On Recursion in Graded Modal Type Theory

Citation for the original published paper (version of record):
Eriksson, O., Abel, A., Danielsson, N. (2026). On Recursion in Graded Modal Type Theory. Proceedings of the ACM on Programming Languages, 10(ICFP). http://dx.doi.org/10.1145/3828677

On Recursion in Graded Modal Type Theory

OSKAR ERIKSSON, University of Gothenburg and Chalmers University of Technology, Sweden
ANDREAS ABEL, University of Gothenburg and Chalmers University of Technology, Sweden
NILS ANDERS DANIELSSON, University of Gothenburg and Chalmers University of Technology, Sweden

We present a graded modal type theory with recursion over natural numbers and prove formally in Agda that it handles resources correctly, in the sense that an abstract machine accesses resources the "correct" number of times. The theory is parametrized, and can for instance be instantiated with grades for erasure, linear types, or affine types. The correctness proof shows that our usage counting is sound. Our eliminator for natural numbers is flexible as it enables different resource-usage patterns and practical in the sense that it can be used both to define functions with expected usage counts for the arguments. Further, it can be used to encode other data types, using large elimination. Finally, we adapt our resource correctness proof to show correctness also for grades tracking information flow, in the form of a non-interference property.

CCS Concepts: • **Theory of computation** → **Type theory**; **Logic and verification**; Linear logic.

Additional Key Words and Phrases: graded modal type theory, quantitative type theory, dependent types, abstract machine, linearity, formalization

ACM Reference Format:
Oskar Eriksson, Andreas Abel, and Nils Anders Danielsson. 2026. On Recursion in Graded Modal Type Theory. *Proc. ACM Program. Lang.* 10, ICFP, Article 279 (August 2026), 29 pages. https://doi.org/10.1145/3828677

1 Introduction

In graded modal type theories the type system is equipped with an algebraic structure, typically some form of semiring, containing *grades*. Grades can be used for data privacy [14, 23, 30], for memory safety [7, 21], to enforce correctness (for instance of communication protocols) [8, 11], and for quantitative aspects, where the number of times variables are being used are counted, like erasure, linear types and affine types [1, 3, 5, 7, 9–13, 19, 21–23].

In the case of quantitative aspects one might like to ensure that variables are referenced the "right" number of times during evaluation, as specified by an operational semantics. This approach is taken by Choudhury et al. [13], who define a heap-based abstract machine with the capability to count heap lookups, corresponding to variable uses, and prove a notion of correctness. On the other hand, in the case of data privacy one might want to prove non-interference, as is done by Choudhury et al. [14].

Our main goal is to find suitable "grade typing" rules for an eliminator for natural numbers. The difficulty with this is connected to recursion. For instance, in a quantitative setting it is difficult to accurately count uses of variables without knowing the depth of the recursion in advance. We have previously considered an eliminator for natural numbers in a graded system [3], but as we discuss in Section 5.1, that approach is problematic in a setting with linear types. Different approaches have

Authors' Contact Information: Oskar Eriksson, oskar.eriksson@gu.se; Andreas Abel, andreas.abel@gu.se; Nils Anders Danielsson, nad@cse.gu.se; Department of Computer Science and Engineering, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden.

ACM 2475-1421/2026/8-ART279
https://doi.org/10.1145/3828677

been presented by e.g. Atkey [6] and Nakov and Forsberg [24] but we argue that the eliminator we present here is more flexible in the sense that it supports multiple usage patterns. For instance, one can define functions that are linear in the natural number, but it is also possible to track linear uses of variables in the zero and successor branches. Our rule is derived from finding necessary conditions for the rule to be sound. We also believe that this approach is sufficiently systematic that it should be applicable to eliminators of other data types as well.

We are primarily interested in quantitative uses. In order to show that our approach to usage counting is correct we build on the graded type theory from our previous work [3] where we showed correctness of erasure, but left the question of correctness for other grade interpretations for future work. We now answer this question for a class of quantitative grade semirings by establishing an operational semantics which counts variable uses, following Choudhury et al. [13]. In particular, we show that the theory can be used to correctly encode information about linear or affine uses. We support large elimination and the empty type, which allows us to use our eliminator to encode other data types like booleans and vectors. While the inclusion of these features does not necessarily cause complications in terms of usage counting, one aim of this work is concerned with the combination of features, ensuring that they work together. The use of dependent types with such features does complicate other aspects of the meta-theory such as normalization and consistency, which our correctness proof relies on.

Our correctness proof is mechanized in Agda [17], building on our previous formalization of graded type theory [2].[1] We believe that this is the first complete formalization of correctness for linearity for a graded type theory with dependent types; the work of Choudhury et al. [13] is partly formalized but the main correctness result is not. The formalization contains the file `README.agda` which lists our definitions, theorems and claims in the order they appear in the paper, linking to their formalized counterparts. The file also includes some discussion about how the formalized statement relates to the version in the paper such as if there are differences between the formalization and the paper. The paper provides links to an HTML rendering of this file for each entry. We mark such links in blue.

In some regards the presentation in the paper differs from the formalization. Notably, the formalized syntax uses well-scoped de Bruijn indices for variables and machine pointers whereas the paper uses names. Consequently, we do not need to worry about potential issues relating to e.g. scoping or name clashes in the formalization. We also ignore such issues in the paper presentation: the definitions and results should be read as if well-scoped de Bruijn indices had been used. Another notable difference is that the formalization supports unit types and identity types. We have chosen to exclude them from the paper but our results hold also when they are included (in some cases with extra conditions). We refer the interested reader to the formalization for details [17]. These differences, as well as others, are explained further in `README.agda`.

The rest of the paper is structured as follows: In Section 2 we give a high-level, informal description of the main ideas we present in the paper and the problem we solve. Specifically, we discuss the difficulty in defining correct grade typing for recursion, our approach to doing so, and how we use an abstract machine to count variable uses. The formal presentation begins in Section 3, where we give an overview of the graded type theory that we will be working with, excluding the natural number eliminator. Then, in Section 4 we define the abstract machine and prove a form of resource correctness for the theory without the eliminator. The eliminator is introduced in Section 5, where we first discuss some issues with the approach used in our previous work

[1] Our formalization is not directly based on the formalization accompanying the paper, but rather on a more recent version to which some additional contributions have been made, including those of Danielsson and Geng [16] and Danielsson et al. [15].

[3] before describing our approach to usage counting. We prove that our approach is sound by extending the correctness proof to include the eliminator We also discuss concretely the practical consequences of our approach to usage counting, namely under what circumstances variables are used e.g. linearly by the eliminator and we show that it can be used to encode booleans and vectors with reasonable eliminators of their own. Section 6 goes beyond quantitative use cases and we describe how our correctness proof can be adapted, with minor changes, to a proof of non-interference, showing that our eliminator behaves correctly also when grades are used for security. Finally, in Section 7 we discuss how our approaches to usage counting and correctness proofs relate to other graded type theories with support for recursion for natural numbers, or other recursive types.

2 The Problem of Usage Counting

Before we begin the main part of the paper we will spend some time giving an overview of some of the difficulties with usage counting and give a high-level explanation of how we approach them in this paper. We first explain the difficulties with static usage analysis that come from recursion and then some subtleties of dynamic usage counting in our semantics. In order to highlight the core issues and ideas, the presentation in this section is in several ways simplified compared to the formal treatment in the rest of the paper and is thus not formalized. For instance, we present functions in a pattern matching style, using Agda notation, rather than using eliminators (though the functions we present can all be written in the eliminator style) and we ignore some of the administrative details of the abstract machine like properly keeping track of variable names, focusing instead on variable usage counting.

2.1 Static Usage Counting

In a quantitative setting we aim to determine, or approximate, how many times variables are used during evaluation, and assign a corresponding grade. In simple cases this can be done by analyzing each function clause individually. For instance, consider the function truncIf which takes a boolean b and a natural number n and is defined by matching on b:

```
truncIf : (b : Bool) (n : ℕ) → ℕ
truncIf true  n = zero
truncIf false n = n
```

How many times are b and n used in truncIf? Clearly, n is used once when b is false and zero times when it is true (at least under call-by-name). Since we cannot know in advance what value b will take we can only conclude that n will be used either zero times or once, and so we may assign any grade which stands for zero or one uses (and possibly more). In both clauses, b is used once through matching but not further, so we can conclude that it is used once.

When we add function calls and recursion into the picture, we cannot simply count variable occurrences, but we need to scale occurrences in function arguments appropriately. Consider the following recursive definitions of addition and multiplication:

```
plus : ℕ → ℕ → ℕ
plus zero    n = n
plus (suc k) n = suc (plus k n)
```

```
mult : ℕ → ℕ → ℕ
mult zero    n = zero
mult (suc k) n = plus n (mult k n)
```

To account for argument uses in plus we may hypothesize that it uses each argument exactly once and confirm our hypothesis for each function clause. The first clause uses the second argument, n, exactly once and matches on the first argument, which should also count as a use, so our hypothesis is confirmed. The second clause uses the arguments just like plus uses them, given that constructors

like suc are treated as linear functions; thus, our hypothesis is true. In fact, any other hypothesis would also be consistent with the second clause, except for zero uses of the first argument, which is ruled out by the match.

For mult, a linear use of the first argument is consistent with both clauses. However, for the second argument we need to be consistent both with zero uses in the first clause, and *one more use*, as the first argument of plus, in the second clause. Here we arrive at a proper recurrence.

Rather than using some kind of solver for recurrences, our approach in this work is to get rid of recursion by instantiating the clauses to all numerals and fully unfold the recursive calls:

```
plus : ℕ → ℕ → ℕ
plus zero              n = n
plus (suc zero)        n = suc n
plus (suc (suc zero)) n = suc (suc n)
                       ⋮
```

```
mult : ℕ → ℕ → ℕ
mult zero              n = zero
mult (suc zero)        n = plus n zero
mult (suc (suc zero)) n = plus n (plus n zero)
                       ⋮
```

Looking at the transformed clauses, it becomes clear that n is used once in plus whereas it might be used any number of times in mult. The uses of the first arguments are perhaps a bit harder to see since they have been fully matched on, but in both cases their only use is the matching, resulting in a single use. A rigorous treatment of these ideas can be found in Section 5.2, detailing how the usage count for each level of unfolding is found and how the counts of all levels are combined.

2.2 Dynamic Usage Counting

We will now switch focus and give a high-level description of the abstract machine, which we will introduce formally in Section 4 and use to prove that our grade typing correctly tracks how many times variables are used during evaluation. The key point of note is how variable uses are counted: this involves some subtlety.

The two main components of the machine are the heap and the stack. The heap assigns a term, which may still need to be evaluated, to each variable in scope. Each variable also gets assigned a grade representing how many times a lookup of that variable may (or must) be performed. The stack keeps track of the context of what is currently being evaluated, acting similarly to a call stack. We write $\langle H; t; S\rangle$ for the state of a machine that evaluates the term t under the heap H and stack S (in Section 4 we extend this with a fourth component, the environment). The overall evaluation strategy can be summarized as follows: When a function call is reached, the function argument(s) are placed on the heap, and then the body of the function is evaluated, starting by evaluating the function's scrutinee. When doing so a "token" is placed on top of the stack, representing the function clauses. Once the scrutinee has been evaluated, the token is removed from the stack and we continue evaluating the corresponding clause of the function.

In order to show the workings of the machine, and the subtleties of usage counting, we will demonstrate how the program dbl (truncIf false zero) is evaluated, where dbl is the following non-recursive function that doubles the predecessor of the input:

```
dbl : ℕ → ℕ
dbl zero     = zero
dbl (suc m) = plus m m
```

It should be clear, given the discussion above, that m is used at most twice in dbl m.

If we evaluate the program, then we get the trace shown in Figure 1. Let us look at how the heap evolves as evaluation progresses. We first insert the argument of dbl. Since the function may use its

$$
\begin{array}{lll}
\langle & ; \mathsf{dbl}\ \mathsf{tfz} & ; \rangle \xrightarrow{(1)} \text{Place argument on heap} \\
\langle m \mapsto^2 \mathsf{tfz} & ; m \,|^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} & ; \rangle \xrightarrow{(2)} \text{Evaluate scrutinee of } \mathsf{dbl} \\
\langle m \mapsto^2 \mathsf{tfz} & ; m & ; |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(3)} \text{Lookup } m \\
\langle m \mapsto^0 \mathsf{tfz} & ; \mathsf{tfz} & ; |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(4)} \text{Place arguments on heap} \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^2 \mathsf{z}, b \mapsto^2 \mathsf{f} & ; b \,|^{\mathsf{true}\to\ldots}_{\mathsf{false}\to\ldots} & ; |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(5)} \text{Evaluate scrutinee of } \mathsf{truncIf} \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^2 \mathsf{z}, b \mapsto^2 \mathsf{f} & ; b & ; |^{\mathsf{true}\to\ldots}_{\mathsf{false}\to\ldots}, |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(6)} \text{Lookup } b \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^2 \mathsf{z}, b \mapsto^0 \mathsf{f} & ; \mathsf{false} & ; |^{\mathsf{true}\to\ldots}_{\mathsf{false}\to\ldots}, |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(7)} \text{Evaluate } \mathsf{false} \text{ clause of } \mathsf{truncIf} \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^2 \mathsf{z}, b \mapsto^0 \mathsf{f} & ; n & ; |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(8)} \text{Lookup } n \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^0 \mathsf{z}, b \mapsto^0 \mathsf{f} & ; \mathsf{zero} & ; |^{\mathsf{zero}\to\ldots}_{\mathsf{suc}\to\ldots} \rangle \xrightarrow{(9)} \text{Evaluate } \mathsf{zero} \text{ clause of } \mathsf{dbl} \\
\langle m \mapsto^0 \mathsf{tfz}, n \mapsto^0 \mathsf{z}, b \mapsto^0 \mathsf{f} & ; \mathsf{zero} & ; \rangle
\end{array}
$$

Fig. 1. Evaluation trace of dbl (truncIf false zero) where truncIf false zero is abbreviated by tfz, zero by z, and false by f.

argument up to two times we associate the binding with a grade representing "at most two uses"[2], which we will here denote 2. In the formal machine, this grade is given by a grade annotation on function applications which we exclude here. The first subtlety appears when we look up this entry. Even though we only perform a single lookup, the stack indicates that we are currently evaluating the argument of dbl, which uses its argument (up to) two times, and the lookup must therefore account for both uses. As a result, zero additional lookups of m are allowed after step (3). If we had instead only counted a single use of m from this lookup it would have been possible to perform another lookup later, meaning that m could be used too many times; two times as the argument to dbl and additional times after the second lookup.

Next, the heap is extended with the arguments to truncIf. Despite truncIf using both its arguments at most once they are inserted with grade 2, indicating that they may be used twice. The reason is once again that we are evaluating the argument to dbl, meaning that although a single call to truncIf uses b and n only once the result will be used (up to) twice and the uses of the arguments are thus increased accordingly. If we had instead only allowed single lookups of b and n, evaluation would get stuck trying to look up two uses of b where only one is allowed. We continue evaluating the argument of dbl, with lookups of two uses of b and n as before, and in the end, we have exhausted all entries in the heap, meaning that they have been used a correct number of times.

If we had instead evaluated dbl (truncIf true zero) evaluation would have proceeded in essentially the same way until step (7), where we would continue by evaluating zero instead of n. This causes n to never be looked up, so it is left with 2 allowed lookups in the final heap. This is still valid since we interpret the grade 2 as meaning at most two uses, including the possibility of zero. If, however, the final heap includes grades which do not allow zero uses, it would indicate that a variable was not used the correct number of times. As an example, when using the linearity semiring which we define in Section 3.1, the grade 1 represents a linear (i.e. exactly one) use. An entry with such a grade would indicate that a variable that was supposed to be used linearly was unused. Our main correctness theorem (Theorem 4.9) shows that this cannot happen for well-typed programs.

[2]Depending on the chosen grade semiring, there might not be a grade that matches this interpretation exactly. For instance, the linearity semiring we define in Section 3.1, does not have such a grade but we can then instead use the grade ω which represents *any* number of uses, including at most two.

3 A Graded Modal Dependent Type Theory

We continue the work on the graded modal type theory from our previous work [3] and its formalization [2]. In this section we will provide some background by describing the key parts of the language. We refer the reader interested in more details to the formalization [17].

3.1 Grade Semirings

Terms of our language are annotated with *grades* that form a semiring-like structure which we call a *grade semiring* or simply *semiring* for short. We will primarily be interested in grades with a quantitative interpretation, allowing us to express how many times some resource will be used when the program is evaluated. Our definition of grade semirings is a slight reformulation of the one presented in our previous work [3, Definition 3.1]:[3]

Definition 3.1. A grade semiring $(\mathcal{G}, +, \cdot, \wedge, 0, 1)$ is a 6-tuple consisting of (from left to right) a set $\mathcal{G}$, three binary operations (addition, multiplication and meet) and zero and unit elements, satisfying the following properties:

- $(\mathcal{G}, +, \cdot, 0, 1)$ forms a semiring: Addition is commutative and associative with 0 as identity. Multiplication is associative with 1 as identity and 0 as absorbing element and distributes over addition. As usual, we will often abbreviate multiplication of p and q as pq.
- $(\mathcal{G}, \wedge)$ forms a semilattice: Meet is commutative, associative and idempotent.
- Both multiplication and addition distribute over meet.
- Equality with 0 is decidable.

For our purposes, we may think of grades as representing a *quantity*—the number of times some resource, like a function argument, is used. From this perspective, the grades 0 and 1 represent *no uses* and a *single use* of some resource, respectively. Addition represents the combined resource usage from different places, usually from two subterms, while multiplication represents a single resource being used several times, for instance, the argument of a function.

As discussed in Section 2.1, when computation can branch the branches may use different amounts of some resource, which we combine with the meet operator. For instance, a function might use its argument either once or not at all, depending on some condition, so the combined usage is $1 \wedge 0$. Thus, in practice, a grade often stands for a set of natural numbers, representing the possible actual uses. For instance, the set of all natural numbers ω represents any number of uses, so it gives us no static information about the actual usage. Singleton sets carry the maximal information as they determine the exact number of uses. Meet is then interpreted as set union, addition as pointwise addition of sets, and multiplication as pointwise multiplication of sets.

Meet induces a partial order defined by $p \leq q$ iff $p = p \wedge q$. We interpret this relation as a form of *compatibility* between grades, in the sense that if $p \leq q$ then a resource that is used q times may alternatively be thought of as being used p times. When grades are interpreted as sets of natural numbers, $p \leq q$ means that p is a superset of q. In particular, the greatest set ω is the least element.

The order explains how to define $+$, $\cdot$ and $\wedge$ on structures $\mathcal{M}$ that represent only *some* sets of numbers, such as the "none-one-tons" semiring $\mathcal{L}$ given by McBride [22]. Unlike McBride, let us present this structure as if it had been defined using actual sets of natural numbers. It contains the sets $0_{\mathcal{L}} := \{0\}$, acting as zero, $1_{\mathcal{L}} := \{1\}$, acting as one, and $\omega_{\mathcal{L}} := \mathbb{N}$, the least element. The meet of $0_{\mathcal{L}}$ and $1_{\mathcal{L}}$ is $\omega_{\mathcal{L}}$, the greatest lower bound of the two. The sum of $1_{\mathcal{L}}$ and itself is $\omega_{\mathcal{L}}$, the smallest superset of $\{1 + 1\} = \{2\}$ that is in the structure. Only the product in this structure is the literal

[3] In comparison with our previous definition [3] we have omitted the operator $\circledast$ used to assign grades to the eliminator for natural numbers. We will return to this in Section 5. In order to support certain uses of identity types the formalization also requires that there is a grade ω satisfying $\omega \leq 1$ and $\omega(p + q) \leq \omega q$.

$$
\begin{array}{rlrlr}
l & \in \mathsf{Nat} & \text{Universe level} & \Gamma, \Delta ::= \epsilon \mid \Gamma.\, x : A & \text{Typing context} \\
p, q, r & \in \mathcal{G} & \text{Grade} & \gamma, \delta ::= \epsilon \mid \gamma.\, x : p & \text{Grade context} \\
s & ::= \otimes \mid \& & \Sigma\text{-type strength} & m ::= 0 \mid 1 & \text{Mode} \\
A, B, t, u & ::= x \mid \lambda^p x.t \mid t\,^p u \mid (^p t, u)_s \mid \mathsf{fst}_p\ t \mid \mathsf{snd}_p\ t & & & \text{Term} \\
& \quad \mid \mathsf{prodrec}^p_r\ (x.A)\ t\ (x_1.x_2.u) \mid \mathsf{emptyrec}_r\ A\ t \mid \mathsf{zero} \mid \mathsf{suc}\ t & & & \\
& \quad \mid \mathsf{U}_l \mid \Pi^q_p\ (x : A)\ B \mid \Sigma^q_{s,p}\ (x : A)\ B \mid \bot \mid \mathbb{N} & & & \text{Type formers}
\end{array}
$$

Fig. 2. The syntax of terms, typing and grade contexts, and modes.

pointwise product of sets, not requiring us to round up to the next available set. (We now drop the subscript $_{\mathcal{L}}$ and only use it when disambiguation is necessary.) We shall refer to this structure, summarized as $0 \geq \omega \leq 1$ with 0 and 1 incomparable, as the linearity semiring. A resource labelled with 1 must here be used *exactly* once.

Changing 1 to be $1_{\mathcal{A}} := \{0, 1\}$, we get the affine types semiring $\mathcal{A}$, summarized as $\omega \leq 1 \leq 0$. A resource labelled with 1 can here be used *at most* once. Setting 1 to be $1_{\mathcal{E}} := \omega$, we get the two-point erasure semiring $\mathcal{E}$, where resources are either unused (0), and thus can be erased, or used any number of times (ω). We also support using natural numbers as grades extended with a grade ω for any number of uses where grade n represents either "exactly n uses", generalizing linear types, or "at most n uses", generalizing affine types. We encourage the reader unfamiliar with these grade semirings to work out their addition and multiplication tables.

For the theory we present below we additionally require grade semirings to have a *well-behaved zero*. This means that $1 \neq 0$, if $p + q = 0$ or $p \wedge q = 0$ then $p = q = 0$, and if $pq = 0$ then either $p = 0$ or $q = 0$. Similar properties have been put forward by McBride [22] and Atkey [5]. The grade semirings we introduced above all have a well-behaved zero.

3.2 Syntax, Typing and Usage

The language we are considering is a dependently typed language with Π- and Σ-types, a natural number type, an empty type, as well as a hierarchy of Russell universes. The syntax is given in Figure 2. There are two different kinds of Σ-types, *weak* with a form of pattern matching (using prodrec) and *strong* with projections and η-equality. We keep the two kinds separate because of their different behavior in terms of resource usage, as we discuss further below. We therefore label Σ-types and pairs with ⊗ or & to indicate the weak and strong versions, respectively. In cases where no distinction is needed, we may omit the label.

Some terms are annotated by grades indicating, for semirings with a quantitative interpretation, how many times some resource should be used. For lambdas and pairs, the grade represents how many times the function argument is used and the number of times the first component should be used, respectively. Their eliminators—applications, projections and prodrec—each have a corresponding annotation. Π^q_p- and Σ^q_p-types have two annotations, with p having the same meaning as the grade on lambdas or pairs, and q instead representing the number of times the function argument is used in the codomain or how many times the first component is used in type of the second component. In addition to the grade already mentioned, the eliminator for $\Sigma_\otimes$-types, $\mathsf{prodrec}^p_r$ has one more annotation, r, which represents how many times the components of the pair are used by u (since a single pair provides p uses of the first component it should be used rp times by u). The eliminator for the empty type ($\mathsf{emptyrec}_r$) has a similar annotation r indicating how many times its argument is used. Both prodrec and emptyrec have side conditions which can be configured to control if certain grade/mode combinations should be disallowed.

$$\frac{}{\mathbf{0} \blacktriangleright^m \mathsf{U}_l} \qquad \frac{}{\mathbf{0} \blacktriangleright^m \mathbb{N}} \qquad \frac{}{\mathbf{0} \blacktriangleright^m \bot} \qquad \frac{\gamma \blacktriangleright^{m\underline{p}} A \qquad \delta . x : mq \blacktriangleright^m B}{p\gamma + \delta \blacktriangleright^m \Pi^q_p\ (x : A)\ B}$$

$$\frac{\gamma \blacktriangleright^{m\underline{p}} A \qquad \delta . x : mq \blacktriangleright^m B}{p\gamma + \delta \blacktriangleright^m \Sigma^q_{s,p}\ (x : A)\ B} \qquad \frac{}{m\boldsymbol{e}_x \blacktriangleright^m x} \qquad \frac{\gamma . x : mp \blacktriangleright^m t}{\gamma \blacktriangleright^m \lambda^p x.t} \qquad \frac{\gamma \blacktriangleright^m t \qquad \delta \blacktriangleright^{m\underline{p}} u}{\gamma + p\delta \blacktriangleright^m t^p u}$$

$$\frac{\gamma \blacktriangleright^{m\underline{p}} t \qquad \delta \blacktriangleright^m u}{p\gamma \wedge \delta \blacktriangleright^m (^p t, u)_\&} \qquad \frac{\gamma \blacktriangleright^m t}{\gamma \blacktriangleright^m \mathsf{fst}_p\ t}\ mp \leq m \qquad \frac{\gamma \blacktriangleright^m t}{\gamma \blacktriangleright^m \mathsf{snd}_p\ t} \qquad \frac{\gamma \blacktriangleright^{m\underline{p}} t \qquad \delta \blacktriangleright^m u}{p\gamma + \delta \blacktriangleright^m (^p t, u)_\otimes}$$

$$\frac{\gamma \blacktriangleright^{mr} t \qquad \delta . x_1 : mrp . x_2 : mr \blacktriangleright^m u \qquad \eta . x : 0 \blacktriangleright^0 A}{r\gamma + \delta \blacktriangleright^m \mathsf{prodrec}^p_r\ (x.A)\ t\ (x_1.x_2.u)}\ \mathsf{Prodrec}\ m\ r\ p \qquad \frac{}{\mathbf{0} \blacktriangleright^m \mathsf{zero}}$$

$$\frac{\gamma \blacktriangleright^m t}{\gamma \blacktriangleright^m \mathsf{suc}\ t} \qquad \frac{\gamma \blacktriangleright^{mr} t \qquad \delta \blacktriangleright^0 A}{r\gamma \blacktriangleright^m \mathsf{emptyrec}_r\ A\ t}\ \mathsf{Emptyrec}\ m\ r \qquad \frac{\gamma \blacktriangleright^m t}{\delta \blacktriangleright^m t}\ \delta \leq \gamma$$

Fig. 3. The usage relation, $\gamma \blacktriangleright^m t$.

The type system consists of five judgements: $\vdash \Gamma$ for well-formed contexts, $\Gamma \vdash A$ and $\Gamma \vdash t : A$ for well-formed types and terms, and $\Gamma \vdash A = B$ and $\Gamma \vdash t = u : A$ for equality of types and terms. Contexts Γ, Δ are snoc-lists of types, see Figure 2. The definition of the typing judgements should be unsurprising except for the inclusion of grades: typing ensures that the annotations on constructors match the annotations on the corresponding eliminators (for instance, $(\lambda^p x.t)^q u$ is well-typed only if $p = q$). In the interest of brevity we thus leave out their definitions, but they can be looked up in the formalization [17] following the links accompanying the judgements above.

Typing does not ensure that assigned grades are correct in the sense that resources are used as many times as indicated. This is instead handled through the *usage relation*, $\gamma \blacktriangleright^m t$, defined in Figure 3. It checks that t is well-resourced under the grade context γ, which assigns a grade to each variable that is in scope in t: the grades correspond to the number of times the variables should be used when a single copy of t is evaluated. Similarly to typing contexts, grade contexts assign a grade to each variable (or pointer) in scope, see Figure 2. We write $\gamma(x)$ for the grade γ associates to x. We define addition ($\gamma + \delta$) and meet ($\gamma \wedge \delta$) of contexts by pointwise application of the respective operator[4]. Similarly, scaling ($p \cdot \gamma$) a context by a grade is defined by pointwise multiplication from the left and we also lift the order relation ($\gamma \leq \delta$) pointwise to contexts. The context $\mathbf{0}$ assigns grade 0 to all variables and $\boldsymbol{e}_x$ assigns 1 to x and 0 to all other variables.

We are using the extended usage relation [3, Section 8] which checks that t is well-resourced at a given *mode* m. In a quantitative setting, the mode can be either 1 or 0. The former is in some sense the default mode, representing computationally relevant settings, while the latter represents erased settings and is used to check terms with no computational relevancy such as the function argument of an erased application ($p = 0$). Intuitively, no variable uses should be counted in this setting since the term will not be evaluated, and we may also be a bit more permissive in the erased mode, owing again to the idea that erased terms are not evaluated. We can convert grades into modes with $\underline{p} = 0$ iff $p = 0$ and 1 otherwise. In the other direction we implicitly convert mode 0 to grade 0 and mode 1 to grade 1. Multiplication for modes is defined such that $mm' = 1$ iff both m and m' are 1, and 0 otherwise.

[4]These operations are only well-defined if the contexts mention the same variables but this is ensured by the de Bruijn representation used in the formalization.

For quantitative grade semirings we may think of the usage relation as counting how many times resources, or variables, are used [3]. To that end, a variable x is well-resourced with respect to the context $m\boldsymbol{e}_x$ which assigns grade m to x and 0 to all other variables. For the term and type constructors (anything but variables and eliminators) the general principle is that, to each subterm, a usage context is assigned which is extended by grades for the variables bound in that subterm. The usage context of a subterm is then scaled by the grade corresponding to how many times the subterm is used in the surrounding term, which is 1 in most cases except for the first component of a pair. The (possibly scaled) contexts of all subterms are then summed up to form the usage context of the whole term. Strong pairs are an exception since they only allow one component to be used through projection while the other is discarded. Here, the contexts are instead combined using meet to ensure a usage context that is compatible regardless of which component is used. For constants every variable is assigned grade 0.

The eliminators follow the same pattern only that scaling is the rule rather than the exception. For instance, an application $t^p u$ uses the function argument p times, which is reflected by a p-fold scaling of the usage context δ of the argument u, which is added to the usage context γ of the function t. We also see that u is checked in mode mp, rather than m, ensuring that it is checked in the erased mode when $p = 0$. The eliminators corresponding to pattern matching, prodrec and emptyrec, all allow scaling of the scrutinee t by an arbitrary r. Variable usage in the motive (target type) A of the eliminators is not accounted for, as motives are not constructed at runtime and thus do not consume resources. For the projections, the usage count of $\mathsf{fst}_p\ t$ or $\mathsf{snd}_p\ t$ is the same as that for t which, as mentioned, is a context compatible with either component of t. For the first projection the side condition $mp \leq m$ restricts p to be bounded by 1 when checked in mode 1. This is because only one of the p copies of the first component remains after taking the projection so we require compatibility between p and 1.

The final rule works like subsumption in subtyping systems. It allows us to throw away information about resource usage, going from a more precise usage context γ to a less precise context δ whenever $\delta \leq \gamma$. For instance, for all our example semirings we have $\omega \leq 0$ and an unused variable (0) can then be considered to be used (ω). Similarly, for the affine types semiring with $1 \leq 0$, an unused variable can be considered to be used 1 time: 1 has the interpretation "at most one use". On the other hand, this is not allowed for the linearity semiring, because 0 and 1 are not comparable.

The reduction relation $\Gamma \vdash t \longrightarrow u : A$ specifies call-by-name weak head reduction and, as usual, $\Gamma \vdash t \longrightarrow^* u : A$ denotes its reflexive, transitive closure. Notably, the reduction relation is typed, yet the reduction rules are otherwise standard and thus omitted. We have a form of subject reduction for the usage relation. If $\gamma \blacktriangleright^m t$ and $\Gamma \vdash t \longrightarrow u : A$ then $\gamma \blacktriangleright^m u$, which we can interpret as saying that the uses of the free variables do not change (up to subsumption) during evaluation.

Several meta-theoretical properties of the type theory are proved through a logical relation, originally due to Abel et al. [4]. We rely on some of these below in our proof of resource correctness. Most notably, this includes normalization, i.e., all well-typed terms reduce to weak head normal form, and consistency, i.e., one cannot construct a closed term of the empty type. But we also depend on properties like inversion of typing and injectivity of Π- and Σ-types and we will further use part of the logical relation itself as an inductive structure for the terms inhabiting data types, in our case, the natural numbers.

4 A Resource Aware Abstract Machine

In [3] we have shown that a usage preservation lemma holds for all grade semirings, and that the usage relation captures the concept of erasure correctly for a class of semirings (those with a "well-behaved zero"). However, "a serious soundness argument for linearity" was left for future work. The goal of this section is to provide such an argument.

In the context of linearity, our intended interpretation of $\gamma \blacktriangleright^1 t$ is that, if $\gamma(x) = 1$, then x is used exactly once during evaluation of t. We specify what it means *to be used* through an abstract machine that keeps track of available resources, and then we prove that the number of times a variable is used is consistent with the usage relation. Our proof works for the linearity semiring, but more generally whenever grade subtraction is definable.

4.1 Subtraction of Grades

While we utilize grades to represent how many times a resource may be used, they do not a priori provide a way to track how this number changes as resources are consumed. There are two questions that we would like to be able to answer. Firstly, "given p copies of some resource, are there q copies available?" and secondly, "if so, how many copies would be left if we used those q copies?". The first question can be answered in the affirmative if there is some r such that $p \leq r + q$, meaning that p can be converted into the sum of q and some residual r. There may be many such r so we should not take just any such r as the answer to the second question. We want to allow as many future uses as possible when converting p into $q + r$ so we should pick the least such r. A least r might not always exist, but when there is such an r for every p and q such that $p \leq r' + q$ for some r', then we say that the grade semiring under consideration *supports subtraction*. In this case we write $p - q = r$. We shall in turn write $p - q \leq r$ whenever $p \leq r + q$.[5]

For example, in the linearity semiring we have both $\omega \leq 1 + 1$ and $\omega \leq \omega + 1$, but since $\omega \leq 1$ we let $\omega - 1 = \omega$. In fact, $\omega - q = \omega$ for any q since ω is the least element: it represents any number of uses. This generalizes to any least element p in any grade semiring since then $p \leq p + q$ for any q. We further always have $p - 0 = p$ for any p since $p \leq p + 0$. For the linear and affine types semirings, the only remaining case is $1 - 1 = 0$; the rest, $0 - 1$, $0 - \omega$, and $1 - \omega$, are undefined (because there is no r such that $0 \leq r + 1$, $0 \leq r + \omega$ or $1 \leq r + \omega$). Notation might suggest $p - p = 0$, but this does not hold in general, for instance $\omega - \omega = \omega$ in the linearity semiring.

All concrete grade semirings we consider support subtraction. Note in particular that, for these semirings, subtraction from 0 is only possible when subtracting by 0. That is, if there are no copies available then we may only "use" zero copies.

4.2 The Abstract Machine

We will now define an abstract machine for the step-wise evaluation of terms. Structure-wise, it is similar to the lazy abstract machine by Sestoft [27], but adapted to a call-by-name setting. In regards to resource tracking we draw inspiration from Choudhury et al. [13]. The choice of call-by-name is due to the usage relation being defined with resource counting for call-by-name in mind. Other evaluation strategies may lead to differences in how many times variables are referenced.

We focus on evaluation of closed programs, but with some modifications our results hold also for open programs for which all free variables are erasable, see Theorem 4.10. Details are available in the formalization.

We follow Sestoft and use machine states s with four parts: a heap H mapping pointers to closures, an evaluation stack S, the (possibly open) term t that is being evaluated (the *head*), and an environment ρ for t, mapping t's free variables to pointers. The syntax for machine states, heaps and stacks is given in Figure 4. Heaps are lists of entries (which are closures consisting of a term and an environment) together with grades indicating how many copies of the entries are available. The dependency structure is like that of a typing context: entries can only refer to preceding entries, making the heap well-founded. We write $\rho(x)$ for the pointer environment ρ assigns to variable x.

[5] Defining subtraction from addition in preordered semirings is a standard application of Galois connections, and the general term is *residuation* [18, Ch. 3].

$$
\begin{array}{rlr}
e & ::= (t, \rho) & \text{Heap entries} \\
H & ::= \epsilon \mid H.\, y \mapsto^{p} e & \text{Heaps} \\
\rho & ::= \epsilon \mid \rho[x \mapsto y] & \text{Environments} \\
s & ::= \langle H;\, t;\, \rho;\, S \rangle & \text{Machine states}
\end{array}
\qquad
\begin{array}{rlr}
S & ::= \epsilon \mid c.\, S & \text{Stacks} \\
c & ::= \bullet^{p} u\ [\rho] \mid \mathsf{fst}_{p}\ \bullet \mid \mathsf{snd}_{p}\ \bullet \mid \mathsf{suc}\ \bullet & \text{Continuations} \\
 & \quad \mid \mathsf{prodrec}^{p}_{r}\ (x.A)\ \bullet\ (x_1.x_2.u)\ [\rho] & \\
 & \quad \mid \mathsf{emptyrec}_{r}\ A\ \bullet\ [\rho] &
\end{array}
$$

Fig. 4. The syntax of the abstract machine.

$$
\frac{p - q = r}{(H.\, y \mapsto^{p} e) \vdash y \mapsto^{q} e\ ;\ (H.\, y \mapsto^{r} e)}
\qquad
\frac{H \vdash y \mapsto^{q} e\ ;\ H'}{(H.\, y' \mapsto^{p} e') \vdash y \mapsto^{q} e\ ;\ (H'.\, y' \mapsto^{p} e')}\ y \neq y'
$$

$$
\frac{}{(H.\, y \mapsto^{p} e) \vdash y \mapsto e}
\qquad
\frac{H \vdash y \mapsto e}{(H.\, y' \mapsto^{q} e') \vdash y \mapsto e}\ y \neq y'
$$

Fig. 5. Heap lookup relations $H \vdash y \mapsto^{q} e\ ;\ H'$ and $H \vdash y \mapsto e$ for lookup of q copies with heap update and lookup without heap update respectively.

The statement $H \vdash y \mapsto^{q} e\ ;\ H'$, defined in Figure 5, means that it is possible to dereference the pointer y into the heap H, obtaining the entry e and a new heap H'. In the new heap, the usage count of the entry has been decreased by q. This kind of lookup can fail if the heap does not contain sufficiently many copies of that entry, i.e. if it is not possible to subtract q from the number of available copies. We also define a separate lookup relation $H \vdash y \mapsto e$ that always succeeds and does not involve a heap update, see Figure 5.

The evaluation stack S is a list of *continuations* c that represent the context of the term that is currently being evaluated, see Figure 4. The continuation $\mathsf{suc}\ \bullet$, corresponding to the successor constructor, is used to fully evaluate programs to numerals. The other continuations are eliminators with the scrutinee removed but complemented with an environment for the remaining subterms (if there are any). For instance, when an application is evaluated, the function argument is put as a closure on the stack while the function itself is evaluated.

We note that a machine state s can be seen as a decomposition of a (closed) term that can be recovered from s. From this perspective heaps can be seen as substitutions replacing each variable with its corresponding entry in the heap. Stacks make up the "spine" of a term, missing only the scrutinee to form a complete term. We can recover the term corresponding to a state by inserting the head as the missing scrutinee and applying a substitution corresponding to the heap. We denote this operation by $(\!|s|\!)$. Going in the other direction there can of course be several different states all corresponding to a term t. The most natural is perhaps $\langle \epsilon;\, t;\, \epsilon;\, \epsilon \rangle$ which we call the *initial* state for t and denote by $\langle t \rangle$.

As noted in Section 2.2 we must be careful with how heap lookups are counted, and account for the fact that arguments of eliminators are not necessarily used only once. When an entry is looked up it is not necessarily the case that 1 should be subtracted from its grade: the grade to subtract depends on how the current head will be used by continued evaluation. Similarly, the grade of a new heap entry depends on how the head is going to be used. For these reasons we will keep track of a grade r that represents how many "copies of the head" the current state is using.

Choudhury et al. [13] parametrize their reduction relation by r, but we compute r from the stack S via the *multiplicity* function $|S|$. In our formalization multiplicity is defined as a *functional relation* $|S| \equiv r$ introduced by the rules in Figure 6, because the function becomes partial once we add the eliminator for natural numbers in Section 5. The multiplicity of a stack is the product of

$$\frac{}{|\epsilon| \equiv 1} \qquad \frac{|S| \equiv p \qquad |e| \equiv q}{|e.\,S| \equiv pq} \qquad \frac{}{|\bullet^p u\ [\rho]| \equiv 1} \qquad \frac{}{|\mathsf{fst}_p\ \bullet| \equiv 1} \qquad \frac{}{|\mathsf{snd}_p\ \bullet| \equiv 1}$$

$$\frac{}{|\mathsf{prodrec}^p_r\ (x.A) \bullet (x_1.x_2.u)\ [\rho]| \equiv r} \qquad \frac{}{|\mathsf{emptyrec}_r\ A \bullet\ [\rho]| \equiv r} \qquad \frac{}{|\mathsf{suc}\ \bullet| \equiv 1}$$

Fig. 6. Multiplicity of stacks and continuations.

$$\begin{array}{lrcll}
\langle\, H; x & ; \rho; S \,\rangle & \to_e & \langle\, H'; t; \rho'; S \,\rangle & \text{if } H \vdash \rho(x) \mapsto^{|S|} (t, \rho')\ ;\ H' \\
\langle\, H; t^p u & ; \rho; S \,\rangle & \to_e & \langle\, H\ ; t; \rho\ ; & \bullet^p u\ [\rho].\, S \,\rangle \\
\langle\, H; \mathsf{fst}_p\ t & ; \rho; S \,\rangle & \to_e & \langle\, H\ ; t; \rho\ ; & \mathsf{fst}_p\ \bullet.\, S \,\rangle \\
\langle\, H; \mathsf{snd}_p\ t & ; \rho; S \,\rangle & \to_e & \langle\, H\ ; t; \rho\ ; & \mathsf{snd}_p\ \bullet.\, S \,\rangle \\
\langle\, H; \mathsf{prodrec}^p_r\ (x.A)\ t\ (x_1.x_2.u); \rho; S \,\rangle & & \to_e & \langle\, H\ ; t; \rho\ ; & \mathsf{prodrec}^p_r\ (x.A) \bullet (x_1.x_2.u)\ [\rho].\, S \,\rangle \\
\langle\, H; \mathsf{emptyrec}_r\ A\ t & ; \rho; S \,\rangle & \to_e & \langle\, H\ ; t; \rho\ ; & \mathsf{emptyrec}_r\ A \bullet\ [\rho].\, S \,\rangle
\end{array}$$

$$\begin{array}{lrcl}
\langle\, H; \lambda^p x.t & ; \rho; \quad \bullet^p u\ [\rho'].\, S \,\rangle & \to_v & \langle\, H.\, y \mapsto^{|S|p} (u, \rho'); t\ ;\ \rho[x \mapsto y]; S \,\rangle \\
\langle\, H; (^p t_1, t_2)_\&; \rho; & \mathsf{fst}_p\ \bullet.\, S \,\rangle & \to_v & \langle\, H\ ; t_1; \rho\ ; S \,\rangle \\
\langle\, H; (^p t_1, t_2)_\&; \rho; & \mathsf{snd}_p\ \bullet.\, S \,\rangle & \to_v & \langle\, H\ ; t_2; \rho\ ; S \,\rangle \\
\langle\, H; (^p t_1, t_2)_\otimes; \rho; \mathsf{prodrec}^p_r\ (x.A) \bullet (x_1.x_2.u)\ [\rho'].\, S \,\rangle & & \to_v & \langle\, H'\ ; u\ ;\ \rho''\ ; S \,\rangle \\
\quad \text{if} \quad H' = H.\, y_1 \mapsto^{|S|rp} (t_1, \rho).\, y_2 \mapsto^{|S|r} (t_2, \rho[x_1 \mapsto y_1]) & & & \\
\qquad \rho'' = \rho'[x_1 \mapsto y_1][x_2 \mapsto y_2] & & &
\end{array}$$

$$\begin{array}{lrcll}
\langle\, H; \mathsf{suc}\ t; \rho; & \mathsf{suc}^k \bullet \,\rangle & \to_n & \langle\, H; t\ ; \rho; \mathsf{suc}^{k+1} \bullet \,\rangle & \text{if } \neg \mathsf{Numeral}\ t \\
\langle\, H; t\ ; \rho; & \mathsf{suc} \bullet.\, S \,\rangle & \to_n & \langle\, H; \mathsf{suc}\ t; \rho; \quad S \,\rangle & \text{if } \mathsf{Numeral}\ t
\end{array}$$

Fig. 7. The machine reduction relations for eliminators, $s \to_e s'$, values, $s \to_v s'$, and numerals $s \to_n s'$. The stack $\mathsf{suc}^k \bullet$ consists (only) of k copies of $\mathsf{suc}\ \bullet$.

the multiplicities of its continuations, and those are 1 in all cases except for the pattern matching eliminators where it is the multiplicity of the scrutinee. Note that the multiplicity of an application continuation is 1: this continuation is used when the function is evaluated, not the argument.

The state transitions $s \to s'$ of our machine, defined in Figure 7, mimic the call-by-name weak head reduction of the language. This will be made formal by a bisimulation argument utilizing the translation $(\!|s|\!)$ from states s to terms. For a given machine state, we can think of one evaluation step as doing one of three things: (1) looking up a variable in the heap, (2) putting a continuation on the stack, or (3) popping a continuation off the stack.

It turns out to be convenient to know not only that one state reduces to another but also in what way. In particular, we separate the third category above from the others and define the relation $s \to_e s'$ capturing reductions in the first two groups and $s \to_v s'$ capturing those of the third, see Figure 7. We denote the union of these by $s \to s'$ and will refer to it as the weak head semantics of the machine. As usual, notation like $s \to^* s'$ is used for reflexive, transitive closures.

The first relation, $s \to_e s'$, concerns cases where the head of the state is either an eliminator or a variable. For eliminators, a corresponding continuation is put on top of the stack and evaluation continues with the scrutinee in head position.In the variable case, we look up $|S|$ copies of the pointer corresponding to the variable and continue evaluating the result under the updated heap. Importantly, the variable rule can fail if an attempt is made to look up more copies than are available, in which case evaluation would halt.

Table 1. Summary of the reduction relations showing for which kinds of head terms they can be applied.

Applies when:	$\rightarrow_e$	$\dashrightarrow_e$	$\rightarrow_v$	$\rightarrow_n$	$\rightarrow$	$\dashrightarrow$	$\twoheadrightarrow$
Head is an eliminator	✓	✓			✓	✓	✓
Head is a value and stack matches			✓		✓	✓	✓
Head is a variable (with resource tracking)	✓				✓		✓
Head is a variable (without resource tracking)		✓				✓	
Head is suc t				✓			✓

The second relation, $s \rightarrow_v s'$, corresponds to states with a value (non-neutrals in weak head normal form) in head position. These rules can fail if the top of the stack does not match the head term, for instance in the case of a projection with a lambda in head position. In the case of a match the continuation on top of the stack is popped off, and new entries are possibly put on the heap. The number of copies put on the heap are given by the grade annotations of the continuation—in analogy with the usage relation—scaled by the stack multiplicity as explained above.

The observant reader might have noticed that it is possible to look up zero copies of a heap entry whenever the stack multiplicity is 0. As can be gleaned from its definition, the multiplicity of a stack is zero if (and only if) it contains $\mathsf{prodrec}^{p}_{0}$ or $\mathsf{emptyrec}_0$, i.e. in the case of an *erased match* [3] on a weak pair or for the empty type. In either case, this indicates that the current head is an erased term which does not have any computational relevance. While lookups may still happen there is no risk of any data on the heap being used more times than allowed as any new heap entries will have grade 0. Nevertheless, depending on what the language is used for one might not want to allow such lookups in erased contexts. Erased matches can be disallowed using the corresponding side conditions in the usage relation. Unless otherwise noted, the results we present here hold regardless of whether erased matches are allowed or not.

We will also make use of a variant of the semantics for which we do not keep track of resource usage: $s \dashrightarrow s'$. Since the variable case is the only one in which this is done we can define this simply by replacing the variable rule of the weak head semantics with the following one:

$$\frac{H \vdash \rho(x) \mapsto (t, \rho')}{\langle H;\, x;\, \rho;\, S\rangle \dashrightarrow_e \langle H;\, t;\, \rho';\, S\rangle}$$

As Theorems 4.4 and 4.6 will show, the weak head semantics mirrors the call-by-name weak head reduction of the language. In order to prove our resource correctness theorem, Theorem 4.9, just weak head reduction is not quite sufficient. The reason is that evaluation to weak head normal form could leave us with states in which the head still contains references to the heap, representing resources that have not yet been used. Since our goal involves verifying that the number of times variables are used is correct we want evaluation to continue until no such references remain. We will therefore evaluate programs (terms of type $\mathbb{N}$) to numerals, rather than just to outermost constructor form.

For this purpose, we define an additional reduction relation $s \rightarrow_n s'$, see Figure 7. This allows suc t to be evaluated further if t is not a numeral. Note that we only allow evaluation under suc when the stack is $\mathsf{suc}^k \bullet$ (for some k), the stack consisting of nothing but k successor eliminators. We add this stipulation to ensure that the reduction semantics stays deterministic once we add the continuation for the natural number eliminator. We also define an extended reduction relation $s \twoheadrightarrow s'$, which is the union of $s \rightarrow_e s'$ and $s \rightarrow_v s'$ and $s \rightarrow_n s'$: we will refer to this as the full semantics of the machine. A summary of the reduction relations for the abstract machine is given in Table 1.

Our goal is to show that this semantics is correct in the sense that all programs (of type $\mathbb{N}$) evaluate to a numeral and use resources correctly: evaluation should not get stuck because the heap did not contain enough resources, and once evaluation finishes, the heap should have no leftover resources that were expected to have been used. Note that, as mentioned in Section 2.2, the latter condition does not mean that the heap should not have any resources remaining in the sense that all entries are annotated with 0: for the linearity semiring it should be fine to have entries annotated with ω, because ω stands for *any* number of uses, including none. Thus, it suffices that all heap entries are annotated with grades that are compatible with 0, i.e., *at most* 0. Note that for the linearity semiring $0 \leq 0$ and $\omega \leq 0$, but $1 \not\leq 0$.

As we will see, evaluation of "sufficiently well-behaved" programs with the full reduction strategy can be summarized in the following way: First evaluation uses the weak head semantics until a state with suc t in head position is reached (we may also reach zero right away in which case evaluation halts). At this point, a successor continuation is put on the stack and evaluation continues using the weak head semantics. This repeats until a state with zero in head position is reached, at which point evaluation finishes by popping all successor continuations off the stack (one by one). In other words, most interesting steps of evaluation are covered by the reduction steps of the weak head semantics. Although our main goal involves full reduction, we will therefore spend most of our effort considering only weak head reduction.

We will show that in each iteration of the procedure described above weak head reduction will lead to a state with either zero or suc t as the head, and use this to show that full reduction matches the procedure described above. Looking at the reduction rules, we can see three ways in which weak head evaluation can get stuck or terminate (two of which were mentioned above): (1) variable lookup fails because there are not enough resources in the heap, (2) there is a value in head position with a non-matching continuation on top of the stack, or (3) there is a value in head position and the stack is empty. In the next few sections we will consider these reasons one by one, leading up to our main correctness proof. The first reason is perhaps the most interesting as it means that a term is trying to access more resources than allowed—for instance attempting to access a linear resource more than once. We will show in Section 4.3 that well-resourced states cannot get stuck in this way. The second reason is an indication of a malformed state, such as applying a projection to a lambda abstraction. As we will show, also in Section 4.3, such situations are ruled out by typing. Conversely, the final reason is the expected way for evaluation to terminate. In Section 4.4 we show that all sufficiently well-behaved programs indeed terminate in this way and in the process use any resources as many times as specified.

4.3 Usage and Typing for the Machine

In order to rule out the possibility of evaluation getting stuck because the heap does not contain enough resources, we will define a usage relation for machine states. The idea is to keep track of both the resources available in the heap and the resources that will be consumed by the head and stack, and make sure that these are compatible, i.e. that the heap contains at least as many resources as will be consumed. We partly follow Choudhury et al. [13].

The first step is to define a usage relation for heaps, representing the available resources, see Figure 8. We write this as $\gamma \blacktriangleright H$ with the interpretation that γ represents the number of available copies of each entry.

When a heap entry $x \mapsto^{p} (t, \rho)$, with $\delta \blacktriangleright \underline{}^{p} t$, is added to H, the new entry is assigned the grade p, representing that it has p copies available. For all preceding entries, however, the number of available copies seems to decrease (from $\gamma + p\delta[\rho]$ to γ). This accounts for any resources that the p copies of t may consume, making them unavailable to be used elsewhere. We use the notation $\delta[\rho]$ to indicate a renaming of the variables of δ to the corresponding pointers according to ρ.

$$\frac{}{\epsilon \blacktriangleright \epsilon} \qquad \frac{\gamma + p\delta[\rho] \blacktriangleright H \qquad \delta \blacktriangleright^{\underline{p}} t}{\gamma.\, y : p \blacktriangleright H.\, y \mapsto^{p} (t, \rho)} \qquad \frac{\gamma \blacktriangleright H}{\delta \blacktriangleright H}\ \gamma \leq \delta$$

$$\frac{\gamma \blacktriangleright^{m\underline{p}} u}{p\gamma[\rho] \blacktriangleright^{m} \bullet^{p} u\ [\rho]} \qquad \frac{}{\mathbf{0} \blacktriangleright^{m} \mathsf{fst}_{p}\ \bullet}\ mp \leq m \qquad \frac{}{\mathbf{0} \blacktriangleright^{m} \mathsf{snd}_{p}\ \bullet}$$

$$\frac{\gamma.\, x_1 : mrp.\, x_2 : mr \blacktriangleright^{m} u}{\gamma[\rho] \blacktriangleright^{m} \mathsf{prodrec}^{p}_{r}\ (x.A)\ \bullet\ (x_1.x_2.u)\ [\rho]}\ \mathsf{Prodrec}\, m\, r\, p \qquad \frac{}{\mathbf{0} \blacktriangleright^{m} \mathsf{emptyrec}_{r}\ A\ \bullet\ [\rho]}\ \mathsf{Emptyrec}\, m\, r$$

$$\frac{}{\mathbf{0} \blacktriangleright \epsilon} \qquad \frac{\delta \blacktriangleright^{\underline{|S|}} c \qquad \gamma \blacktriangleright S}{\gamma + |S|\delta \blacktriangleright c.\, S} \qquad \frac{\gamma \blacktriangleright H \qquad \delta \blacktriangleright^{\underline{|S|}} t \qquad \eta \blacktriangleright S}{\blacktriangleright \langle H;\, t;\, \rho;\, S \rangle}\ \gamma \leq |S| \cdot \delta[\rho] + \eta$$

Fig. 8. Usage relations for heaps, $\gamma \blacktriangleright H$, continuations, $\gamma \blacktriangleright^{m} c$, stacks, $\gamma \blacktriangleright S$, and states, $\blacktriangleright s$.

As for terms, we allow a form of subsumption for heaps. This one goes the other way around, because while the usage relation for terms represents resources that will be consumed, usage for heaps represents resources that are available.

The head and stack make up the "consuming" parts of a state. The usage relation for terms (Figure 3) describes the resources consumed by the head, and also forms the basis for defining a usage relation for stacks. In short, the resources consumed by a continuation are the same as the ones for the corresponding term with the resources for the scrutinee removed and the assumptions regarding eliminators' motives dropped. This is expressed by $\gamma \blacktriangleright^{m} c$, defined in Figure 8. Note that the lack of a usage rule for suc $\bullet$ is deliberate; with the weak head semantics we are currently considering such continuations will never be placed on the stack. For stacks, the usage counts of all continuations are simply added together, scaled by the stack multiplicity to account for the number of copies being evaluated, see Figure 8. We write this as $\gamma \blacktriangleright S$.

With this we get both (an approximation of) what resources a state has available for use and (an approximation of) what resources are required to evaluate it. For a well-resourced state, these should be in balance, i.e. there should always be enough resources available. We define usage of states, $\blacktriangleright s$ in Figure 8. The first three conditions ensure that the heap, head and stack are all well-resourced with respect to certain usage contexts. The side condition is key for this definition. The right-hand side, $|S| \cdot \delta[\rho] + \eta$, represents the total number of resources that will be consumed by the stack and the ($|S|$ copies of) the head, so this condition ensures that the heap contains at least as many resources as will be consumed. We get that, for well-resourced states, reduction cannot get stuck due to failing heap lookups caused by insufficient resources in the heap:

Theorem 4.1. *If* $\blacktriangleright \langle H;\, x;\, \rho;\, S \rangle$, *then* $H \vdash \rho(x) \mapsto^{|S|} e\ ;\ H'$ *for some entry* e *and heap* H'.

Furthermore the property of being well-resourced is preserved by the weak head semantics:

Theorem 4.2. *If* $\blacktriangleright s$ *and* $s \longrightarrow s'$ *or* $s \longrightarrow^{*} s'$ *then* $\blacktriangleright s'$.

Note that the contexts assigned to the heap, head and stack of s and s' are not necessarily the same: they could change due to e.g. heap lookups.

Thus we see that, starting with a well-resourced state, reduction will never fail because the variable rule could not be applied due to insufficient resources. In order to also ensure that reduction cannot get stuck for states with non-empty stacks, due to a non-matching continuation, we define a typing judgement for states, $\vdash s : A$. Like typing for terms, typing for states is mostly unrelated to usage counting so we do not spell out the details here and instead refer to the formalization. Like

usage, typing is preserved by machine reduction and any state with a value in head position and a non-empty stack reduces. We can now show that two of three ways in which reduction could get stuck cannot happen for well-resourced and well-typed states:

Theorem 4.3. *If $\vdash s : A$ and $\blacktriangleright s$ for some state s that does not evaluate further then the head of s is a value and its stack is empty.*

4.4 Resource Correctness

At this point we have shown that there is only one way in which (well-typed and well-resourced) programs can terminate, but it remains to show that all programs do terminate and that heap entries are used a sufficient number of times. We will deduce this from normalization of call-by-name reduction after first showing that the machine and call-by-name reduction relations are bisimilar.

The first direction, showing that machine evaluation steps are matched by call-by-name reduction steps, is relatively straightforward:

Theorem 4.4. *For any machine states s and s':*

- *If $s \to_e s'$ then $(\!|s|\!) = (\!|s'|\!)$.*
- *If $\vdash s : A$ and $s \to_v s'$ then $\epsilon \vdash (\!|s|\!) \to (\!|s'|\!) : A$.*
- *If $\vdash s : A$ and $s \to s'$ then $\epsilon \vdash (\!|s|\!) \to^* (\!|s'|\!) : A$.*
- *If $\vdash s : A$ and $s \to^* s'$ then $\epsilon \vdash (\!|s|\!) \to^* (\!|s'|\!) : A$.*

The second direction is more involved. We begin by showing that every well-resourced state can be evaluated to a state with a value in head position using only $\to_e$, reduction for continuations and variables. We say that such states are in *normal form*. The procedure for normalizing a state is as follows: (1) the head is a value, in which case we are done, (2) the head is an eliminator, in which case we put the corresponding continuation on the stack and continue, (3) the head is a variable, in which case we look up the variable and continue. The final case poses some issues since variable lookups might not be allowed. In principle, this could be avoided using Theorem 4.1 since heap lookups are always allowed for well-resourced states, but this does not lend itself well to the induction scheme that we use to prove that normalization is always possible. We therefore show normalization first for the non-tracking semantics, avoiding the issue of possibly failing lookups. By showing the tracking and non-tracking semantics to be bisimilar for well-resourced states we also get normalization for the tracking semantics.

Theorem 4.5. *For any machine state s:*

- *There is some state s' in normal form with $s \dashrightarrow_e^* s'$.*
- *If $\blacktriangleright s$ then there is some state s' in normal form such that $s \to_e^* s'$.*

To go from normalization to bisimilarity we note that normalized states with non-empty stacks reduce using $\to_v$. Thus, a single call-by-name reduction step corresponds to normalizing the corresponding machine state, followed by a single evaluation step using $\to_v$, popping the topmost continuation off the stack.

Theorem 4.6. *If $\epsilon \vdash (\!|s|\!) \to u : A$ and $\vdash s : B$ and $\blacktriangleright s$, then there is some state s' such that $s \to^* s'$ and $(\!|s'|\!) = u$. If additionally u is in weak head normal form then s' does not evaluate further: the head of s' is a value and the stack is empty.*

As a corollary we get that evaluation with the weak head semantics terminates:

Corollary 4.7. *If $\vdash s : A$ and $\blacktriangleright s$ then $s \to^* s'$ for some state $s' = \langle H; t; \rho; \epsilon \rangle$ where t is a value and s' does not evaluate further.*

So far we have only considered the weak head semantics of the machine, but it is now finally time to turn our attention to full evaluation to numerals. We can use Corollary 4.7 to reach a state with either zero or suc t in head position. In the latter case, a successor continuation is placed on the stack and Corollary 4.7 is applied again repeatedly until the head becomes zero, at which point the continuations are popped off the stack, and we reach a state with a numeral in head position.

Theorem 4.8. *If $\vdash s : \mathbb{N}$ and $\blacktriangleright\, s$ then there is a numeral n, a heap H and an environment ρ such that $s \twoheadrightarrow^* \langle H; n; \rho; \epsilon\rangle$ and $\blacktriangleright\, \langle H; n; \rho; \epsilon\rangle$.*

The proof is done by induction on the logical relation for natural numbers provided by Abel et al. [4], essentially representing the numeral which t eventually evaluates to. Notably, the theorem implies that evaluation cannot get stuck due to over-use of some resource.

Of course, there is still one part missing—namely that resources are also not used fewer times than specified. This follows more or less directly from Theorem 4.8 by noting that since n is a numeral and the stack is empty, the state we end up in consumes no resources. Since the state is well-resourced this means that $\gamma \blacktriangleright H$ for some context $\gamma \leq \mathbf{0}$ from which it follows that the number of copies of each entry in H is bounded by 0.

Theorem 4.9. *If $\epsilon \blacktriangleright^1 t$ and $\epsilon \vdash t : \mathbb{N}$ then there is a numeral n, a heap H and an environment ρ such that $\langle t\rangle \twoheadrightarrow^* \langle H; n; \rho; \epsilon\rangle$ and the grade associated with each entry in H is bounded by 0.*

For the linearity semiring, the final heap cannot contain any remaining linear entries. Thus, any such entries that were placed on the heap must have been looked up exactly once.

With some modifications our results hold also for evaluation of open programs, as long as all free variables are assigned grade 0. In order to ensure that such variables cannot be used in a computationally relevant way, our proof assumes that the grade 0 is well-behaved (regardless of whether the setting with two or one modes is used). This ensures that one cannot subtract a non-zero quantity from zero. This extends our resource correctness theorem as follows:

Theorem 4.10. *If Δ is consistent, erased matches for* prodrec *are disallowed, $\mathbf{0} \blacktriangleright^1 t$, and $\Delta \vdash t : \mathbb{N}$, then there is a numeral n, a heap H and an environment ρ such that $\langle t\rangle \twoheadrightarrow^* \langle H; n; \rho; \epsilon\rangle$ and the grade associated with each entry in H is bounded by 0.*

As a reminder, an erased match here refers to occurrences of prodrec^{p}_{0}. These can be disallowed using the corresponding side condition of the usage relation. Note that the assumptions of this theorem (a consistent context, no erased matches, all free variables are erasable) are similar to those of the proof of soundness of erasure [3, Theorem 6.13]. All these assumptions are necessary and the theorem fails if one is removed (though we can replace consistency with the requirement that erased matches are not allowed for emptyrec).

5 Natural Number Recursion

In the previous sections we omitted the eliminator for natural numbers. In this section we extend the syntax of our language with such a term, $\text{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s)\ n$, corresponding to the following Agda function, and discuss how to properly count usage for it.

```
natrec : (A : (x_n : ℕ) → Set) (z : A zero) (s : (x_p : ℕ) (x_r : A x_p) → A (suc x_p)) (n : ℕ) → A n
natrec A z s zero    = z
natrec A z s (suc n) = s n (natrec A z s n)
```

We take the our previous method [3] as a starting point. As we will see this approach fails to capture, for instance, linearity. We will instead propose a different method for usage counting for natrec, and extend the abstract machine and its correctness proof to show that the method works correctly.

5.1 Natrec-star

In previous work [3], we state the usage rule[6] for natrec using an additional operator $\circledast$ named natrec-star. The operator is assumed to satisfy some distributivity properties, an "interchange" property, as well as the two inequalities $p \circledast_r q \leq p$ and $p \circledast_r q \leq q + r(p \circledast_r q)$. They use the following usage rule, with $\circledast_r$ lifted pointwise to contexts:

$$\frac{\gamma_z \blacktriangleright^m z \qquad \gamma_s . x_p : mp . x_r : mr \blacktriangleright^m s \qquad \gamma_n \blacktriangleright^m n \qquad \eta . x_n : 0 \blacktriangleright^0 A}{(\gamma_z \wedge \gamma_n) \circledast_r (\gamma_s + p\gamma_n) \blacktriangleright^m \mathsf{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n}$$

Here z and s are the zero and suc branches, respectively, and n is the natural number argument. The s argument binds two variables, one corresponding to the predecessor of the natural number, assigned grade p, and one corresponding to the recursive call, assigned grade r. Similarly to other eliminators there is an explicit motive A.

A detailed explanation of this rule has been given previously [3] where our focus is on proving that usage is preserved by reduction and correctly models erasure. However, the rule is not sufficient to express *linearity* correctly.

One problem arises when considering addition of natural numbers, defined in the following way: $\mathtt{plus}\ k\ n = \mathsf{natrec}^1_0\ (x_n.\mathbb{N})\ k\ (x_p.x_r.(\mathsf{suc}\ x_r))\ n$. This corresponds to the following Agda program, which one might expect to be linear in both arguments:

```
plus : ℕ → ℕ → ℕ
plus k zero    = k
plus k (suc n) = suc (plus k n)
```

For the linearity semiring, however, the suggested definition of $\circledast$ is $p \circledast_0 q = p \wedge q$ and $p \circledast_1 q = p + \omega q$ and $p \circledast_\omega q = \omega(p + q)$, which is the (pointwise) largest possible definition [2]. This would mean that `plus` we have $\gamma \wedge \delta \blacktriangleright^m \mathtt{plus}\ k\ n$ if $\gamma \blacktriangleright^m k$ and $\delta \blacktriangleright^m n$. For $\mathtt{plus}\ x_1\ x_2$ the usage count of both x_1 and x_2 is then $0 \wedge 1 = \omega$. This is safe but suboptimal since it fails to capture linearity. A more problematic situation is provided by $\mathtt{plus}\ x\ x$, for which the usage count is $1 \wedge 1 = 1$, despite the fact that the variable is used twice.

5.2 A Resource-Correct Usage Rule

Let us now define an alternative usage rule for natrec. If the pattern of the other eliminators is followed, then the usage count for $\mathsf{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n$ should be the usage count for n scaled by some factor plus the usage contributions from z and s; as usual, there is no contribution from the motive A. If we leave the other parts of the usage rule above unchanged, then we get a rule of the following form:

$$\frac{\gamma_z \blacktriangleright^m z \qquad \gamma_s . x_p : mp . x_r : mr \blacktriangleright^m s \qquad \gamma_n \blacktriangleright^m n \qquad \eta . x_n : 0 \blacktriangleright^0 A}{x\gamma_n + \chi \blacktriangleright^m \mathsf{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n}\ \text{Side conditions to be determined}$$

Here x is some grade indicating the number of copies of n that are used and χ is a context representing the usage contributions from z and s. It seems reasonable that the uses of n, i.e. x, should only depend on how the number is being used in s, i.e. on p and r. We want to find values for x and χ that are sound in the sense that they provide an accurate usage count, while also not being overly approximate. We will choose the values in such a way that subject reduction can be proved: a linear variable should not stop being linear due to evaluation. Our approach is therefore to determine necessary conditions on x and χ for subject reduction to hold, given our ansatz.

[6] We also present an alternative usage rule [3, Section 7.1.4]. Similar issues apply to that one.

The argument is as follows: Given certain assumptions, we can, for any contexts γ_z and γ_s, construct terms z and s such that $\gamma_z \blacktriangleright^1 z$ and $\gamma_s . x_p : p . x_r : r \blacktriangleright^1 s$ as nested pairs of variables to accumulate the required usage of each variable. For any numeral n we have $\mathbf{0} \blacktriangleright^m n$ and for $t = \mathrm{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ n$ we have $\chi \blacktriangleright^1 t$ by assumption. In the case where $n = \mathrm{zero}$, we have $t \longrightarrow z$ and subject reduction thus gives $\chi \leq \gamma_z$. When $n = \mathrm{suc\ zero}$ we have $t \longrightarrow s[\mathrm{zero}/x_p, \mathrm{natrec}^r_p\ (x_n.A)\ z\ (x_p.x_r.s)\ \mathrm{zero}/x_r]$, so subject reduction gives $\chi \leq \gamma_s + r\gamma_z$, and so on. In total, a necessary condition is given by $\chi \leq \mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$, where we have defined the function nr_i by $\mathrm{nr}_0\ r\ q_z\ q_s = q_z$ and $\mathrm{nr}_{i+1}\ r\ q_z\ q_s = q_s + r \cdot \mathrm{nr}_i\ r\ q_z\ q_s$, lifted pointwise to contexts in its last two arguments. We can see $\mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$ as representing the usage contributions from z and s given that natrec is unfolded i times. Zero unfoldings just yield the base case z with contributions γ_z, a single unfolding uses the recursive call, which here is the zero branch, r times and the successor branch once giving a total usage count of $\gamma_s + r\gamma_z$, and so on. The inequalities can thus be seen as requiring that the usage contribution from z and s should be compatible with any number of unfoldings. We pick the greatest such context (assuming that such a context exists) and thus let[7] $\chi = \bigwedge_i \mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$. Taking the largest context for which subject reduction holds should avoid overapproximations and give us the most general usage rule for our ansatz—subsumption allows going to any smaller context.

Proceeding similarly for x we get the inequality $x \leq p + rx$, corresponding to the uses in the successor case. On its own, this constraint would allow x to be 0 whenever $p = 0$ which means that n would be considered to be erased even though its value affects evaluation. Unlike above, we do not get any necessary condition corresponding to the zero case so to prevent this we add the requirement $x \leq 1$. We can interpret this as having reached the zero case, we have fully matched on n counting as a single use. If we again take x to be the greatest solution to these inequalities, then the condition can be written as $x = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p$, where we can think of $\mathrm{nr}_i\ r\ 1\ p$ as representing how many times n is used given i unfoldings.

We get our usage rule by adding the side conditions $x = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p$ and $\chi = \bigwedge_i \mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$ to the rule given above. With this rule we can prove subject reduction for the usage relation given some additional assumptions for the grade semiring:

Definition 5.1. A grade semiring is said to have well-behaved greatest lower bounds if it satisfies the following properties:

(1) If $p = \bigwedge_i p_i$ then $q + p = \bigwedge_i (q + p_i)$.
(2) If $p = \bigwedge_i p_i$ then $q \cdot p = \bigwedge_i (q \cdot p_i)$.
(3) If $p = \bigwedge_i p_i$ then $p \cdot q = \bigwedge_i (p_i \cdot q)$.
(4) If $p = \bigwedge_i \mathrm{nr}_i\ r\ q_z\ q_s$ and $p' = \bigwedge_i \mathrm{nr}_i\ r\ q'_z\ q'_s$ then $\hat{p} = \bigwedge_i \mathrm{nr}_i\ r\ (q_z + q'_z)\ (q_s + q'_s)$ exists and $p + p' \leq \hat{p}$.

The first three properties are essentially generalizations of distributivity over meet to greatest lower bounds of grade sequences. The final property corresponds to the following "sub-interchange" property between addition and meet which follows from the grade semiring laws: $(p \wedge q) + (p' \wedge q') \leq (p + p') \wedge (q + q')$. Unlike the preceding three properties we restrict this property to sequences of the form nr_i. This is for practical purposes as proving constructively that all our example semirings satisfy the sub-interchange property for arbitrary sequences does not seem feasible. All grade semirings we have mentioned above (e.g. those for linearity and affine types) satisfy these properties.

We can prove subject reduction for grade semirings with well-behaved greatest lower bounds, using the following "characteristic inequalities" (which also appeared above for natrec-star): if

[7] Given a sequence $p_\_$ of grades (a function from natural numbers to grades) we let $p = \bigwedge_i p_i$ mean that p_i has a greatest lower bound p, and similarly for contexts.

$$\langle H;\, \mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s)\ n;\, \rho;\, S \rangle \ \to_e\ \langle H;\, n;\, \rho;\, \mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s) \bullet [\rho].\, S \rangle$$

$$\begin{array}{lll} \langle H;\, \mathsf{zero}\ ;\, \rho;\, \mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s) \bullet [\rho'].\, S \rangle & \to_v & \langle H\ ;\, z;\, \rho' \qquad\qquad\qquad ;\, S \rangle \\ \langle H;\, \mathsf{suc}\ t;\, \rho;\, \mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s) \bullet [\rho'].\, S \rangle & \to_v & \langle H';\, s;\, \rho'[x_p \mapsto y_p][x_r \mapsto y_r];\, S \rangle \end{array}$$

$$\begin{array}{rcl} \text{if} \quad q & = & \bigwedge_i \mathrm{nr}_i\ r\ 1\ p \\ H' & = & H.\, y_p \mapsto^{|S|q} (t, \rho').\, y_r \mapsto^{|S|r} (\mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s)\ x_p, \rho'[x_p \mapsto y_p]) \end{array}$$

Fig. 9. Extension of the machine reductions for eliminators and values from Figure 7 by natrec.

$p = \bigwedge_i \mathrm{nr}_i\ r\ q_z\ q_s$ then $p \leq q_z$ and $p \leq q_s + rp$. In addition, the main results presented previously [3] still hold with our new rule (for semirings with well-behaved greatest lower bounds), in particular the substitution lemma and correctness of erasure.

It remains to show that the resource correctness from Section 4 extends to natrec. However, let us first note that for the linearity semiring the natural number will be considered to be used linearly ($x = 1$) only when $p = 0$ and $r = 1$ or if $p = 1$ and $r = 0$. For the affine types semiring we also have $x = 1$ when $p = r = 0$. In all other cases $x = \omega$. For `plus` we have $\gamma + \delta \blacktriangleright^m \mathtt{plus}\ k\ n$ if $\gamma \blacktriangleright^m k$ and $\delta \blacktriangleright^m n$: both arguments are treated as if they are used once (for any grade semiring).

5.3 Resource Correctness

As a first step towards extending our proof of resource correctness to natrec we add a corresponding continuation $\mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s) \bullet [\rho]$ and define its multiplicity. The usage rule we defined above indicates that the multiplicity should be the greatest lower bound of $\mathrm{nr}_i\ r\ 1\ p$, so we extend the multiplicity definition above (Figure 6) with $|\mathsf{natrec}^{r}_{p}\ (x_n.A)\, z\, (x_p.x_r.s) \bullet [\rho]| \equiv q'$ iff $q' = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p$. Since this greatest lower bound does not necessarily exist for all semirings, the stack multiplicity is no longer guaranteed to exist. It is still the case, however, that the stack multiplicity relation $|S| \equiv r$ is functional since the greatest lower bound is unique if it does exist. We therefore still write $|S|$ as before for the unique multiplicity of a stack, with an implicit assumption that it exists.

We get three new reduction rules: one for putting the continuation on the stack and two for popping it off, corresponding to the zero and successor cases, see Figure 9. In the successor case, two new entries are put on the heap. The first contains $|S|q'$ copies of the natural number, where $q' = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p$ corresponds to all of the number's uses, both by s and by the recursive call. The second entry contains $|S|r$ copies of the recursive call in the form of a closure. Its natural number argument x_p refers to the previous entry, prompting a lookup if the recursive call gets evaluated. The reduction relation is still deterministic, because of the constraint on the reduction rule that puts suc $\bullet$ on the stack.

A consequence of the addition of natrec is that the circumstances under which reduction can get stuck are now slightly different. A state with a value in head position and a non-empty stack is now stuck if it is not the case that (1) the stack and head match and (2) the stack multiplicity exists. Similarly to what we did in Section 4, we will show that for well-resourced and well-typed states, this cannot happen.

The usage relation for continuations, Figure 8, is extended with the following rule:

$$\frac{\gamma_z \blacktriangleright^m z \qquad \gamma_s.\, x_p : mp.\, x_r : mr \blacktriangleright^m s \qquad \eta.\, x_n : 0 \blacktriangleright^0 A}{\chi[\rho] \blacktriangleright^m \mathsf{natrec}^{r}_{p}\ (x_n.A)\ z\ (x_p.x_r.s) \bullet [\rho]}\ \chi = \bigwedge_i \mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$$

This rule is obtained from the usage rule for the corresponding term through the removal of the contribution from the scrutinee (i.e. the natural number argument). The only difference from the pattern established above is that we have included the assumption that the motive is well-resourced.

Table 2. Usage counting for natrec for linear and affine types.

r	$\chi = \bigwedge_i \mathrm{nr}_i\ r\ \gamma_z\ \gamma_s$	$x = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p$		
		$p = 0$	$p = 1$	$p = \omega$
0	$\gamma_z \wedge \gamma_s$	$\omega_{\mathcal{L}}/1_{\mathcal{A}}$	1	ω
1	$\gamma_z + \omega\gamma_s$	1	ω	ω
ω	$\omega(\gamma_z + \gamma_s)$	ω	ω	ω

This is used to conclude that the motive in the recursive call is well-resourced when the recursive call is added to the heap.

A priori, machine states can now get stuck if the stack multiplicity is undefined. However, we can prove that for well-resourced states, the stack multiplicity always exists. We get that well-resourced states do not get stuck due to the stack multiplicity not existing and that the theorems related to usage in Section 4.3 still hold.

Typing for machine states is similarly updated to account for the added continuation. We can now prove everything from Sections 4.3 and 4.4 as stated. In particular, the main resource correctness result, Theorem 4.9, still holds.

To wrap this section up, let us return to the addition example presented above. Let us evaluate `plus` k n, for closed numbers k and n. The evaluation progresses by recursion over n, with each iteration inserting a natural number in the heap as well as an unevaluated recursive call, both assigned grade 1. Note that even though the natural number is unused by the successor branch ($p = 0$), it is used once by the recursive call through matching. The correctness theorem shows that once this evaluation finishes, lookups will have been performed in such a way that all these entries will have their remaining uses bounded by 0. For the linearity semiring it must then be the case that there are 0 copies left, since one cannot obtain ω by (repeated) subtraction from 1.

5.4 Usage Counting of the Natural Number Eliminator for Linear and Affine Types

We have shown that our proposed usage rule for natrec is sound, but the rule is perhaps not very enlightening, as it might not be immediately clear what the greatest lower bound of $\mathrm{nr}_i\ r\ q_z\ q_s$ is (or whether such a bound even exists). For that reason we will now investigate natrec for two specific grade semirings: those for linear and affine types. For these semirings it is straightforward to determine the relevant greatest lower bounds for concrete values of r and p. We summarize these in Table 2.

As we noted above, there are two cases when natrec uses its natural number argument linearly. The first is when the recursive call is used linearly and the predecessor is erased, that is when $r = 1$ and $p = 0$. The addition function discussed above is an example of this. The other situation is when instead the predecessor is used linearly and the recursive call is erased. An example of this is the predecessor function $\mathtt{pred}\ n = \mathtt{natrec}^{0}_{1}\ (x_n.\mathbb{N})\ \mathtt{zero}\ (x_p.x_r.x_p)\ n$ which is considered to be linear since $r = 0$ and $p = 1$. This corresponds to the following Agda program:

```
pred : ℕ → ℕ
pred zero    = zero
pred (suc n) = n
```

For affine types the number is used at most once in the given two cases and also when $p = r = 0$. In the latter case the number is only used through a match on the outermost constructor, so it is not necessarily "fully used".

For both semirings all other cases yield an unrestricted (ω) number of uses (and the number is therefore never considered to be erased). This should not be too surprising, as the exact usage

count can depend on the specific number of recursive calls (i.e. on the eliminator's natural number argument), and little can therefore be said about the usage count in the general case.

Table 2 also shows the combined usage contributions from the zero and successor branches, allowing us to determine uses of variables that appear there. Associating γ_z with the usage counts of the zero branch and γ_s with the successor branch we have that for $r = 0$, their contribution (χ in the usage rule above) is $\gamma_z \wedge \gamma_s$. In this case the recursive calls are not used, so only one of the zero and successor branches will be used (but we do not know which). Note that if a variable is used once in the zero branch, once in the successor branch, and not at all in the scrutinee, then it counts as being used once. For instance, the program $\mathsf{f}\ m\ n = \mathsf{natrec}^{0}_{1}\ (x_n.\mathbb{N})\ m\ (x_p.x_r.(\mathtt{plus}\ m\ x_p))\ n$ is linear in both arguments. This program corresponds to the following Agda code:

```
f : ℕ → ℕ → ℕ
f m zero    = m
f m (suc n) = m + n
```

The argument m is used linearly in both the zero and successor branches, resulting in its combined use being linear as well.

For $r = 1$ we get $\gamma_z + \omega\gamma_s$: every recursive call is used once, so the zero case will inevitably be used once, while the successor cases could be used any number of times. This was the case for `plus`, where the linear use of the number we do not recurse over comes from its use in the zero branch (γ_z).

Finally, when $r = \omega$, the recursive calls are used an unrestricted number of times so we get $\omega(\gamma_z + \gamma_s)$, using both branches an unrestricted number of times.

5.5 Derived Usage Rules for Encoded Booleans and Vectors

Knowing what the usage counts are for concrete values of p and r we are now able to encode data types[8] like booleans and vectors and figure out what their "usage rules" end up being. It follows from our correctness proof that these rules are "safe" but not necessarily that they are as "exact" as one might expect. For instance, a rule that always assigns ω to all variables would be safe but not particularly useful.

Booleans can be encoded as pairs containing a natural number and a predicate stating that the number is either zero or one. Our encoding corresponds roughly to the following Agda code—though of course we use eliminators instead of explicit pattern matching:[9]

```
Bool : Set
Bool = Σ ℕ IsZeroOrOne

boolrec :
  (P : Bool → Set) → P true → P false → (b : Bool) → P b
boolrec P t f (zero , tt)          = f
boolrec P t f (suc zero , tt)      = t
boolrec P t f (suc (suc n) , not-ok) = emptyrec not-ok

false true : Bool
false = zero , tt
true = suc zero , tt

IsZeroOrOne : ℕ → Set
IsZeroOrOne zero          = ⊤
IsZeroOrOne (suc zero)    = ⊤
IsZeroOrOne (suc (suc n)) = ⊥
```

The encoding of the eliminator `boolrec` uses natrec with $p = 1$ and $r = 0$. This results in the following admissible usage rule where the boolean argument is used linearly and the usage contributions of the true and false branches are combined using the meet operation (some side conditions related

[8] Our encodings use unit types: we have not discussed them in the paper, but they are present in the formalization.
[9] We use a variant of emptyrec which allows us to assign any grade context to the impossible case. This variant is implemented with the help of a strong unit type.

to allowing certain matches are omitted):

$$\frac{\gamma_b \blacktriangleright^m b \qquad \gamma_t \blacktriangleright^m t \qquad \gamma_f \blacktriangleright^m f \qquad \eta.\, x_b : 0 \blacktriangleright^0 P}{\gamma_b + (\gamma_t \wedge \gamma_f) \blacktriangleright^m \mathsf{boolrec}\ P\ t\ f\ b}$$

This rule is exactly what we would expect, and it holds for any grade semiring.

Vectors of length n can be encoded as n-iterated Σ-types. Our encoding roughly corresponds to the following Agda code:

```
Vec : Set → ℕ → Set
Vec A zero    = ⊤
Vec A (suc n) = A × Vec A n

nil : (A : Set) → Vec A zero
nil A = tt

cons :
  (A : Set) (n : ℕ) →
  A → Vec A n → Vec A (suc n)
cons A n x xs = x , xs
```

```
vecrec :
  {A : Set} →
  (P : (n : ℕ) → Vec A n → Set) →
  P zero (nil A) →
  ((n : ℕ) (x : A) (xs : Vec A n) → P n xs →
    P (suc n) (cons A n x xs)) →
  (n : ℕ) (xs : Vec A n) → P n xs
vecrec P nl cs zero    tt       = nl
vecrec P nl cs (suc n) (x , xs) =
  cs n x xs (vecrec P nl cs n xs)
```

Unlike the encoding of `boolrec`, the grades used for natrec in the encoding of `vecrec` are not fixed and instead depend on the grades assigned to the head (p_h), tail (p_t), length of the tail (p_n) and recursive call (r) in the cons-branch. As a consequence of this we have greatest lower bounds in the admissible usage rule (some side conditions are again omitted):

$$\frac{\begin{array}{c}\eta_A \blacktriangleright^0 A \qquad \eta_P.\, x_n : 0.\, x_v : 0 \blacktriangleright^0 P \\ \gamma_{nl} \blacktriangleright^m nl \qquad \gamma_{cs}.\, x_n : m p_n.\, x_h : m p_h.\, x_t : m p_t.\, x_r : m r \blacktriangleright^m cs \qquad \delta_n \blacktriangleright^m n \qquad \delta_{xs} \blacktriangleright^{m \underline{q_{xs}}} xs \\ q_n = \bigwedge_i \mathrm{nr}_i\ r\ 1\ p_n \qquad q_{xs} = \bigwedge_i \mathrm{nr}_i\ r\ p_h\ p_t \qquad \chi = \bigwedge_i \mathrm{nr}_i\ r\ \gamma_{nl}\ \gamma_{cs} \qquad \bigwedge_i \mathrm{nr}_i\ 1\ \mathbf{0}\ \eta_A\ \text{exists}\end{array}}{\chi + q_n \delta_n + q_{xs} \delta_{xs} \blacktriangleright^m \mathsf{vecrec}^r_{p_n, p_h, p_t}\ A\ P\ nl\ cs\ n\ xs}$$

The occurrences of greatest lower bounds makes this rule somewhat difficult to interpret: it is not clear to us whether it, in general, gives an accurate estimate or an overapproximation. However, we can note that the contributions from the nil (γ_{nl}) and cons (γ_{cs}) branches are treated in the same way as in the rule for natrec, and the natural number is also treated in the same way.

Though it may be hard to say something general about the usage rule we can consider the concrete grade semirings for linear and affine types as we did in Section 5.4, making use of Table 2 to find values for the bounds. Doing so indicates that we get reasonable usage counts for the list in at least some cases. For instance, the list is used linearly ($q_{xs} = 1$) when either the recursive call is erased and the head and tail are used linearly, or when the recursive call is used linearly and the tail is erased while the head is used linearly. In the affine types case these all constitute affine uses, together with the cases where the recursive call is erased, exactly one of the head and tail is affine, and exactly one of the head and tail is erased.

Since this encoding uses the length of the vector in a computationally relevant way, the argument n cannot be erased ($q_n \neq 0$), though we do not rule out that another encoding could make this possible. On the other hand, since the recursion is defined over the length and not the list itself, the list can be erased.

5.6 Decidability of Usage

For the system with the natrec-star operator, it has been shown that usage is decidable: it is decidable whether, for a given term t and mode m, there is a context γ such that $\gamma \blacktriangleright^m t$ [3]. The presented

decision procedure involves a function that takes terms to grade contexts in such a way that, for a well-resourced term, the returned grade context is optimal. The context is constructed using the contexts for the sub-terms and the operators of the grade semiring, following the rules of the usage relation. A drawback of the usage rule we proposed in Section 5 is that this approach no longer works in general, because the new rule involves greatest lower bounds that might not exist.

For some grade semirings the sequence $\mathrm{nr}_i\ r\ p\ q$ has a greatest lower bound for all r, p and q; this is the case for all our example semirings. For such semirings the optimal grade context function and the decision procedure for usage can be recovered.

If this is the case, why did we not replace the natrec-star operator with some function providing the greatest lower bound of $\mathrm{nr}_i\ r\ p\ q$ and formulate our usage rule in terms of this? This approach would likely have been feasible, but it would rule out grade semirings for which not all bounds exist. An example of this is provided by the natural numbers (without ω). Since there is no global least element, many grade combinations cause $\mathrm{nr}_i\ r\ p\ q$ to not have any lower bound. Consequently we have not proved decidability of usage for this semiring.[10] Having a "non-computational" usage rule for natrec thus allows for some flexibility in which grade semirings are allowed.

6 Information Flow

Our main motivation has been grades with a quantitative interpretation, but programs can be run in the abstract machine regardless of the grade semiring and most of the results we have shown only require that it supports subtraction and has well-behaved greatest lower bounds. For instance, we may instantiate the theory with a (bounded, distributive) lattice of security levels, following Choudhury et al. [14]. In this case, the addition and multiplication of the semiring are given by the meet and join of the lattice, respectively. We can interpret larger grades as representing data that is "more secret", and 0 and 1 become the maximum and minimum elements, representing maximally secret and maximally public information, respectively. If we allow any grade to be used as a mode[11] with mode multiplication coinciding with grade multiplication, then the usage relation can track information flow: the mode can change for instance when checking a function argument or the scrutinee of prodrec. As an example, when $\mathrm{prodrec}_r^p\ (x.A)\ t\ (x_1.x_2.u)$ is checked, the scrutinee t is checked with access to level r. We can also note that our usage rule for natrec prevents leaking of information, since the use of the scrutinee must be bounded by 1. Since 1 is the least element, this means that we always consider the result of matching on a number to be fully public. In the end, the context γ associates each variable to a security level, representing how secret it is.

Using the abstract machine, with minor modifications, we can show correctness also for these grade semirings in the form of a non-interference theorem. In this interpretation, the multiplicity of a continuation can be thought of as the security level needed to evaluate the scrutinee. For instance, in $\mathrm{prodrec}_r^p\ (x.A)\ t\ (x_1.x_2.u)$, the components of the pair are used at level at least r by u, so no information at a higher level should be allowed while evaluating t. For stacks the multiplicity is given by the product, here join, of all its continuations, but for the empty stack we assign a fixed security level ℓ_0 which represents the security level the user should have access to. The multiplicity of a stack is thus the greatest lower bound of its continuations and the user's clearance level.

The semantics of the machine is left unchanged, but we note that for bounded, distributive lattices, subtraction $p - q = r$ is equivalent to $p \leq q$ and $p = r$. This has two implications for the semantics. Firstly, the variable rule only allows lookups for entries that are at most as secret as the level given by $|S|$, and secondly, the grades assigned to entries are not changed by lookups. This means that the grades assigned to heap entries can now be thought of as representing the security

[10] To be clear, we do not claim that usage is undecidable for this semiring, only that our approach does not work.
[11] We do not require the semiring to have a well-behaved zero in this case.

level needed to access them, and the variable evaluation rule can be thought of as a *monitor* [26], preventing lookups that should not be allowed.

Given two heaps and a security level ℓ we define ℓ-equivalence of the heaps to mean that they are equal on all entries which are at least as public as ℓ, but possibly different elsewhere:

$$\frac{}{\epsilon \sim_\ell \epsilon} \qquad \frac{H_1 \sim_\ell H_2}{H_1.\, y \mapsto^{\ell'} (t, \rho) \sim_\ell H_2.\, y \mapsto^{\ell'} (u, \rho)}\ \ell' \leq \ell \to t = u$$

Using a similar argument as for resource correctness, we can then prove a non-interference property [29]. Unlike Theorem 4.9 we now consider evaluation of open terms but do so only under heaps containing entries for each free variable.

Theorem 6.1. *If secret matches for* prodrec *are disallowed,* $\gamma \blacktriangleright^{\ell_0} t$ *and* $\Gamma \vdash t : \mathbb{N}$*, and* $H_1 \sim_{\ell_0} H_2$ *for some heaps where the heap entries are well-typed and their grades are given by* $\gamma[\rho]$*, then there is a numeral* n *and heaps* H_1' *and* H_2' *such that* $\langle H_1; t; \rho; \epsilon\rangle \twoheadrightarrow^* \langle H_1'; n; \rho'; \epsilon\rangle$ *and* $\langle H_2; t; \rho; \epsilon\rangle \twoheadrightarrow^* \langle H_2'; n; \rho'; \epsilon\rangle$ *and* $H_1' \sim_{\ell_0} H_2'$.

A "secret match" here refers to a use of prodrec^p_ℓ where $\ell \not\leq \ell_0$ in a mode that is not less public than ℓ_0, that is a match on a pair that would reveal information that is "more secret" than the user's clearance. This is similar to the concept of "erased matches" discussed above for quantitative grade semirings.

The theorem says that we can evaluate a program t under two different ℓ_0-equivalent heaps whose entries have security labels matching the security levels of the corresponding variables in t. Doing so yields the same numeral and the entries of the resulting heaps are still ℓ_0-equivalent, differing only on entries that are secret from the perspective of the user.

7 Related Work

This work adds to a large collection of theoretical investigations of graded type theory that occupy various points in the design space in regards to, for instance, the types that are included, support for erased matches, and usage counting in types as well as terms [1, 3, 5, 9, 12, 13, 19, 22, 23]. The dependently typed functional language Idris 2 [11] is a practical implementation of graded type theory, incorporating many of the language features that we investigate in this work, albeit restricted to the one-none-many semiring [22] with grade typing rules based on Quantitative Type Theory (QTT) [5].

We build on our own previous formalization [3], and do not repeat the discussion of related work presented in that text. We focus on correctness for graded modal types, in particular for quantitative interpretations, and on graded type theories with support for some form of inductive data types.

7.1 Resource Correctness

Our correctness proof is very much inspired by the work of Choudhury et al. [13]. Like us, they define an abstract machine that tracks usage counts for variables and prevents lookups if allowed usage would be exceeded. In many respects we follow their approach; we use call-by-name reduction, we use similar methods to track resources in heaps and to check that sufficient resources are available for each evaluation step. We also base our correctness proof on a bisimilarity argument. Yet in some respects, we departed from their design. For one, in contrast to their big-step semantics we define a small-step semantics that handles continuations explicitly. We also use slightly different approaches for tracking variable uses. For instance, while our use of a stack-based machine allows us to infer how many copies are currently being evaluated, their reduction relation is annotated with a grade for this purpose. Heap lookups are also handled a little differently. They allow lookups

of r copies whenever there are $q + r$ copies available and the resulting heap has q copies remaining. This does not uniquely determine the number of remaining copies in all cases. In contrast, we defined subtraction by always taking the least such q, obtaining *deterministic* reduction. We also differ in which type formers are covered: Unlike Choudhury et al. we include natural numbers, strong Σ-types, and an empty type. On the other hand, Choudhury et al. include sum types, which we can encode in our setting using large elimination.

We are not the first to build on Choudhury et al. [13] when giving correctness arguments for graded type systems. This is done also by Bianchini et al. [9, 10], similarly using an abstract machine as an argument for resources being used to correct number of times. Torczon et al. [28] use a similar abstract machine where the focus is on using grades to track effects and coeffects. Despite the different application, the "usage counting" is treated in the same way, with "resources" being removed when variables are evaluated.

In terms of correctness in an operational sense, a different approach is to show a form of subject reduction for resources, that is, that the amount of resources used by a term is preserved by reduction. This is the case, for instance, for Orchard et al. [25] and Moon et al. [23]. As we have argued in Section 5.1, subject reduction alone is not sufficient for showing correctness in a quantitative sense, though it may serve as a reasonable "sanity check" as part of a larger correctness argument. For instance, [3] we do not only prove subject reduction, but also show correctness for erasure in the sense that removing erased content during compilation is safe.

Atkey [5] shows resource-correctness for QTT, a dependent type theory with linearity and erasure, by means of a logical relation between a set-theoretic model of QTT and an interpretation into linear combinatory algebras called BCI algebras. This logical relation is constructed as an instance of a generic model that uses categories with families enriched with quantitative information.

Abel and Bernardy [1] define a relational semantics that is used to obtain "free theorems" for linear (non-dependent) function spaces, in particular that a linear function of type $A \times A \to^1 A \times A$ (polymorphic in A) must either be the identity or the swap function.

7.2 Natural Numbers and Recursion

Let us now discuss some existing methods for integrating one or more inductive data types into graded type theory. We have already discussed in some detail how the natrec-star operator [3] would, at least as presented, be insufficient for quantitative settings in general. However, the main correctness argument of that work concerned erasure, and we can note that for the erasure semiring the usage rule we suggest here coincides with the one using natrec-star.

While QTT [5] does not include recursive data types, Idris 2 [11] has support for linearity and erasure, based on QTT, and allows user-defined data types. To our knowledge the exact rules for how quantities and data types interact in Idris are not documented. However, we have implemented variants of natrec_p^r corresponding to the different values of p and r for the linearity semiring. Through our testing, it appears that our usage rules coincide with those used by Idris in this case.

Nakov and Forsberg [24] extend QTT with polynomial functors, allowing the encoding of inductive data types. These data types come with eliminators that are linear in the matched argument. As a concrete example they consider the natural numbers, deriving a usage rule for which the successor branch may use the natural number 0 times and the recursive call 1 time. In our setting, this is the same as setting $p = 0$ and $r = 1$. In our case we also get linear usage of the scrutinee when $p = 1$ and $r = 0$ (this case is not discussed by Nakov and Forsberg).

Atkey [6] adds some inductive types to (a variant of) QTT in a systematic way. The focus is on polynomial-time computation, and the data types' eliminators are restricted to allow recursion only in the erased fragment. Doing this avoids many of the problems with usage counting in the presence of recursion, though working with such data types might be a bit restrictive in practice. In

addition to the restricted data types there is a type of natural numbers with support for recursion: The eliminator's successor branch only allows uses of the natural number in erased positions, while the recursive call is used once. In addition, both the zero and successor branches are restricted to have no uses of any other variables. The first restriction appeared also in the approach used by Nakov and Forsberg [24], and the second corresponds to having $\gamma_z = \gamma_s = \mathbf{0}$ in our setting.

Another approach is taken by Brunel et al. [12], who support recursion for natural numbers through the combination of pattern matching and a fixpoint combinator for which the "recursive call" is assigned a grade p. The total usage count q of the fixpoint combinator is then given as a solution to $q \leq 1 + pq$. We recognize this inequality as, basically, one of the characteristic inequalities we describe in Section 5.2.

A somewhat similar style of recursion is supported by Bianchini et al. [10], though recursion is achieved not through a fixpoint combinator but by directly allowing lambda expressions to be recursive. They also support encoding natural numbers. Since they allow recursion in a more general sense than us it is somewhat difficult to compare our grade typing to theirs. A notable difference is how usage counts of free variables in the zero and successor branches are handled. In their case, these are required to always be equal in both the zero and successor cases, leading to variables that are used in one branch but unused in the other to be assigned an unrestricted use count. As we discuss in Section 5.4 we are able to give a more precise usage count, at least in some cases. That kind of analysis is presumably not feasible for general recursion since one would not have the same guarantee that the zero case will be reached (exactly one time).

Huang and Yallop [20] extend QTT with a type of lists together with an eliminator, and suggest that the method that is used should be extendable to arbitrary inductive families. In contrast to us, the eliminator is not annotated with grades indicating how many times resources are used by the eliminator. Instead, this information is encoded in the constructors, allowing them to contain several copies of the head and tail of the list—similar to our Σ-types. The eliminator is then linear in the sense that it uses the supplied list once and requires that all available copies of the head and tail are used (up to subsumption).

8 Conclusions

We have investigated a graded modal language with dependent types, formally proving its usage accounting to be correct with respect to an operational semantics for a certain class of grade semirings with quantitative interpretations. In particular, we can instantiate this result with semirings for erasure, linearity and affine types. We have also considered how one can support natural numbers in this kind of system, and how to correctly define usage counting for their eliminators. We believe that our method of dividing the usage contributions of the scrutinee and branches, and deriving side conditions, should be applicable to a larger group of inductive types beyond natural numbers.

The formalization underlying this work [17] is available for further extensions. One obvious idea is to try to add support for more recursive types to the formalization, based on the approach we used for natural numbers. Currently, while we can encode recursive types such as vectors using natural numbers and large elimination, it is unclear whether they can be used as flexibly as natrec. One could also consider whether our method can be adapted to support constructors that take several copies, as supported by Huang and Yallop [20].

Our abstract machine is rather high-level. It might be worthwhile to develop similar results for a more low-level machine that uses a more realistic evaluation strategy, and that incorporates actual erasure. It could also be interesting to make actual use of the support for linear or affine types, for instance by adding support for safe mutability in the style of Bernardy et al. [7].

Finally, one could explore the possibility of implementing a practical type-checker based on our formalization.

Acknowledgments

We would like to thank Dominic Orchard and the anonymous reviewers of this and previous submissions for their feedback, helping us improve this paper. We would also like to thank Naïm Camille Favier, Eve Geng, Gaëtan Gilbert, Ondřej Kubánek, Wojciech Nawrocki, Joakim Öhman, and Andrea Vezzosi for their contributions to the formalization.

Andreas Abel and Oskar Eriksson acknowledge support by *Vetenskapsrådet* (the Swedish Research Council) via project 2019-04216 *Modal typteori med beroende typer* (Modal Dependent Type Theory). Oskar Eriksson additionally acknowledges support by *Knut and Alice Wallenberg Foundation* via project 2019.0116. Nils Anders Danielsson acknowledges support from Vetenskapsrådet (2023-04538).

References

[1] Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP, Article 90 (Aug. 2020), 28 pages. doi:10.1145/3408972

[2] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. *An Agda Formalization of a Graded Modal Type Theory with a Universe and Erasure.* doi:10.5281/zenodo.8119348

[3] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (Aug. 2023), 35 pages. doi:10.1145/3607862

[4] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. doi:10.1145/3158111

[5] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *LICS '18: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science.* ACM Press, 56–65. doi:10.1145/3209108.3209189

[6] Robert Atkey. 2024. Polynomial Time and Dependent Types. *Proc. ACM Program. Lang.* 8, POPL, Article 76 (Jan. 2024), 30 pages. doi:10.1145/3632918

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. doi:10.1145/3158093

[8] Jean-Philippe Bernardy and Patrik Jansson. 2025. Domain-specific tensor languages. *Journal of Functional Programming* 35 (2025), e9. doi:10.1017/S0956796825000048

[9] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Multi-Graded Featherweight Java. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (LIPIcs, Vol. 263).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 3:1–3:27. doi:10.4230/LIPIcs.ECOOP.2023.3

[10] Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. 2023. Resource-Aware Soundness for Big-Step Semantics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 267 (Oct. 2023), 29 pages. doi:10.1145/3622843

[11] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021 (LIPIcs, Vol. 194).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 26 pages. doi:10.4230/LIPIcs.ECOOP.2021.9

[12] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems, 23rd European Symposium on Programming, ESOP 2014 (LNCS, Vol. 8410).* Springer, 351–370. doi:10.1007/978-3-642-54833-8_19

[13] Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A Graded Dependent Type System with a Usage-Aware Semantics. *Proc. ACM Program. Lang.* 5, POPL, Article 50 (Jan. 2021), 32 pages. doi:10.1145/3434331

[14] Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *Programming Languages and Systems, 31st European Symposium on Programming, ESOP 2022 (LNCS, Vol. 13240).* Springer, 403–430. doi:10.1007/978-3-030-99336-8_15

[15] Nils Anders Danielsson, Naïm Camille Favier, and Ondřej Kubánek. 2026. Normalisation for First-Class Universe Levels. *Proceedings of the ACM on Programming Languages* 10, POPL (2026), 25 pages. doi:10.1145/3776645

[16] Nils Anders Danielsson and Eve Geng. 2025. A Formalization of Opaque Definitions for a Dependent Type Theory. In *TyDe '25, Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development.* 39–51. doi:10.1145/3759538.3759653

[17] Oskar Eriksson, Andreas Abel, and Nils Anders Danielsson. 2026. *An Agda Formalization of "On Recursion in Graded Modal Type Theory".* doi:10.5281/zenodo.20489398

[18] Nikolaos Galatos, Peter Jipsen, Tomasz Kowalski, and Hiroakira Ono. 2007. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*. Studies in Logic and the Foundations of Mathematics, Vol. 151. Elsevier, Amsterdam. doi:10.1016/S0049-237X(07)80005-X

[19] Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems, 23rd European Symposium on Programming, ESOP 2014 (LNCS, Vol. 8410)*. Springer, 331–350. doi:10.1007/978-3-642-54833-8_18

[20] Yulong Huang and Jeremy Yallop. 2024. Towards Quantitative Inductive Families. In *30th International Conference on Types for Proofs and Programs, TYPES 2024 - Abstracts*, Patrick Bahr and Rasmus Ejlers Møgelberg (Eds.). IT University of Copenhagen, 84–85. https://vipwww.itu.dk/research/types2024/abstracts.pdf

[21] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (Aug. 2024), 30 pages. doi:10.1145/3674642

[22] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World (LNCS, Vol. 9600)*. Springer, 207–233. doi:10.1007/978-3-319-30936-1_12

[23] Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems, 30th European Symposium on Programming, ESOP 2021 (LNCS, Vol. 12648)*. Springer, 462–490. doi:10.1007/978-3-030-72019-3_17

[24] Georgi Nakov and Fredrik Nordvall Forsberg. 2021. Quantitative Polynomial Functors. In *27th International Conference on Types for Proofs and Programs (TYPES 2021) (LIPIcs, Vol. 239)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 10:1–10:22. doi:10.4230/LIPICS.TYPES.2021.10

[25] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. doi:10.1145/3341714

[26] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50. doi:10.1145/353323.353382

[27] Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7, 3 (1997), 231–264. doi:10.1017/S0956796897002712

[28] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 310 (Oct. 2024), 27 pages. doi:10.1145/3689750

[29] Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. *SIGPLAN Not.* 39, 9 (Sept. 2004), 115–125. doi:10.1145/1016848.1016868

[30] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of Computer Security* 4, 2–3 (April 1996), 167–187. doi:10.3233/JCS-1996-42-304

Received 2026-02-19; accepted 2026-05-13