# Denotational Recurrence Extraction for Amortized Analysis

JOSEPH W. CUTLER, Wesleyan University, USA
DANIEL R. LICATA, Wesleyan University, USA
NORMAN DANNER, Wesleyan University, USA

A typical way of analyzing the time complexity of functional programs is to extract a recurrence expressing the running time of the program in terms of the size of its input, and then to solve the recurrence to obtain a big-O bound. For recurrence extraction to be compositional, it is also necessary to extract recurrences for the size of outputs of helper functions. Previous work has developed techniques for using logical relations to state a formal correctness theorem for a general recurrence extraction translation: a program is bounded by a recurrence when the operational cost is bounded by the extracted cost, and the output value is bounded, according to a value bounding relation defined by induction on types, by the extracted size. This previous work supports higher-order functions by viewing recurrences as programs in a lambda-calculus, or as mathematical entities in a denotational semantics thereof. In this paper, we extend these techniques to support amortized analysis, where costs are rearranged from one portion of a program to another to achieve more precise bounds. We give an intermediate language in which programs can be annotated according to the banker's method of amortized analysis; this language has an affine type system to ensure credits are not spent more than once. We give a recurrence extraction translation of this language into a recurrence language, a simply-typed lambda-calculus with a cost type, and state and prove a bounding logical relation expressing the correctness of this translation. The recurrence language has a denotational semantics in preorders, and we use this semantics to solve recurrences, e.g analyzing binary counters and splay trees.

CCS Concepts: • **Software and its engineering** → *Functional languages*; • **Theory of computation** → **Program analysis**; **Program verification**.

Additional Key Words and Phrases: recurrence extraction, resource analysis, amortized analysis, cost semantics, higher order recurrences, denotational semantics

**ACM Reference Format:**
Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational Recurrence Extraction for Amortized Analysis. *Proc. ACM Program. Lang.* 4, ICFP, Article 97 (August 2020), 29 pages. https://doi.org/10.1145/3408979

## 1 INTRODUCTION

A common technique for analyzing the asymptotic resource complexity of functional programs is the *extract-and-solve* method, in which one extracts a recurrence expressing an upper bound on the cost of the program in terms of the size of its input, and then solves the recurrence to obtain a big-$O$ bound. Typically, the connection between the original program and the extracted recurrence is left informal, relying on an intuitive understanding that the extracted recurrence correctly models the program. Previous work [Danner and Licata 2020; Danner et al. 2015, 2013; Hudson, Bowornmet 2016; Kavvos et al. 2019] has begun to explore more formal techniques for

Authors' addresses: Joseph W. Cutler, Wesleyan University, Middletown, Connecticut, USA, jwcutler@wesleyan.edu; Daniel R. Licata, Wesleyan University, Middletown, Connecticut, USA, dlicata@wesleyan.edu; Norman Danner, Wesleyan University, Middletown, Connecticut, USA, ndanner@wesleyan.edu.

2475-1421/2020/8-ART97
https://doi.org/10.1145/3408979

relating programs and extracted recurrences. The process of extracting a recurrence consists of two phases. The first is a monadic translation into the writer monad $\mathbb{C} \times \cdot$, translating a program to also "output" its cost along with its value. We call the result a *syntactic recurrence*, and at function type, the result is essentially a function that maps a value to a pair consisting of the cost of evaluating that function along with its result. At higher type, the syntactic recurrence maps a recurrence for the argument to a recurrence for the result. A *bounding logical relation* relates programs to syntactic recurrences, and the fundamental *bounding theorem* states that a program and its syntactic recurrence are related, which in particular implies that its actual runtime cost is bounded by the extracted prediction. Since inductive values are translated to (essentially) themselves, this phase does not abstract values to sizes; in effect, the syntactic recurrence describes the cost of the program in terms of its actual arguments. The second phase performs this size abstraction by interpreting (the language of) syntactic recurrences in a denotational model. The interpretation of each type is intended to be a domain of sizes for values of that type, and different models can implement different notions of size. For example, a list value (i.e., the list type and constructors) may be interpreted by its length in one model, or even more exotic notions of size, such as the number of pairwise inversions (as required for an analysis of insertion sort) for a list of numbers. Thus the interpretation of the syntactic recurrence extracted from a source program (what we might call the *semantic recurrence*) is a function that maps sizes (of source-program values) to a bound on the cost of that program on those values. It is these semantic recurrences that match the recurrences that arise from the typical "extract-and-solve" approach to analyzing program cost. Our previous work develops this methodology for functional programs with numbers and lists [Danner et al. 2013], inductive types with structural recursion [Danner et al. 2015], general recursion [Kavvos et al. 2019], and let-polymorphism [Danner and Licata 2020].

As an example that demonstrates both the approach and a weakness of the underlying technique for cost analysis that it formalizes, let us consider the binary increment function, a standard motivating example for amortized analysis:

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{inc} & : & \mathtt{bit\ list} \to \mathtt{bit\ list} & \mathtt{set} & : & \mathtt{nat} \to \mathtt{bit\ list} \\
\mathtt{inc}\,[\,] & = & [1] & \mathtt{set}\,0 & = & [\,] \\
\mathtt{inc}\,(0 :: bs) & = & 1 :: bs & \mathtt{set}\,(S\,n) & = & \mathtt{inc}(\mathtt{set}\,n) \\
\mathtt{inc}\,(1 :: bs) & = & 0 :: \mathtt{inc}\,bs & & &
\end{array}$$

The value part of a monadic translation of a function into $\mathbb{C} \times \cdot$ is a function into a pair, but here we sugar that into a pair of functions, which may be mutually recursive. We denote the cost and value components by $(\cdot)_c$ and $(\cdot)_p$, respectively (this notation is explained in Section 3.1), and charge one unit of cost for each :: operation:

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{inc}_c & : & \mathtt{bit\ list} \to \mathbb{C} & \mathtt{inc}_p & : & \mathtt{bit\ list} \to \mathtt{bit\ list} \\
\mathtt{inc}_c\,[\,] & = & 1 & \mathtt{inc}_p\,[\,] & = & [1] \\
\mathtt{inc}_c\,(0 :: bs) & = & 1 & \mathtt{inc}_p\,(0 :: bs) & = & 1 :: bs \\
\mathtt{inc}_c\,(1 :: bs) & = & 1 + \mathtt{inc}_c\,bs & \mathtt{inc}_p\,(1 :: bs) & = & 0 :: \mathtt{inc}_p\,bs \\
\\
\mathtt{set}_c & : & \mathtt{nat} \to \mathbb{C} & \mathtt{set}_p & : & \mathtt{nat} \to \mathtt{bit\ list} \\
\mathtt{set}_c\,0 & = & 0 & \mathtt{set}_p\,0 & = & [\,] \\
\mathtt{set}_c\,(S\,n) & = & \mathtt{set}_c(n) + \mathtt{inc}_c(\mathtt{set}_p\,n) & \mathtt{set}_p\,(S\,n) & = & \mathtt{inc}_p(\mathtt{set}_p\,n)
\end{array}$$

We obtain the usual recurrences that we expect when we interpret these syntactic recurrences in an appropriate denotational semantics. We interpret `bit list` and `nat` by $\mathbb{N}$, the natural numbers, and interpret the constructors so that a `bit list` is interpreted by its length and a `nat` by its value.

Doing so, we obtain semantic recurrences for the the cost and size of inc:

$$\begin{aligned} T_{\mathtt{inc}}(0) &= 1 & S_{\mathtt{inc}}(0) &= 1 \\ T_{\mathtt{inc}}(n+1) &= \max\{1, 1 + T_{\mathtt{inc}}(n)\} & S_{\mathtt{inc}}(n+1) &= \max\{1+n, 1 + S_{\mathtt{inc}}(n)\} \end{aligned}$$

The usual techniques (in the semantics) then allow us to conclude that $T_{\mathtt{inc}}(n) \le n+1$ and $S_{\mathtt{inc}}(n) \le n+1$, which are correct and tight bounds on the cost and size of the inc function. The semantic recurrences for set are

$$\begin{aligned} T_{\mathtt{set}}(0) &= 0 & S_{\mathtt{set}}(0) &= 0 \\ T_{\mathtt{set}}(n+1) &= T_{\mathtt{set}}(n) + T_{\mathtt{inc}}(S_{\mathtt{set}}(n)) & S_{\mathtt{set}}(n+1) &= S_{\mathtt{inc}}(S_{\mathtt{set}}(n)) \\ &\le T_{\mathtt{set}}(n) + S_{\mathtt{set}}(n) + 1 & &\le S_{\mathtt{set}}(n) + 1 \end{aligned}$$

and so we conclude that $S_{\mathtt{set}}(n) \le n$ and hence $T_{\mathtt{set}}(n) \in O(n^2)$, both of which are correct, but not tight, bounds.

On the one hand, through syntactic recurrence extraction, the bounding theorem, and soundness of the semantics, we have a formal connection between the original programs and the semantic recurrences that bound their cost and size. On the other, this example demonstrates a well-understood weakness in the informal technique: while the cost of a composition of functions is bounded by the composition of their costs, the bound is not necessarily tight. The tight bound is usually established with some form of amortized analysis, and *the goal of this paper is to provide a formalization of the banker's method for amortized analysis comparable to the formalization of [Danner et al. 2015, 2013; Hudson, Bowornmet 2016] for non-amortized analysis.*

The *banker's method* for amortized analysis [Tarjan 1985] permits one to "prepay" time cost to generate "credits" that are "spent" later to reduce time cost, rearranging the accounting of costs from one portion of a program to another (in particular, generating a credit costs 1 unit of time, while spending a credit reduces the cost by 1 unit of time). In this example, we maintain the invariant that one credit is attached to every 1 bit in the counter representation. The *amortized cost* of flipping a bit from 0 to 1 is then 2 units of time—one for the actual bit flip plus one to generate the credit. However, the amortized cost of flipping a bit from 1 to 0 is 0 units of time—the bit flip takes one unit of time, but that is paid for by the credit. Using these new amortized costs, we can see that $T_{\mathtt{inc}}(n)$ is $O(1)$ amortized: in the case where the first bit is 0, we flip it to 1, which costs 2 units of time, and stop. In the case where the first bit is 1, we flip it *for free* to 0, and then make a recursive call, which inductively is bounded by 2. So $T_{\mathtt{inc}}(n) = 2$, which means that $T_{\mathtt{set}}(n) = 2n$, amortized. Since a single run of set starts with no credits, its actual cost will be bounded by the amortized cost $2n$: all of the credits spent during the call to set, which subtract from the cost, must have been created earlier, incurring a cost which balances out the gain garnered from spending it.

Formalizing recurrence extraction for the banker's method for amortized analysis requires us to move from a relatively standard source language based on the simply-typed $\lambda$-calculus with inductive datatypes to a more specialized one. We do not expect amortization policies (e.g. generate a credit when flipping a bit from 0 to 1, to be spent when flipping a bit from 1 to 0) to be automatically inferable in the general case—these policies are the part of an amortized analysis that requires the most cleverness. To notate these policies, we use an *intermediate language* $\lambda^A$ (Section 2), which has "effectful" operations for generating and spending credits (create and spend), as well as a modal type operator $!_\ell$ for associating credits with values (e.g. storing a credit with each 1 in a bit list). The type $!_\ell A$ classifies a value of type $A$ that has $\ell$ credits associated with it. To correctly manage credits, this intermediate language is based on a form of linear logic, which prevents spending the same credit more than once; in particular, $\lambda^A$ is an affine lambda calculus with all of the standard connectives $\otimes, \oplus, \&, \multimap, !$ plus multiplicities $!^k A$ (where $k$ is a positive number) for tracking multiple-use values. The type structure of the intermediate language is inspired by

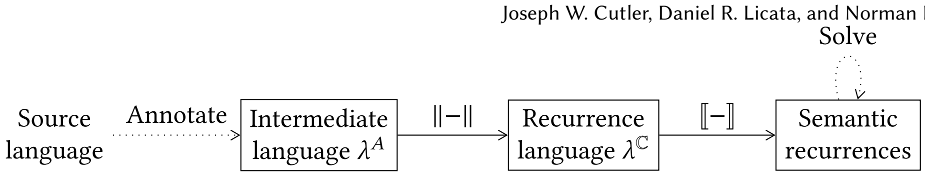
Source language
Annotate
Intermediate language $\lambda^A$
$\|-\|$
Recurrence language $\lambda^{\mathbb{C}}$
$[\![-]\!]$
Semantic recurrences
Solve

Fig. 1. Recurrence Extraction Pipeline

the credits (written as $\diamond$) of [Hofmann 2002, 2003], $n$-linear types (e.g. [Atkey 2018; Girard et al. 1992; McBride 2016; Reed [n.d.]]), and the uses of credits and linear logic in in automatic amortized resource analysis (AARA) (e.g. [Hoffmann et al. 2012; Hofmann and Jost 2003; Knoth et al. 2019]).

The target of the monadic translation is the *recurrence language* $\lambda^{\mathbb{C}}$, which, following [Danner et al. 2015; Hudson, Bowornmet 2016], is a standard simply-typed $\lambda$-calculus with a base type for costs (linearity is not needed at this stage). It is equipped with an inequality judgment $E \leq_T E'$ that can be used to express upper bounds. The translation we define here extracts a recurrence for the *amortized* cost of the program (where the costs have been "rearranged"), by translating the credit generation and spending operations in $\lambda^A$ to modifications of the cost. We define a bounding relation (a cross-language logical relation) for the amortized case, and prove that a term is related to its extraction. As a corollary, we obtain that the amortized cost of running a program from $\lambda^A$ is bounded by the cost component of its translation into $\lambda^{\mathbb{C}}$; for programs that use no external credits, this gives a bound on its actual cost as well. The recurrence language, recurrence extraction and bounding theorem are described in Section 3. Next, we use a denotational semantics of the recurrence language in preorders, similar to [Danner et al. 2015], to justify the consistency of the recurrence language $\leq$ judgment, and to simplify and solve extracted recurrences (Section 4).

The version of $\lambda^A$ and the recurrence extraction presented through Section 4 allows a statically fixed number of credits to be stored with each element of a data structure (e.g. 1 credit on element of a list, so $n$ credits overall). For some analyses, it is necessary to choose the number of credits stored with an element dynamically. For example, when analyzing splay trees [Sleator and Tarjan 1985], the number of credits stored at each node in the tree is a function of the size of the subtree rooted at that node, which varies for different tree nodes. To support such analyses, we extend $\lambda^A$ with existential quantifiers over credit variables in Section 5, and use them to code a portion of Okasaki [1998]'s analysis of splay trees in our system.

The process of extracting and solving a recurrence in diagrammed in Figure 1. While automation of the annotation and solving steps is a worthwhile goal (something we discuss in Section 7), our main motivation in this paper is to formally justify the extract-and-solve method for amortized analysis, a technique that we teach and that is typically used by practitioners. Connecting the extracted recurrence in terms of user-defined notions of size to the operational cost is the least justified step in this process, and so a formal account of it has important foundational value. It could likewise have important practical value: because students and practitioners are trained in the use of cost recurrences, reverse-engineering a recurrence that yields a worse-than-expected cost bound to the (mis)implementation may require a lower cognitive load than doing the same with more sophisticated techniques. Moreover, though this technique is less automated than others, it can handle at least some examples that existing techniques cannot—to our knowledge, splay trees cannot be analyzed by the existing automatic techniques. We give a detailed comparison with related work in Section 6.

## 2 INTERMEDIATE LANGUAGE $\lambda^A$

In this section we discuss the static and operational semantics of $\lambda^A$, which is an *affine* lambda calculus—it permits weakening (unused variables) but not contraction (duplication of variables). It

$$
\begin{array}{lll}
\text{Types} & A, B, C & ::= \mathbb{N} \mid \texttt{List}\,(A) \mid A \multimap B \mid A \otimes B \mid A \oplus B \mid A \& B \mid !^{k}_{\ell} A \\
\text{Terms} & M, N & ::= x \mid \texttt{tick}\,;\, M \mid \texttt{create}_{\ell}\, M \mid \texttt{spend}_{\ell}\, M \mid \texttt{save}^{k}_{\ell}\, M \mid \texttt{transfer}_{k'}\, !^{k}_{\ell}\, x = M \texttt{ to } N \\
 & & \quad \mid \lambda x.M \mid M\, N \mid \texttt{inl}\, M \mid \texttt{inr}\, M \mid \texttt{case}_{k'}(M, x.N_1, y.N_2) \mid \langle M, N \rangle \mid \pi_1 M \mid \pi_2 M \\
 & & \quad \mid \texttt{split}(M,\ x.y.N) \mid 0 \mid S(M) \mid \texttt{nrec}(M, N_1, N_2) \mid \texttt{[]} \mid M :: N \mid \texttt{lrec}(M, N_1, N_2)
\end{array}
$$

Fig. 2. $\lambda^A$ Grammar

includes some standard connectives of linear logic, such as positive/eager/multiplicative products ($\otimes$ and 1), sums/coproducts ($\oplus$), and functions ($\multimap$), as well as negative/lazy/additive products ($\&$). The language has two basic datatypes, natural numbers ($\mathbb{N}$) and (eager) lists ($\texttt{List}\,(A)$), both with structural recursion (though we expect these techniques to extend to all strictly positive inductive types [Danner and Licata 2020; Danner et al. 2015]).

In addition to these, $\lambda^A$ contains some constructs specific to its role as an intermediate language for expressing amortized analyses. First, instead of fixing the operational costs of $\lambda^A$'s programs themselves, we include a `tick` operation which costs 1 unit of time, and assume that the translation of a program into $\lambda^A$ has annotated the program with sufficient ticks to model the desired operational cost [Danielsson 2008] (for example, we can charge only for bit flips in the above binary counter program).

Second, we have operations `create` and `spend` for creating and spending credits, which respectively increase and decrease the *amortized* cost of the program *without changing* the true operational cost.

Third, we have a type constructor $!_{\ell} A$, where a value of this type is a value of type $A$ with $\ell$ credits attached; its introduction and elimination rules allow for the movement of credits around a program. The combination of of `spend` and the $!_{\ell}$ modality motivates our affine type system: because spending credits decreases the amortized cost of a program, we must ensure that a credit is spent only once, so credits should not be duplicated; because credits can be stored in values, values cannot in general be duplicated as well. However, $\lambda^A$ does allow credit weakening—choosing not to spend available credits—because this increases the amortized cost (relative to spending the credits), and we are interested in upper bounds on running time. While the basic affine type system allows a variable to be used only once, to simplify the expression of programs that use a variable a fixed number of times, we use $n$-linear types (see e.g. [Atkey 2018; Girard et al. 1992; McBride 2016; Reed [n.d.]]), where variables are annotated with a multiplicity $k$, and can be used at most $k$ times.[1] This is internalized by a modality $!^{k} A$, which represents an $A$ that can be used at most $k$ times. We additionally allow $k$ to be $\infty$, in which case $!^{\infty} A$ is the usual exponential of linear logic, allowing unrestricted use. Using this modality, standard functional programs can be coded in $\lambda^A$, but our current recurrence extraction does not handle the $!^{\infty}$ fragment very well, as explained below—at present, we use $!^{\infty}$ mainly as a technical device for typing recursors. It is technically convenient to combine the two modalities into one type former $!^{k}_{\ell} A$, which represents an $A$ that can be used $k$ times, which also has $\ell$ credits attached (total, not $\ell$ credits with each use). Because $k$ is a coefficient but $\ell$ is an additive constant, the individual modalities are recovered as $!^{k} A := !^{k}_{0} A$ and $!_{\ell} A := !^{1}_{\ell} A$. In pure affine logic, one can think of $!^{k}_{\ell} A$ as $X \otimes \ldots \otimes X \otimes A \otimes \ldots \otimes A$ with $\ell$ $X$s and $k$ $A$'s (in the case where $k$ and $\ell$ are finite), for an atomic proposition $X$ representing a single credit. However, our judgmental presentation is easier to work with for our bounding relation and theorem below, and the $n$-linear modality $!^{k} A$ ensures that additional invariant that it is the *same* value that can be used $k$ times, i.e. it only allows the diagonal of $A \otimes \ldots \otimes A$.

[1]While Girard's notation for multiplicities is $!_{k} A$ [Girard et al. 1992], we write superscripts following Atkey [2018], and write subscripts for the credit-storing modality, which is used more frequently in our system.

## 2.1 Type System

In Fig. 3 we define a typing judgment of the form $\Gamma \vdash_f M : A$, where $\Gamma$ is a standard context $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ and $f$ is a *resource* term of the form $a_1x_1 + a_2x_2 + \ldots + a_nx_n + \ell$, where $x_1, \ldots, x_n$ are the variables in $\Gamma$ and $a_i$ and $\ell$ are natural numbers or $\infty$. The resource term $f$ can be thought of as annotating each variable $x_i$ with the number of times $a_i$ that it is allowed to occur, and additionally annotating the judgment with a nonnegative "bank" $\ell$ of available credits that can be used. For example, the judgment $x : A, y : B, z : C \vdash_{3x+2y+0z+2} M : D$, means that $M$ is a term of type $D$, which may use $x$ at most 3 times, $y$ at most twice, $z$ not at all, and has access to 2 credits. We consider these resource terms up to the usual arithmetic identities (associativity, unit, commutativity, distributivity, $0f = 0$, $\infty k = \infty$ otherwise, etc.). In the admissible substitution rule, we write $g[f/x]$ to denote the result of normalizing the textual substitution of $f$ for $x$ in $g$ according to these identities; e.g. $(3x + 2y + 2)[10a + 11b + 3/x] = 30a + 33b + 2y + 11$. Our judgmental presentation of $n$-linear types differs from some existing ones– the reader more familiar with Girard's BLL [Girard et al. 1992] may read $\Gamma \vdash_f M : A$ as analogous to $!_{\vec{f}}\Gamma \vdash M : A$ – but this type system was derived as an instance of a general framework for modal types [Licata et al. 2017], which, for our purposes, simplifies the presentation of standard metatheorems like substitution. Note that the resource terms $f$ play a different role than the resource polynomials in Bounded Linear Logic and AARA [Girard et al. 1992; Hoffmann et al. 2012], which provide a mechanism for measuring the size and credit allocation in a data structure. The resource terms are also affine in the sense of a polynomial—the exponent of every variable is 1, except for the constant term $\ell$—but we will avoid this meaning of affine to avoid confusion with "affine logic" (allowing weakening but not contraction).

*2.1.1 Structural Rules.* The rules make three structural principles admissible:

Theorem 2.1 (Admissible structural rules).

- *Resource Weakening: Write $g \geq f$ for the coefficient-wise partial order on resource terms ($a_1x_1 + a_2x_2 + \ldots + \ell \geq b_1x_1 + b_2x_2 + \ldots + \ell'$ iff $a_i \geq b_i$ for all i and $\ell \geq \ell'$). Then if $\Gamma \vdash_f M : A$ and $g \geq f$ then $\Gamma \vdash_g M : A$.*
- *Variable Weakening: If $\Gamma \vdash_f M : A$ and $y$ does not occur in $\Gamma$, then $\Gamma, y : B \vdash_{f+0y} M : A$.*
- *Substitution: If $\Gamma \vdash_f M : A$ and $\Gamma, x : A \vdash_g N : B$, then $\Gamma \vdash_{g[f/x]} N[M/x] : B$*

Proof. By induction on derivations. □

First, we can weaken the resource subscript, allowing more uses of a variable or more credits in the bank (e.g. if $\cdot \vdash_3 M : A$, then $\cdot \vdash_5 M : A$). Second, we can weaken a context to include an unused variable (we write $f + 0y$ for emphasis, but by equating resource terms up to arithmetic identities, this is just $f$). Third, we can substitute one term into another, performing the corresponding substitution on resource terms. The idea is that, if $N$ uses a variable $x$ say 3 times, then it requires 3 times the resources needed to make $M$ to duplicate $M$ three times; this multiplication occurs when substituting $f$ for the occurrence of $x$ in $g$.

*2.1.2 Multiplicative/Additive Rules in n-linear Style.* In the $n$-linear types style of presentation, rules of linear logic that traditionally split the context (e.g. $\otimes$ introduction, $\multimap$ elimination) sum the resources used in each premise, but keep the same underlying variable context $\Gamma$ in all premises. For example, in a positive pair $(M, N) : A \otimes B$, if $M$ is allowed to use $x$ 3 times and $N$ is allowed to use $x$ 4 times, then the whole pair must be allowed to use $x$ 7 times. As a special case, if a variable is not allowed to occur in, e.g., $N$, it can be marked with a coefficient of 0. On the other hand, rules for additives (e.g. pairing for $A\&B$) use the same resource term in multiple premises. While

$$\text{Admissible:} \quad \frac{\Gamma \vdash_f M : A \quad g \geq f}{\Gamma \vdash_g M : A} \qquad \frac{\Gamma \vdash_f M : A}{\Gamma, y : B \vdash_{f+0y} M : A} \qquad \frac{\Gamma \vdash_f M : A \quad \Gamma, x : A \vdash_g N : B}{\Gamma \vdash_{g[f/x]} N[M/x] : B}$$

$$\frac{}{\Gamma, x : A \vdash_{f+x} x : A} \qquad \frac{\Gamma \vdash_f M : A}{\Gamma \vdash_f \texttt{tick}\ ;\ M : A} \qquad \frac{\Gamma \vdash_{f+\ell} M : A}{\Gamma \vdash_f \texttt{create}_\ell\ M : A} \qquad \frac{\Gamma \vdash_f M : A}{\Gamma \vdash_{f+\ell} \texttt{spend}_\ell\ M : A}$$

$$\frac{\Gamma \vdash_f M : A \quad kf + \ell \leq g}{\Gamma \vdash_g \texttt{save}^k_\ell\ M :!^k_\ell A} \qquad \frac{\Gamma \vdash_f M :!^k_\ell A \quad \Gamma, x : A \vdash_{g+k'(kx+\ell)} N : B}{\Gamma \vdash_{k'f+g} \texttt{transfer}_{k'}\ !^k_\ell\, x = M \texttt{ to } N : B}$$

$$\frac{\Gamma, x : A \vdash_{f+x} M : B}{\Gamma \vdash_f \lambda x.M : A \multimap B} \qquad \frac{\Gamma \vdash_f M : A \multimap B \quad \Gamma \vdash_g N : A}{\Gamma \vdash_{f+g} M\, N : B}$$

$$\frac{\Gamma \vdash_f M : A}{\Gamma \vdash_f \texttt{inl}\, M : A \oplus B} \qquad \frac{\Gamma \vdash_f M : B}{\Gamma \vdash_f \texttt{inr}\, M : A \oplus B} \qquad \frac{\begin{array}{l}\Gamma \vdash_f M : A \oplus B \\ \Gamma, x : A \vdash_{g+k'x} N_1 : C \\ \Gamma, y : B \vdash_{g+k'y} N_2 : C\end{array}}{\Gamma \vdash_{k'f+g} \texttt{case}_{k'}\ (M,\ x.N_1\ ,\ y.N_2) : C}$$

$$\frac{\Gamma \vdash_f M : A \quad \Gamma \vdash_f N : B}{\Gamma \vdash_f \langle M, N \rangle : A \& B} \qquad \frac{\Gamma \vdash_f M : A_1 \& A_2}{\Gamma \vdash_f \pi_i M : A_i}$$

$$\frac{}{\Gamma \vdash_f () : 1} \qquad \frac{\Gamma \vdash_f M : A \quad \Gamma \vdash_g N : B}{\Gamma \vdash_{f+g} (M, N) : A \otimes B} \qquad \frac{\Gamma \vdash_f M : A \otimes B \quad \Gamma, x : A, y : B \vdash_{g+k'(x+y)} N : C}{\Gamma \vdash_{k'f+g} \texttt{split}_{k'}(M,\ x.y.N) : C}$$

$$\frac{}{\Gamma \vdash_f 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash_f M : \mathbb{N}}{\Gamma \vdash_f S(M) : \mathbb{N}} \qquad \frac{\begin{array}{l}\Gamma \vdash_f M : \mathbb{N} \\ \Gamma \vdash_{g_1} N_1 : 1 \multimap C \\ \Gamma \vdash_{g_2} N_2 :!^\infty_0(\mathbb{N} \otimes (1 \multimap C) \multimap C)\end{array}}{\Gamma \vdash_{f+g_1+g_2} \texttt{nrec}\,(M, N_1, N_2) : C}$$

$$\frac{}{\Gamma \vdash_f \texttt{[]} : \texttt{List}\,(A)} \qquad \frac{\Gamma \vdash_f M_1 : A \quad \Gamma \vdash_g M_2 : \texttt{List}\,(A)}{\Gamma \vdash_{f+g} M_1\ ::\ M_2 : \texttt{List}\,(A)} \qquad \frac{\begin{array}{l}\Gamma \vdash_f M : \texttt{List}\,(A) \\ \Gamma \vdash_{g_1} N_1 : 1 \multimap C \\ \Gamma \vdash_{g_2} N_2 :!^\infty_0(A \otimes (\texttt{List}\,(A)\, \& C) \multimap C)\end{array}}{\Gamma \vdash_{f+g_1+g_2} \texttt{lrec}\,(M, N_1, N_2) : C}$$

Fig. 3. $\lambda^A$ Typing Rules

the elimination rule for $\oplus$ is additive in sequent calculus style, in natural deduction there is some summing because it builds in a cut for the term being case-analyzed.

*2.1.3 Ticks, and Creating/Spending Credits.* The `tick ; ` $M$ construct is used to mark program points that are intended to incur one unit of time cost (e.g. bit flips in the binary counter example); it uses the same resources as $M$.

`create` is the means to create credits, where $\texttt{create}_\ell$ gives $M$ access to $\ell$ extra credits to use, along with whatever resources are present in the ambient context; formally, this is represented by adding to the "bank" in the premise of the typing rule for $M$. In the operational semantics and recurrence extraction below, `create` adds $\ell$ steps to the amortized cost of $M$—it is used to "prepay" for later costs.

spend is the means to spend credits, where $\texttt{spend}_\ell$ spends $\ell$ credits; because credits can only be spent once, these $\ell$ credits in the conclusion of the typing rule are not also available in the premise for $M$. In the operational semantics/recurrence extraction, spend subtracts $\ell$ steps from the amortized cost of $M$—it is used to take advantage of prepaid steps. Note that spend satisfies the same typing judgments as an instance of resource weakening (because $f + \ell \geq f$); the "silent" weakening does not change the amortized cost, but instead is a case where our recurrence extraction might obtain a non-tight upper-bound.

*2.1.4 $!^k_\ell$ Modality.* Instead of having two separate modalities, one for $n$-use types and the other for types storing credits, we combine them into a single modality $!^k_\ell A$. A value of type $!^k_\ell A$ is a $k$-use $A$ with $\ell$ credits attached (not $k \cdot \ell$ credits, which is what one would expect if each use had $\ell$ credits attached—though that could be modeled by the type $!^k_0(!^1_\ell A)$). While we write $a$ and $\ell$ for nonnegative numbers or $\infty$, we restrict $k$ to range over a *positive* number or $\infty$ – i.e. we do not allow a "zero-use" modality $!^0_\ell A$, which would complicate the erasure of $\lambda^A$ to regular simply typed lambda calculus.

The introduction rule for $!^k_\ell$ says that if we can prove $M$ has type $A$ with $f$ resources, then a version of $M$ that can be used $k$ times requires $kf$ resources. If in addition, $\ell$ credits are to be attached, then $kf + \ell$ resources are required. Intuitively, one can think of $\texttt{save}^k_\ell\ M$ as the act of running $M$ once to obtain its value, but repeating whatever requirement it imposes on the bank $k$ times, which justifies making $k$ uses of its value, and then attaching $\ell$ credits to this value. In order to make resource weakening admissible in general, it is necessary to build weakening into this rule (the second premise).

The elimination rule for the modality allows for the credit stored on a term to be released into the ambient context of another in order to be redistributed or spent. We first present a simplified version, and then explain the general version. Given $\Gamma \vdash_f M :!^k_\ell A$, we essentially have $k$ copies of an $A$, along with $\ell$ extra credits. Given a term $N$ which can use $k$ copies of an $A$ and $\ell$ credits, $\Gamma, y : A \vdash_{ky+\ell} N : C$, we can form the term $\Gamma \vdash_f \texttt{transfer}\ !^k_\ell y = M \texttt{ to } N : C$, which, intuitively, deconstructs $M$ into its $k$-usable value and $\ell$ credits, and moves them to $N$, where they can be used. On top of this version, we make two modifications. Firstly, $N$ should have access to resources other than just what's provided to it by $M$– so we add a resource term $g$ available in $N$ (and therefore required to type the transfer). Secondly, it may be necessary at the site of the transfer to further duplicate the $M :!^k_\ell A$ — this is required to prove a fusion law below, for example. To support this, we parameterize the transfer term by another number, $k'$, arriving at the version of the rule presented in Figure 3, which should be thought of as eliminating $k'$ copies of a $!^k_\ell A$ at once. The rules for other positive types ($\oplus, \otimes$) similarly permit elimination of multiple copies at once.

The ! modality satisfies the following interactions with other logical connectives, where we write $A \dashv\vdash B$ to mean interprovability/functions in both directions:

Theorem 2.2 (Fusion Laws).

*(1)* $!^{k_1 k_2}_{\ell_1 + k_1 \cdot \ell_2} A \dashv\vdash !^{k_1}_{\ell_1} !^{k_2}_{\ell_2} A$
*(2)* $!^k_{\ell_1+\ell_2}(A \otimes B) \dashv\vdash !^k_{\ell_1} A \otimes !^k_{\ell_1} B$
*(3)* $!^k_\ell(A \oplus B) \dashv\vdash !^k_\ell A \oplus !^k_\ell B$

*2.1.5 Natural Number Recursor.* For natural numbers, while the rules for zero and successor are standard, the recursor takes a bit of explanation. We think of the recursor nrec as a function constant of type $\mathbb{N} \multimap (1 \multimap C) \multimap !^\infty_0(\mathbb{N} \times (1 \multimap C) \multimap C) \multimap C$. The base case is "thunked" because we think of $\multimap$ as a call-by-value function type, but the base case should not be evaluated until the recurrence argument is 0. The ordinary type for the step function (inductive case) would be $(\mathbb{N} \times C \multimap C)$, but we also suspend the recursive call, to allow for a simple case analysis that chooses not to use the recursive call. The $!^\infty_0$ modality surrounding the step function is needed to ensure that the step function itself does not use any ambient credits, which is necessary because the step function is applied repeatedly by the recursor ($n$ times if the value of $M$ is $n$). Without this restriction, one could, for example, iterate a step function that spends $k$ credits to subtract $Mk$ credits from the amortized cost, while only having $k$ credits in the bank to spend. For example, without the use of $!^{\infty}_{0}$, the term $\cdot \vdash_1 \texttt{nrec}\,(7, \lambda\_.0, \lambda\_.\texttt{spend}_1\ 0) : \mathbb{N}$ typechecks with only one credit in the ambient bank, but intuitively subtracts 7 from the amortized cost, rather than just the 1 credit that was allowed. We solve this problem using the type $!^{\infty}_{0}A$ (where $A$ is the ordinary type of the step function $\mathbb{N} \otimes (1 \multimap C) \multimap C$), which represents an infinitely duplicable $A$ that stores no additional credits. Being infinitely duplicable is an over-approximation, because the step function really only needs to be run $M$ times, but being more precise would require reasoning about such values in the type system.

In the common case, the step function will use other infinite-use variables but no credits from the bank. A typical typing derivation for this case, where $H$ is the type of a helper function and $A$ is the type of the step function, would be

$$\frac{f : H \vdash_{\infty f} N_2' : A}{f : H \vdash_{\infty(\infty f) = \infty f} \texttt{save}^{\infty}_{0}\ N_2' :!^{\infty}_{0}A}$$

Using this as the third premise of the typing rule of nrec, we see that such an nrec itself requires only the credits demanded by the number argument ($M$) and base case ($N_1$), assuming $f$ is substituted by a helper function that uses no credits.

The way in which the $!^{\infty}$ modality "prevents" the use of credits from the bank is somewhat subtle: a step function *can* use credits from the bank, but this will require the bank to be infinite in the conclusion. This is because the introduction rule for $!^{\infty}_{0}$ inflates any finite resources to $\infty$ in the conclusion:

$$\frac{f : H \vdash_{2x+3} N_2' : A}{f : H \vdash_{\infty(2f+3) = \infty f + \infty} \texttt{save}^{\infty}_{0}\ N_2' :!^{\infty}_{0}A}$$

Thus, the step function is only permitted to use credits from the bank when the bank has $\infty$ credits in the conclusion, while we are generally interested in programs that use finitely many credits.

*2.1.6 List Recursor.* The list recursor $\texttt{lrec}\,(M, N_1, N_2)$ has the same "credit capture" problem as the recursor on naturals, which we solve using $!^{\infty}_{0}$. The list recursor has another challenge, though, because unlike a natural number, the values of the list can themselves store credits. Because of this, to prevent credits from being duplicated, in the cons case, the recursor may use *either* the tail of the list or the recursive result, but not both. We code this using an internal choice/negative product &. The negative product will itself be treated as a lazy type constructor, where an $A \,\&\, B$ pair is a value even when the $A$ and $B$ are not, so we do not need to further thunk the recursive result $C$ here.

## 2.2 Operational Semantics for $\lambda^A$

We present a call-by-value big-step operational semantics for $\lambda^A$ in Figure 4, whose primary judgment form is $M \downarrow^{(n,r)} v$, which means that $M$ evaluates to the value $v$ with cost $(n, r)$. The first component of the cost, $n$ (a non-negative number) indicates the *real cost* of evaluating $M$, in this case the number of ticks performed while evaluating $M$. The second component, $r$ (which can be any integer), tracks creates and spends — the (possibly negative) sum total of credits created and spent while evaluating $M$, where creating is positive and spending is negative. The *amortized cost* of evaluating $M$ is $n + r$: the number of "actual" steps taken, plus the number of credits created, minus the number spent.

One reason we separate $n$ and $r$ in the judgment form is that there is a straightforward *erasure* of $\lambda^A$ to ordinary simply typed $\lambda$-calculus (STLC with a tick operation), in which evaluating the

STLC program has cost (number of ticks) $n$. Briefly, this translation translates $!^k_\ell A$ to $A$, translates all of the linear connectives to their unrestricted counterparts, drops all `create`, `spend`, `save` term constructors, and translates `transfer` to a `let`. The definition of $n$ in each of our inference rules for $M \downarrow^{(n,r)} v$ is the same as the usual cost for STLC with a tick operation, so this erasure preserves cost. Because of this erasure, the $n$ in $M \downarrow^{(n,r)} v$ is a meaningful cost to bound. Further, the distinction between $n$ and $r$ is why we have separate terms `create` and `tick`: `tick` increases the operational cost which should be preserved under erasure, while `create` increase the amortized cost only.

As discussed in Section 2.1.3, $\texttt{create}_\ell\ M$ creates $\ell$ credits for $M$ to use for the price of $\ell$ units of time cost, whereas `spend` subtracts from the amortized cost of an expression — a speedup which is paid for by the $\ell$ credits which the body is no longer allowed to use. Both are reflected by corresponding changes to $r$.

The operational intuition for $\texttt{save}^k_\ell\ M : !^k_\ell A$ is that it runs $M$ once, but repeats whatever effect this had on the credit bank $k$ times, which justifies using the credits in the value of $M$ $k$ times. (The erasure to STLC discussed above runs $M$ only once, not $k$ times—which would be challenging when $k$ is $\infty$.) Formally, this means that the $n$ in the conclusion is just the $n$ in the premise, but the $r$ is multiplied by $k$. Running $\texttt{save}^k_\ell$ does *not* add $\ell$ to the $r$ component because `save` does not create credits (adding to the amortized cost), but only attaches some already existing credits to the value $v$. Recall that `transfer` detaches the credits from a $!^k_\ell$ value, and allows for them, along with the $k$ copies of the value, to be used in another term. The evaluation rule says that, in order to evaluate $\texttt{transfer}_{k'}\ !^k_\ell\ x = M\ \texttt{to}\ N$, we first evaluate $M$ to a `save` value, and then evaluate the substitution instance $N[v_1/x]$. The $k'$ in `transfer` means to repeat the evaluation of $M$ $k'$ times, allowing $k \cdot k'$ uses in the body of $N$, so this (similarly to `save`) repeats the credit effects $r_1$ of $M$ $k'$ times in the conclusion. The other positive elimination forms are similar.

## 2.3 Syntactic Properties

In the operational semantics judgment $M \downarrow^{(n,r)} v$, we think of $n + r$ (the actual cost $n$ plus the credit difference $r$) as the amortized cost of the program. A key property of amortized analysis is that the amortized cost is an upper bound on the true cost, which means in this case that $n + r \geq n$, so we would like $r \geq 0$. While $r$ is in general allowed to be a negative number, it is controlled by the credits $a$ of the typing judgment $\cdot \vdash_a M : A$, intuitively because it is only `spend` operations that subtract from $r$, and `spend` operations are only allowed when the type system deems there to be sufficient credits available. Thus, we will be able to prove that $r \geq 0$ for well-typed terms. To do so, we strengthen the induction hypotheses to prove that $\cdot \vdash_a M : A$ and $M \downarrow^{(n,r)} v$ imply $a + r \geq 0$, which gives $r \geq 0$ for closed programs that use no external credits (so $a = 0$), which is what a "main" function is expected to be (e.g. `set` in the binary counter example). It is technically convenient to combine this with a preservation result, stating that the credits of $v$ is in fact $a + r$ (the resource term in a typing judgment must be non-negative, so $a + r \geq 0$ is in fact a prerequisite for even asserting that $\cdot \vdash_{a+r} v : A$). The proofs of the following are relatively straightforward and may be found in the full version of this paper [Cutler et al. 2020].

Theorem 2.3 (Preservation Bound). *If $\cdot \vdash_a M : A$ and $M \downarrow^{(n,r)} v$, then $a+r \geq 0$ and $\cdot \vdash_{a+r} v : A$.*

We also have that values evaluate in 0 steps:

Theorem 2.4. *If $v$ is a value, and $v \downarrow^{(n,r)} v$, then $n = r = 0$.*

and that values of type $\mathbb{N}$ contain no credits:

Theorem 2.5 (Resource strengthening for $\mathbb{N}$). *If $\cdot \vdash_a v : \mathbb{N}$, then $\cdot \vdash_0 v : \mathbb{N}$*

$$\frac{M \downarrow^{(n,r)} v}{\texttt{tick} \ ; \ M \downarrow^{(1+n,r)} v} \qquad \frac{M \downarrow^{(n,r)} v}{\mathsf{create}_\ell \ M \downarrow^{(n,r+\ell)} v} \qquad \frac{M \downarrow^{(n,r)} v}{\mathsf{spend}_\ell \ M \downarrow^{(n,r-\ell)} v}$$

$$\frac{M \downarrow^{(n,r)} v}{\mathsf{save}^k_\ell \ M \downarrow^{(n,kr)} \mathsf{save}^k_\ell \ v} \qquad \frac{M \downarrow^{(n_1,r_1)} \mathsf{save}^k_\ell \ v_1 \quad N[v_1/x] \downarrow^{(n_2,r_2)} v}{\texttt{transfer}_{k'} \ !^k_\ell \, x = M \ \texttt{to} \ N \downarrow^{(n_1+n_2,k'r_1+r_2)} v}$$

$$\frac{}{\lambda x.M \downarrow^{(0,0)} \lambda x.M} \qquad \frac{M \downarrow^{(n_1,r_1)} \lambda x.M' \quad N \downarrow^{(n_2,r_2)} v_1 \quad M'[v_1/x] \downarrow^{(n_3,r_3)} v}{M \ N \downarrow^{(n_1+n_2+n_3,r_1+r_2+r_3)} v}$$

$$\frac{M \downarrow^{(n,r)} v}{\texttt{inr}\, M \downarrow^{(n,r)} \texttt{inr}\, v} \qquad \frac{M \downarrow^{(n_1,r_1)} \texttt{inr}\, v_1 \quad N_2[v_1/x] \downarrow^{(n_2,r_2)} v}{\texttt{case}_{k'} \ (M, \ x.N_1 \ , \ y.N_2) \downarrow^{(n_1+n_2,k'r_1+r_2)} v}$$

$$\frac{M \downarrow^{(n,r)} v}{\texttt{inl}\, M \downarrow^{(n,r)} \texttt{inl}\, v} \qquad \frac{M \downarrow^{(n_1,r_1)} \texttt{inl}\, v_1 \quad N_1[v_1/x] \downarrow^{(n_2,r_2)} v}{\texttt{case}_{k'} \ (M, \ x.N_1 \ , \ y.N_2) \downarrow^{(n_1+n_2,k'r_1+r_2)} v}$$

$$\frac{}{\langle M, N \rangle \downarrow^{(0,0)} \langle M, N \rangle} \qquad \frac{M \downarrow^{(n_1,r_1)} \langle N_1, N_2 \rangle \quad N_i \downarrow^{(n_2,r_2)} v}{\pi_i M \downarrow^{(n_1+n_2,r_1+r_2)} v}$$

$$\frac{M \downarrow^{(n_1,r_1)} v_1 \quad N \downarrow^{(n_2,r_2)} v_2}{(M, N) \downarrow^{(n_1+n_1,r_1+r_2)} (v_1, v_2)} \qquad \frac{M \downarrow^{(n_1,r_1)} (v_1, v_2) \quad N[v_1/x, v_2/y] \downarrow^{(n_2,r_2)} v}{\texttt{split}_{k'}(M, \ x.y.N) \downarrow^{(n_1+n_2,k'r_1+r_2)} v}$$

$$\frac{}{0 \downarrow^{(0,0)} 0} \qquad \frac{M \downarrow^{(n,r)} v}{S(M) \downarrow^{(n,r)} S(v)} \qquad \frac{}{() \downarrow^{(0,0)} ()}$$

$$\frac{M \downarrow^{(n_1,r_1)} 0 \quad N_1 \downarrow^{(n_2,r_2)} \lambda x.N_1' \quad N_2 \downarrow^{(n_3,r_3)} v' \quad N_1'[()/x] \downarrow^{(n_4,r_4)} v}{\mathsf{nrec}\,(M, N_1, N_2) \downarrow^{(n_1+n_2+n_3+n_4,r_1+r_2+r_3+r_4)} v}$$

$$\frac{\begin{array}{l} M \downarrow^{(n_1,r_1)} S(v_1) \\ N_2 \downarrow^{(n_2,r_2)} \mathsf{save}^\infty_0 \ (\lambda x.N_2') \\ N_1 \downarrow^{(n_3,r_3)} \lambda x.N_1' \\ N_2'[(v_1, \lambda z.(\mathsf{nrec}\left(v_1, \lambda x.N_1', \mathsf{save}^\infty_0 \ (\lambda x.N_2')\right)))/x] \downarrow^{(n_4,r_4)} v \end{array}}{\mathsf{nrec}\,(M, N_1, N_2) \downarrow^{(n_1+n_2+n_3+n_4,r_1+r_2+r_3+r_4)} v}$$

$$\frac{M \downarrow^{(n_1,r_1)} [] \quad N_1 \downarrow^{(n_2,r_2)} \lambda x.N_1' \quad N_2 \downarrow^{(n_3,r_3)} \mathsf{save}^\infty_0 \ (\lambda x.N_2') \quad N_1'[()/x] \downarrow^{(n_4,r_4)} v}{\texttt{lrec}\,(M, N_1, N_2) \downarrow^{(n_1+n_2+n_3+n_4,r_1+r_2+r_3+r_4)} v}$$

$$\frac{\begin{array}{l} M \downarrow^{(n_1,r_1)} v_1 \ :: \ v_2 \\ N_2 \downarrow^{(n_2,r_2)} \mathsf{save}^\infty_0 \ (\lambda x.N_2') \\ N_1 \downarrow^{(n_3,r_3)} \lambda x.N_1' \\ N_2'[(v_1, \langle v_2, \texttt{lrec}\left(v_2, \lambda x.N_1', \mathsf{save}^\infty_0 \ (\lambda x.N_2')\right)\rangle)/x] \downarrow^{(n_4,r_4)} v \end{array}}{\texttt{lrec}\,(M, N_1, N_2) \downarrow^{(n_1+n_2+n_3+n_4,r_1+r_2+r_3+r_4)} v}$$

Fig. 4. $\lambda^A$ Operational Semantics

$$
\begin{aligned}
\cdot \vdash_0 \texttt{inc} := \lambda b.\texttt{lrec}(b, \lambda\_.\texttt{tick}\ ;\ \texttt{create}_1\ (\texttt{inl}\,(\texttt{save}_1^1\ ()))\ ::\ \texttt{[]},\\
\texttt{save}_0^\infty(\lambda(a, tr).\texttt{case}_1(a, \_.\texttt{tick}\ ;\ \texttt{create}_1\ (\texttt{inl}\,(\texttt{save}_1^1\ ()))\ ::\ \pi_1 tr,\\
y.\texttt{transfer}_1\ !_{1\_}^1 = y\ \texttt{to}\\
\texttt{spend}_1\ (\texttt{tick}\ ;\ \texttt{inl}\,()\ ::\ \pi_2 tr)))) : \texttt{List}\,(\texttt{bit}) \multimap \texttt{List}\,(\texttt{bit})
\end{aligned}
$$

$$
\vdash_0 \texttt{set} := \lambda n.\texttt{nrec}(n, \lambda\_.\texttt{[]}, \texttt{save}_0^\infty(\lambda p.\texttt{split}_1(p,\ \_.x.\texttt{inc}\ (x\ ()))))) : \mathbb{N} \multimap \texttt{List}\,(\texttt{bit})
$$

Fig. 5. Binary Counter Terms in $\lambda^A$

## 2.4 Binary Counter Annotation

As an example, we translate the binary counter program from Section 1 to $\lambda^A$, decorating the program with `create`, `spend`, `save`, and `transfer` in order to emulate the analysis described in Section 1. Since the analysis stores credits on 1 bits, the type of bits is $\texttt{bit} = 1 \oplus !_1^1 1$; a value $\texttt{inl}\,()$ represents a 0 bit, and a value $\texttt{inr}\,(\texttt{save}_1^1\ ())$ represents a 1 bit, with a credit attached. A binary number is represented as a list of bits, `List (bit)`. The cost of interest is the number of bit flips, so we insert `tick`s everywhere a bit is flipped from 0 to 1 or vice versa. Next, to handle the credits, we `create` and subsequently `save` a credit when we flip a bit from 0 to 1, and `transfer` then `spend` when flipping bits from 0 to 1. This annotation is shown in Figure 5 – for simplicity, we use `inc` as a meta-level name for the term implementing the function, so its occurrence in `set` really means a copy of that entire term (to do this at the object level, we could alternatively think of a top-level definition of `inc` as binding an infinite-use variable).

## 3 RECURRENCE LANGUAGE $\lambda^{\mathbb{C}}$, AMORTIZED RECURRENCE EXTRACTION, AND BOUNDING THEOREM

Next, we define a translation from $\lambda^A$ into a *recurrence language* $\lambda^{\mathbb{C}}$. Unlike $\lambda^A$, $\lambda^{\mathbb{C}}$ has a fully structural (weakening and contraction) type system, and no special constructs for amortized analysis (it is mostly unchanged from [Danner et al. 2015; Hudson, Bowornmet 2016]). Further, because we view $\lambda^{\mathbb{C}}$ as a syntatx for mathematical expressions, it is designed as a call-by-name language– this is in contrast to $\lambda^A$, which is by-value. The recurrence translation takes a function in $\lambda^A$ to a function that outputs the original function's cost in $\lambda^{\mathbb{C}}$, using a cost type $\mathbb{C}$ (which we will often take to be integers). Formally, $\mathbb{C}$ can be any commutative ring with an $\infty$ element, the typical example being the ("tropical") max-plus ring on the integers, i.e. integers with addition and binary maxes. Some of the typing rules for $\lambda^{\mathbb{C}}$ are presented in Figure 6.

Relative to our previous work, the main conceptual change for supporting amortized analysis is that, instead of extracting recurrences for the true cost of a program ($n$ in $M \downarrow^{(n,r)} v$), we extract recurrences that given an upper bound on the program's amortized cost $n + r$, which is itself a bound on the true cost for programs which begin with an empty bank of credits.

### 3.1 Monadic Translation from $\lambda^A$ to $\lambda^{\mathbb{C}}$

Following [Danner et al. 2015, 2013], a function $A \multimap B$ in $\lambda^A$ will be translated to a function $\langle\!\langle A \rangle\!\rangle \to \mathbb{C} \times \langle\!\langle B \rangle\!\rangle$, where for a $\lambda^A$ type $A$, a value of $\lambda^{\mathbb{C}}$ type $\langle\!\langle A \rangle\!\rangle$ represents the size of a value in $\lambda^A$. Intuitively, this means that a function in $\lambda^A$ is translated to a $\lambda^{\mathbb{C}}$ function that, in terms of the size of the input, gives the cost of running the function on that argument and the size of the output. Generalized to higher-type, "size" is properly viewed as "use-cost;" it is a property that tells us how the value affects the cost of a computation that uses it. In an unfortunate terminological clash, prior work [Danner and Royer 2007] refers to this concept as *potential* (as in "potential cost" or "future cost"), with no intentional connotation of potential functions from the physicist's method of amortized analysis. In order to keep this work consistent with the sequence of papers it follows,

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{k \in \mathbb{Z}}{\Gamma \vdash k : \mathbb{C}} \qquad \frac{\Gamma \vdash E_1 : \mathbb{C} \quad \Gamma \vdash E_2 : \mathbb{C}}{\Gamma \vdash E_1 + E_2 : \mathbb{C}} \qquad \frac{}{\Gamma \vdash () : 1}$$

$$\frac{\Gamma \vdash E_1 : T_1 \to T_2 \quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ E_2 : T_2} \qquad \frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash \lambda x.E : T_1 \to T_2} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash E : T_1 \times T_2}{\Gamma \vdash \pi_i E : T_i}$$

$$\frac{\Gamma \vdash E : T_1}{\Gamma \vdash \mathtt{inl}\, E : T_1 + T_2} \qquad \frac{\Gamma \vdash E : T_2}{\Gamma \vdash \mathtt{inr}\, E : T_1 + T_2} \qquad \frac{\Gamma \vdash E : T_1 + T_2 \quad \Gamma, x : T_1 \vdash E_1 : T \quad \Gamma, y : T_2 \vdash E_2 : T}{\Gamma \vdash \mathsf{case}\,(E,\ x.E_1\ ,\ y.E_2) : T}$$

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash E : \mathbb{N}}{\Gamma \vdash S(E) : \mathbb{N}} \qquad \frac{\Gamma \vdash E : \mathbb{N} \quad \Gamma \vdash E_1 : 1 \to T \quad \Gamma \vdash E_2 : \mathbb{N} \times T \to T}{\Gamma \vdash \mathsf{nrec}\,(E, E_1, E_2) : T}$$

$$\frac{}{\Gamma \vdash \mathtt{[]} : \mathtt{List}\,(T)} \qquad \frac{\Gamma \vdash E_1 : T \quad \Gamma \vdash E_2 : \mathtt{List}\,(T)}{\Gamma \vdash E_1\ ::\ E_2 : \mathtt{List}\,(T)} \qquad \frac{\begin{array}{l}\Gamma \vdash E : \mathtt{List}\,(T_1) \\ \Gamma \vdash E_1 : 1 \to T \\ \Gamma \vdash E_2 : T_1 \times (\mathtt{List}\,(T_1) \times T) \to T\end{array}}{\Gamma \vdash \mathtt{lrec}\,(E, E_1, E_2) : T}$$

Fig. 6. Recurrence Language $\lambda^{\mathbb{C}}$ Definition

and since $\lambda^A$ is based on the banker's method, we will only use "potential" to refer to the use-cost of a value, and so call $\langle\!\langle A \rangle\!\rangle$ the *potential type* for $A$ and a value of type $\langle\!\langle A \rangle\!\rangle$ a *potential*. The size of the output is needed for the translation to be compositional: the recurrence extracted for a term should be composed of the recurrences extracted for its subterms, but the cost of e.g. a function application depends on the size of the argument itself, not just its cost. A recurrence extraction of this form can be packaged as a monadic translation into the writer monad $\mathbb{C} \times A$.

As discussed in Section 1, the proper notion of size for a specific datatype may vary from analysis to analysis. To this end, we follow [Danner et al. 2015] in deferring the abstraction of values as sizes to denotational semantics of $\lambda^{\mathbb{C}}$ defined in Section 4, which allows the same recurrence extraction and bounding theorem to be reused for multiple models with different notions of size.

We call the pair of a cost and a potential a *complexity*. The translation consists of three separate functions, the definitions of which are shown in Figure 7. Firstly, $\langle\!\langle \cdot \rangle\!\rangle$ takes a type $A$ in $\lambda^A$ and maps it to the type $\langle\!\langle A \rangle\!\rangle$ whose elements are the potentials of type $A$. We extend this to contexts pointwise: $\langle\!\langle \Gamma, x : A \rangle\!\rangle = \langle\!\langle \Gamma \rangle\!\rangle, x : \langle\!\langle A \rangle\!\rangle$. The second is $\|A\| := \mathbb{C} \times \langle\!\langle A \rangle\!\rangle$, which takes a type $A$ to the corresponding type of complexities. Finally, we overload $\|\cdot\|$ to denote the recurrence extraction function from terms of $\lambda^A$ to terms in $\lambda^{\mathbb{C}}$. For convenience, when $E : \mathbb{C} \times T$, we often write $\pi_1 E$ as $E_c$ (cost) and $\pi_2 E$ as $E_p$ (potential). [2] We also use special notation for adding a cost to a complexity, writing $E +_c E'$ for $(E + E'_c, E'_p)$ when $E : \mathbb{C}$ and $E' : \mathbb{C} \times T$.

Overall, the idea is that a term is translated to a function from potentials of its context to complexities of its type:

Theorem 3.1 (Extraction Preserves Types). *If* $\Gamma \vdash_a M : A$ *then* $\langle\!\langle \Gamma \rangle\!\rangle \vdash \|M\| : \|A\|$

We comment on some of the less obvious aspects of this translation:

- $!^k_\ell A$: The type translation erases the $!^k_\ell$ modality.
- $A \& B$: Since the negative product in $\lambda^A$ is lazy, a value of type $A \& B$ is a pair of un-evaluated terms. Thus, the potential of a term of type $A \& B$ must include the cost of evaluating each term, since that will factor into the cost of using such a value.
- `tick`: Since `tick ;` $M$ evaluates with (true cost and) amortized cost 1 higher than $M$'s, the cost component of $\|$`tick ;` $M\|$ is $1 + \|M\|_c$.

[2] We regard the subscript notation as binding tighter than ordinary projection: i.e. $\pi_1 E_p = \pi_1(E_p)$.

$$\|A\| = \mathbb{C} \times \langle\!\langle A \rangle\!\rangle$$

$$\langle\!\langle 1 \rangle\!\rangle = 1 \qquad \langle\!\langle A \otimes B \rangle\!\rangle = \langle\!\langle A \rangle\!\rangle \times \langle\!\langle B \rangle\!\rangle \qquad \langle\!\langle A \oplus B \rangle\!\rangle = \langle\!\langle A \rangle\!\rangle + \langle\!\langle B \rangle\!\rangle \qquad \langle\!\langle A \multimap B \rangle\!\rangle = \langle\!\langle A \rangle\!\rangle \to \|B\|$$

$$\left\langle\!\!\left\langle !^k_\ell A \right\rangle\!\!\right\rangle = \langle\!\langle A \rangle\!\rangle \qquad \langle\!\langle A \& B \rangle\!\rangle = \|A\| \times \|B\|$$

$$\langle\!\langle \mathbb{N} \rangle\!\rangle = \mathbb{N} \qquad \langle\!\langle \texttt{List}\,(A) \rangle\!\rangle = \texttt{List}\,(\langle\!\langle A \rangle\!\rangle)$$

$$\|x\| = (0, x)$$

$$\|()\| = (0, ()) \qquad \|(M, N)\| = (\|M\|_c + \|N\|_c\,, (\|M\|_p\,, \|N\|_p)) \qquad \|\pi_i M\| = \|M\|_c +_c \pi_i \left(\|M\|_p\right)$$

$$\|\texttt{inl}\, M\| = (\|M\|_c\,, \texttt{inl}\ \|M\|_p) \qquad \|\texttt{inr}\, M\| = (\|M\|_c\,, \texttt{inr}\ \|M\|_p)$$

$$\|\texttt{case}_{k'}\ (M,\ x.N_1\ ,\ y.N_2)\| = k'\ \|M\|_c + \texttt{case}\ (\|M\|_p\ ,\ x.\ \|N_1\|\ ,\ y.\ \|N_2\|)$$

$$\|\lambda x.M\| = (0, \lambda x.\ \|M\|) \qquad \|M\ N\| = (\|M\|_c + \|N\|_c) +_c \|M\|_p\ \ \|N\|_p$$

$$\|\langle M, N \rangle\| = (0, (\|M\|\ , \|N\|))$$

$$\|\texttt{split}_{k'}(M,\ x.y.N)\| = k'\ \|M\|_c +_c \|N\|\ [\pi_1\ \|M\|_p\ /x, \pi_2\ \|M\|_p\ /y]$$

$$\|0\| = (0, 0) \qquad \|S(M)\| = (\|M\|_c\,, S(\|M\|_p))$$

$$\|\texttt{[]}\| = (0, \texttt{[]}) \qquad \|M\ ::\ N\| = (\|M\|_c + \|N\|_c\,, \|M\|_p\ ::\ \|N\|_p)$$

$$\|\texttt{tick}\ ;\ M\| = 1 +_c \|M\|$$

$$\left\|\texttt{transfer}_{k'}\ !^k_\ell\ x = M\ \texttt{to}\ N\right\| = k'\ \|M\|_c +_c \|N\|\ [\|M\|_p\ /x] \qquad \left\|\texttt{save}^k_\ell\ M\right\| = (k\ \|M\|_c\,, \|M\|_p)$$

$$\|\texttt{create}_\ell\ M\| = \ell +_c \|M\| \qquad \|\texttt{spend}_\ell\ M\| = (-\ell) +_c \|M\|$$

$$\|\texttt{nrec}\,(M, N_1, N_2)\| = (\|M\|_c + \|N_1\|_c + \|N_2\|_c) +_c \texttt{nrec}\left(\|M\|_p\,, \|N_1\|_p\,, \lambda x.\ \|N_2\|_p\ (\pi_1 x, \lambda z.\pi_2 x)\right)$$

$$\|\texttt{lrec}\,(M, N_1, N_2)\| = (\|M\|_c + \|N_1\|_c + \|N_2\|_c) +_c \texttt{lrec}\left(\|M\|_p\,, \|N_1\|_p\,, \lambda x.\ \|N_2\|_p\ (\pi_1 x, ((0, \pi_1\pi_2 x), \pi_2\pi_2 x))\right)$$

Fig. 7. Recurrence Extraction

- $\texttt{save}^k_\ell$: The extracted amortized cost of $\texttt{save}^k_\ell\ M$ is $k$ times the extracted cost of $M$, with the potential remaining the same. This is in principle a non-exact bound, because we are conceptually multiplying the operational amortized cost of $M \downarrow^{(n,r)} v$, which is $n + r$, by $k$, whereas the operational semantics gives the more precise $n + kr$. We view this as a consequence of the fact that amortized analyses extract recurrences for the amortized cost $n + r$, rather than $n$ and $r$ separately. However, this inflation is not a problem for our uses of $!^\infty$ in typing recursors because the branches of the recursor are usually values, which have 0 cost, and $\infty \times 0 = 0$. In future work, we might consider a recurrence translation into the $\mathbb{C} \times \mathbb{C} \times A$ monad, with separate extractions of $n$ and $r$, if more precision is needed. This would allow for $\lambda^A$ to be used in the place of the (linear fragment) of the source language in previous work [Danner et al. 2015]. Embedding that language into the $!^\infty$ fragment of $\lambda^A$ and then extracting recurrences into $\mathbb{C} \times \mathbb{C} \times A$ would yield the same results as applying the non-amortized recurrence extraction. We emphasize that the loss of precision from not making this change has no bearing on *amortized* algorithm analyses, it would only allow for *non-amortized* analyses to also be performed with $\lambda^A$– but such analyses are already handled by prior work [Danner et al. 2015; Kavvos et al. 2019]
- `transfer`: A similar imprecision arises with respect to the multiplicity $k'$ here, but otherwise `transfer` is translated like a `let`.

$$C ::= [] \mid \pi_0 C \mid \pi_1 C \mid C\ E \mid \mathtt{case}\,(C,\ x.E\ ,\ y.E') \mid \mathtt{nrec}\,(C, E_1, E_2) \mid \mathtt{lrec}\,(C, E_1, E_2)$$

$$\frac{\Gamma, x : T' \vdash C[x] : T \quad \Gamma \vdash E_0 \leq_{T'} E_1}{\Gamma \vdash C[E_0] \leq_T C[E_1]} \qquad \frac{}{\Gamma \vdash E \leq_T E} \qquad \frac{\Gamma \vdash E_1 \leq_T E_2 \quad \Gamma \vdash E_2 \leq_T E_3}{\Gamma \vdash E_1 \leq_T E_3}$$

$$\frac{}{\Gamma \vdash E_1[E/x] \leq_T \mathtt{case}\,(\mathtt{inl}\,E,\ x.E_1\ ,\ y.E_2)} \qquad \frac{}{\Gamma \vdash E_2[E/x] \leq_T \mathtt{case}\,(\mathtt{inr}\,E,\ x.E_1\ ,\ y.E_2)}$$

$$\frac{}{\Gamma \vdash E[E'/x] \leq_T (\lambda x.E)\ E'} \qquad \frac{}{\Gamma \vdash E_i \leq_{T_i} \pi_i(E_1, E_2)}$$

$$\frac{}{\Gamma \vdash E_1\ () \leq_T \mathtt{nrec}\,(0, E_1, E_2)} \qquad \frac{}{\Gamma \vdash E_2\ (E, \mathtt{nrec}\,(E, E_1, E_2)) \leq_T \mathtt{nrec}\,(S(E), E_1, E_2)}$$

$$\frac{}{\Gamma \vdash E_1\ () \leq_T \mathtt{lrec}\,([], E_1, E_2)} \qquad \frac{}{\Gamma \vdash E_2\ (E, (E', \mathtt{lrec}\,(E', E_1, E_2))) \leq_T \mathtt{lrec}\,(E :: E', E_1, E_2)}$$

Fig. 8. Syntactic Ordering on $\lambda^{\mathbb{C}}$

- nrec: As in the operational semantics, because we think of the recursor as a call-by-value function constant, some cost is in principle incurred for evaluating the branches to function values, though the branches are usually values in practice.
- lrec: The type of the step function in a list recursor is $!_0^\infty(A \otimes (\mathtt{List}\,(A)\,\&C) \multimap C)$, and the potential translation of this type is $\langle\!\langle A \rangle\!\rangle \times ((\mathbb{C} \times \mathtt{List}\,(\langle\!\langle A \rangle\!\rangle)) \times (\mathbb{C} \times \langle\!\langle C \rangle\!\rangle)) \to \mathbb{C} \times \langle\!\langle C \rangle\!\rangle$. However, this does not match the required type of the step function of the list recursor in $\lambda^{\mathbb{C}}$, which must be $T_1 \times (\mathtt{List}\,(T_1) \times T_2) \to T_2$. Taking $T_1 = \langle\!\langle A \rangle\!\rangle$ and $T_2 = \mathbb{C} \times \langle\!\langle C \rangle\!\rangle$, the translation of the step function additionally requires a $\mathbb{C}$ input representing the cost of the tail of the list. However, lists are eager, so the step function is always applied to a value, so we can supply 0 cost here.

## 3.2 Recurrence Language Inequality Judgment

$\lambda^{\mathbb{C}}$ has a syntactic inequality judgment $\Gamma \vdash E_1 \leq_T E_2$ (Figure 8), which intuitively means that the recurrence $E_1$ is bounded above by $E_2$. For now, we include only those inequalities that are necessary to prove the bounding theorem; this allows for the most models of the recurrence language, and additional axioms valid in particular models can be added in order to simplify recurrences syntactically. The necessary axioms are congruence in the principal positions of elimination forms, as well as the fact that $\beta$-reducts are bounded above by their redexes. We often omit the context and type subscript from $\Gamma \vdash E_1 \leq_T E_2$, writing $E_1 \leq_T E_2$ or $E_1 \leq E_2$, though formally it is a relation on well-typed terms in context. This relation is primarily a technical device to provide closure properties for the bounding relation. Because of this, we omit a more lengthy discussion of the relation here, and refer the reader to the prior work [Danner et al. 2015] which introduces this type of relation.

## 3.3 Bounding Relation and Its Closure Properties

The correctness of the recurrence extraction is stated in terms of a logical relation between terms in $\lambda^A$ and terms in $\lambda^{\mathbb{C}}$. The intended meaning is that the $\lambda^{\mathbb{C}}$ recurrence term is an upper bound on the $\lambda^A$ term's cost and potential.

*Definition 3.2 (Bounding Relation).* When $\cdot \vdash_a M : A$ and $\cdot \vdash E : \|A\|$, then $M \sqsubseteq^{A,a} E$ if and only if, when $M \downarrow^{(n,r)} v$,

- $n \leq E_c - r$

- $v \sqsubseteq_{\texttt{val}}^{A, a+r} E_p$

When $\cdot \vdash_a v : A$ and $\cdot \vdash E : \langle\!\langle A \rangle\!\rangle$, we define $v \sqsubseteq_{\texttt{val}}^{A,a} E$ by induction on $A$.

- $\texttt{save}_\ell^k\ v \sqsubseteq_{\texttt{val}}^{!_\ell^k A, c} E$ if there exists $d \geq 0$ so that $kd + \ell \leq c$, and $v \sqsubseteq_{\texttt{val}}^{A,d} E$
- $\lambda x.M \sqsubseteq_{\texttt{val}}^{A \multimap B, c} E$ if whenever $v \sqsubseteq_{\texttt{val}}^{A,d} E'$, we have that $M[v/x] \sqsubseteq^{B, c+d} E\ E'$
- $(v_1, v_2) \sqsubseteq_{\texttt{val}}^{A_1 \otimes A_2, a} E$ if there are $a_1, a_2$ such that $a_1 + a_2 = a$ and $v_i \sqsubseteq_{\texttt{val}}^{A_i, a_i} \pi_i E$ for $i \in \{1, 2\}$
- $[] \sqsubseteq_{\texttt{val}}^{\texttt{List}(A), a} E$ iff $[] \leq_{\texttt{List}(\langle\!\langle A \rangle\!\rangle)} E$
- $v_1 :: v_2 \sqsubseteq_{\texttt{val}}^{\texttt{List}(A), a} E$ iff there are $E_1, E_2$ with $E_1 :: E_2 \leq_{\texttt{List}(\langle\!\langle A \rangle\!\rangle)} E$, and there are $a_1, a_2$ such that $a_1 + a_2 = a$ such that $v_1 \sqsubseteq_{\texttt{val}}^{A, a_1} E_1$ and $v_2 \sqsubseteq_{\texttt{val}}^{\texttt{List}(A), a_2} E_2$.
- $0 \sqsubseteq_{\texttt{val}}^{\mathbb{N}, a} E$ iff $0 \leq E$
- $S(v) \sqsubseteq_{\texttt{val}}^{\mathbb{N}, a} E$ iff there is some $E'$ such that $S(E') \leq_{\mathbb{N}} E$, and $v \sqsubseteq_{\texttt{val}}^{\mathbb{N}, a} E'$
- $\texttt{inl}\ v \sqsubseteq_{\texttt{val}}^{A \oplus B, a} E$ if there exists $E'$ such that $\texttt{inl}\ E' \leq_{\langle\!\langle A \rangle\!\rangle} E$ and $v \sqsubseteq_{\texttt{val}}^{A, a} E'$.
- $\texttt{inr}\ v \sqsubseteq_{\texttt{val}}^{A \oplus B, a} E$ if there exists $E'$ such that $\texttt{inr}\ E' \leq_{\langle\!\langle B \rangle\!\rangle} E$ and $v \sqsubseteq_{\texttt{val}}^{B, a} E'$.
- $() \sqsubseteq_{\texttt{val}}^{1, a} E$ if $() \leq_1 E$.
- $\langle M, N \rangle \sqsubseteq_{\texttt{val}}^{A \& B, a} E$ if $M \sqsubseteq^{A, a} \pi_1 E$, and $N \sqsubseteq^{B, a} \pi_2 E$.

We extend the value bounding relation to substitutions pointwise: $\theta \sqsubseteq_{\texttt{sub}}^{\Gamma, \sigma} \Theta$ if for all $x : A \in \Gamma$, $\theta(x) \sqsubseteq_{\texttt{val}}^{A, \sigma(x)} \Theta(x)$. Finally, we define the bounding relation for open terms: when $\Gamma \vdash_f M : A$, we say that $M \sqsubseteq E$ if for all $\theta \sqsubseteq_{\texttt{sub}}^{\Gamma, \sigma} \Theta$, we have $M[\theta] \sqsubseteq^{A, f[\sigma]} E[\Theta]$.

The *term/expression bounding relation* $M \sqsubseteq^{A,a} E$ says first that the cost component of $E$ is an upper bound on the amortized cost of $M$, which is $n + r \leq E_c$ (since we will eventually be interested in bounding the actual cost of evaluating $M$, we write this as $n \leq E_c - r$). Additionally, expression bounding says that the potential component of $E$ is an "upper bound" on the value that $M$ evaluates to; this is expressed via a mutually-defined type-varying *value bounding relation* $M \sqsubseteq_{\texttt{val}}^{A,a} E$. The value bounding relation is defined first by induction on the type $A$, and the cases for natural numbers and lists have a local induction on the number/list value as well.[3] We write the credit bank $a$ as a parameter of the bounding relations, but it is a presupposition that this number is the same one that was used to type check $\cdot \vdash_a \{M, v\} : A$ (because the bounding relation is on closed terms, the resource subscript is just a single number $a$).

We extend the bounding relation to open terms by considering all closing substitutions: a term $\Gamma \vdash_f M : A$ is bounded by $E$ if for every substitution $\theta$ which is bounded pointwise by $\Theta$ with some credit function $\sigma$, then the closed term $M[\theta]$ is bounded by $E[\Theta]$ with $f[\sigma]$ credits. In this definition, $\sigma$ gives a number of credits $a_i$ for each variable $x_i$, because $\theta$ is a substitution of closed terms for variables $(\cdot \vdash_{a_1} v_1 : A_1)/x_1, (\cdot \vdash_{a_2} v_2 : A_2)/x_2, \ldots$.

## 3.4 Bounding Theorem

As usual for a logical relation, we first require some lemmas about the bounding relation, before a main loop proving the fundamental theorem that terms are related to their extractions. The proofs of the following theorems can be found in the full version of this paper [Cutler et al. 2020].

First, we have an analogue of Theorem 2.5:

Theorem 3.3 ($\mathbb{N}$-strengthening). *For all $\cdot \vdash_a v : \mathbb{N}$, if $v \sqsubseteq_{\texttt{val}}^{\mathbb{N}, a} E$, then $v \sqsubseteq_{\texttt{val}}^{\mathbb{N}, 0} E$.*

Second, we can weaken a bound by recurrence language inequality:

[3] In general, it is necessary to define the relations for inductive types inductively [Danner et al. 2015], but the values of $\mathbb{N}$ and $\texttt{List}\ (A)$ are simple enough that induction on values suffices here.

$$
\begin{aligned}
\cdot \vdash \|\mathtt{inc}\| := (0, \lambda bs.\mathtt{lrec}(bs, \lambda\_.(2, (\mathtt{inr}\,()) :: []),\\
&\lambda p.(\lambda x.\mathtt{case}(\pi_1 x,\\
&\quad \_.(2 + (\pi_1\pi_2 x)_c, (\mathtt{inr}\,()) :: (\pi_1\pi_2 x)_p)\\
&\quad \_.((\pi_2\pi_2 x)_c, (\mathtt{inl}\,()) :: (\pi_2\pi_2 x)_p))\\
&)(\pi_1 p, ((0, \pi_1\pi_2 p), \pi_2\pi_2 p)))) : \mathbb{C} \times (\mathtt{List}\,(1+1) \to \mathbb{C} \times \mathtt{List}\,(1+1)
\end{aligned}
$$

$$
\begin{aligned}
\cdot \vdash \|\mathtt{set}\| := (0, \lambda n.\mathtt{nrec}(n, \lambda\_.(0, []), \lambda u.(0, \lambda p.(\pi_2 p\ ())_c +_c \|\mathtt{inc}\|_p\ (\pi_2 p\ ())_p)_p\\
&(\pi_1 u, \lambda\_.\pi_2 u))) : \mathbb{C} \times (\mathbb{N} \to \mathbb{C} \times \mathtt{List}\,(1+1))
\end{aligned}
$$

Fig. 9. Binary Counter Recurrences in $\lambda^{\mathbb{C}}$

Theorem 3.4 (Weakening).

*(1) If $M \sqsubseteq^{A,a} E$, and $E \leq_{\|A\|} E'$, then $M \sqsubseteq^{A,a} E'$*

*(2) If $v \sqsubseteq^{A,a}_{\mathtt{val}} E$, and $E \leq_{\langle\!\langle A \rangle\!\rangle} E'$, then $v \sqsubseteq^{A,a}_{\mathtt{val}} E'$*

Next, we have an analogue of resource weakening in Theorem 2.1:

Theorem 3.5 (Credit Weakening). *If $a_1 \leq a_2$, then:*

*(1) If $M \sqsubseteq^{A,a_1} E$, then $M \sqsubseteq^{A,a_2} E$*

*(2) If $v \sqsubseteq^{A,a_1}_{\mathtt{val}} E$, then $v \sqsubseteq^{A,a_2}_{\mathtt{val}} E$*

Next, we have inductive lemmas that will be used in the recursor cases of the fundamental theorem:

Theorem 3.6 ($\mathbb{N}$-Recursor). *If $\lambda x.N_1' \sqsubseteq^{1 \multimap C, c_3}_{\mathtt{val}} E_1$, $\lambda x.N_2' \sqsubseteq^{\mathbb{N}\otimes(1\multimap C)\multimap C, d}_{\mathtt{val}} E_2$ with $d \geq 0$, then $\forall n \geq 0$, if $\overline{n} \sqsubseteq^{\mathbb{N},0}_{\mathtt{val}} E$, then $\mathtt{nrec}\left(\overline{n}, \lambda x.N_1', \mathtt{save}^{\infty}_0\ (\lambda x.N_2')\right) \sqsubseteq^{C, c_3 + \infty \cdot d} \mathtt{nrec}\,(E, E_1, \lambda p.E_2\ (\pi_1 p, \lambda z.\pi_2 p))$*

Theorem 3.7 ($\mathtt{List}\,(A)$-Recursor). *If $\lambda x.N_1' \sqsubseteq^{1\multimap C, c_1}_{\mathtt{val}} E_1$ and $\lambda x.N_2' \sqsubseteq^{A\otimes(\mathtt{List}(A)\&C)\multimap C, c_2}_{\mathtt{val}} E_2$, then for all values $\cdot \vdash_d v : \mathtt{List}\,(A)$ such that $v \sqsubseteq^{\mathtt{List}(A), d}_{\mathtt{val}} E$, we have that*
*$\mathtt{lrec}\left(v, \lambda x.N_1', \mathtt{save}^{\infty}_0\ (\lambda x.N_2')\right) \sqsubseteq^{C, c_1 + d + \infty\cdot c_2} \mathtt{lrec}\,(E, E_1, \lambda x.E_2(\pi_1 x, ((0, \pi_1\pi_2 x), \pi_2\pi_2 x)))$*

Using these, we prove the main result:

Theorem 3.8 (Bounding Theorem). *If $\Gamma \vdash_f M : A$, then $M \sqsubseteq^{A} \|M\|$*

Finally, for terms that use no external credits, the true cost is bounded by the extracted recurrence:

Corollary 3.9 (True cost bounding). *If $\cdot \vdash_0 M : A$ and $M \downarrow^{(n,r)} v$ then $n \leq \|M\|_c$.*

Proof. By Theorem 3.8, we have $n \leq \|M\|_c - r$, but by preservation (Theorem 2.3), we have that $0 + r \geq 0$, so $n \leq \|M\|_c$. □

## 3.5 Binary Counter Recurrences

As an example, the binary counter program in $\lambda^{A}$ (Figure 5) is translated by the recurrence extraction translation to the terms in Figure 9. Next, we will use a denotational semantics of the recurrence language to simplify these recurrences to the desired closed form.

## 4 RECURRENCE LANGUAGE SEMANTICS

The final step of our technique is to simplify recurrences to closed forms. This can be done semantically, in a denotational model of the recurrence languages, or syntactically, by adding axioms to the inequality judgment $\Gamma \vdash E \leq_T E'$ corresponding to properties true in a particular model. Here, we will work in a denotational model of $\lambda^{\mathbb{C}}$ in preorders, which mostly follows previous work [Danner et al. 2015, 2013; Hudson, Bowornmet 2016].

## 4.1 Semantic Interpretation

We describe the semantic interpretation of $\lambda^{\mathbb{C}}$ in preorders here, and highlight the differences from [Hudson, Bowornmet 2016], which gives a similar presentation with mechanized proofs.

The semantics of types and terms is given in Figure 10, omitting function and product types, which are interpreted using the standard cartesian product and exponential objects of preorders. For each type $A$ of $\lambda^{\mathbb{C}}$, we associate a partially ordered set $[\![A]\!]$ equipped with a top element ($\infty$) and binary maximums ($\vee$) for which the top element is an annihilator. We write 1 for the one-element poset, and $\mathbb{N} \cup \infty$ for the natural numbers with an infinite element added, with the usual $0 \leq 1 \leq 2 \leq \ldots \leq \infty$ total order, and $\mathbb{Z} \cup \infty$ for the integers with an infinite element added, with the usual total order. We write $P \times Q$ for the cartesian product of posets with the pointwise order, and $Q^P$ for the poset of monotone functions from $P$ to $Q$, ordered pointwise; these have binary maxes and top elements given pointwise. We write $P + Q/\!\sim$ for the "coalesced" sum, which first takes the disjoint union of $P$ and $Q$, with only $\texttt{inl}(x) \leq \texttt{inl}(y)$ if $x \leq_P y$ and similarly for $\texttt{inr}$, and then equates $\texttt{inl}(\infty_P)$ and $\texttt{inr}(\infty_Q)$ to create a top element $\infty_{P+Q/\sim}$; binary maxes are defined using maxes in $P$ and $Q$ for two elements whose injections match, and to be $\infty$ otherwise. The translation on types is extended to contexts: $[\![\cdot]\!] = 1$, $[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$. Finally, we interpret terms of $\lambda^{\mathbb{C}}$ as *monotone* (but not necessarily infinity- or max-preserving) maps[4] from the interpretation of their contexts into the interpretation of their types. These maps are morphisms in the category **Poset** of partially ordered sets and monotone maps, and so we write them as elements of $\mathrm{Hom}_{\mathbf{Poset}}(A, B)$, the set of monotone maps between posets $A$ and $B$.

In Figure 10, we show some representative cases of the interpretation of terms for sums, natural numbers and lists. For costs, the interpretation of cost constants and addition uses the elements and addition of $\mathbb{Z} \cup \infty$. In this model, we interpret both natural numbers and lists as $\mathbb{N} \cup \infty$; for lists, this interprets a list as its length. $\mathbb{N} \cup \infty$ has a 0 element and a monotone successor function $S$, where $S(\infty) = \infty$; these are used to interpret 0/the empty list and successor/cons. The elimination forms for positives are more complex, and use some auxiliary monotone functions (which are the morphisms in the category of posets):

Theorem 4.1. *For any posets $A, B, C, G$ with $\infty$ and $\vee$,*

*(1)* $\texttt{snrec} \in \mathrm{Hom}_{Poset}\left(\left(C^1\right)^G \times \left(C^{\mathbb{N}\times C}\right)^G, C^{G\times\mathbb{N}}\right)$

*(2)* $\texttt{slrec} \in \mathrm{Hom}_{Poset}\left(\left(C^1\right)^G \times \left(C^{A\times(\mathbb{N}\times C)}\right)^G, C^{G\times\mathbb{N}}\right)$

*(3)* $\texttt{scase} \in \mathrm{Hom}_{Poset}\left(C^{G\times A} \times C^{G\times B}, C^{G\times(A+B)}\right)$

The definition of $\texttt{scase}$ is required to respect the quotienting $\texttt{inl}(\infty) = \texttt{inr}(\infty)$; by maxing each branch the image of $\infty$ from the other branch, we obtain $f(\gamma, \infty) \vee g(\gamma, \infty)$ as the image of both of those. The definition of $\texttt{snrec}$ is required to be monotone in the $0 \leq 1 \leq \ldots \leq \infty$ ordering; taking the maximum of the base case and the inductive step achieves this, because it forces the image of 1 to dominate the image of 0. The definition of $\texttt{slrec}$ is similar; the new question that arises is that, because we have abstracted lists as their lengths, forgetting the elements, we do not have a value for the head of the list to supply to $g$ (which, when we use this operation, will be the translation of the cons branch given to the $\lambda^{\mathbb{C}}$ recursor). Here, we always supply $\infty$ as the head list element, which is sufficient when the analysis really does not require any information about the elements of the list (otherwise, one can make a model where lists are interpreted more precisely than as their lengths [Danner and Licata 2020; Danner et al. 2015]).

[4] We write the composition of maps $f : A \to B$ and $g : B \to C$ in diagrammatic order, $f; g : A \to C$.

$$\begin{aligned}
&[\![\mathbb{C}]\!] = \mathbb{Z} \cup \{\infty\}\\
&[\![\mathbb{N}]\!] = \mathbb{N} \cup \{\infty\}\\
&[\![\texttt{List}\,(T)]\!] = \mathbb{N} \cup \{\infty\}\\
&[\![T_1 + T_2]\!] = ([\![T_1]\!] + [\![T_2]\!]) \;/ \sim \text{ where } \texttt{inl}\,\infty \sim \texttt{inr}\,\infty
\end{aligned}$$

$$[\![\Gamma, x : T, \Gamma' \vdash x : T]\!] = \pi_1^k ; \pi_2 \text{ where } |\Gamma'| = k$$

$$\begin{aligned}
&[\![\Gamma \vdash k : \mathbb{C}]\!] = \text{const}\,(k)\\
&[\![\Gamma \vdash E_1 + E_2 : \mathbb{C}]\!] = ([\![\Gamma \vdash E_1 : \mathbb{C}]\!], [\![\Gamma \vdash E_2 : \mathbb{C}]\!]); +
\end{aligned}$$

$$[\![\Gamma \vdash () : 1]\!] = \text{const}\,(())$$

$$\begin{aligned}
&[\![\Gamma \vdash \texttt{inl}\,E : T_1 + T_2]\!] = [\![\Gamma \vdash E : T_1]\!]; \texttt{inl}\\
&[\![\Gamma \vdash \texttt{inr}\,E : E_1 + E_2]\!] = [\![\Gamma \vdash E : E_2]\!]; \texttt{inr}\\
&[\![\Gamma \vdash \texttt{case}\,(E, x.E_1\,, y.E_2) : T]\!] = \left(1_{[\![\Gamma]\!]}, [\![\Gamma \vdash E : T_1 + T_2]\!]\right); \texttt{scase}([\![\Gamma, x : T_1 \vdash E_1 : T]\!], [\![\Gamma, y : T_2 \vdash E_2 : T]\!])\\
&\texttt{scase} \in \text{Hom}_{Poset}\left(C^{G\times A} \times C^{G\times B}, C^{G\times(A+B)}\right)\\
&\texttt{scase}(f,g)(\gamma, \texttt{inl}\,a) = f(\gamma, a) \vee g(\gamma, \infty)\\
&\texttt{scase}(f,g)(\gamma, \texttt{inr}\,b) = f(\gamma, \infty) \vee g(\gamma, b)
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \vdash 0 : \mathbb{N}]\!] = \text{const}\,(0)\\
&[\![\Gamma \vdash S(M) : \mathbb{N}]\!] = [\![\Gamma \vdash M : \mathbb{N}]\!]; S\\
&[\![\Gamma \vdash \texttt{nrec}\,(E, E_1, E_2) : T]\!] = \left(1_{[\![\Gamma]\!]}, [\![\Gamma \vdash E : \mathbb{N}]\!]\right); \texttt{snrec}([\![\Gamma \vdash E_1 : 1 \to T]\!], [\![\Gamma \vdash E_2 : \mathbb{N} \times T \to T]\!])\\
&\texttt{snrec} \in \text{Hom}_{Poset}\left(\left(C^1\right)^G \times \left(C^{\mathbb{N}\times C}\right)^G, C^{G\times\mathbb{N}}\right)\\
&\texttt{snrec}(f,g)(\gamma, 0) = f(\gamma)()\\
&\texttt{snrec}(f,g)(\gamma, n+1) = g(\gamma)(n, \texttt{snrec}(f,g)(\gamma, n)) \vee f(\gamma)()
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \vdash \texttt{[]} : \texttt{List}\,(A)]\!] = \text{const}\,(0)\\
&[\![\Gamma \vdash E_1 :: E_2 : \texttt{List}\,(A)]\!] = [\![\Gamma \vdash E_2 : \texttt{List}\,(A)]\!]; S\\
&[\![\Gamma \vdash \texttt{lrec}\,(E, E_1, E_2) : T]\!] = \left(1_{[\![\Gamma]\!]}, [\![\Gamma \vdash E : \texttt{List}\,(T)']\!]\right);\\
&\qquad\qquad\qquad\qquad \texttt{slrec}([\![\Gamma \vdash E_1 : 1 \to T]\!], [\![\Gamma \vdash E_2 : T' \times (\texttt{List}\,(T)' \times T) \to T]\!])\\
&\texttt{slrec} \in \text{Hom}_{Poset}\left(\left(C^1\right)^G \times \left(C^{A\times(\mathbb{N}\times C)}\right)^G, C^{G\times\mathbb{N}}\right)\\
&\texttt{slrec}(f,g)(\gamma, 0) = f(\gamma)()\\
&\texttt{slrec}(f,g)(\gamma, n+1) = g(\gamma)(\infty, (n, \texttt{slrec}(f,g)(\gamma, n))) \vee f(\gamma)()
\end{aligned}$$

Fig. 10. Semantic Interpretation Definition

The interpretation satisfies standard soundness theorems, the proofs of which can be found in the full version of this paper [Cutler et al. 2020].

Theorem 4.2 (Compositionality). *If* $\Gamma, x : T_1 \vdash E : T_2$, *and* $\Gamma \vdash E' : T_1$, *then* $[\![\Gamma \vdash E[E'/x] : T_2]\!] = \left(1_{[\![\Gamma]\!]}, [\![\Gamma \vdash E' : T_1]\!]\right); [\![\Gamma, x : T_1 \vdash E : T_2]\!]$

Theorem 4.3 (Soundness (Terms)). *If* $\Gamma \vdash E : T$, *then* $[\![\Gamma \vdash E : T]\!] \in \text{Hom}\,([\![\Gamma]\!], [\![T]\!])$

Theorem 4.4 (Soundness (Inequality)). *If* $\Gamma \vdash E \leq E'$, *then for all* $\gamma \in [\![\Gamma]\!]$, $[\![\Gamma \vdash E : T]\!](\gamma) \leq [\![\Gamma \vdash E' : T]\!](\gamma)$

$$
\begin{aligned}
[\![ \|\mathtt{inc}\|_p ]\!] = \lambda\gamma.\lambda bs.\mathtt{slrec}(&\lambda\gamma.\lambda z.(\boxed{2}, 1),\\
&\lambda\gamma.\lambda p.\mathtt{case}(\lambda x.(\boxed{2}, 1 + \pi_1\pi_2 p),\\
&\qquad \lambda x.(\boxed{\pi_1\pi_2\pi_2 p}, 1 + \pi_2\pi_2\pi_2 p)\\
&\qquad )(\gamma', \pi_1\pi_1 p)\\
&\qquad \text{where } \gamma' = ((\gamma, p), (\pi_1 p, ((0, \pi_1\pi_2 p), \pi_2\pi_2 p)))\\
&)(\gamma, bs)\\
[\![ \|\mathtt{set}\|_p ]\!] = \lambda\gamma.(0, \lambda n.\mathtt{snrec}(&\lambda\gamma'.\lambda x.(\boxed{0}, 0)\\
&\lambda\gamma.\lambda p.(\boxed{\pi_1\pi_2 p + \pi_2([\![ \|inc\|_p ]\!]()(\pi_2\pi_2 p))}, \pi_2([\![ \|inc\|_p ]\!]()(\pi_2\pi_2 p)))))
\end{aligned}
$$

Fig. 11. Binary Counter Recurrences Interpreted

## 4.2 Binary Counter Conclusion

We interpret the binary counter recurrences from Figure 9 in preorders by unfolding the definitions in Figure 10; the result is shown in Figure 11. For the function `inc`, this yields a monotone map $[\![ \|\mathtt{inc}\|_p ]\!] \in \mathrm{Hom}(1, \mathbb{N} \to \mathbb{Z} \times \mathbb{N})$, which is (essentially) a function from an input list size to the cost of evaluation and the length of the output. For the function `set`, this yields a monotone map $[\![ \|\mathtt{set}\| ]\!] \in \mathrm{Hom}(1, \mathbb{Z} \times (\mathbb{N} \to \mathbb{Z} \times \mathbb{N}))$, which is a pair of a cost (the cost of evaluating the function definition — 0 since `set` is a value) and a function from input size to the cost of evaluation and the length of the output.

We have boxed the parts of the term that are related to computing the cost. The boxed portions of `inc` express that its amortized cost is 2 on the empty list (to create a 1 bit with a credit), is 2 when the bit is 0, and is exactly the same number of steps as the recursive call when the bit is 1. The boxed portions of `set` express that for zero it costs 0, and for successor it costs the recursive call plus the cost of `inc` on the potential of the output of the recursive call. However, because we will show that `inc` turns out to be constant amortized time, we do not need to bound the potential of the output of `set`. Intuitively, to see that `inc` has constant amortized time, observe that the `slrec` will always supply the $\infty$ bit as the head of the list, which by definition of the coalesced sum is both true and false, so the case is effectively the maximum of 2 and $\pi_1\pi_2\pi_1 p$. Thus, we effectively have recurrence where $T_{\mathtt{inc}}(0) = 2$ and $T_{\mathtt{inc}}(n) = 2 \vee T_{\mathtt{inc}}(n-1)$, which solves to $T(n) = 2$ by induction. Substituting this into the recurrence for `set`, we have essentially $T_{\mathtt{set}}(0) = 0$ and $T_{\mathtt{set}}(n) = T_{\mathtt{set}}(n-1) + 2$, which is of course $O(n)$. More formally, we can show by induction that for all $n \geq 0$, $([\![ \|\mathtt{inc}\|_p ]\!]()(n))_c \leq 2$, and that for all $n$, $([\![ \|\mathtt{set}\|_p ]\!]()(n))_c \leq 2n$, establishing bounds on these recurrences in this denotational semantics in preorders.

By the bounding theorem (Corollary 3.9), we have that, for the true operational cost $m$ of evaluating $\mathtt{set}(n) \downarrow^{(m,r)} v$, we have $m \leq_{\mathbb{C}} \|\mathtt{set}\|_p(n)_c$ in terms of the syntactic preorder judgment in $\lambda^{\mathbb{C}}$. By the soundness of the interpretation in preorders (Theorem 4.4), we have that $m \leq_{\mathbb{Z} \sqcup \infty} [\![ \|\mathtt{set}\|_p ]\!]()(n)_c$ in the preorder model. Therefore, by transitivity, we have $m \leq 2n$ in the preorder model, so our technique proves that the true operational cost $m$ of setting the binary counter to $n$ is in fact $O(n)$, as desired.

## 5 VARIABLE-CREDIT EXTENSION

The version of $\lambda^A$ described thus far supports amortized analyses where the amount of credit stored on each element of a data structure is fixed (e.g. `List` $(!_2 A)$ is a list with 2 credits on each element). However, in some important amortized analyses, different amounts of credit must be stored in different parts of a data structure—e.g. for balanced binary search trees implemented via splay trees [Sleator and Tarjan 1985], the number of credits stored on each node is a function of the size of the subtree rooted at that node. In this section, we show that adding existential quantification

$$\frac{\Delta|\Gamma \vdash_f M : A[c/\alpha] \quad \Delta,\alpha \vdash A \quad \Delta \vdash c\ \texttt{credit}}{\Delta|\Gamma \vdash_f \mathsf{pack}_{\alpha=c} M : \exists\alpha.A} \qquad \frac{M \downarrow^{(n,r)} v}{\mathsf{pack}_{\alpha=\ell} M \downarrow^{(n,r)} \mathsf{pack}_{\alpha=\ell} v}$$

$$\frac{\Delta|\Gamma \vdash_f M : \exists\alpha.A \quad \Delta,\alpha|\Gamma, x : A \vdash_{g+x} N : C \quad \Delta \vdash C}{\Delta|\Gamma \vdash_{f+g} \texttt{unpack}\ (\alpha, x) = M\ \texttt{in}\ N : C} \qquad \frac{M \downarrow^{(n_1,r_1)} \mathsf{pack}_{\alpha=\ell} v_1 \quad N[\ell/\alpha, v_1/x] \downarrow^{(n_2,r_2)} v}{\texttt{unpack}\ (\alpha, x) = M\ \texttt{in}\ N \downarrow^{(n_1+n_2, r_1+r_2)} v}$$

Fig. 12. Extension of $\lambda^A$ with existential types

over credit amounts to $\lambda^A$ suffices to analyze such examples, using a portion of splay trees as an example. Using existentials, a value of type $\exists\alpha.!_\alpha A$ is a value of type $A$ which carries $\alpha$ credits, for some $\alpha$; for example, a tree whose elements are of type $\exists\alpha.!_\alpha\mathbb{N}$ stores a variable number of credits with the number on each node. In keeping with our methodology of doing as much of an analysis as possible in the recurrence language and its semantics, the fact that a particular piece of code uses existentials to implement a desired credit policy will not be tracked by the type system, but proved after recurrence extraction. An alternative approach would be to enrich $\lambda^A$ with some form of indexed or dependent types to track the sizes of data structures in the type system, but such an extension is not necessary for our approach. The proofs of the results in this section can be found in the full version of this paper [Cutler et al. 2020].

## 5.1 Existential Types in $\lambda^A$

To support existential quantifiers over credits, we extend the main typing judgment to be one of the form $\Delta|\Gamma \vdash_f M : A$, where $\Delta = \alpha_1, \ldots, \alpha_n$ is a list of "credit variables". Any of the $\alpha_i$ can occur free in the types in $\Gamma$, the resource term $f$, the term $M$, or the type $A$. Credit variables $\alpha$ range over *credit terms* $c$, which are (finite) sums of credit variables like $\alpha, \beta$ and credit constants $\ell$ — i.e. $\alpha_1 + \alpha_2 + \ldots + \alpha_n + l$. We write $\Delta \vdash c\ \texttt{credit}$ to mean that a credit term is well-formed from the variables in $\Delta$. We consider credit terms up to the usual equations for addition on natural numbers. These credit terms can then be used as the "bank" in resource terms: the resource term $3x + 2y + (\alpha + 2)$ describes a context where one can use $x$ 3 times, $y$ twice, and has access to the credit term $\alpha + 2$ credits. Most importantly, credit terms are now allowed to appear in the subscript of the ! modality (generalizing the natural number constants $\ell$ allowed above): a term $\alpha \mid \Gamma \vdash_f M :!_\alpha A$ with is an $A$ with $\alpha$ credits attached. We add a new type $\exists\alpha.A$ for existentially quantifying over credit variables. A value of type $\exists\alpha.A$ is a value of type $A[c/\alpha]$, for some credit term $c$. Such a value does not store the ability to *use* the credits $c$ — it stores a number of credits itself. However, combining the existential with the ! modality, a value of type $\exists\alpha.!_\alpha A$ is an $A$ with $c$ credits attached, for some credit term $c$. The operational semantics is defined for terms with no free credit variables, so its structure remains unchanged.

The typing rules and operational semantics for existential types are presented in Figure 12. The terms for existentials are standard pack/unpack terms. The operational semantics of pack and unpack are also standard; because we only evaluate closed terms, the credit term being packed/unpacked with the value will always be a (closed) natural number $\ell$.

The rest of the rules for $\lambda^A$ are mostly unchanged, so we do not repeat them: they are obtained from the rules in Figure 3 by carrying the credit variable context $\Delta$ through all of the rules, and, in the $!^k_c$ modality and the save, transfer, create, and spend terms, the natural number constants $\ell$ are generalized to credit terms $c$ constructed from these variables. Finally, since the resource terms may contain free credit variables, the ordering judgment on resource terms must be augmented with a credit variable context, and the ordering itself extended to contain the coefficient-wise ordering on credit variables. The operational semantics for these constructs in unchanged, because closed credit terms are precisely the credit values $\ell$ used above.

$$\begin{array}{rcl}
\langle\!\langle \exists\alpha.A \rangle\!\rangle & = & \$ \times \langle\!\langle A \rangle\!\rangle \\
\|\mathsf{pack}_{\alpha=c} M\| & = & (\|M\|_c \,, (c, \|M\|_p)) \\
\|\mathtt{unpack}\ (\alpha, x) = M\ \mathtt{in}\ N\| & = & \|M\|_c +_c \|N\| \, [\pi_1 \|M\|_p / \alpha, \pi_2 \|M\|_p / x] \\
\|\mathsf{create}_c\ M\| & = & (\mathrm{to}\mathbb{C}(c) + \|M\|_c \,, \|M\|_) \\
\|\mathsf{spend}_c\ M\| & = & (-\mathrm{to}\mathbb{C}(c) + \|M\|_c \,, \|M\|_p)
\end{array}$$

Fig. 13. Recurrence extraction for credit existentials

For this extension, substitution and type preservation are stated as follows:

Theorem 5.1 (Substitution).

- *If* $\Delta \vdash c\ \mathtt{credit}$ *and* $\Delta, \alpha \vdash c'\ \mathtt{credit}$, *then* $\Delta \vdash c'[c/\alpha]\ \mathtt{credit}$
- *If* $\Delta \vdash c\ \mathtt{credit}$ *and* $\Delta, \alpha | \Gamma \vdash_f M : A$, *then* $\Delta | \Gamma[c/\alpha] \vdash_{f[c/\alpha]} M[c/\alpha] : A[c/\alpha]$

Theorem 5.2 (Preservation). *If* $\cdot | \cdot \vdash_a M : A$ *and* $M \downarrow^{(n,r)} v$, *then* $a + r \geq 0$ *and* $\cdot | \cdot \vdash_{a+r} v : A$.

## 5.2 Extracting Recurrences for Existentials

Recall that the recurrence extraction in Figure 7 erases the $!^k_\ell A$ modalities and translates $\mathsf{create}_\ell\ M$ and $\mathsf{spend}_\ell\ M$ by adding/subtracting $\ell$ to/from the amortized cost. Since we now allow credit variables $\alpha$, such as those coming from unpacking an existential type, in the credit position of create/spend, the recurrence extraction will need to refer to the values chosen for $\alpha$ in order to know how much to add/subtract to/from the amortized cost. Thus, we add a type \$ to the recurrence language, the values of which are numbers of credits, represented by natural numbers. The credit context $\Delta$ is translated to recurrence language variables of type \$ (i.e. $\langle\!\langle \Delta, \alpha \rangle\!\rangle = \langle\!\langle \Delta \rangle\!\rangle, \alpha : \$$), while existential types $\exists\alpha.A$ are translated to pairs $\$ \times \langle\!\langle A \rangle\!\rangle$. A simple pair suffices because the ! modality is erased by $\langle\!\langle \cdot \rangle\!\rangle$, and this is the only place where credit terms can occur in the syntax of types, so all occurrences of $\alpha$ under the binder are removed, and $\langle\!\langle A \rangle\!\rangle$ is a closed type.

We show the new and changed cases of recurrence extraction in Figure 13. The introduction and elimination rules for $\exists\alpha.A$ translate to the corresponding introduction and elimination forms for $\$ \times \langle\!\langle A \rangle\!\rangle$. For create and spend, in principle, we would like the cost component of $\mathsf{create}_c\ M$ to be $c + \|M\|_c$, but this will not type check, given that $c : \$$ but $\|M\|_c : \mathbb{C}$. Recalling that costs $\mathbb{C}$, though axiomatized as a monoid with some operations, are morally integers, we add a coercion $\mathrm{to}\mathbb{C} : \$ \to \mathbb{C}$, which is morally the inclusion of natural numbers into integers.

Theorem 5.3 (Extraction Preserves Types). *If* $\Delta | \Gamma \vdash_f M : A$, *then* $\langle\!\langle \Delta \rangle\!\rangle, \langle\!\langle \Gamma \rangle\!\rangle \vdash \|M\| : \|A\|$

## 5.3 Bounding Relation and Bounding Theorem

The definition of the bounding relation for values (Definition 3.2) is extended with

- $\mathsf{pack}_{\alpha=\ell} v \sqsubseteq^{\exists\alpha.A, a}_{\mathtt{val}} E$ iff $\ell \leq_\$ \pi_1 E$ and $v \sqsubseteq^{A[\ell/\alpha], a}_{\mathtt{val}} \pi_2 E$

Recalling that $E : \langle\!\langle \exists\alpha.A \rangle\!\rangle = \$ \times \langle\!\langle A \rangle\!\rangle$, this simply states that the amount of credit packed by $\alpha$ is bounded by the amount described by $\pi_1 E$, and that the value packed with the credit amount is in fact bounded by $\pi_2 E$. We remark that this definition may give the careful reader pause– inducting on a substitution instance of an existential type where the existential variable ranges over *types* leads to well-definedness issues. But, our existential variables range over *credits*, so we may simply regard a closed substitution instance of a type $\alpha \vdash A$ type as a smaller type than $A$.

The definition of the bounding relation for open terms must also be modified to quantify over closing substitutions for the credit context, as well as the term context. First, if $\omega$ is a substitution of credit amounts $\ell$ for credit variables, and $\Omega$ is a substitution of closed terms of type \$ for recurrence language variables, then $\omega \sqsubseteq^\Delta \Omega$ means that for all $\alpha \in \Delta$, $\omega(\alpha) \leq_\$ \Omega(\alpha)$. Then for $\Delta | \Gamma \vdash_f M : A$ we

$$\begin{array}{rcl} N_1 & = & \lambda\_.\mathsf{pack}_{\alpha=0}(\mathsf{save}^1_0\ ()) \\ N_2 & = & \lambda(\_, (\alpha, \mathsf{save}^\alpha_1\ ())).\mathsf{create}_1\ (\mathsf{pack}_{\beta=\alpha+1}\mathsf{save}^{\alpha+1}_1\ ()) \\ \mathsf{spawn}(n) & = & \mathsf{nrec}\left(n, N_1, \mathsf{save}^\infty_0\ N_2\right) : \exists\alpha.!^1_\alpha 1 \end{array}$$

Fig. 14. $\lambda^A$ term for the spawn function

write $M \sqsubseteq^A E$ if for all $\omega \sqsubseteq^\Delta \Omega$ and for all $\theta \sqsubseteq^{\Gamma[\omega],\sigma} \Theta$, we have that $M[\omega,\theta] \sqsubseteq^{A[\omega],f[\omega,\sigma]} E[\Omega,\Theta]$. Using this notation, the bounding theorem is

Theorem 5.4 (Bounding Theorem). *If* $\Delta|\Gamma \vdash_f M : A$*, then* $M \sqsubseteq^A \|M\|$

and the cases which differ from the original Theorem 3.8 are proved in the supplementary materials.

## 5.4 Splay Tree Analysis

We now describe somewhat informally how to use the above machinery to analyze splay trees; the complete formalism is given in the full version of this paper [Cutler et al. 2020]. Following Okasaki's presentation [Okasaki 1998], the key operation is a `split` : $(A \times \mathtt{tree}\ (A)) \to \mathtt{tree}\ (A) \times \mathtt{tree}\ (A)$ function that splits a given tree into elements larger and smaller than a given pivot. Insertion, deletion, union, intersection, difference etc. can be all implemented from `split` and a `join` operation that combines two sorted trees where all the elements of the first are less than the elements of the second. Showing that `split` is amortized $O(\log n)$ time, where $n$ is the size of the tree, is the most difficult part of the amortized analysis, and implies the desired time bounds for the other operations. The key idea of splay trees is that each access rearranges the tree so that accessing the same element twice in a row is quicker the second time. In Okasaki's presentation, this rearrangement takes place in `split`, which performs a series of tree rotations. These rotations ensure that the amortized cost of `split` (amortized over any sequence of binary search tree operations) is $O(\log n)$, even though the tree is not always balanced. The most challenging cases of the code unpack the tree to depth two, and rotate the output if they traverses the same direction twice while searching for the pivot:

$$\begin{array}{l} split\ p\ (N(x, N(y, a_{11}, a_{12}), N(z, a_{21}, a_{22})))|\ x \geq p\ \&\&\ y \geq p = \\ \quad (small, N(y, big, N(x, a_{12}, N(z, a_{21}, a_{22}))))\ \ \mathtt{where}\ \ (small, big) = \mathtt{split}\ p\ a_{11} \end{array}$$

Okasaki's analysis of split maintains the invariant that there are $\varphi(t) = \lceil \lg(|t| + 1) \rceil$ credits associated with the root of every subtree $t$ in a splay tree, and uses the potential/physicists method to analyze the amortized cost.

The addition of existentials to $\lambda^A$ allows us to encode this analysis, by giving `split` the type $A \otimes \mathtt{tree}\left(\exists\alpha.!^1_\alpha A\right) \multimap \mathtt{tree}\left(\exists\alpha.!^1_\alpha A\right) \otimes \mathtt{tree}\left(\exists\alpha.!^1_\alpha A\right)$, and using code to maintain the invariant that each of these $\alpha$'s are precisely $\varphi(t)$.

*5.4.1 Creating Variable Amounts of Credit.* To maintain this invariant, we will sometimes need to `create` amounts of credit determined by a run-time natural number, like $\varphi(t)$ for some tree $t$—but the primitive $\mathtt{create}_c\ M$ term allows for waiting only for a credit term $c$, which cannot depend on run-time values. However, we can write a recursive loop that spawns a number of credits dependent on a run-time value, and package this as a function $\mathsf{spawn} : \mathbb{N} \multimap \exists\alpha.!^1_\alpha 1$ such that the $\alpha$ packed in the result of $\mathsf{spawn}(n)$ is (the credit term representing) $n$. The implementation of spawn is shown in Figure 14—at a high level, the term loops $\mathtt{create}_1$ in a $\mathbb{N}$-recursor, using a credit existential as a counter variable. In this example, and throughout this section, we use pattern-matching notation as syntactic sugar for the elimination rules for positive types like $\exists, !, \otimes$, with the convention that matching on the result of a thunked recursive call implicitly forces it.

In Section 2.2, we argued that the $n$ component in the operational cost semantics $M \downarrow^{n,r} v$ captures the actual operational cost of an erasure to simply-typed $\lambda$-calculus, as long as `ticks` in $\lambda^A$ are inserted for each STLC $\beta$-redex. Because we do not include any `tick` terms in `spawn`, its abstract operational cost $n$ is zero. Thus, to realize this cost semantics, `spawn` must be erased before actually running the program. Fortunately, a simple program optimization suffices to do this: translate $\lambda^A$ to simply-typed $\lambda$-calclus by dropping both the $\exists$ and ! types and the associated terms, at which point `spawn` has type $\mathbb{N} \to 1$; then replace all terms of type 1 with the trivial value. That is, we think of `spawn` as a *ghost loop* — code that is meant for the extracted recurrence, but not intended to actually be run.

*5.4.2 Definition of Trees in $\lambda^A$.* Extending $\lambda^A$ with the requisite `tree` type constructor and its rules follows both previous work [Danner et al. 2015] and the pattern illustrated with lists above. The type of trees is essentially $\texttt{tree}\,(A) = \texttt{Emp} \mid \texttt{N of } A \otimes \mathbb{N} \otimes \texttt{tree}\,(A) \otimes \texttt{tree}\,(A)$. The $\mathbb{N}$ argument caches the size of the tree, making the function $\texttt{size} : \texttt{tree}\,(A) \multimap \mathbb{N} \otimes \texttt{tree}\,(A)$ — which projects out that field and then rebuilds the tree[5] — constant time. To support coding the `split` function described above, we directly add a recursor that performs a two-level pattern match, with cases for the empty tree, for a node with one child or the other empty and the other is another node, and for a node with two nodes as children; in the latter case, the recursor provides recursive calls on all four subtrees.

*5.4.3 Splay Tree Implementation.* We define a *splay tree* to be a binary search tree $t : \texttt{tree}\,(\exists\alpha.!^{\infty}_{\alpha}A)$ satisfying the property that if $\texttt{size}(t) = n$, then if $t = N(\_, m, t_0, t_1)$, then $t_0$ and $t_1$ are splay trees, and for $[\![\|t\|_p]\!] = N((\alpha, \_), \_, \_)$, we have $\alpha = \phi(n)$. In other words, the credit invariant holds at each node in the tree. We note that each element of the tree not only carries $\alpha$ credits, but is also infinitely usable since we are required to compare nodes in the tree more than constantly many times. This causes no issues for the extracted recurrences, because keys in the tree are always values. We then prove a lemma which states that `split` preserves the splay tree property — i.e. that the existentially quantified credits stored in the tree satisfy the desired invariant.

Lemma 5.5. *If $t : \texttt{tree}\,(\exists\alpha.!^{\infty}_{\alpha}A)$ is a splay tree and $\texttt{split}(t) \downarrow (t_0, t_1)$, then $t_0$ and $t_1$ are also splay trees.*

To illustrate the $\lambda^A$ term for `split`, we show one key case of the recursor, which corresponds to the snippet given at the beginning of this section and to [Okasaki 1998, Theorem 5.2]. For this case, we are in the situation where the root, labeled by $x$, has two subtrees, $y$ with subtrees $a_{11}, a_{12}$, and $z$ with subtrees , $a_{21}, a_{22}$. If the pivot is less than both $x$ and $y$, we recur on the leftmost subtree $a_{11}$, which produces the elements of $a_{11}$ that are smaller and bigger than the pivot. Then *smaller* contains all the elements of the original tree smaller than the pivot. The elements bigger than the pivot are *bigger* and everything else from the original tree; we combine these together into a new tree, performing a rotation to put $y$ at the root.

The $\lambda^A$ version of this term, presented in Figure 15, annotates the above code with some additional information about the sizes of trees, and with some code for manipulating credits. The variables $x, y, z$ are the values of type $A$ at the root and its immediate children; these come with existentially-quantified numbers of credits $\alpha, \beta, \gamma$ ($\alpha$ credits are stored with $x$, $\beta$ with $y$, and $\gamma$ with $z$), and also with natural numbers caching the sizes of the subtrees that they are the roots of ($n_1, n_2, n_3$ respectively). The variables $a_{ij}$ stand for the four subtrees with their (suspended) recursive call outputs; we write $\texttt{split}(p, a_{11})$ for projecting and forcing the recursive call, and write $a_{ij}$ for projecting the other subtrees. The credit manipulation involves spending the credits $\alpha$ and $\beta$ stored with $x$ and $y$ in the input tree (we do not spend $z$, because the $z$ node is left unchanged in the

[5]The tree can be rebuilt because values of type $\mathbb{N}$ are duplicable— there is a diagonal map $\mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$. Also, we will often use `size` as a function $\texttt{tree}\,(A) \multimap \mathbb{N}$, and silently contract the second projection for re-use of the argument.

```
$\lambda((\alpha, \mathsf{save}^{\infty}_{\alpha}\ x), n_1, (\beta, \mathsf{save}^{\infty}_{\beta}\ y), n_2, (\gamma, \mathsf{save}^{\infty}_{\gamma}\ z), n_3, a_{11}, a_{12}, a_{21}, a_{22}).$
if $x \geq p$ && $y \geq p$ then tick; let
      $(small, big)$         $= \mathsf{spend}_{\alpha+\beta}\ (\mathsf{split}(p, a_{11}))$              $d$      $= N(\mathsf{pack}_{\alpha=\gamma}(\mathsf{save}^{\infty}_{\gamma}\ z), n_3, a_{21}, a_{22})$
      $n_{12}$               $= \mathsf{size}(a_{12})$                                  $n_{big}$ $= \mathsf{size}(big)$
      $t'_{size}$            $= 1 + n_{12} + n_3$                                   $s'_{size}$ $= 2 + n_{big} + n_{12} + n_3$
      $((\alpha', \_), (\beta', \_))$ $= (\mathsf{spawn}(\varphi(t'_{size})), \mathsf{spawn}(\varphi(s'_{size})))$   $t'$ $= N(\mathsf{pack}_{\alpha=\alpha'}(\mathsf{save}^{\infty}_{\alpha'}\ x), t'_{size}, a_{12}, d)$
      $s'$                   $= N(\mathsf{pack}_{\alpha=\beta'}(\mathsf{save}^{\infty}_{\beta'}\ y), s'_{size}, big, t')$
   in $(small, s')$ end
else...
```

Fig. 15. Part of the $\lambda^A$ term for split

output), calculating the sizes of the new nodes $t'$ and $s'$ that will be part of the output, and spawning credits corresponding to $\varphi$ of these sizes. The term presented in Figure 15 is one branch of one of the step functions passed to the `treerec` which forms the outermost structure of `split`.

To analyze splay trees, we pass this $\lambda^A$ term through recurrence extraction and the preorder semantics and then prove the following:

Theorem 5.6. *If* $t$ : tree $(\exists\alpha.!^{\infty}_{\alpha}A)$ *is a splay tree with* size$(t) = n$, *then for any* $v : A$, $[\![\|\mathsf{split}(t, v)\|_c]\!] \leq 1 + 2\varphi([\![\|\mathsf{size}\|_p\ (t)]\!]) \in O(\lg n)$.

Proof. As an example, we show the case for the code in Figure 15. The cost component of the extracted recurrence is

$$1 - \alpha - \beta + [\![\|\mathsf{split}(p, a_{11})\|]\!] + \varphi(1 + n_{12} + n_3) + \varphi(2 + n_{big} + n_{12} + n_3)$$

The 1 comes from the tick; $\alpha$ and $\beta$ are subtracted because they are spent; and the $\varphi$ of the sizes of $t'$ and $s'$ are added because they are created. By definition, $1 + n_{12} + n_3 = [\![\|\mathsf{size}\|_p\ (t')]\!]$ and $2 + n_{big} + n_{12} + n_3 = [\![\|\mathsf{size}\|_p\ (s')]\!]$. By the credit invariant, $\alpha = \varphi([\![\|\mathsf{size}\|_p\ (t)]\!])$, and $\beta = \varphi([\![\|\mathsf{size}\|_p\ (s)]\!])$, where $s$ is the subtree of $t$ rooted at $y$. Rewriting by these and commuting terms, the extracted recurrence is precisely

$$1 + [\![\|\mathsf{split}(p, a_{11})\|]\!] + \varphi([\![\|\mathsf{size}\|_p\ (s')]\!]) + \varphi([\![\|\mathsf{size}\|_p\ (t')]\!]) - \varphi([\![\|\mathsf{size}\|_p\ (s)]\!]) - \varphi([\![\|\mathsf{size}\|_p\ (t)]\!])$$

which Okasaki [Okasaki 1998, Theorem 5.2] proves is bounded by $1 + 2\varphi(\mathsf{size}(t))$, as required. □

## 6 RELATED WORK

Techniques for extracting (asymptotic) cost information from high-level program source code is a project that is almost as old as studying programming languages. For non-amortized analysis of functional languages, we have examples from the 1970s and 1980s by Wegbreit [1975], Le Métayer [1988], and Rosendahl [Rosendahl 1989]. The idea of simultaneously extracting information about cost and size, and defining the size of a function to be a function itself (leading to higher-order recurrences) has its roots in Danner and Royer [2007], which in turn draws from ideas in Shultis [1985], Sands [1990], and Stone [2003]. Using bounded modal operators to describe resource usage goes back at least to Girard et al. [1992], and Orchard et al. have recently incorporated these ideas into the Granule language [Orchard et al. 2019]. Perhaps the work that is closest in spirit to ours is Benzinger's ACA system for analyzing call-by-name Nuprl programs [Benzinger 2004]. From a cost-annotated operational semantics, he extracts a "symbolic semantics" that is similar in flavor to our recurrence language and extracted recurrences, although without amortization. The symbolic semantics yields higher-order recurrences, which he reduces to first-order recurrences that can be analyzed with a computer algebra system.

There is also extensive work on recurrence extraction from first-order imperative languages. The COSTA project [Albert et al. 2011, 2012, 2013] takes Java bytecode as its source language, extracts cost relations (essentially, non-deterministic cost recurrences), and solves them for upper bounds. In this line of work, Alonso-Blas and Genaim [2012] and Flores-Montoya [2016] investigate the failure to derive tight upper bounds in settings where amortized analysis is typically deployed. They trace the issue to the fact that typically cost relations do not depend on the results of the analyzed functions. Making this possible allows more precise constraints which, when solved, yield tighter bounds. The dependency on output corresponds roughly to total accumulated savings, and they infer an appropriate potential function (in the terminology of the physicist's method), modulo a choice of templates. To analogize with our work, they delay the determination of the credit policy until solving for upper bounds of extracted recurrences, whereas we specify the credit policy as part of the source program, which directly yields a recurrence for cost that takes the policy into account.

Two recent approaches that handle amortized analysis for functional programs are Timed ML (TiML, [Wang et al. 2017]) and automatic amortized resource analysis (AARA, [Hoffmann et al. 2012, 2017; Hoffmann and Shao 2015; Niu and Hoffmann 2018]). In TiML, ML type and function definitions are annotated with indices that convey size information. The notion of size is left unspecified and the indices are very flexible, and can include constraints such as those required to define red-black trees. Type inference generates verification conditions. Depending on the details of the annotations, solving the verification conditions provides exact or asymptotic bounds on the cost of the original program. The focus is on worst-case analysis, but the annotation language is sufficiently rich to encode the physicist's method of amortized analysis. Although it is not part of their focus, the formalism does not appear to enable analysis of higher-order functions whose cost depends on the complexity behavior of the function arguments.

AARA provides a type inference system for resource bound analysis of higher-order functional programs that incorporates amortization. Credit allocation is built into the type system itself. Soundness says that the net credit change during evaluation is bounded by the net credit change described by the typing. AARA focuses primarily on strict languages, but Jost et al. [2017] use similar ideas to analyze programs under lazy evaluation. In AARA, the credit allocation and usage is described in the type judgment. Type inference generates constraints, and the solution of these constraints is essentially a credit allocation strategy. Our approach describes usage in the type judgment, but requires the strategy to be explicit in the program (via `save`, `create`, `spend`, etc.), which places a greater burden on the programmer. However, reasoning about that strategy (e.g., establishing a credit invariant) in the semantics may provide more flexibility, though that requires more investigation.

We note that the technical differences between TiML and AARA and our approach arise from a difference in what we might consider the philosophical underpinnings. TiML and AARA introduce novel type systems with a goal of inferring cost bounds to the greatest extent possible. Those bounds are extracted as part of the type inference procedure. This is not how most programmers conceptualize a cost analysis, and our interest is in staying as close to typical informal analyses as we can. While $\lambda^A$ is a novel type system, the novelty exists solely in order to make the programmer be explicit about how credits are allocated and used. This task is part of a banker's-method analysis, though it is usually stated informally ("put one credit on each 1 in the bit list"). After that, it is extraction of ordinary (semantic) recurrences which one hopes to be able to bound using whatever methods are at the programmer's disposal.

## 7 FUTURE WORK

We expect that the techniques used in Kavvos et al. [2019] to handle general recursion in the source language can be adapted to the approach we have taken here to handle amortization, though work remains to be done to see whether typical non-structurally recursive amortized algorithms would satisfy the necessary typing constraints. A useful project would then be to do the analyses that Okasaki [1998] describes via recurrence extraction, which focus on amortized cost of sequences of arbitrary data structure operations (e.g., typical usage of a functional queue), where the data structure is used ephemerally. Adding a type for memoizing thunks to the source language, or more generally lazy evaluation and datatypes, would permit analysis of persistent usage.

Recalling our “big-picture” goal of formalizing as closely as possible the process by which programmers actually perform cost analyses, we are not there yet. Let us consider how an analysis of the usual splay-tree implementation of the abstract set type actually proceeds. First we define the splay-tree implementation, and then reason about its operations *in isolation* to conclude that, provided the it is used ephemerally, the amortized cost of those operations is $O(\lg n)$, where $n$ is the size of the tree. We then would typically analyze an algorithm that uses the abstract set type ephemerally *under the assumption that the set operations are* $O(\lg n)$*-time,* where $n$ is the size of the set. In other words, the analysis of the data structure (which may use techniques such as amortized analysis, and depends crucially on the structure of the tree) is separated from the analysis of the program that uses the interface that it implements (which uses only information about the size of the set). In the context of what we have presented here, while the splay-tree type would be something like our $\texttt{tree}\,(\exists\alpha.!^{\infty}_{\alpha}A)$, the programs that use it would use an abstract $\texttt{set}\,(A)$ type, and in particular the abstract type would not refer to the type constructors we have introduced to manage credits. Codifying this would require something like abstract type declarations (or more generally existential types as in [Mitchell and Plotkin 1985]), but where the denotation of the abstract type (corresponding to the abstract notion of size) is not the same as that of the concrete type of the implementation. This is an ongoing project.

We have neglected any discussion of automating either the front end of this process (annotating a program with the constructs used for amortization) or the back end (automatic solving of recurrences). Fully automating the annotation of a source program may be too much to ask (amortized analysis is hard!), but one could hope for a process that elaborates a program with higher-level annotations (e.g. written as comments) into $\lambda^{A}$, inserting `create`, `spend`, etc. On the back end, the syntactic inequality judgment from Figure 8 can be used to simplify recurrences in $\lambda^{\mathbb{C}}$ as opposed to interpreting into a model of the recurrence language and then simplifying there. Ideally, one could add enough rules to the judgment (and perhaps enrich its structure) to be able to simplify a large class of standard recurrences, and then apply proof search techniques to automate the process. We would still have higher-order recurrences, and it would be worthwhile to see if the techniques used by Benzinger [Benzinger 2004] can be used to reduce them to first-order recurrences that could be solved by a recurrence solver such as OCRS [Kincaid et al. 2017].

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous ICFP reviewers for their very thoughtful feedback. We also thank the reviewers from ESOP for suggesting an analysis of splay trees as a way to demonstrate the expressiveness of $\lambda^{A}$. Finally, the first author would like to thank Alex Kavvos for numerous clarifying conversations during the course of this work.

This material is based upon work supported by the National Science Foundation under Grant Number CCF-1618203, the Air Force Office of Scientific Research under award number FA9550-16-1-0292, and the United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government or Carnegie Mellon University.

## REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning* 46 (2011), 161–203. Issue 2. https://doi.org/10.1007/s10817-010-9174-1

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science* 413, 1 (2012), 142–159. https://doi.org/10.1016/j.tcs.2011.07.009

Elvira Albert, Samir Genaim, and Abu Naser Masud. 2013. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic* 14, 3 (2013), 22:1–22:35. https://doi.org/10.1145/2499937.2499943

Diego Esteban Alonso-Blas and Samir Genaim. 2012. On the limits of the classical approach to cost analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Antoine Miné and David Schmidt (Eds.), Vol. 7460. Springer Berlin Heidelberg, 405–421. https://doi.org/10.1007/978-3-642-33125-1_27

Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018* (Oxford, United Kingdom). ACM Press, 56–65. https://doi.org/10.1145/3209108.3209189

Ralph Benzinger. 2004. Automated higher-order complexity analysis. *Theoretical Computer Science* 318, 1-2 (2004), 79–103. https://doi.org/10.1016/j.tcs.2003.10.022

Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis. arXiv:2006.15036

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, George Necula and Philip Wadler (Eds.). ACM Press, 133–144. https://doi.org/10.1145/1328438.1328457

Norman Danner and Daniel R. Licata. 2020. Denotational semantics as a foundation for cost recurrence extraction for functional languages. arXiv:2002.07262v1

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015* (Vancouver, BC, Canada), Kathleen Fisher and John Reppy (Eds.). ACM Press, 140–151. https://doi.org/10.1145/2784731.2784749

Norman Danner, Jennifer Paykin, and James S. Royer. 2013. A static cost analysis for a higher-order language. In *Proceedings of the 7th workshop on Programming languages meets program verification, PLPV 2013*, Matthew Might and David Van Horn (Eds.). ACM Press, 25–34. https://doi.org/10.1145/2428116.2428123

Norman Danner and James S. Royer. 2007. Adventures in time and space. *Logical Methods in Computer Science* 3, 9 (2007), 1–53. https://doi.org/10.2168/LMCS-3(1:9)2007

Antonio Flores-Montoya. 2016. Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations. In *FM 2016: Formal Methods (Lecture Notes in Computer Science)*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.), Vol. 9995. Springer International Publishing, 254–273. https://doi.org/10.1007/978-3-319-48989-6_16

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), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM Transactions on Programming Languages and Systems* 34, 3 (2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM Press, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Zhong Shao. 2015. Automatic static cost analysis for parallel programs. In *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015 (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer-Verlag, 132–157. https://doi.org/10.1007/978-3-662-46669-8_6

Martin Hofmann. 2002. The Strength of Non-Size Increasing Computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 260–269. https://doi.org/10.1145/503272.503297

Martin Hofmann. 2003. Linear Types and Non-Size-Increasing Polynomial Time Computation. *Information and Computation* 183, 1 (2003), 57–85. https://doi.org/10.1016/S0890-5401(03)00009-9

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Alex Aiken and Greg Morrisett (Eds.). ACM Press, 185–197. https://doi.org/10.1145/604131.604148

Hudson, Bowornmet. 2016. *Computer-Checked Recurrence Extraction for Functional Programs*. Master's thesis. Wesleyan University.

Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning* 59, 1 (2017), 87–120. https://doi.org/10.1007/s10817-016-9398-9

G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2019. Recurrence Extraction for Functional Programs through Call-by-Push-Value. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 15 (2019).

https://doi.org/10.1145/3371083

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-linear Reasoning for Invariant Synthesis. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 54 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158142

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Daniel Le Métayer. 1988. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems* 10, 2 (1988), 248–266. https://doi.org/10.1145/42190.42347

Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A fibrational framework for substructural and modal logics. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017 (Leibniz International Proceedings in Informatics (LIPIcs))*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 25:1–25:22. https://doi.org/10.4230/LIPIcs.FSCD.2017.25

Conor McBride. 2016. I got plenty o' Nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.), Vol. 9600. Springer-Verlag. https://doi.org/10.1007/978-3-319-30936-1_12

John C. Mitchell and Gordon D. Plotkin. 1985. Abstract types have existential types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1985*. 37–51. https://doi.org/10.1145/318593.318606

Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.), Vol. 57. EasyChair, 543–563. https://doi.org/10.29007/xkwx

Chris Okasaki. 1998. *Purely Functional Data Structures.* Cambridge University Press.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Jason Reed. [n.d.]. Names are (mostly) useless. ([n. d.]). https://www.cis.upenn.edu/~sweirich/wmm/wmm08-programme.html Presented at 3rd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory 2008.

Mads Rosendahl. 1989. Automatic complexity analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989*, Joseph E. Stoy (Ed.). ACM Press, 144–156. https://doi.org/10.1145/99370.99381

David Sands. 1990. *Calculi for Time Analysis of Functional Programs.* Ph.D. Dissertation. University of London.

Jon Shultis. 1985. *On the complexity of higher-order programs.* Technical Report CU-CS-288-85. University of Colorado at Boulder.

Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM* 32, 3 (1985).

Kathryn Van Stone. 2003. *A Denotational Approach to Measuring Complexity in Functional Programs.* Ph.D. Dissertation. Carnegie Mellon University.

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318. https://doi.org/10.1137/0606031

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 79 (2017), 26 pages. https://doi.org/10.1145/3133903

Ben Wegbreit. 1975. Mechanical program analysis. *Communications of the Association for Computing Machinery* 18, 9 (1975), 528–539. https://doi.org/10.1145/361002.361016