_1 \multimap \tau_2$ would no longer translate to $!_{a<I}\, \mathbb{M}\, 0 (\!|\tau_1|\!) \multimap [I]\, \mathbf{1} \multimap \mathbb{M}\, 0\, (\!|\tau_2|\!)$, but would instead translate to $!_{a<I}\, \mathbb{M}\, 0 (\!|\tau_1|\!) \multimap \mathbb{M}\, I\, (\!|\tau_2|\!)$. Further, in the (lam) case, from the premise we would have a term $e_t$ of type $\mathbb{M}(J + \mathit{count}(\Gamma) + I)\ (\!|\tau_2|\!)$ in a context that includes $x : !_{a<I}\, \mathbb{M}\, 0 (\!|\tau_1|\!)$. We would want to construct a term of type $\mathbb{M}(J + \mathit{count}(\Gamma))\ \big(!_{a<I}\, \mathbb{M}\, 0 (\!|\tau_1|\!) \multimap \mathbb{M}\, I\, (\!|\tau_2|\!)\big)$. However, such a term does not exist in $\lambda$-amor or any calculus with indexed/graded monads that we know of. To understand this, let's abstract $J + \mathit{count}(\Gamma)$ to $K$, $!_{a<I}\, \mathbb{M}\, 0 (\!|\tau_1|\!)$ to $\sigma$, and $(\!|\tau_2|\!)$ to $\tau$. Then, we are asking that the type $(\sigma \multimap \mathbb{M}(K + I)\ \tau) \multimap \mathbb{M}\, K\ (\sigma \multimap \mathbb{M}\, I\, \tau)$ be inhabited. It should be easy to see that this goes well beyond standard properties of graded monads. Adding a *primitive term* of this type would be sound but dissatisfying.

To summarize, it seems that potentials are somewhat fundamental to embedding a coeffect-based cost analysis (as in $d\ell$PCF) in a monadic or effect-based system like $\lambda$-amor. Whether and how this generalizes to quantitative properties beyond cost remains an open question.

*Semantic properties.* We proved above that our translation is type preserving. In addition, we prove that it preserves reduction costs up to constant factors. This requires us to relate reduction of a $d\ell$PCF term $e$ in the Krivine machine to the reduction of its translation in $\lambda$-amor. This is technically tedious and, due to lack of space here, we cover it in full detail only in our technical appendix. Briefly, we define a type-preserving "decompilation" of Krivine states to $d\ell$PCF terms. By composing this with the translation of $d\ell$PCF terms to $\lambda$-amor, we get a translation from Krivine states to $\lambda$-amor. We then show the following result about this translation (formalized as Lemma 52 in the appendix): A Krivine reduction of $k$ steps can be simulated by its $\lambda$-amor translation in $i$ steps where $k \leq (i + 1)|t|$ and $|t|$ is the size of the initial Krivine state. This result immediately allows us to re-derive $d\ell$PCF's soundness theorem from that of $\lambda$-amor.

*Completeness.* Finally, we note that Dal Lago and Gaboardi [2011] proved that $d\ell$PCF is relatively complete for PCF programs: Every simply typed, closed PCF program of type $Nat$ that reduces in $k$ steps to a value can be typed with cost $k$ in $d\ell$PCF. Since our translation from $d\ell$PCF to $\lambda$-amor trivially preserves cost annotations for closed terms, this means that $\lambda$-amor can also type (the translation of) a PCF term that reduces in $k$ steps with cost $k$. Hence, $\lambda$-amor is also relatively complete for PCF.

## 7 RELATED WORK

The literature on cost analysis is very vast; we summarize and compare to only a representative subset of the literature, covering several prominent *styles* of cost analysis.

*Type and effect systems.* Several type and effect system have been proposed for amortized analysis using the method of potentials. Early approaches [Hofmann and Jost 2003; Jost et al. 2009] allow the potential associated with a value to only be a linear function of the value's size. Univariate RAML [Hoffmann and Hofmann 2010] generalizes this to polynomial potentials. Multivariate RAML [Hoffmann et al. 2011] is a further generalization where a single potential, that is a polynomial of the sizes of several input variables, can be associated to all of them together. These approaches work with an assumption that functions are freely duplicable and not under the purview of affineness. As a result, potential captured in partially applied functions renders these approaches unsound. This is prevented by simply *disallowing* closures with captured potential in the type system. In contrast, $\lambda$-amor can effortlessly handle closures that capture potential since the type system is fully affine. We already showed how to embed Univariate RAML in $\lambda$-amor in Section 4. We believe that the embedding can be extended to Multivariate RAML with some effort.

Later work tries to address some of the limitations of RAML. For instance, [Jost et al. 2010] extends RAML with *limited* support for closures and higher-order functions. In particular, [Jost et al. 2010] can handle curried functions only when the potential is associated with the last argument. More recently, the type system of Knoth et al. [2019] works by capturing the number of times a partially applied function can be used, thereby supporting closures which have enough potential for the allowed number of copies. All of these approaches are somewhat ad hoc in nature and do not fix the root cause in the type theory, which is the lack of affineness. In contrast, $\lambda$-amor, being fully affine, does not have such limitations.

Some prior work such as the unary fragment of Çiçek et al. [2017] uses effect-based type systems for *non-amortized* cost analysis. Handley et al. [2020] show how to perform (non-amortized) cost analysis using refinement types of Liquid Haskell. A significant line of work tracing lineage back to at least Crary and Weirich [2000] uses sized types and cost represented in a writer monad for cost analysis. More recently, Danner et al. [2015] show how to extend this idea to extract sound cost recurrences from programs. These recurrences can be solved to establish cost bounds. However, none of this work supports potentials or amortized analysis. Conceptually, it is simpler than the previously-mentioned work on amortized cost analysis (it corresponds to RAML functions where the input and output potentials are both 0). We believe that all systems mentioned in this paragraph so far can be embedded in $\lambda$-amor. Kavvos et al. [2020] extend the work of Danner et al. [2015] to the call-by-push-value (CBPV) setting. While we have not considered CBPV here, we believe that changing $\lambda$-amor's semantics from monadic to CBPV is feasible and not particularly difficult.

*Cost analysis using program logics.* As an alternative to type systems, a growing line of work uses variants of Hoare logic for amortized cost analysis [Carbonneaux et al. 2015; Charguéraud and Pottier 2019; Mével et al. 2019]. The common idea is to represent the potential before and after the execution of a code segment as ghost state in the pre- and post-condition of the segment, respectively. Conceptually, this idea is not very different from how we encode potentials using our $[p]\,\tau$ construct in the inputs and outputs of functions (e.g., in embedding RAML in Section 4). However, unlike $\lambda$-amor, prior work on program logics for cost analysis shows neither embeddings of existing frameworks, nor any (relative) completeness result. Haslbeck and Nipkow [2018] present completeness results for some Hoare logics but only in the context of a simple imperative language. The real challenge for completeness is handling of higher-order functions with arbitrary recursion, which $\lambda$-amor covers. [Mével et al. 2019] introduce a new concept called *time receipts*, which are useful for lower-bound analysis, something that we are extending $\lambda$-amor to.

*Cost analysis of lazy programs.* Some prior work [Danielsson 2008; Jost et al. 2017; Madhavan et al. 2017] develops methods for cost analysis of lazy programs (call-by-need evaluation). It turns out that affineness is not required for amortized cost analysis of lazy programs, as an expression is evaluated only once due to memoization even if it is used several times. As mentioned in Section 1, the semantics of call-by-need is fundamentally incompatible with the semantics of $\lambda$-amor, so we did not consider call-by-need in this paper. However, at the level of types, there is considerable similarity between Danielsson [2008] and our work. For example, Danielsson [2008] uses an indexed monad to track costs as we do. Interestingly, he does not have a separate type-theoretic construct to associate potential. Instead, he introduces a primitive coercion "pay" of type $\mathbb{M}(\kappa_1 + \kappa_2)\,\tau \multimap \mathbb{M}\,\kappa_1\,(\mathbb{M}\,\kappa_2\,\tau)$, which, in a way, represents paying $\kappa_1$ part of the cost $\kappa_1 + \kappa_2$ using potential from the outside. An interesting question is whether, assuming a coercion of this type, we could do away with potentials altogether and still embed $d\ell$PCF. It seems that the answer is no. As we noted at the end of Section 6.2, the axiom about indexed monads we really need in order to embed $d\ell$PCF without potentials is $(\sigma \multimap \mathbb{M}(\kappa_1 + \kappa_2)\,\tau) \multimap \mathbb{M}\,\kappa_1\,(\sigma \multimap \mathbb{M}\,\kappa_2\,\tau)$, and this is stronger than "pay".

*Coeffect-based cost analysis.* $d\ell$PCF [Dal Lago and Gaboardi 2011] and $d\ell\text{PCF}_\text{V}$ [Dal Lago and Petit 2012] are coeffect-based type systems for non-amortized cost analysis of PCF programs in the call-by-name and call-by-value settings, respectively. Both systems count the number of variable lookups during execution on an abstract machine (the Krivine machine for call-by-name and the CEK machine for call-by-value [Felleisen and Friedman 1987; Krivine 2007]). This is easily done by tracking (as a coeffect) the number of uses of each variable in an affine type system with a dependent sub-exponential borrowed from Bounded Linear Logic (BLL) [Girard et al. 1992] ($\lambda$-amor also borrows the same dependent sub-exponential, but does not use coeffects for tracking cost). A common limitation of $d\ell$PCF and $d\ell\text{PCF}_\text{V}$ is that they cannot internalize the cost of a program into its type; instead the cost is a function of the typing derivation. We showed in Section 6 that $\lambda$-amor can embed $d\ell$PCF and internalize its costs into types. Hence, $\lambda$-amor advances beyond $d\ell$PCF. We expect that $\lambda$-amor can also embed $d\ell\text{PCF}_\text{V}$, but have not tried this embedding yet.

Atkey [2018] presents QTT, a quantitative dependent type theory with coeffects. QTT and $\lambda$-amor are very different in their goals. QTT focuses on the interaction between dependent types and coeffects whereas $\lambda$-amor is based on effects (but we show how to simulate coeffects in $\lambda$-amor). Technically, QTT only considers non-dependent coeffects, as in $x :_n \tau$ ($n$ copies of the same $\tau$) while $\lambda$-amor includes coeffects with parameterized linear dependencies from the dependent sub-exponential, as in $x :_{a<n} \tau$ ($n$ different instances of $\tau$).

## 8 CONCLUSION

We developed $\lambda$-amor with the broader goal of unifying existing type systems for cost analysis. $\lambda$-amor introduces a new modal type constructor to represent potential at the level of types and uses affine types with a dependent sub-exponential. Through two embeddings, we have shown that $\lambda$-amor can simulate cost analysis for different evaluation strategies (call-by-name and call-by-value), in different styles (effect-based and coeffect-based), and with or without amortization.

*Acknowledgments.* We thank our anonymous reviewers and our shepherd David Van Horn for their feedback. Vineet Rajani conducted most of this research while he was a graduate student at the Max Planck Institute for Software Systems and Saarland University. Marco Gaboardi was supported by the National Science Foundation under awards 1718220 and 1845803. Jan Hoffmann was supported by DARPA under AA Contract FA8750-18-C-0092 and by the National Science Foundation under SaTC Award 1801369, CAREER Award 1845514, and SHF Awards 1812876 and 2007784. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

## REFERENCES


Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Ph.D. Dissertation. Princeton university.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Annual ACM/IEEE Symposium on Logic in Computer Science*.

Martin Avanzini and Ugo Dal Lago. 2017. Automating Sized-type Inference for Complexity Analysis. *Proc. ACM Program. Lang.* 1, ICFP (2017).

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proceedings of the European Symposium on Programming Languages and Systems*.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reasoning* 62, 3 (2019).

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science* 8, 4 (2011).

Ugo Dal Lago and Barbara Petit. 2012. Linear Dependent Types in a Call-by-value Scenario. *Science of Computer Programming* 84 (2012).

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Norman Danner, Daniel R. Licata, and Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. 140–151.

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the $\lambda$-calculus. In *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992).

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL (2020).

Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds - A Study in Meta Theory. In *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 10805.

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

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*.

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential: A Static Inference of Polynomial Bounds for Functional Programs. In *Proceedings of the European Conference on Programming Languages and Systems*.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *Proceedings of Formal Methods*.

Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *J. Autom. Reason.* 59, 1 (2017).

G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2020. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.* 4, POPL (2020).

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Jean-Louis Krivine. 2007. A Call-by-name Lambda-calculus Machine. *Higher Order Symbolic Computation* 20, 3 (2007).

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based Resource Verification for Higher-order Functions with Memoization. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*.

Dylan McDermott and Alan Mycroft. 2018. Call-by-need effects via coeffects. *Open Comput. Sci.* 8, 1 (2018).

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming*.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (1991).

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *J. Funct. Program.* 21, 4-5 (2011).

Chris Okasaki. 1996. *Purely Functional Data Structures*. Ph.D. Dissertation. Carnegie Mellon University.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-dependent Computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages, and Programming - International Colloquium*.

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1 (2004).

Robert E. Tarjan. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985).

Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (2007).