Monadic refinements for relational cost analysis

IVAN RADIČEK, TU-Wien, Austria
GILLES BARTHE, IMDEA Software Institute, Spain
MARCO GABOARDI, University at Buffalo, SUNY, USA
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
FLORIAN ZULEGER, TU-Wien, Austria

Formal frameworks for cost analysis of programs have been widely studied in the unary setting and, to a limited extent, in the relational setting. However, many of these frameworks focus only on the cost aspect, largely side-lining functional properties that are often a prerequisite for cost analysis, thus leaving many interesting programs out of their purview. In this paper, we show that elegant, simple, expressive proof systems combining cost analysis and functional properties can be built by combining already known ingredients: higher-order refinements and cost monads. Specifically, we derive two syntax-directed proof systems, U^C and R^C, for unary and relational cost analysis, by adding a cost monad to a (syntax-directed) logic of higher-order programs. We study the metatheory of the systems, show that several nontrivial examples can be verified in them, and prove that existing frameworks for cost analysis (RelCost and RAML) can be embedded in them.

CCS Concepts: • **Theory of computation → Logic and verification**; **Proof theory**; **Higher order logic**;

Additional Key Words and Phrases: Cost analysis, monads, relational verification, higher-order logic, refinement types

ACM Reference Format:
Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL, Article 36 (January 2018), 31 pages. https://doi.org/10.1145/3158124

1 INTRODUCTION

Cost analysis aims to statically establish upper and lower bounds on the cost of evaluating a program. It is useful for resource allocation and scheduling problems, especially in embedded and real-time systems, and for the analysis of algorithmic complexity. *Relational* cost analysis aims to statically establish the difference between the costs of two evaluations of one program on different inputs, or the evaluation of two different programs. It is useful for comparing the efficiency of two programs, for reasoning about side-channel security of programs, for the analysis of the complexity of incremental programs and for stability analysis in algorithmic complexity. Both unary and relational cost analyses are supported by a broad range of techniques, including static analyses and type-and-(co)effect systems. Avanzini and Dal Lago [2017]; Bonfante et al. [2011]; Dal Lago and Gaboardi [2011]; Dal Lago and Petit [2013]; Danielsson [2008]; Grobauer [2001]; Gulwani et al. [2009]; Hermenegildo et al. [2005]; Hoffmann et al. [2012] are prominent examples of systems for

Authors' addresses: Ivan Radiček, TU-Wien, Austria; Gilles Barthe, IMDEA Software Institute, Spain; Marco Gaboardi, University at Buffalo, SUNY, USA; Deepak Garg, Max Planck Institute for Software Systems, Germany; Florian Zuleger, TU-Wien, Austria.

2475-1421/2018/1-ART36
https://doi.org/10.1145/3158124

unary cost analysis, whereas type systems for relational cost analysis are presented by Çiçek et al. [2017] and Ngo et al. [2017].

Precise cost analysis is often *value-sensitive* and often requires *functional verification* as a prerequisite. For example, the cost of inserting an element into a sorted list depends on the value of the element to be inserted and the values of the list's elements. In the relational setting, precisely establishing the relative cost of two runs of insertion sort (to establish the sensitivity of the algorithm's cost to input changes) requires proving the functional correctness of the algorithm first. Similarly, proving (via amortized analysis) that incrementing an n-bit counter k times causes only $O(k)$ bit flips requires treating bits with values 0 and 1 differently.

However, formal frameworks for cost analysis usually do not support value-sensitivity and functional verification. We are not aware of any such support in the relational setting and, even in the unary setting, this support is rather limited [Atkey 2011; Danielsson 2008]. Hence, our goal in this paper is to build a formal framework for analyzing cost and, in particular, relative cost that may be value-sensitive or may depend on complex functional properties. Our approach is simple: We start from a sufficiently expressive logic for reasoning about pure programs, extend it with a monad for encapsulating cost-relevant computations and add refinements to the monad to capture precise (unary and relational) costs. This approach has significant merits. First, it is highly expressive. We are able to verify several new examples, both unary and relational, that prior work on cost analysis cannot handle. Second, the resulting system can be used as a *meta framework* for embedding other cost analyses. As instances, we show how to embed RelCost, a type-and-effect system for relational analysis [Çiçek et al. 2017] and the unary amortized analysis of RAML [Hoffmann et al. 2012]. Third, the use of a monad not only separates the cost-relevant computations from the existing pure framework, but also syntactically separates *reasoning* about costs from reasoning about functional properties, thus improving clarity in proofs.

In principle, this general approach can be instantiated for any sufficiently expressive (relational) logic. Here, we choose to build on a recent, theoretically lightweight but expressive logic, RHOL [Aguirre et al. 2017b]. RHOL is a syntax-directed relational program/refinement logic for a simply typed variant of PCF. It manipulates judgments of the form $\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi$, where Γ is a simply typed context, τ_1 and τ_2 are the simple types of e_1 and e_2, Ψ is a set of assumed assertions about free variables and ϕ is a HOL assertion about e_1, e_2.[1] ϕ can be read either as a postcondition for e_1, e_2 or as a relational refinement for the types τ_1, τ_2. This form of judgment, and the associated typing rules, retain the flavor of refinement types—for example, the rules are syntax-directed—but achieve far greater expressiveness.

To reason about costs, we add a new syntactic class of monadic expressions m that explicitly carry cost annotations[2], and a new judgment form

$$\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$$

Informally, the judgment states that, if m_1, m_2, when forced, evaluate with costs n_1, n_2, then $n_1 - n_2 \leq n$ and ϕ holds of the two results. Hence, n is an upper bound on the difference of the costs of m_1 and m_2. We call this proof system R^{C}. We also develop a corresponding unary system, U^{C}, that establishes upper and lower bounds on the cost of a single program.

By its very design, R^{C}'s new judgment syntactically distinguishes reasoning about functional correctness (ϕ) from reasoning about costs (n). This improves clarity in proofs. The rules of R^{C} are syntax-directed. They exploit similarities in the two expressions (e_1, e_2 or m_1, m_2) by analyzing their

[1] HOL is a standard abbreviation for "higher-order simple predicate logic" [Jacobs 1999, Chapter 5]. This is a logic over higher-order programs. It includes quantification at arbitrary types, but excludes impredicative quantification over types and predicates.

[2] The idea of using a separate syntactic class for monadic expressions is due to Pfenning and Davies [2001].

common constructs simultaneously. When the two expressions are dissimilar, additional rules allow analyzing either expression in isolation, or falling back to unary reasoning in U^C, or even falling back to equational reasoning in HOL. This provides R^C great expressiveness—in fact, we show that its expressiveness equals that of HOL. Finally, R^C allows relating the costs of two programs even when their *types* are different, a feature that no prior framework for cost analysis offers.

Despite the expressiveness of the rules, the metatheory of R^C/U^C is simple; we prove the framework sound in a set-theoretic model through a cost-passing interpretation of the monadic expressions. We illustrate R^C/U^C's working on several examples that were out of reach of previous systems. Moreover, we demonstrate that R^C/U^C can be used for embedding other existing cost analyses.

To summarize, the contributions of this work are:

- We present the logic/refinement frameworks R^C and U^C for verifying relational and unary costs of higher-order programs, even when the costs depend on program values or complex functional properties.
- We study the metatheory of both frameworks and prove them sound in a set-theoretic model.
- We show that several nontrivial examples, outside the purview of existing work on cost analysis, can be verified in R^C and U^C.
- We demonstrate that R^C and U^C can be used as meta frameworks for cost analysis by translating two existing systems—RelCost (for relational cost analysis) and RAML (for amortized unary cost analysis)—into them.

A supplementary appendix available from the authors' homepages provides proofs of theorems, omitted rules, details of some examples and additional examples.

Scope and limitations. Our focus is on understanding the fundamentals of cost analysis when costs depend on functional values and cost verification depends on functional invariants. An implemention of R^C and U^C is out of the scope of this paper. Nonetheless, since R^C and U^C separate refinements from typing syntactically, an implementation of interactive proofs based on constraint solving seems feasible.

The programming language underlying R^C and U^C currently lacks nonterminating computations and mutable state. These limitations are inherited from RHOL, on which we build. Further, the cost monad presented here supports only what are called *additive* costs. Although these limitations do not seem to be fundamental, extending R^C and U^C to address them remains an open problem that we plan to address in future work. Section 9 describes our initial ideas in this direction.

2 A LANGUAGE WITH A MONAD FOR COSTS

In this section we present the language of programs we consider in the rest of the paper. The language is a simply typed λ-calculus that syntactically separates pure, cost-free expressions, e, from monadic expressions, m, that have cost. The syntax is shown below.

Types	τ	::=	$\mathsf{b} \mid \theta \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \mathbb{C}(\tau)$
(Pure) expr.	e, n, k, ℓ	::=	$x \mid \langle e, e\rangle \mid \pi_i\ e \mid \mathsf{inj}_i\ e \mid \mathsf{case}\ e\ \mathsf{of}\ e; e \mid e\ e \mid \lambda x.e \mid c \mid K(\overline{e})$
			$\mid \mathsf{match}\ e\ \mathsf{with}\ \overline{K_i \mapsto e_i} \mid \mathsf{rec}\, f(x).e \mid \{m\}$
Monadic expr.	m	::=	$\mathsf{cret}(e) \mid \mathsf{cbind}(e_1, \{x\}.\, m_2) \mid \mathsf{cstep}_n(m)$

Base types are generically denoted b. We assume at least one base type, $\mathbb{R}^\infty$, containing real numbers and $\{-\infty, \infty\}$. Costs are pure expressions of this type. We often use n, k and ℓ in place of e to distinguish costs from other expressions. $\mathbb{C}(\tau)$ is the type of monadic computations that return a result of type τ when forced. θ denotes a defined first-order inductive data type (described later).

Pure expressions are mostly standard. c denotes a constant of a base type. case e of $e_1; e_2$ is case analysis over sum types: If $e = \mathsf{inj}_i\ e'$, then (case e of $e_1; e_2$) reduces to $(e_i\ e')$. $K(\overline{e})$

constructs an expression of a data type by applying the *constructor* K to the terms $\overline{e}$. The expression (match e with $\overline{K_i \mapsto e_i}$) is the corresponding case analysis. If $e = K_j(e'_1, \ldots, e'_k)$, then (match e with $\overline{K_i \mapsto e_i}$) reduces to $(e_j\ e'_1 \ldots e'_k)$. $\mathrm{rec}\, f(x).e$ defines the recursive function f over a variable x, which must be of a data type. The body e can apply f only to arguments smaller than x. In contrast, the type of the argument in a non-recursive function $\lambda x.e$ can be arbitrary. The construct $\{m\}$ is an injection from monadic to pure expressions.

The interesting part of the language are the monadic expressions. The cost of monadic expressions is best understood through a *forcing evaluation*, $m \Downarrow^n e$, which means that the *closed* monadic expression m eventually returns the pure expression e and incurs cost n.

$$\frac{}{\mathrm{cret}(e) \Downarrow^0 e} \qquad \frac{e_1 \to^* \{m_1\} \quad m_1 \Downarrow^n e'_1 \quad m_2[e'_1/x] \Downarrow^{n_2} e_2}{\mathrm{cbind}(e_1, \{x\}.m_2) \Downarrow^{n+n_2} e_2} \qquad \frac{m \Downarrow^{n'} e}{\mathrm{cstep}_n(m) \Downarrow^{n+n'} e}$$

$\mathrm{cret}(e)$, when forced, returns e immediately with 0 cost. This is the usual "return" or "unit" of the monad. Forcing $\mathrm{cbind}(e_1, \{x\}.m_2)$ first evaluates e_1 purely to some $\{m_1\}$ ($\to$ is the effect-free, small-step reduction described later), then forces m_1 to some e'_1 with some cost n and then forces $m_2[e'_1/x]$. The total cost is n plus the cost of forcing $m_2[e'_1/x]$. cbind is the usual "bind" of the monad. $\mathrm{cstep}_n(m)$ is the only non-standard construct. When forced, it forces m, but adds an additional cost n. This is the only way to represent non-zero cost in the language. Note that forcing defines the cost semantics, not the equational theory of monadic expressions, which is defined later.

This style of presenting the monad by separating pure and monadic expressions syntactically owes lineage to the work of Pfenning and Davies [2001, Section 8]. It is crucial to our later development.

Data types and simple typing. A data type is defined by an equation of the form $\theta = K_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \ldots + K_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n})$. This defines a data type θ with n distinct constructors $K_1, \ldots, K_n$. The types of the arguments of the constructors, denoted σ, must be either base types b or the same or other data types. This supports inductive and mutually inductive definitions.[3] For example, the type of lists of integers can be defined as $\mathrm{list} = \mathrm{nil}() + \mathrm{cons}(\mathbb{Z} \times \mathrm{list})$. All data type definitions are assumed to be collected in a context Θ, which we leave implicit in judgments.

A typing environment Γ assigns types to variables, as usual. We define two typing judgments, $\Gamma \vdash e : \tau$ and $\Gamma \vdash m \div \tau$, for pure and monadic expressions, respectively. In the second typing judgment, τ is the type of the expression eventually returned by the monadic expression m (the cost always has type $\mathbb{R}^\infty$). Selected typing rules for the monadic constructs and data types are shown in Figure 1. The remaining rules (for pure expressions) are standard, except for the rule for typing $\mathrm{rec}\, f(x).e$. This rule requires that the body of the recursive definition satisfy a predicate $\mathcal{D}ef(f, x, e)$, which ensures that all recursive calls are performed on smaller arguments. The language has standard metatheoretic properties like subject reduction.

β-reduction and equational theory. The equality relation $\doteq$ defines when two pure or monadic expressions are equal to each other. As usual, we define $\doteq$ as the congruence closure of small-step β-reduction $\to$. On pure expressions, the rules for $\to$ are the expected ones. As an example, $(\mathrm{rec}\, f(x).e)\ e' \to e[(\mathrm{rec}\, f(x).e)/f][e'/x]$. We allow reduction on open expressions and in all contexts (even under binders) since reduction is pure (effect-free).

Following Pfenning and Davies [2001], we also define reduction on monadic expressions. This reduction represents the so-called commuting conversions for the monad. Typically, commuting conversions arrange nested bind operators into a spine; here, we also have additional conversions

[3] Our appendix generalizes data types to allow monadic types $\mathbb{C}(\cdot)$ in definitions, thus permitting reasoning with lazy data structures. The appendix also includes examples of such reasoning.

$$\frac{K(\sigma_1 \times \cdots \times \sigma_n) \in \Theta(\theta) \qquad \Gamma \vdash e_i : \sigma_i \text{ for all } 1 \le i \le n}{\Gamma \vdash K(e_1, \ldots, e_n) : \theta}$$

$$\frac{\Gamma \vdash e : \theta \qquad \begin{array}{c}\theta = K_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \cdots + K_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n}) \in \Theta \\ \Gamma \vdash e_i : \sigma_{i,1} \to \cdots \to \sigma_{i,a_i} \to \tau \text{ for all } 1 \le i \le n\end{array}}{\Gamma \vdash \text{match } e \text{ with } K_1 \mapsto e_1; \cdots ; K_n \mapsto e_n : \tau}$$

$$\frac{\Gamma, f : \theta \to \tau, x : \theta \vdash e : \tau \qquad \mathcal{D}ef(f, x, e)}{\Gamma \vdash \text{rec}\, f(x).e : \theta \to \tau} \qquad \frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \{m\} : \mathbb{C}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{cret}(e) \div \tau}$$

$$\frac{\Gamma \vdash n : \mathbb{R}^\infty \qquad \Gamma \vdash m \div \tau}{\Gamma \vdash \text{cstep}_n(m) \div \tau} \qquad \frac{\Gamma \vdash e : \mathbb{C}(\tau') \qquad \Gamma, x : \tau' \vdash m \div \tau}{\Gamma \vdash \text{cbind}(e, \{x\}.\, m) \div \tau}$$

Fig. 1. Simple typing rules (selected)

for the cstep construct. The main rules are listed below.

$$\text{cstep}_0(m) \to m \qquad \text{cstep}_{n_1}(\text{cstep}_{n_2}(m)) \to \text{cstep}_{n_1+n_2}(m) \qquad \text{cbind}(\{m_1\}, \{x\}.m_2) \to \{\!|m_1/x|\!\}m_2$$

The new substitution $\{\!|m_1/x|\!\}m_2$ is defined by induction on m_1 (not m_2):

$$\{\!|\text{cret}(e)/x|\!\}m_2 \triangleq m_2[e/x] \qquad \{\!|\text{cbind}(e, \{y\}.m)/x|\!\}m_2 \triangleq \text{cbind}(e, \{y\}.\{\!|m/x|\!\}m_2)$$

$$\{\!|\text{cstep}_n(m)/x|\!\}m_2 \triangleq \text{cstep}_n(\{\!|m/x|\!\}m_2)$$

Together, these rules have the effect of *rearranging* all cstep constructs by pushing them outwards and then *coalescing* them at the top-level. For example, we have: $\text{cbind}(\{\text{cstep}_{n_1}(\text{cret}(e_1))\}, \{x\}.\ \text{cstep}_{n_2}(\text{cret}(e_2))) \to \text{cstep}_{n_1}(\text{cstep}_{n_2}(\text{cret}(e_2[e_1/x]))) \to \text{cstep}_{n_1+n_2}(\text{cret}(e_2[e_1/x]))$. This suggests that every closed, well-typed monadic expression has a "normal form" of the shape $\text{cstep}_n(\text{cret}(e))$ where e is the pure expression returned by the monadic expression and n is the cost. This is, in fact, true, as established by the following lemma.

Lemma 2.1. *If m is closed and $m \Downarrow^n e$, then $m \doteq \text{cstep}_n(\text{cret}(e))$.*

Set-theoretic model. We give types and expressions a simple interpretation in set theory. Types are interpreted as sets: Base types map to corresponding sets, e.g., $[\![\mathbb{R}^\infty]\!] \triangleq \mathbb{R} \cup \{-\infty, \infty\}$; products, sums and arrows map homomorphically to their set-theoretic analogues, e.g., $[\![\tau_1 \to \tau_2]\!] \triangleq [\![\tau_1]\!] \to [\![\tau_2]\!]$; data types θ map to initial (tree) algebras. The interpretation of $\mathbb{C}(\tau)$ is more interesting: $[\![\mathbb{C}(\tau)]\!] \triangleq [\![\tau]\!] \times \mathbb{R}^\infty$, representing both the returned pure expression and the cost. (Technically, this makes $\mathbb{C}(\tau)$ a specific *writer* monad.)

The interpretation $(\!|\cdot|\!)_\rho$ of expressions is indexed by a valuation ρ for free variables. Pure expressions have the expected interpretations, e.g., $(\!|\langle e_1, e_2\rangle|\!)_\rho \triangleq \langle (\!|e_1|\!)_\rho, (\!|e_2|\!)_\rho\rangle$. The recursive function definition is interpreted via a fixpoint: $(\!|\text{rec}\, f(x).e|\!)_\rho \triangleq \text{fix}(\lambda F.\, \lambda x.\, (\!|e|\!)_{\rho, f \mapsto F})$. The fixpoint is unique for well-typed functions because e can recursively apply f only to arguments smaller than x.

The interesting interpretation is that of monadic expressions and the construct $\{m\}$. A monadic expression $m \div \tau$ is interpreted as an element of $[\![\tau]\!] \times \mathbb{R}^\infty$, representing the returned expression and the cost. The cost is accumulated over binds using the addition operator + on $\mathbb{R} \cup \{-\infty, \infty\}$.

General rules for equality. Here, $u ::= e \mid m$.

$$\frac{u_1 \to u_2}{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} u_1 \doteq u_2}\ \text{BETA} \qquad \frac{}{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} u \doteq u}\ \text{REFL} \qquad \frac{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} u_1 \doteq u_2 \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi[u_1/x]}{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi[u_2/x]}\ \text{SUBST}$$

Rules for symmetry (SYM) and transitivity (TRANS) of $\doteq$ can be derived

Axioms specific to monadic expression equality

$\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi$ if ϕ is one of:

$$\forall x, y \div \tau.\ \{x\} \doteq \{y\} \Rightarrow x \doteq y \qquad \forall x : \mathbb{C}(\tau).\ \exists y \div \tau.\ x \doteq \{y\}$$

$$\forall x \div \tau.\ \exists y : \mathbb{R}^{\infty}, z : \tau.\ x \doteq \mathrm{cstep}_y(\mathrm{cret}(z))$$

$$\forall x_1, x_2 : \mathbb{R}^{\infty},\ y_1, y_2 : \tau.\ \mathrm{cstep}_{x_1}(\mathrm{cret}(y_1)) \doteq \mathrm{cstep}_{x_2}(\mathrm{cret}(y_2)) \Rightarrow x_1 \doteq x_2 \wedge y_1 \doteq y_2$$

Rules for data types

$$\frac{\begin{array}{c}\theta = K_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \cdots + K_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n}) \in \Theta \\ \Gamma \vdash e : \theta \qquad \Gamma, x_1 : \sigma_{i,1}, \ldots, x_{a_i} : \sigma_{i,a_i}; \Psi, e \doteq K_i(x_1, \ldots, x_{a_i}) \vdash_{\mathrm{L}^{\mathrm{C}}} \phi \\ \text{for all } 1 \le i \le n \qquad \text{where } x_1, \ldots, x_{a_i} \notin \phi\end{array}}{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi}\ \text{ELIM}$$

$$\frac{\theta \in \Theta \qquad \Gamma, x : \theta; \Psi, \forall y : \theta.\ |y| < |x| \Rightarrow \phi[y/x] \vdash_{\mathrm{L}^{\mathrm{C}}} \phi}{\Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \forall x : \theta.\ \phi}\ \text{IND}$$

Fig. 2. L^{C} rules (selected)

This interpretation is reminiscent of Danner et al. [2015] and Grobauer [2001].

$$(\!|\{m\}|\!)_\rho \triangleq (\!|m|\!)_\rho \qquad (\!|\mathrm{cret}(e)|\!)_\rho \triangleq ((\!|e|\!)_\rho, 0) \qquad (\!|\mathrm{cstep}_n(m)|\!)_\rho \triangleq (\pi_1(\!|m|\!)_\rho, (\!|n|\!)_\rho + \pi_2(\!|m|\!)_\rho)$$

$$(\!|\mathrm{cbind}(e, \{x\}.\ m)|\!)_\rho \triangleq \mathsf{let}\ y \leftarrow (\!|e|\!)_\rho\ \mathsf{in}\ \mathsf{let}\ x \leftarrow \pi_1 y\ \mathsf{in}\ \mathsf{let}\ z \leftarrow (\!|m|\!)_\rho\ \mathsf{in}\ (\pi_1 z, \pi_2 y + \pi_2 z)$$

Theorem 2.2 (Soundness). *Let $\rho \vDash \Gamma$ mean that for each $x \in dom(\Gamma)$, $\rho(x) \in [\![\Gamma(x)]\!]$. Then: (1) If $\Gamma \vdash e : \tau$ and $\rho \vDash \Gamma$, then $(\!|e|\!)_\rho \in [\![\tau]\!]$. (2) If $\Gamma \vdash m \div \tau$ and $\rho \vDash \Gamma$, then $(\!|m|\!)_\rho \in [\![\tau]\!] \times \mathbb{R}^{\infty}$.*

Remark. In our language costs must be explicitly specified using the construct cstep_n. Consequently, a program's cost analysis is correct only to the extent that it carries correct cstep_n annotations. Danielsson [2008] calls this style of analysis "semi-formal". We argue that this style aids expressiveness since cstep_n can model different kinds of costs (see Section 6 for examples). Further, our embeddings of RelCost (Section 7) and RAML (Section 8) show how programs written in languages with in-built cost-semantics can be translated to our language automatically.

3 L^{C}: THE ASSERTION LOGIC

L^{C} is a logic of assertions over pure and monadic expressions. It extends HOL [Jacobs 1999, Chapter 5] with quantification and equality over monadic expressions. Formulae are denoted ϕ. The letter u

denotes either a pure or a monadic expression ($u ::= e \mid m$).

$$\phi ::= P(u_1, \ldots, u_n) \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \forall x : \tau.\phi \mid \exists x : \tau.\phi \mid \forall x \div \tau.\phi \mid \exists x \div \tau.\phi$$

P denotes an atomic predicate. Predicates are defined through axioms. We pre-define expression equality, $u_1 \doteq u_2$, and add more predicates for typing examples as needed. The remaining constructs are standard. Variables $x : \tau$ and $x \div \tau$ represent pure and monadic expressions of type τ, respectively.

The logic establishes judgments of the form $\Gamma; \Psi \vdash_{\mathrm{L}^C} \phi$ where Γ is a typing context with assumptions of the forms $x : \tau$ and $x \div \tau$, and Ψ is a context of assumed formulae. The rules for all logical connectives are standard. We show in Figure 2 some selected rules pertaining to equality and data types. Importantly, the rule BETA subsumes reduction $\rightarrow$ into the logic's equality $\doteq$. The axioms in the middle of the figure specify important properties of $\doteq$ on monadic expressions. The third axiom formalizes the normal form of monadic expressions described at the end of Section 2. Other interesting properties of equality can be derived. For instance, $\mathrm{cstep}_n(m) \doteq \mathrm{cstep}_{n'}(\mathrm{cret}(e)) \Rightarrow \exists n''.\ n' \doteq n + n'' \wedge m \doteq \mathrm{cstep}_{n''}(\mathrm{cret}(e))$.

The rules ELIM and IND allow case analysis and induction on data types. In the rule IND, the notation $|e|$ stands for the *depth* of e (which must be of some data type θ). Informally, the depth is the maximum number of constructor applications on any path in the normal form of e viewed as a tree. (The appendix defines this formally.)

Model. L^C has a straightforward model in set theory. Connectives are interpreted as expected, e.g., $\Rightarrow$ is interpreted as implication in set theory. The logic's equality $\doteq$ maps to equality in set theory. We write this interpretation as $(\!|\phi|\!)_\rho$. All rules (and axioms) of L^C are sound in this model.

THEOREM 3.1 (SOUNDNESS). *If* $\Gamma; \Psi \vdash_{L^C} \phi$, $\rho \vDash \Gamma$ *and* $\bigwedge_{\phi' \in \Psi} (\!|\phi'|\!)_\rho$, *then* $(\!|\phi|\!)_\rho$.

4 U^C: UNARY COST ANALYSIS

Next, we present U^C, a syntax-directed proof system for *unary* cost analysis. Since cost analysis often depends on functional properties, U^C builds-in an expressive program logic using the assertions of L^C. In this aspect, U^C is inspired by UHOL, a program logic/refinement type system for proving functional properties of pure expressions [Aguirre et al. 2017b]. U^C uses two judgments:

$$\Gamma; \Psi \vdash e : \tau \mid \phi \qquad\qquad \Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi$$

The first judgment, also called the pure judgment, means that under assumptions Ψ, e (of simple type τ) satisfies $\phi[e/\mathbf{r}]$, where $\mathbf{r}$ is a distinguished variable in ϕ representing the result of e. ϕ can be viewed as either a postcondition for e or a refinement for the type τ. The second judgment, also called the monadic judgment, is central to cost analysis. It means that m, when forced, returns a pure expression e which satisfies $\phi[e/\mathbf{r}]$ and k and ℓ are expressions that denote, respectively, *lower and upper bounds* on the cost of m. In verifying programs, we use ϕ to represent functional properties of the output, and use k and ℓ to bound the costs.[4]

Figure 3 shows the rules for establishing the two judgments. The rules are mostly *syntax-directed*, which simplifies verification of examples. For pure expressions, we show rules for only a few constructs. (Rules for the remaining constructs are the same as those in UHOL.) We can establish any refinement ϕ for a variable x if we can show $\phi[x/\mathbf{r}]$ from the assumptions Ψ in the assertion logic (rule U-VAR). For function types $\tau \rightarrow \tau'$, the refinement has the shape $\forall x.\phi \Rightarrow \phi'$, where ϕ is a refinement on the argument x and ϕ' is a refinement on the result. In the rule U-LETREC for $\mathrm{rec}\, f(x).e$, we make the assumption that the refinement holds for the function f for all arguments y with

[4]The cost bounds k and ℓ are drawn from $\mathbb{R}^\infty$ and can be $-\infty$ and ∞. Since our language is terminating, no program actually has unbounded costs, but we find these extreme bounds handy in verification when we do not care about a more precise bound on one side. Section 6.3 of our appendix presents an example of such a situation.

Rules for the pure judgment $\Gamma; \Psi \vdash e : \tau \mid \phi$ (selected)

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi[x/\mathbf{r}]}{\Gamma; \Psi \vdash x : \tau \mid \phi}\ \text{U-VAR}$$

$$\frac{\begin{array}{c}\mathcal{D}ef(f, x, e) \\ \Gamma, x : \theta, f : \theta \to \tau; \Psi, \phi, \forall y.\ |y| < |x| \Rightarrow \phi[y/x] \Rightarrow \phi'[y/x][f\ y/\mathbf{r}] \vdash e : \tau \mid \phi'\end{array}}{\Gamma; \Psi \vdash \mathsf{rec}\, f(x).e : \theta \to \tau \mid \forall x.\ \phi \Rightarrow \phi'[\mathbf{r}\ x/\mathbf{r}]}\ \text{U-LETREC}$$

$$\frac{\Gamma; \Psi \vdash e_1 : \tau \to \tau' \mid \forall x.\phi \Rightarrow \phi'[\mathbf{r}\ x/\mathbf{r}] \qquad \Gamma; \Psi \vdash e_2 : \tau \mid \phi[\mathbf{r}/x]}{\Gamma; \Psi \vdash e_1\ e_2 : \tau' \mid \phi'}\ \text{U-APP}$$

$$\frac{\begin{array}{c}K(\sigma_1 \times \cdots \times \sigma_n) \in \Theta(\theta) \qquad \Gamma; \Psi \vdash e_i : \sigma_i \mid \phi_i \ \text{ for all } 1 \le i \le n \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \phi_1[x_1/\mathbf{r}] \Rightarrow \cdots \Rightarrow \phi_n[x_n/\mathbf{r}] \Rightarrow \phi[K(x_1, \ldots, x_n)/\mathbf{r}]\end{array}}{\Gamma; \Psi \vdash K(e_1, \ldots, e_n) : \theta \mid \phi}\ \text{U-CONS}$$

$$\frac{\begin{array}{c}\theta = K_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \cdots + K_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n}) \in \Theta \\ \Gamma; \Psi \vdash e : \theta \mid \phi' \qquad \text{For all } 1 \le i \le n : \quad \Gamma; \Psi \vdash e_i : \sigma_{i,1} \to \cdots \sigma_{i,a_i} \to \tau \mid \phi_i' \quad \text{where} \\ \phi_i' \equiv \forall x_1 : \sigma_{i,1}, \ldots, x_{a_i} : \sigma_{i,a_i}.\ \phi'[K_i(x_1, \ldots, x_{a_i})/\mathbf{r}] \Rightarrow \phi[(\mathbf{r}\ x_1\ \cdots\ x_{a_i})/\mathbf{r}]\end{array}}{\Gamma; \Psi \vdash \mathsf{match}\ e\ \mathsf{with}\ K_1 \mapsto e_1; \ldots; K_n \mapsto e_n : \tau \mid \phi}\ \text{U-MATCH}$$

$$\frac{\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi[\mathbf{r}/x]}{\Gamma; \Psi \vdash \{m\} : \mathbb{C}(\tau) \mid \mathbb{C}_{\mathrm{u}}(\mathbf{r}, k, \ell, x.\phi)}\ \text{U-MONAD}$$

Rules for the monadic judgment $\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi$

$$\frac{\Gamma; \Psi \vdash e : \tau \mid \phi}{\Gamma; \Psi \vdash \mathsf{cret}(e) \div \tau \mid 0 \mid 0 \mid \phi}\ \text{U-RET} \qquad \frac{\Gamma \vdash n : \mathbb{R}^{\infty} \qquad \Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi}{\Gamma; \Psi \vdash \mathsf{cstep}_n(m) \div \tau \mid k + n \mid \ell + n \mid \phi}\ \text{U-STEP}$$

$$\frac{\Gamma; \Psi \vdash e_1 : \mathbb{C}(\tau_1) \mid \mathbb{C}_{\mathrm{u}}(\mathbf{r}, k', \ell', x.\phi_1) \qquad \Gamma, x : \tau_1; \Psi, \phi_1 \vdash m_2 \div \tau_2 \mid k \mid \ell \mid \phi_2 \qquad x \notin k, \ell, \phi_2}{\Gamma; \Psi \vdash \mathsf{cbind}(e_1, \{x\}.m_2) \div \tau_2 \mid k' + k \mid \ell' + \ell \mid \phi_2}\ \text{U-BIND}$$

Structural rules (selected)

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash e : \tau \mid \phi' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi'[e/\mathbf{r}] \Rightarrow \phi[e/\mathbf{r}]\end{array}}{\Gamma; \Psi \vdash e : \tau \mid \phi}\ \text{U-SUB} \qquad \frac{\begin{array}{c}\Gamma; \Psi \vdash m \div \tau \mid k' \mid \ell' \mid \phi \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} k \le k' \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \ell' \le \ell\end{array}}{\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi}\ \text{U-SUBC}$$

Admissible rules (selected)

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash m \div \tau \mid k' \mid \ell' \mid \phi' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} m \doteq \mathsf{cstep}_n(\mathsf{cret}(e)) \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} k \le n \le \ell \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi'[e/\mathbf{r}] \Rightarrow \phi[e/\mathbf{r}]\end{array}}{\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi}\ \text{U-SUBM1} \qquad \frac{\begin{array}{c}\Gamma; \Psi \vdash m \div \tau \mid k' \mid \ell' \mid \phi' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} k \le k' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \ell' \le \ell \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \forall \mathbf{r}.\ \phi' \Rightarrow \phi\end{array}}{\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi}\ \text{U-SUBM2}$$

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash e : \tau \mid \phi \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} e \doteq e' \qquad \Gamma \vdash e' : \tau\end{array}}{\Gamma; \Psi \vdash e' : \tau \mid \phi}\ \text{U-EQ-PURE} \qquad \frac{\begin{array}{c}\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} m \doteq m' \qquad \Gamma \vdash m' \div \tau\end{array}}{\Gamma; \Psi \vdash m' \div \tau \mid k \mid \ell \mid \phi}\ \text{U-EQ-MONADIC}$$

Fig. 3. U^{C} rules

$|y| < |x|$, and show that e has the postcondition ϕ'. Rule U-APP allows applying a function if the argument satisfies the pre-condition.

Rule U-CONS says that a data type constructor $K(e_1, \ldots, e_n)$ can be typed with refinement ϕ if the constructor transforms arguments with refinements $\{\phi_i\}_{i=1}^n$ to ϕ and the arguments actually have these refinements. Dually, U-MATCH allows establishing refinement ϕ for (match e with $K_1 \mapsto e_1; \ldots; K_n \mapsto e_n$) when e has some refinement ϕ' and each function e_i maps arguments $x_1, \ldots, x_{a_i}$ satisfying $\phi'[K_i(x_1, \ldots, x_{a_i})/\mathbf{r}]$ to a result satisfying ϕ.

Next, we *define* a refinement $\mathbb{C}_{\mathbf{U}}(e, k, \ell, x.\phi)$ for the monadic type $\mathbb{C}(\tau)$.

$$\mathbb{C}_{\mathbf{U}}(e, k, \ell, x.\phi) \triangleq \exists n, y.\ (e \doteq \{\mathrm{cstep}_n(\mathrm{cret}(y))\}) \wedge (k \leq n \leq \ell) \wedge \phi[y/x]$$

Note that x is locally bound in ϕ. In words, the refinement means that e is (up to $\doteq$) $\{\mathrm{cstep}_n(\mathrm{cret}(y))\}$, i.e., a suspended computation which eventually returns y with cost n, that the cost n is lower- and upper-bounded by k and ℓ, respectively, and that y satisfies ϕ. Due to Lemma 2.1, a consequence is that if $e \to^* \{m\}$, then m, when forced, returns a pure expression (named y) that satisfies ϕ and the cost of this forcing is lower- and upper-bounded by k and ℓ, respectively.

Rule U-MONAD says that to verify the expression $\{m\}$, we should verify the monadic expression m using the monadic judgment, showing that m has some cost lower and upper bounds k and ℓ and some postcondition ϕ. Then, the postcondition of $\{m\}$ is $\mathbb{C}_{\mathbf{U}}(\mathbf{r}, k, \ell, x.\phi[x/\mathbf{r}])$.

Monadic expressions. The crux of U^{C} are the rules for typing monadic expressions. Rule U-RET says that $\mathrm{cret}(e)$ has postcondition ϕ if e has postcondition ϕ, and that the cost of $\mathrm{cret}(e)$ is both lower- and upper-bounded by 0. This represents the fact that $\mathrm{cret}(e)$ forces to e with 0 cost. Rule U-STEP says that $\mathrm{cstep}_n(m)$ has cost bounds $k + n$ and $\ell + n$, and postcondition ϕ if m has cost bounds k and ℓ and the same postcondition ϕ. This represents the fact that $\mathrm{cstep}_n(m)$ forces like m, but with additional cost n. Finally, to type $\mathrm{cbind}(e_1, \{x\}.m_2)$, we first verify e_1 (which has a monadic type) purely with a refinement $\mathbb{C}_{\mathbf{U}}(\mathbf{r}, k', \ell', x.\phi_1)$, then verify m_2 assuming that x satisfies ϕ_1. The bounds on the cost of $\mathrm{cbind}(e_1, \{x\}.m_2)$ are obtained by adding k' and ℓ' to the bounds of m_2, and the postcondition is the same as that of m_2. Again, this directly reflects how $\mathrm{cbind}(e_1, \{x\}.m_2)$ forces.

Note that our monadic judgment separates reasoning about functional properties from reasoning about costs *syntactically*. This elegance is a consequence of setting up the monad in the judgmental style of Pfenning and Davies [2001], which isolates all reasoning about the effect (cost in this case) in a separate monadic judgment.

Structural and admissible rules. The rule U-SUB weakens the postcondition of a pure expression. The rule U-SUBC weakens the cost bounds of an monadic expression. This rule does *not* weaken the postcondition. Rules to weaken the postcondition of monadic expressions are, in fact, admissible in the system (U-SUBM1 and U-SUBM2). The admissible rules U-EQ-PURE and U-EQ-MONADIC show that U^{C}'s judgments are closed under $\doteq$.

Metatheory. Our main metatheoretic result about U^{C} is that it has a sound and complete interpretation in L^{C}.

Theorem 4.1 (Equivalence of U^{C} and L^{C}). *The following hold.*

(1) $\Gamma; \Psi \vdash e : \tau \mid \phi$ *if and only if* $\Gamma; \Psi \vdash_{L^C} \phi[e/\mathbf{r}]$ *(and* $\Gamma \vdash e : \tau$*).*

(2) $\Gamma; \Psi \vdash m \div \tau \mid k \mid \ell \mid \phi$ *if and only if* $\Gamma; \Psi \vdash_{L^C} \exists y, n.\ (m \doteq \mathrm{cstep}_n(\mathrm{cret}(y))) \wedge (k \leq n \leq \ell) \wedge \phi[y/\mathbf{r}]$ *(and* $\Gamma \vdash m \div \tau$*).*

Proof. The $\Rightarrow$ direction of (1) and (2) follows by induction on the given derivations. The $\Leftarrow$ direction is established by showing that every well-typed pure or monadic expression can be given a trivial refinement $\top$, and then using rule U-SUB for pure expressions and another small induction on typing derivations for monadic expressions. □

This theorem has several useful consequences. First, it *explains* the meaning of U^{C}'s judgments. Second, the $\Rightarrow$ direction implies that U^{C} has sound set-theoretic semantics (since L^{C} has them). Third, the theorem is directly useful in verification: It allows switching to the assertion logic L^{C} in a proof. This is useful in many of our examples. Fourth, at a conceptual level, the theorem establishes that the mostly syntax-directed style of U^{C} does not reduce expressiveness—it is as expressive as L^{C}, which already includes the entire equational theory of our language. Fifth, the theorem immediately implies that the rules marked admissible in Figure 3 are indeed admissible. Finally, a corollary to this theorem and Lemma 2.1 is the following subject reduction for the monadic judgment with respect to the forcing semantics that also takes costs into account.

Theorem 4.2 (Forcing subject reduction). *If* $\vdash m \div \tau \mid k \mid \ell \mid \phi$ *and* $m \Downarrow^n e$, *then* $\vdash_{L^C} k \leq n \leq \ell$ *and* $\vdash e : \tau \mid \phi$.

5 R^{C}: RELATIONAL COST ANALYSIS

Finally, we present R^{C}, a syntax-directed proof system for *relational* cost analysis. Like U^{C}, R^{C} is based on two judgments, but these judgments relate *pairs* of pure and monadic expressions.

$$\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi \qquad \Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$$

The first judgment, also called the pure judgment, is based on RHOL [Aguirre et al. 2017b] and means that under assumptions Ψ, e_1 (of simple type τ_1) and e_2 (of simple type τ_2) satisfy $\phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2]$. Here, $\mathbf{r}_1$ and $\mathbf{r}_2$ are distinguished variables in the relational assertion ϕ representing the results of e_1 and e_2 respectively. The second judgment, also called the monadic judgment, is the foundation of our relational cost analysis. It means that m_1 and m_2, when forced, evaluate respectively with some costs n_1 and n_2 to some pure expressions e_1 and e_2, $\phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2]$ holds and n is an upper bound on the *relative cost* $n_1 - n_2$ of m_1 with respect to m_2. In verifying programs, we use ϕ to represent relational properties of the outputs, and use n to bound the relative cost.[5]

Figures 4 and 5 show the syntax-directed rules for establishing the two judgments. The *two-sided* rules (Figure 4) apply when both expressions have the same top-level construct; they analyze this common construct. The *one-sided* rules of Figure 5 analyze either the left or the right expression, thus allowing verification to proceed even when the expressions are dissimilar. (Figure 5 shows only the left rules.) For pure expressions, we show rules for only a few constructs. Rules for the remaining constructs are taken as-is from RHOL. The two sided rule R-VAR relates the variables x_1, x_2 at ϕ when $\phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2]$ holds. The one-sided rule R-VAR-L relates x_1 to an arbitrary e_2 when $\phi[x_1/\mathbf{r}_1]$ holds and $\mathbf{r}_2$ does not appear in ϕ (however, ϕ may contain e_2). The two-sided rule R-LETREC applies to two recursive function definitions $\mathrm{rec}\, f_1(x_1).e_1$ and $\mathrm{rec}\, f_2(x_2).e_2$. In the premise, we get to assume the relational refinement for all arguments y_1, y_2 with $(|y_1|, |y_2|) < (|x_1|, |x_2|)$, which holds when both $|y_1| \leq |x_1|$ and $|y_2| \leq |x_2|$ and at least one of the inequalities is strict. The corresponding one-sided rule R-LETREC-L allows unfolding a recursive function definition on the left side only. The application rules R-APP and R-APP-L are dual. The rule R-MATCH relates two case-analyses on data types. We show here only a simplified version of the rule where the analyzed data type θ is the same on both sides. For a data type with n constructors, the rule has n^2 cases in the premises. The corresponding one-sided rule is elided here.

Next, we *define* a relational refinement $\mathbb{C}_{\mathbf{r}}(e_1, e_2, n, x_2.x_2.\phi)$ for the monadic type $\mathbb{C}(\tau)$:

$$\exists y_1, n_1, y_2, n_2.\ e_1 \doteq \{\mathrm{cstep}_{n_1}(\mathrm{cret}(y_1))\} \wedge e_2 \doteq \{\mathrm{cstep}_{n_2}(\mathrm{cret}(y_2))\} \wedge \phi[y_1/x_1][y_2/x_2] \wedge n_1 - n_2 \leq n$$

[5] Our current development only supports establishing a specific kind of predicate on n_1, n_2, namely, an upper bound on $n_1 - n_2$. We believe this can be generalized to arbitrary predicates on n_1, n_2 but we haven't worked out the generalization since none of our examples so far have required this.

Two-sided rules for the pure judgment $\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi$ (selected)

$$\frac{\Gamma \vdash x_1 : \tau_1 \qquad \Gamma \vdash x_2 : \tau_2 \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2]}{\Gamma; \Psi \vdash x_1 : \tau_1 \sim x_2 : \tau_2 \mid \phi} \text{R-VAR}$$

$$\frac{\begin{array}{c}\mathcal{D}ef(f_1, x_1, e_1) \qquad \mathcal{D}ef(f_2, x_2, e_2) \\ \Gamma, x_1 : \theta_1, x_2 : \theta_2, f_1 : \theta_1 \to \tau_1, f_2 : \theta_2 \to \tau_2; \\ \Psi, \phi, \forall y_1 y_2.\ (|y_1|, |y_2|) < (|x_1|, |x_2|) \Rightarrow \phi[y_1/x_1][y_2/x_2] \Rightarrow \phi'[y_1/x_1][y_2/x_2][f_1\ y_1/\mathbf{r}_1][f_2\ y_2/\mathbf{r}_2] \\ \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi'\end{array}}{\Gamma; \Psi \vdash \mathrm{rec}\, f_1(x_1).e_1 : \theta_1 \to \tau_1 \sim \mathrm{rec}\, f_2(x_2).e_2 : \theta_2 \to \tau_2 \mid \forall x_1 x_2.\ \phi \Rightarrow \phi'[\mathbf{r}_1\ x_1/\mathbf{r}_1][\mathbf{r}_2\ x_2/\mathbf{r}_2]} \text{R-LETREC}$$

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash e_1 : \tau_1 \to \tau_1' \sim e_2 : \tau_2 \to \tau_2' \mid \forall x_1 x_2.\phi \Rightarrow \phi'[\mathbf{r}_1\ x_1/\mathbf{r}_1][\mathbf{r}_2\ x_2/\mathbf{r}_2] \\ \Gamma; \Psi \vdash e_1' : \tau_1 \sim e_2' : \tau_2 \mid \phi[\mathbf{r}_1/x_1][\mathbf{r}_2/x_2]\end{array}}{\Gamma; \Psi \vdash e_1\ e_1' : \tau_1' \sim e_2\ e_2' : \tau_2' \mid \phi'} \text{R-APP}$$

$$\frac{\begin{array}{c}\theta = K_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \cdots + K_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n}) \in \Theta \qquad \Gamma; \Psi \vdash e : \theta \sim e' : \theta \mid \phi' \\ \text{For all } 1 \le i, j \le n:\ \Gamma; \Psi \vdash e_i : \sigma_{i,1} \to \cdots \to \sigma_{i,a_i} \to \tau_1 \sim e_j' : \sigma_{j,1} \to \cdots \to \sigma_{j,a_j} \to \tau_2 \mid \phi_{i,j}' \text{ where} \\ \phi_{i,j}' \equiv \forall x_1 : \sigma_{i,1}, \ldots, x_{a_i} : \sigma_{i,a_i}, y_1 : \sigma_{j,1}, \ldots, y_{a_j} : \sigma_{j,a_j}.\ \phi'[K_i(x_1, \ldots, x_{a_i})/\mathbf{r}_1][K_j(y_1, \ldots, y_{a_j})/\mathbf{r}_2] \\ \Rightarrow \phi[(\mathbf{r}_1\ x_1\ \cdots\ x_{a_i})/\mathbf{r}_1][(\mathbf{r}_2\ y_1\ \cdots\ y_{a_j})/\mathbf{r}_2]\end{array}}{\Gamma; \Psi \vdash \mathrm{match}\ e\ \mathrm{with}\ K_1 \mapsto e_1; \ldots; K_n \mapsto e_n : \tau_1 \sim \mathrm{match}\ e'\ \mathrm{with}\ K_1 \mapsto e_1'; \ldots; K_n \mapsto e_n' : \tau_2 \mid \phi} \text{R-MATCH}$$

$$\frac{\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}{\Gamma; \Psi \vdash \{m_1\} : \mathbb{C}(\tau_1) \sim \{m_2\} : \mathbb{C}(\tau_2) \mid \mathbb{C}_{\mathrm{r}}(\mathbf{r}_1, \mathbf{r}_2, n, \mathbf{r}_1.\mathbf{r}_2.\phi)} \text{R-MONAD}$$

Two-sided rules for the monadic judgment $\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$

$$\frac{\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi}{\Gamma; \Psi \vdash \mathrm{cret}(e_1) \div \tau_1 \sim \mathrm{cret}(e_2) \div \tau_2 \mid 0 \mid \phi} \text{R-RET}$$

$$\frac{\Gamma \vdash n_1 : \mathbb{R}^{\infty} \qquad \Gamma \vdash n_2 : \mathbb{R}^{\infty} \qquad \Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}{\Gamma; \Psi \vdash \mathrm{cstep}_{n_1}(m_1) \div \tau_1 \sim \mathrm{cstep}_{n_2}(m_2) \div \tau_2 \mid n + n_1 - n_2 \mid \phi} \text{R-STEP}$$

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash e_1 : \tau_1' \sim e_2 : \tau_2' \mid \mathbb{C}_{\mathrm{r}}(\mathbf{r}_1, \mathbf{r}_2, n', x_1.x_2.\phi') \\ \Gamma, x_1 : \tau_1', x_2 : \tau_2'; \Psi, \phi' \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi \qquad x_1, x_2 \notin n, \phi\end{array}}{\Gamma; \Psi \vdash \mathrm{cbind}(e_1, \{x_1\}.\, m_1) \div \tau_1 \sim \mathrm{cbind}(e_2, \{x_2\}.\, m_2) \div \tau_2 \mid n' + n \mid \phi} \text{R-BIND}$$

Fig. 4. R^{C} two-sided rules

Here, x_1, x_2 are locally bound in ϕ. The relational refinement means that e_1 and e_2 are (up to $\doteq$) $\{\mathrm{cstep}_{n_1}(\mathrm{cret}(y_1))\}$ and $\{\mathrm{cstep}_{n_2}(\mathrm{cret}(y_2))\}$, i.e., two suspended computation which eventually return y_1 and y_2 with costs n_1 and n_2 respectively, that the relative cost $n_1 - n_2$ is upper-bounded by n, and that y_1 and y_2 satisfy the relational assertion ϕ. Lemma 2.1 then implies that if $e_1 \to^* \{m_1\}$ and $e_2 \to^* \{m_2\}$, then m_1 and m_2, when forced, return pure expressions (named y_1 and y_2) that satisfy ϕ and the difference in the costs of forcing is upper-bounded by n.

Rule R-MONAD says that to verify that the expressions $\{m_1\}$ and $\{m_2\}$ are related at the assertion $\mathbb{C}_{\mathrm{r}}(\mathbf{r}_1, \mathbf{r}_2, n, \mathbf{r}_1.\mathbf{r}_2.\phi)$, we should verify, using the monadic judgment, that the monadic expressions m_1 and m_2 are related at the assertion ϕ and that their relative cost is at most n.

One-sided rules for the pure judgment $\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi$ (selected)

$$\frac{\Gamma \vdash x_1 : \tau_1 \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi[x_1/\mathbf{r}_1] \qquad \mathbf{r}_2 \notin FV(\phi) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma; \Psi \vdash x_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi}\ \text{R-VAR-L}$$

$$\frac{\begin{array}{c}\mathcal{D}ef(f,x,e) \\ \Gamma, x : \theta, f : \theta \to \tau_1; \Psi, \phi, \forall y.\ |y| < |x| \Rightarrow \phi[y/x] \Rightarrow \phi'[y/x][f\ y/\mathbf{r}_1][e_2/\mathbf{r}_2] \vdash e : \tau_1 \sim e_2 : \tau_2 \mid \phi'\end{array}}{\Gamma; \Psi \vdash \mathrm{rec}\, f(x).e : \theta \to \tau_1 \sim e_2 : \tau_2 \mid \forall x.\ \phi \Rightarrow \phi'[\mathbf{r}_1\ x_1/\mathbf{r}_1]}\ \text{R-LETREC-L}$$

$$\frac{\Gamma; \Psi \vdash e : \tau \to \sigma_1 \sim e_2 : \sigma_2 \mid \forall x.\phi \Rightarrow \phi'[\mathbf{r}\ x/\mathbf{r}] \qquad \Gamma; \Psi \vdash e' : \tau \mid \phi[\mathbf{r}/x]}{\Gamma; \Psi \vdash e\ e' : \sigma_1 \sim e_2 : \sigma_2 \mid \phi'}\ \text{R-APP-L}$$

One-sided rules for the monadic judgment $\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$

$$\frac{\Gamma \vdash e_1 \div \tau_1 \qquad \Gamma; \Psi \vdash m_2 \div \tau_2 \mid k \mid \ell \mid \phi[e_1/\mathbf{r}_1][\mathbf{r}/\mathbf{r}_2]}{\Gamma; \Psi \vdash \mathrm{cret}(e_1) \div \tau_1 \sim m_2 \div \tau_2 \mid -k \mid \phi}\ \text{R-RET-L}$$

$$\frac{\Gamma \vdash n_1 : \mathbb{R}^{\infty} \qquad \Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}{\Gamma; \Psi \vdash \mathrm{cstep}_{n_1}(m_1) \div \tau_1 \sim m_2 \div \tau_2 \mid n + n_1 \mid \phi}\ \text{R-STEP-L}$$

$$\frac{\Gamma; \Psi \vdash e_1 : \mathbb{C}(\tau_1') \mid \mathbb{C}_{\mathrm{u}}(\mathbf{r}, k, \ell, x.\phi') \qquad \Gamma, x : \tau_1'; \Psi, \phi' \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}{\Gamma; \Psi \vdash \mathrm{cbind}(e_1, \{x\}.\, m_1) \div \tau_1 \sim m_2 \div \tau_2 \mid \ell + n \mid \phi}\ \text{R-BIND-L}$$

Fig. 5. R^{C} one-sided rules

Structural rules

$$\frac{\begin{array}{c}\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi'[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2] \Rightarrow \phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2]\end{array}}{\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi}\ \text{R-SUB} \qquad \frac{\begin{array}{c}\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n' \mid \phi \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} n' \le n\end{array}}{\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}\ \text{R-SUBC}$$

Admissible rules

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n' \mid \phi' \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} m_1 \doteq \mathrm{cstep}_{n_1}(\mathrm{cret}(e_1)) \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} m_2 \doteq \mathrm{cstep}_{n_2}(\mathrm{cret}(e_2)) \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} n_1 - n_2 \le n \\ \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \phi'[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2] \Rightarrow \phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2]\end{array}}{\Psi; \Gamma \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}\ \text{R-SUBM1} \qquad \frac{\begin{array}{c}\Psi; \Gamma \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n' \mid \phi' \\ \Psi; \Gamma \vdash_{\mathrm{L}^{\mathrm{C}}} n' \le n \\ \Psi; \Gamma \vdash_{\mathrm{L}^{\mathrm{C}}} \forall \mathbf{r}_1, \mathbf{r}_2.\ \phi' \Rightarrow \phi\end{array}}{\Psi; \Gamma \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi}\ \text{R-SUBM2}$$

$$\frac{\Gamma; \Psi \vdash m_1 \div \tau_1 \mid k_1 \mid \ell_1 \mid \phi_1 \qquad \Gamma; \Psi \vdash m_2 \div \tau_2 \mid k_2 \mid \ell_2 \mid \phi_2 \qquad \Gamma; \Psi \vdash_{\mathrm{L}^{\mathrm{C}}} \ell_1 - k_2 \le n}{\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi_1[\mathbf{r}_1/\mathbf{r}] \wedge \phi_2[\mathbf{r}_2/\mathbf{r}]}\ \text{R-SPLIT}$$

Fig. 6. R^{C} structural and admissible rules (selected)

Monadic expressions. Rule R-RET says that we can relate $\mathrm{cret}(e_1)$ and $\mathrm{cret}(e_2)$ at ϕ if e_1 and e_2 are related at ϕ, and that the relative cost is upper-bounded by 0. This corresponds to the fact that $\mathrm{cret}(e)$ forces to e with 0 cost. The one-sided rule R-RET-L relates $\mathrm{cret}(e_1)$ to an arbitrary m_2. The relative cost is $-k$, where k is a lower bound on the *unary* cost of forcing m. Rule R-STEP relates

two expressions $\mathrm{cstep}_{n_1}(m_1)$ and $\mathrm{cstep}_{n_2}(m_2)$ at ϕ if m_1 and m_2 are related at ϕ. The relative cost is $n + n_1 - n_2$, where n is the relative cost of m_1 and m_2. This corresponds to the fact that $\mathrm{cstep}_n(m)$ forces like m, but with additional cost n. The one-sided rule R-STEP-L analyzes only a left expression of the shape $\mathrm{cstep}_{n_1}(m_1)$; it increases the relative cost by n_1. Finally, the rule R-BIND can be used to relate $\mathrm{cbind}(e_1, \{x_1\}.m_1)$ and $\mathrm{cbind}(e_2, \{x_2\}.m_2)$. This requires that we relate the monadic expressions m_1 and m_2 and the pure expressions e_1 and e_2. The relative cost is bounded by the sum of the bounds on the relative cost of m_1 with respect to m_2 and the relative cost of e_1 with respect to e_2. The corresponding one-sided rule R-BIND-L performs a unary analysis of e_1.

Structural and admissible rules. Figure 6 shows structural and admissible rules. The rule R-SUB weakens the postcondition of two pure expressions. The rule R-SUBC weakens the relative cost of two monadic expressions. The *admissible* rules R-SUBM1 and R-SUBM2 allow weakening of relational postconditions. The admissible rule R-SPLIT allows falling back to unary reasoning at any point in a relational proof. It derives an upper-bound on the relative cost by taking a difference of the unary upper bound on the left and the unary lower bound on the right. (A similar, simpler rule for pure expressions is elided here.)

Metatheory. As for U^{C}, our main metatheoretic result about R^{C} is that it has a sound and complete interpretation in L^{C}.

Theorem 5.1 (Equivalence of R^{C} and L^{C}). *The following hold:*

(1) $\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi$ *if and only if* $\Gamma; \Psi \vdash_{L^C} \phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2]$ *(and* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau_2$*).*

(2) $\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$ *if and only if* $\Gamma; \Psi \vdash_{L^C} \exists e_1, e_2, n_1, n_2.\, m_1 \doteq \mathrm{cstep}_{n_1}(\mathrm{cret}(e_1)) \wedge m_2 \doteq \mathrm{cstep}_{n_2}(\mathrm{cret}(e_2)) \wedge \phi[e_1/\mathbf{r}_1][e_2/\mathbf{r}_2] \wedge n_1 - n_2 \leq n$ *(and* $\Gamma \vdash m_1 \div \tau_1$ *and* $\Gamma \vdash m_2 \div \tau_2$*).*

Proof. On the same lines as that for U^{C}. □

Again, this theorem has very useful consequences: it explains the meaning of R^{C}'s judgments, it gives R^{C} a set-theoretic semantics (via the semantics of L^{C}), it aids verification by allowing switching to L^{C} freely, and it shows that R^{C} is as expressive as the full equational theory of L^{C}. Finally, the theorem implies the following subject reduction property for forcing.

Theorem 5.2 (Forcing subject reduction). *If* $\vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 \mid n \mid \phi$ *and* $m_1 \Downarrow^{n_1} e_1$ *and* $m_2 \Downarrow^{n_2} e_2$*, then* $\vdash_{L^C} n_1 - n_2 \leq n$ *and* $\vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi$.

6 EXAMPLES

We present several examples of verification in U^{C} and R^{C} that highlight the importance of value-dependence, functional correctness and non-standard invariants for cost analysis and some non-standard features of R^{C} such as the one-sided rules. Our appendix contains several additional examples, including examples of unary and relational cost analysis on lazy data structures.

Notation. To aid readability, we use simplified notation in examples. We often elide cret, writing e in place of $\mathrm{cret}(e)$. We use $\uparrow^n m$ as alternate notation for $\mathrm{cstep}_n(m)$. The scope of $\uparrow^n$ extends to the end of the expression or the next closing bracket. We write $x \leftarrow e_1;\ m_2$ in place of $\mathrm{cbind}(e_1, \{x\}.m_2)$. In examples involving the defined data type list, we often use the infix form :: in place of the prefix form cons. Finally, we often omit contexts from judgments. The contexts can be reconstructed by following the sequence of applied rules. We also implicitly apply reasoning in the assertion logic L^{C}, and skip trivial applications of subsumption rules that weaken cost bounds or postconditions.

Insert into a sorted list. Our first example highlights the need to reason about values in a data structure to establish a precise cost (value-dependence). Consider the following function, a standard

component of insertion sort, that inserts an element x into a *sorted* list ℓ, where the list type is defined as usual with the equation $\text{list}_{\mathbb{N}} = \text{nil}() + \text{cons}(\mathbb{N} \times \text{list}_{\mathbb{N}})$.

$$
\begin{array}{l}
\mathit{insert} \triangleq \lambda x.\ \text{rec}\, f(\ell). \\
\quad \text{match } \ell \text{ with nil} \mapsto \{x :: \text{nil}\}; \\
\qquad\qquad \text{cons} \mapsto \lambda h, t.\ \text{match } x \leq h \text{ with tt} \mapsto \{x :: (h :: t)\}; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ff} \mapsto \{t' \leftarrow f\ t;\ \uparrow^1 h :: t'\}
\end{array}
$$

The cost of interest is the number of recursive calls of *insert*, modeled by the $\uparrow^1$ after the recursive call to f. It is straightforward to establish an upper bound of $|\ell|$ on the cost of *insert*. However, we want to establish a more precise bound—the number of elements in ℓ that x is larger than. A crucial precondition for this bound to hold is that ℓ be sorted ascending. Hence, the precise cost depends on the value of x, the values of the elements of ℓ, as well as a nontrivial refinement of ℓ (sortedness). We define three refinements (predicates) in L^{C} to capture these properties.

$$
\begin{array}{rl}
\forall x, \ell, n.\ \text{LargerThan}(x, \ell, n) \Leftrightarrow & (n \doteq 0 \wedge \ell \doteq \text{nil}) \\
& \vee\ (\exists h, t.\ \ell \doteq h :: t \wedge x \leq h \wedge \text{LargerThan}(x, t, n)) \\
& \vee\ (\exists h, t, n'.\ \ell \doteq h :: t \wedge x \not\leq h \wedge \text{LargerThan}(x, t, n') \wedge n \doteq n' + 1) \\
\forall \ell, n.\ \text{Unsorted}(\ell, n) \Leftrightarrow & (n \doteq 0 \wedge \ell \doteq \text{nil}) \\
& \vee\ (\exists h, t, n_1, n_2.\ \ell \doteq \text{cons}(h, t) \wedge \text{LargerThan}(h, t, n_1) \wedge \\
& \qquad \text{Unsorted}(t, n_2) \wedge n \doteq n_1 + n_2) \\
\forall l.\ \text{Sorted}(\ell) \Leftrightarrow & \text{Unsorted}(\ell, 0)
\end{array}
$$

$\text{LargerThan}(x, \ell, n)$ states that x is larger than n elements of ℓ; $\text{Unsorted}(\ell, n)$ states that the *unsortedness measure* of ℓ is n (the unsortedness measure is the sum of LargerThan predicates on all suffixes); and $\text{Sorted}(\ell)$ states that ℓ is sorted (its unsortedness measure is 0).

We show two additional properties of *insert*, for later use in our verification of insertion sort: (1) The list output by *insert* is also sorted, and (2) For any y, if y is larger than q elements of $x :: \ell$, then y is larger than q elements of the output list. Formally, we show in U^{C} that:

$$
\begin{array}{l}
\vdash \mathit{insert} : \mathbb{N} \to \text{list}_{\mathbb{N}} \to \mathbb{C}(\text{list}_{\mathbb{N}}) \mid \forall x, \ell.\ \text{Sorted}(\ell) \Rightarrow \forall n.\ \text{LargerThan}(x, \ell, n) \Rightarrow \mathbb{C}_{\mathbf{u}}(\mathbf{r}\ x\ \ell, n, n, \mathbf{r}.\phi) \\
\text{where } \phi \triangleq \text{Sorted}(\mathbf{r}) \wedge \forall y, q.\ \text{LargerThan}(y, \text{cons}(x, \ell), q) \Rightarrow \text{LargerThan}(y, \mathbf{r}, q)
\end{array}
$$

Following the syntax of *insert*, the proof first applies a U^{C} rule for λ-abstraction, which is similar to U-LETREC, but without the inductive hypothesis. Next we apply U-LETREC to get the inductive hypothesis (IH): $\forall m.\ |m| < |\ell| \Rightarrow\ \text{Sorted}(m) \Rightarrow \forall n.\ \text{LargerThan}(x, m, n) \Rightarrow \mathbb{C}_{\mathbf{u}}(f\ m, n, n, \mathbf{r}.\phi[m/\ell])$.

Next, by rule U-MATCH we need to consider two cases. For the case $\ell \doteq \text{nil}$ we need to show $\vdash \{x :: \text{nil}\} : \mathbb{C}(\text{list}_{\mathbb{N}}) \mid \mathbb{C}_{\mathbf{u}}(\mathbf{r}, n, n, \mathbf{r}.\phi)$. The assumptions $\ell \doteq \text{nil}$ and $\text{LargerThan}(\ell, x, n)$ force $n \doteq 0$. The cost part follows trivially by the rules U-MONAD and U-RET. The proof of ϕ is also easy.

For the case when $\ell \doteq \text{cons}(h, t)$ we further apply the rule for λ twice and the rule U-MATCH. This yields two sub-cases: $x \leq h$ and $x \not\leq h$ (note that we overload $\leq$: It is a predicate in L^{C} and an operator in the language). For the sub-case $x \leq h$, it remains to show $\vdash \{x :: (h :: t)\} : \mathbb{C}(\text{list}_{\mathbb{N}}) \mid \mathbb{C}_{\mathbf{u}}(\mathbf{r}, n, n, \mathbf{r}.\phi)$. Next, $\ell \doteq \text{cons}(h, t)$, $\text{Sorted}(\ell)$, $x \leq h$ and $\text{LargerThan}(x, \ell, n)$ force $n \doteq 0$. It is crucial that ℓ is sorted, otherwise we would not be able to conclude this here. The rest of this sub-case is then very similar to the $\ell \doteq \text{nil}$ case above.

For the sub-case $x \not\leq h$, it remains to show $\vdash \{t' \leftarrow f\ t;\ \uparrow^1 h :: t'\} : \mathbb{C}(\text{list}_{\mathbb{N}}) \mid \mathbb{C}_{\mathbf{u}}(\mathbf{r}, n, n, \mathbf{r}.\phi)$. From $\ell \doteq \text{cons}(h, t)$, $\text{LargerThan}(x, \ell, n)$ and $x \not\leq h$, it follows $\text{LargerThan}(x, t, n')$ for some $n' \in \mathbb{N}$ s.t. $n \doteq n' + 1$. From $\text{Sorted}(\ell)$, it follows that $\text{Sorted}(t)$. Then, by applying the IH and using Theorem 4.1, we get $\vdash f\ t : \mathbb{C}(\text{list}) \mid \mathbb{C}_{\mathbf{u}}(\mathbf{r}, n', n', t'.\phi[t/\ell][t'/\mathbf{r}])$. Using the rules U-BIND and SUBC (with $n' + 1 \doteq n$) it suffices to show $\vdash \uparrow^1 h :: t' \div \text{list}_{\mathbb{N}} \mid 1 \mid 1 \mid \phi$. The cost part follows from the rules U-STEP and U-RET. Proving ϕ needs additional reasoning in L^{C}, which we defer to the appendix.

Count-up/count-down binary counters. Next, we consider an example of *relational* cost analysis. The example presents a situation where the relative cost of two programs is very easy to establish, but a unary analysis of either program is not. The example also highlights the use of a non-standard relational invariant. Consider two binary counters of the same width, each represented as a list of bits coded as booleans ($0 \triangleq \mathsf{ff}$, $1 \triangleq \mathsf{tt}$), with the least significant bit at the head. One counter starts at 0 (all bits 0) and counts up—it can be incremented through the function *inc* defined below. The other counter starts at the maximum possible value (all bits 1) and counts down—it can be decremented through the *dec* function defined below.

$$\begin{array}{l}
\mathsf{bool} = \mathsf{tt}() + \mathsf{ff}() \qquad \mathsf{list} = \mathsf{nil}() + \mathsf{cons}(\mathsf{bool} \times \mathsf{list}) \\
\mathit{inc} \triangleq \mathsf{rec}\, f_1(\ell_1).\ \mathsf{match}\ \ell_1\ \mathsf{with} \\
\qquad\qquad \mathsf{nil} \mapsto \{\mathsf{nil}\}; \\
\qquad\qquad \mathsf{cons} \mapsto \lambda x_1, t_1.\ \mathsf{match}\ x_1\ \mathsf{with}\ \ \mathsf{ff} \mapsto \{\uparrow^1 \mathsf{tt} :: t_1\};\ \mathsf{tt} \mapsto \{t_1' \leftarrow f_1\ t_1;\ \uparrow^1 \mathsf{ff} :: t_1'\} \\
\mathit{dec} \triangleq \mathsf{rec}\, f_2(\ell_2).\ \mathsf{match}\ \ell_2\ \mathsf{with} \\
\qquad\qquad \mathsf{nil} \mapsto \{\mathsf{nil}\}; \\
\qquad\qquad \mathsf{cons} \mapsto \lambda x_2, t_2.\ \mathsf{match}\ x_2\ \mathsf{with}\ \ \mathsf{ff} \mapsto \{t_2' \leftarrow f_2\ t_2;\ \uparrow^1 \mathsf{tt} :: t_2'\};\ \mathsf{tt} \mapsto \{\uparrow^1 \mathsf{ff} :: t_2\}
\end{array}$$

The cost of interest is the number of *bit flips* during the operations, as represented by the $\uparrow^1$ annotations in the code. We want to show that for any natural number k, the *relative* cost of incrementing the first counter k times and decrementing the second counter k times is 0, i.e., the number of bit flips in these sequences of operations is the same. Informally, this property follows from the fact that the counters always remain bitwise duals of each other. Formally, it follows from a strong invariant: If two counters are bitwise duals of each other, then incrementing one counter once and decrementing the other once incurs 0 relative cost, and the resulting counters are still bitwise duals. The property we want to prove follows trivially from this invariant by induction on k, since the counters start out as bitwise duals (one is all 0s, the other is all 1s).

We show here how to prove the invariant. We start by defining an assertion $\mathrm{dual}(\ell_1, \ell_2)$ on lists of booleans, which says that ℓ_1 and ℓ_2 have the same length and are pointwise dual.

$$\begin{array}{l}
\forall \ell_1, \ell_2.\ \mathrm{dual}(\ell_1, \ell_2) \Leftrightarrow (\ell_1 \doteq \ell_2 \doteq \mathsf{nil}) \vee \\
\qquad\qquad (\exists x_1, t_1, x_2, t_2.\ \ell_1 \doteq \mathsf{cons}(x_1, t_1) \wedge \ell_2 \doteq \mathsf{cons}(x_2, t_2) \wedge x_1 \neq x_2 \wedge \mathrm{dual}(t_1, t_2))
\end{array}$$

We can then state our invariant as an R^{C} judgment:

$$\begin{array}{l}
\vdash \mathit{inc} : \mathsf{list} \to \mathbb{C}(\mathsf{list}) \sim \mathit{dec} : \mathsf{list} \to \mathbb{C}(\mathsf{list}) \mid \forall \ell_1, \ell_2.\ \mathrm{dual}(\ell_1, \ell_2) \\
\qquad\qquad \Rightarrow \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1\ \ell_1, \mathbf{r}_2\ \ell_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2))
\end{array}$$

The proof of this judgment is straightforward since *inc* and *dec* have very similar structure. We first apply the rule R-LETREC (which introduces the IH into the context), then apply R-MATCH. This yields only two cases since the assumption $\mathrm{dual}(\ell_1, \ell_2)$ forces ℓ_1, ℓ_2 to either both be empty or both be nonempty. The case $\ell_1 \doteq \ell_2 \doteq \mathsf{nil}$ is straightforward. In the case $\ell_1 \doteq x_1 :: t_1$ and $\ell_2 \doteq x_2 :: t_2$, we continue into the structure of the functions and apply R-MATCH again (for the case analysis of the booleans x_1 and x_2). At this point, we get four cases, but from $\mathrm{dual}(\ell_1, \ell_2)$, we also know that $x_1 \neq x_2$, so we have only the cases $x_1 \doteq \mathsf{ff}, x_2 \doteq \mathsf{tt}$ and $x_1 \doteq \mathsf{tt}, x_2 \doteq \mathsf{ff}$.

When $x_1 \doteq \mathsf{ff}, x_2 \doteq \mathsf{tt}$, our goal reduces to proving $\vdash \{\uparrow^1 \mathsf{tt} :: t_1\} : \mathbb{C}(\mathsf{list}) \sim \{\uparrow^1 \mathsf{ff} :: t_2\} : \mathbb{C}(\mathsf{list}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2))$. Applying the rules R-MONAD, R-STEP and R-RET, this reduces to $\vdash \mathsf{ff} :: t_1 : \mathsf{list} \sim \mathsf{tt} :: t_2 : \mathsf{list} \mid \mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2)$, which follows immediately by switching to L^{C} since $\mathrm{dual}(t_1, t_2)$.

When $x_1 \doteq \mathsf{tt}, x_2 \doteq \mathsf{ff}$, our goal is $\vdash \{t_1' \leftarrow f_1\ t_1;\ \uparrow^1 \mathsf{ff} :: t_1'\} : \mathbb{C}(\mathsf{list}) \sim \{t_2' \leftarrow f_2\ t_2;\ \uparrow^1 \mathsf{tt} :: t_2\} : \mathbb{C}(\mathsf{list}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2))$. Using R-MONAD, this reduces to $\vdash\ t_1' \leftarrow f_1\ t_1;\ \uparrow^1 \mathsf{ff} :: t_1' \div \mathsf{list} \sim t_2' \leftarrow f_2\ t_2;\ \uparrow^1 \mathsf{tt} :: t_2 \div \mathsf{list} \mid 0 \mid \mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2)$. From the IH, we derive (via L^{C}) that $\vdash f_1\ t_1 : \mathbb{C}(\mathsf{list}) \sim f_2\ t_2 : \mathbb{C}(\mathsf{list}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2))$. Hence, by rule R-BIND, it suffices to prove that

$\vdash \uparrow^1 \mathsf{ff} :: t_1' \div \mathsf{list} \sim \uparrow^1 \mathsf{tt} :: t_2' \div \mathsf{list} \mid 0 \mid \mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2)$, under the assumption $\mathrm{dual}(t_1', t_2')$. Applying rules R-STEP and R-RET, we further reduce to $\vdash \mathsf{ff} :: t_1' : \mathsf{list} \sim \mathsf{tt} :: t_2' : \mathsf{list} \mid \mathrm{dual}(\mathbf{r}_1, \mathbf{r}_2)$, which follows immediately (in L^{C}) from the assumption $\mathrm{dual}(t_1', t_2')$. This completes the proof.

Even though this relational proof is very straightforward, a unary cost analysis of a binary counter is not—it requires amortized counting. We show this unary analysis for a slightly more general counter in Section 8.

Boolean expression evaluation. Next, we show an example of relational cost analysis where the relative cost depends *not* on a measure of the size of a data structure, but on another nontrivial property of it. Consider the following type bexpr of boolean expressions (bool constants, and the connectives "and" and "or"):

$$\mathsf{bexpr} = \mathsf{const}(\mathsf{bool}) + \mathsf{and}(\mathsf{bexpr} \times \mathsf{bexpr}) + \mathsf{or}(\mathsf{bexpr} \times \mathsf{bexpr})$$

We write two functions to evaluate a bexpr to a bool. The first function is a naive implementation that recurses on the whole bexpr, while the second one more intelligently short cuts (skips) the evaluation of e_2 in $\mathsf{and}(e_1, e_2)$ when e_1 evaluates to ff (and analogously for or).

$$\begin{aligned}
\mathit{eval}_1 \triangleq\ & \mathsf{rec}\, f_1(e_1).\mathsf{match}\ e_1\ \mathsf{with} \\
& \mathsf{const} \mapsto \lambda b'.\{b'\} \\
& \quad \mathsf{and} \mapsto \lambda e', e''.\{x_1 \leftarrow f_1\ e';\ y_1 \leftarrow f_1\ e'';\ \uparrow^1 \mathsf{match}\ x_1\ \mathsf{with}\ \mathsf{tt} \mapsto y_1; \mathsf{ff} \mapsto \mathsf{ff}\} \\
& \quad\ \mathsf{or} \mapsto \lambda e', e''.\{x_1 \leftarrow f_1\ e';\ y_1 \leftarrow f_1\ e'';\ \uparrow^1 \mathsf{match}\ x_1\ \mathsf{with}\ \mathsf{tt} \mapsto \mathsf{tt}; \mathsf{ff} \mapsto y_1\} \\
\mathit{eval}_2 \triangleq\ & \mathsf{rec}\, f_2(e_2).\mathsf{match}\ e_2\ \mathsf{with} \\
& \mathsf{const} \mapsto \lambda b'.\ \{b'\} \\
& \quad \mathsf{and} \mapsto \lambda e', e''.\ \{x_2 \leftarrow f_2\ e';\ y_2 \leftarrow (\mathsf{match}\ x_2\ \mathsf{with}\ \mathsf{tt} \mapsto f_2\ e''; \mathsf{ff} \mapsto \{\mathsf{ff}\});\ \uparrow^1 y_2\} \\
& \quad\ \mathsf{or} \mapsto \lambda e', e''.\ \{x_2 \leftarrow f_2\ e';\ y_2 \leftarrow (\mathsf{match}\ x_2\ \mathsf{with}\ \mathsf{tt} \mapsto \{\mathsf{tt}\}; \mathsf{ff} \mapsto f_2\ e'');\ \uparrow^1 y_2\}
\end{aligned}$$

The cost of interest here is the number of matches performed on bools, which is represented by a $\uparrow^1$ after every match. One obvious relational property is that on the same bexpr, the cost of eval_1 is no less than the cost of eval_2. This property can be established trivially in R^{C}. Here, we define a refinement noshort on bexprs, which ensures that the costs of eval_1 and eval_2 are *equal* and show that this is actually the case.

$$\begin{aligned}
\forall e, b.\ \mathrm{noshort}(e, b) \Leftrightarrow\ & (\exists b'.\ e \doteq \mathsf{const}(b') \wedge b \doteq b') \vee \\
& (\exists e_1, e_2, b.\ e \doteq \mathsf{and}(e_1, e_2) \wedge \mathrm{noshort}(e_1, \mathsf{tt}) \wedge \mathrm{noshort}(e_2, b)) \vee \\
& (\exists e_1, e_2, b.\ e \doteq \mathsf{or}(e_1, e_2) \wedge \mathrm{noshort}(e_1, \mathsf{ff}) \wedge \mathrm{noshort}(e_2, b))
\end{aligned}$$

In words, $\mathrm{noshort}(e, b)$ states that the bexpr e evaluates to the bool b, and short-cutting is inapplicable to the evaluation of e, since the left sub-expression of every nested "and" evaluates to tt, and the left sub-expression of every nested "or" evaluates to ff.

We want to show that eval_1 and eval_2, when applied to the same bexpr e satisfying $\mathrm{noshort}(e, b)$, have relative cost 0, and the result in each case is b. Formally:

$$\begin{aligned}
\vdash \mathit{eval}_1 : \mathsf{bexpr} \to \mathbb{C}(\mathsf{bool}) \sim \mathit{eval}_2 : \mathsf{bexpr} \to \mathbb{C}(\mathsf{bool}) \mid\ & \forall e_1, e_2.\ e_1 \doteq e_2 \Rightarrow \forall b.\ \mathrm{noshort}(e_1, b) \\
& \Rightarrow \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1\ e_1, \mathbf{r}_2\ e_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathbf{r}_1 \doteq \mathbf{r}_2 \doteq b)
\end{aligned}$$

The proof is mostly synchronous and follows the structure of eval_1 and eval_2. It starts by applying the R^{C} rules R-LETREC and R-MATCH. The latter requires considering 9 cases (all possible pairs of bexpr constructors), but because $e_1 \doteq e_2$ is assumed, we immediately reduce to only 3 cases, where the constructors on the two sides are the same. In the case $e_1 \doteq e_2 \doteq \mathsf{const}(b')$, we get from the assumption $\mathrm{noshort}(e_1, b)$ that $b \doteq b'$. We need to show $\vdash \{b'\} : \mathbb{C}(\mathsf{bool}) \sim \{b'\} : \mathbb{C}(\mathsf{bool}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, 0, \mathbf{r}_1.\mathbf{r}_2.\mathbf{r}_1 \doteq \mathbf{r}_2 \doteq b)$, which follows immediately by applying the rules R-MONAD and R-RET and then switching to L^{C} to show $b \doteq b \doteq b'$.

In the case $e_1 \doteq e_2 \doteq \mathsf{and}(e', e'')$, we have $\mathsf{noshort}(e', \mathsf{tt})$ and $\mathsf{noshort}(e'', b)$. By the IH on e' followed by Theorem 5.1, we get $\vdash f_1\ e' : \mathbb{C}(\mathsf{bool}) \sim f_2\ e' : \mathbb{C}(\mathsf{bool}) \mid \mathbb{C}_\mathbf{r}(\mathbf{r}_1, \mathbf{r}_2, 0, x_1.x_2.x_1 \doteq x_2 \doteq \mathsf{tt})$. By applying this result to the original goal, and reducing the inner match constructs (on x_1 and x_2) in both functions, it remains to show $\vdash (y_1 \leftarrow f_1\ e''; \uparrow^1 y_1) \div \mathsf{bool} \sim (y_2 \leftarrow f_2\ e''; \uparrow^1 y_2) \div \mathsf{bool} \mid 0 \mid \mathbf{r}_1 \doteq \mathbf{r}_2 \doteq b$. This follows by the IH on e''. The case $e_1 \doteq e_2 \doteq \mathsf{or}(e', e'')$ is similar.

List length. Our next example is very simple. Its purpose is to demonstrate the use of asynchronous (one-sided) rules in the monadic part of R^C (the use of asynchronous rules in functional verification is well-understood in prior work), and a situation where two expressions of different *types* need to be related. Consider two implementations, length_1 and length_2, of the list length function. length_1 is tail-recursive, and uses a helper function length_h, while length_2 is the standard recursive implementation.

$$\begin{array}{lcl} \mathit{length}_h & \triangleq & \mathsf{rec}\, f_1(\ell_1).\lambda n.\ \mathsf{match}\ \ell_1\ \mathsf{with}\ \mathsf{nil} \mapsto \{n\}; \mathsf{cons} \mapsto \lambda\_, t_1.\ \{x_1 \leftarrow f_1\ t_1\ (n+1);\ x_1\} \\ \mathit{length}_1 & \triangleq & \lambda\ell.\ \mathit{length}_h\ \ell\ 0 \\ \mathit{length}_2 & \triangleq & \mathsf{rec}\, f_2(\ell_2).\mathsf{match}\ \ell_2\ \mathsf{with}\ \mathsf{nil} \mapsto \{0\}; \mathsf{cons} \mapsto \lambda\_, t_2.\ \{x_2 \leftarrow f\ t_2;\ \uparrow^1 x_2 + 1\} \end{array}$$

length_2 incurs a unit cost on every recursive call, while there is no such cost in length_h; the intent is to model the number of allocated stack-frames. We want to show that the relative cost of length_1 and length_2 is determined by the length of the input list and, additionally, that both functions implement the length function. To state our goal, we first define a list length predicate:

$$\forall\ell.\ \mathrm{Len}(\ell, 0) \Leftrightarrow \ell \doteq \mathsf{nil} \qquad \forall\ell, n.\ \mathrm{Len}(\ell, n+1) \Leftrightarrow \exists h, t.\ \ell \doteq \mathsf{cons}(h, t) \wedge \mathrm{Len}(t, n)$$

Then, formally, we want to show:

$$\vdash \mathit{length}_1 : \mathsf{list} \to \mathbb{C}(\mathbb{N}) \sim \mathit{length}_2 : \mathsf{list} \to \mathbb{C}(\mathbb{N}) \mid \forall\ell_1, \ell_2.\ \ell_1 \doteq \ell_2 \Rightarrow \forall m.\ \mathrm{Len}(\ell_1, m) \Rightarrow \mathbb{C}_\mathbf{r}(\mathbf{r}_1\ \ell_1, \mathbf{r}_2\ \ell_2, -m, \mathbf{r}_1.\mathbf{r}_2.\mathbf{r}_1 \doteq \mathbf{r}_2 \doteq m)$$

The cost part of this property means that the cost of length_1 minus the cost of length_2 is upper-bounded by $-m$ or, equivalently, the cost of length_2 is lower-bounded by the cost of length_1 plus m (where m is the length of the input list). Since length_1 merely calls length_h with second argument 0, this can be easily reduced to showing:

$$\vdash \mathit{length}_h : \mathsf{list} \to \mathbb{N} \to \mathbb{C}(\mathbb{N}) \sim \mathit{length}_2 : \mathsf{list} \to \mathbb{C}(\mathbb{N}) \mid \forall\ell_1, \ell_2, n.\ \ell_1 \doteq \ell_2 \Rightarrow \forall m.\ \mathrm{Len}(\ell_1, m) \Rightarrow \mathbb{C}_\mathbf{r}(\mathbf{r}_1\ \ell_1\ n, \mathbf{r}_2\ \ell_2, -m, \mathbf{r}_1.\mathbf{r}_2.\mathbf{r}_1 \doteq \mathbf{r}_2 + n \doteq m + n)$$

Note that length_h and length_2 have different types. We first apply the R^C rule R-LETREC, and then the one-sided rule for λ (since length_h has an extra λ). The latter rule is similar to R-LETREC-L, but without the IH. Next, we apply the two-sided rule R-MATCH. Since $\ell_1 \doteq \ell_2$ is assumed, we get two cases in the proof. In the case $\ell_1 \doteq \ell_2 \doteq \mathsf{nil}$, $\mathrm{Len}(\ell_1, m)$ forces $m \doteq 0$. We need to show $\vdash \{n\} : \mathbb{C}(\mathbb{N}) \sim \{0\} : \mathbb{C}(\mathbb{N}) \mid \mathbb{C}_\mathbf{r}(\mathbf{r}_1, \mathbf{r}_1, 0, \mathbf{r}_1.\mathbf{r}_2.\mathbf{r}_1 \doteq \mathbf{r}_2 + n \doteq 0 + n)$, which follows from the rules R-MONAD, R-RET and reasoning in L^C.

In the case $\ell_1 \doteq \mathsf{cons}(h_1, t_1)$ and $\ell_2 \doteq \mathsf{cons}(h_2, t_2)$, we have $h_1 \doteq h_2$ and $t_1 \doteq t_2$ (from $\ell_1 \doteq \ell_2$), and $\mathrm{Len}(t_1, m')$ and $\mathrm{Len}(t_2, m')$, for some $m' \in \mathbb{N}$ s.t. $m \doteq m' + 1$. Then by IH we have:

$$\vdash f_1\ t_1\ (n+1) : \mathbb{C}(\mathbb{N}) \sim f_2\ t_2 : \mathbb{C}(\mathbb{N}) \mid \mathbb{C}_\mathbf{r}(\mathbf{r}_1, \mathbf{r}_2, -m', x_1.x_2.x_1 \doteq x_2 + (n+1) \doteq m' + (n+1))$$

Hence, by the rule R-BIND, it remains to show (under the assumption $x_1 \doteq x_2 + (n+1) \doteq m' + (n+1)$):

$$\vdash x_1 \div \mathbb{N} \sim \uparrow^1 x_2 + 1 \div \mathbb{N} \mid -1 \mid \mathbf{r}_1 \doteq \mathbf{r}_2 + n \doteq m + n$$

At this point, we apply the one-sided rule R-STEP-R (which is completely analogous to R-STEP-L) to reduce the goal to $\vdash x_1 \div \mathbb{N} \sim x_2 + 1 \div \mathbb{N} \mid 0 \mid \mathbf{r}_1 \doteq \mathbf{r}_2 + n \doteq m + n$. This follows immediately by the rule R-RET and reasoning in L^C.

Insertion sort. We perform a precise relational analysis of insertion sort. Insertion sort calls the *insert* function for which we proved a *unary* property earlier. We use that property now. Hence, this example demonstrates an interaction between relational and unary analysis in R^{C}. The analysis also relies on nontrivial functional properties of insertion sort itself.

The insertion sort function, *isort*, is defined below:

$$\begin{aligned} \mathit{isort} \triangleq \mathsf{rec}\, f(\ell).\mathsf{match}\ \ell\ \mathsf{with}\ \mathsf{nil} \mapsto \{\mathsf{nil}\};\\ \mathsf{cons} \mapsto \lambda h, t.\ \{t' \leftarrow f\ t;\ z \leftarrow \mathit{insert}\ h\ t';\ \uparrow^1 z\} \end{aligned}$$

As for *insert*, the cost here is the number of recursive calls. Next, we define $\mathrm{UnsortedDiff}(\ell_1, \ell_2, n)$, which means that the unsortedness of lists ℓ_1 and ℓ_2 (of the same length) *differs* by n. Note that unsortedness of a single list was defined by the predicate $\mathrm{Unsorted}(\ell, n)$ in the analysis of *insert*.

$$\begin{aligned} \forall \ell_1, \ell_2, n.\ \mathrm{UnsortedDiff}(\ell_1, \ell_2, n) \Leftrightarrow \exists u_1, u_2, m.\ \mathrm{Unsorted}(\ell_1, u_1) \wedge \mathrm{Unsorted}(\ell_2, u_2) \wedge\\ n \doteq u_1 - u_2 \wedge \mathrm{Len}(\ell_1, m) \wedge \mathrm{Len}(\ell_2, m) \end{aligned}$$

Our goal is to show that if $\mathrm{UnsortedDiff}(\ell_1, \ell_2, n)$, then the *relative cost* of running *isort* on ℓ_1 and ℓ_2 is upper-bounded by n. To prove this, we need to show two additional functional properties: (a) That *isort* produces a sorted list, and (b) For any y, the number of elements larger than y in the input and output lists of *isort* is the same. Formally, we show that:

$$\begin{aligned} &\vdash \mathit{isort} : \mathrm{list} \to \mathbb{C}(\mathrm{list}) \sim \mathit{isort} : \mathrm{list} \to \mathbb{C}(\mathrm{list}) \mid \forall \ell_1, \ell_2, n.\ \mathrm{UnsortedDiff}(\ell_1, \ell_2, n)\\ &\qquad\qquad \Rightarrow \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1\ \ell_1, \mathbf{r}_2\ \ell_2, n, \mathbf{r}_1.\mathbf{r}_2.\phi)\\ &\text{where } \phi \triangleq \mathrm{Sorted}(\mathbf{r}_1) \wedge \mathrm{Sorted}(\mathbf{r}_2) \wedge (\forall y, q.\ \mathrm{LargerThan}(y, \ell_1, q) \Rightarrow \mathrm{LargerThan}(y, \mathbf{r}_1, q)) \wedge\\ &\qquad (\forall y, q.\ \mathrm{LargerThan}(y, \ell_2, q) \Rightarrow \mathrm{LargerThan}(y, \mathbf{r}_2, q)) \end{aligned}$$

The proof starts by applying the R^{C} rule R-LETREC, which introduces the induction hypothesis, and then R-MATCH, which causes a case analysis on the input lists. Since the lists have the same length (by assumption $\mathrm{UnsortedDiff}(\ell_1, \ell_2, n)$), we need to consider only the cases when either both lists are nil or both have at least one element. The first case is straightforward. In the second case, $\ell_1 \doteq \mathrm{cons}(h_1, t_1)$ and $\ell_2 \doteq \mathrm{cons}(h_2, t_2)$. The definition of UnsortedDiff yields u_1, u_2 such that $n \doteq u_1 - u_2$, $\mathrm{Unsorted}(\ell_1, u_1)$ and $\mathrm{Unsorted}(\ell_2, u_2)$. The definition of Unsorted now yields $u_1 \doteq n_1' + u_1'$, $u_2 \doteq n_2' + u_2'$, $\mathrm{LargerThan}(h_1, t_1, n_1')$, $\mathrm{Unsorted}(t_1, u_1')$, $\mathrm{LargerThan}(h_2, t_2, n_2')$, and $\mathrm{Unsorted}(t_2, u_2')$ for some u_1', n_1', u_2', n_2'. Further, we also have $\mathrm{UnsortedDiff}(t_1, t_2, n')$, s.t. $n' \doteq u_1' - u_2'$, and $\mathrm{Len}(t_1, m')$ and $\mathrm{Len}(t_2, m')$, where $m \doteq m' + 1$. Hence, from the IH we get:

$$\vdash f_1\ t_1 : \mathbb{C}(\mathrm{list}) \sim f_2\ t_2 : \mathbb{C}(\mathrm{list}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, n', t_1'.t_2'.\phi')$$

for $\phi' \triangleq \phi[t_1/\ell_1][t_2/\ell_2][t_1'/\mathbf{r}_1][t_2'/\mathbf{r}_2]$. From this ϕ' we have $\mathrm{Sorted}(t_1')$ and $\mathrm{LargerThan}(h_1, t_1', n_1')$. Applying these to the unary property we proved for *insert* earlier, we get:

$$\begin{aligned} \vdash \mathit{insert}\ h_1\ t_1' : \mathbb{C}(\mathrm{list}) \mid \mathbb{C}_{\mathbf{u}}(\mathbf{r}, n_1', n_1', z.\mathrm{Sorted}(z) \wedge \forall y, q.\ \mathrm{LargerThan}(y, \mathrm{cons}(h_1, t_1'), q)\\ \Rightarrow \mathrm{LargerThan}(y, z, q)) \end{aligned}$$

and a similar property for *insert* $h_2\ t_2'$. Next, we combine these two unary properties of *insert* into a relational property using the admissible R^{C} rule R-SPLIT:

$$\vdash \mathit{insert}\ h_1\ t_1' : \mathbb{C}(\mathrm{list}) \sim \mathit{insert}\ h_2\ t_2' : \mathbb{C}(\mathrm{list}) \mid \mathbb{C}_{\mathbf{r}}(\mathbf{r}_1, \mathbf{r}_2, n_1' - n_2', z_1.z_2.\phi'')$$

where $\phi'' \triangleq \mathrm{Sorted}(z_1) \wedge \mathrm{Sorted}(z_2) \wedge \forall y, q.\ \mathrm{LargerThan}(y, \mathrm{cons}(h_1, t_1'), q) \Rightarrow \mathrm{LargerThan}(y, z_1, q)$ $\wedge\ \forall y, q.\ \mathrm{LargerThan}(y, \mathrm{cons}(h_1, t_2'), q) \Rightarrow \mathrm{LargerThan}(y, z_2, q)$.

Note that the additional property ϕ' obtained for (a recursive call to) *isort* enabled us to derive the results for *insert*. Without this, we would not have been able to derive $\mathrm{Sorted}(t_i')$ and $\mathrm{LargerThan}(h_i, t_i', n_i')$ from $\mathrm{Sorted}(t_i)$ and $\mathrm{LargerThan}(h_i, t_i, n_i)$ (for $i \in \{1, 2\}$). In turn, the additional property ϕ'' of *insert* obtained above is required to show ϕ.

To close the proof, we apply R-BIND twice and use subsumption with $n' + n'_1 + n'_2 + 0 \leq n$, reducing the goal to $\vdash \uparrow^1 z_1 \div \text{list} \sim \uparrow^1 z_2 \div \text{list} \mid 0 \mid \phi$. The cost part follows trivially from the rules R-STEP and R-RET. Showing ϕ needs additional reasoning in L^{C}; we defer the details to the appendix.

7 EMBEDDING OF RELCOST

Next, we show that $\mathrm{R}^{\mathrm{C}}/\mathrm{U}^{\mathrm{C}}$ can be used as meta frameworks to embed other cost analyses. This section shows an embedding of RelCost, while Section 8 shows an embedding of RAML. RelCost [Çiçek et al. 2017] is a type-and-effect system for unary and relational cost analysis. It includes lightweight refinements, also known as index refinements, in the style of DML [Xi and Pfenning 1999]. Many examples, including all examples presented in the paper so far, cannot be verified in RelCost since its index refinements are not expressive enough. We now present an embedding of RelCost in $\mathrm{U}^{\mathrm{C}}/\mathrm{R}^{\mathrm{C}}$, thus establishing that RelCost's approach is strictly less expressive than ours. We restrict our attention to a core of RelCost with non-recursive functions and lists at base types; this suffices to explain all the key ideas.

RelCost's types and selected typing rules are shown in Figure 7. RelCost has unary types A (for unary expressions and their costs), and relational types τ (for pairs of expressions and their relative cost). The only data type supported by RelCost is primitive lists. Unary types are mostly standard, except for the annotation $\text{exec}(k, \ell)$ on the arrow- and universal-types that represents lower and upper bounds, k and ℓ respectively, on the cost of the body of the closure. The index n on the list type is the length of the list. The relational type int_r is the diagonal relation on integers. The n in the annotation $\text{diff}(n)$ on the arrow- and universal-types in the relational types is an upper bound on the relative cost of the two closures. The annotation α on list types is an upper bound on the Hamming distance of the two lists. The relational type UA contains pairs of arbitrary elements of unary type A, while $\Box\tau$ is the diagonal subrelation of τ. Quantifiers range over index variables i, which are distinct from (program) expressions e. Expressions are standard; they have a *call-by-value* semantics. Cost is incurred only at elimination constructs like function application.

There are two typing judgments in RelCost—one unary ($\Delta; \Phi; \Omega \vdash^{\ell}_{k} e : A$) and one relational ($\Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim n : \tau$). In both judgments, Δ is an environment for index variables, and Φ are assumed constraints over the index variables. The unary judgment states that, under a unary typing environment Ω, e has type A, and its execution cost is lower- and upper-bounded by k and ℓ, respectively. The relational judgment states that, under a relational typing environment Γ, expressions e_1 and e_2 have the relational type τ, and their relative cost is at most n. The function $\overline{\tau}$ embeds relational types into unary types. (RelCost uses the notation $|\cdot|$ for this function; we use a different notation to avoid confusion with the erasure function below.) Constants like c_{app} in the rules represent the costs of individual reductions, e.g., function application.

Our embedding of RelCost consists of several translations, shown in Figure 8. We describe these translations below.

Type translation. We first define an erasing type translation that removes all refinement and effect annotations from RelCost's types, yielding simple types. This translation, written $|\cdot|$ for the unary types and $\|\cdot\|$ for the relational types, follows the standard embedding of call-by-value in a monadic type system [Moggi 1991], e.g., $A_1 \xrightarrow{\text{exec}(k,\ell)} A_2$ translates to $|A_1| \to \mathbb{C}(|A_2|)$. We assume that our language has a type unit and a family of list types, list_σ, both of which can be defined as inductive data types.

Expression translation. Following the standard embedding of call-by-value in a monadic type system, we translate a RelCost expression of type A to a pure expression of type $\mathbb{C}(|A|)$. To capture

$$A ::= \text{int} \mid \text{list}[n]\ A \mid A_1 \xrightarrow{\text{exec}(k,\ell)} A_2 \mid \forall i \overset{\text{exec}(k,\ell)}{::} S.\ A \qquad \text{Unary types}$$

$$\tau ::= \text{int}_r \mid \text{list}[n]^{\alpha}\ \tau \mid \tau_1 \xrightarrow{\text{diff}(n)} \tau_2 \mid \forall i \overset{\text{diff}(n)}{::} S.\ \tau \mid UA \mid \Box\tau \qquad \text{Relational types}$$

$$\boxed{\Delta; \Phi; \Omega \vdash_k^{\ell} e : A} \qquad \text{Unary typing judgment}$$

$$\frac{\Omega(x) = A}{\Delta; \Phi; \Omega \vdash_0^0 x : A} \qquad \frac{\Delta; \Phi; \Omega, x : A_1 \vdash_k^{\ell} e : A_2}{\Delta; \Phi; \Omega \vdash_0^0 \lambda x.\ e : A_1 \xrightarrow{\text{exec}(k,\ell)} A_2}$$

$$\frac{\Delta; \Phi; \Omega \vdash_{k_1}^{\ell_1} e_1 : A_1 \xrightarrow{\text{exec}(k,\ell)} A_2 \qquad \Delta; \Phi; \Omega \vdash_{k_2}^{\ell_2} e_2 : A_1}{\Delta; \Phi; \Omega \vdash_{k_1+k_2+k+c_{app}}^{\ell_1+\ell_2+\ell+c_{app}} e_1\ e_2 : A_2}$$

$$\boxed{\Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim n : \tau} \qquad \text{Relational typing judgment}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Phi; \Gamma \vdash x \ominus x \lesssim 0 : \tau} \qquad \frac{\Delta; \Phi; x : \tau_1, \Gamma \vdash e_1 \ominus e_2 \lesssim n : \tau_2}{\Delta; \Phi; \Gamma \vdash \lambda x.\ e_1 \ominus \lambda x.\ e_2 \lesssim 0 : \tau_1 \xrightarrow{\text{diff}(n)} \tau_2}$$

$$\frac{\Delta; \Phi; \overline{\Gamma} \vdash_{\ell_1}^{k_1} e_1 : A \qquad \Delta; \Phi; \overline{\Gamma} \vdash_{\ell_2}^{k_2} e_2 : A}{\Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim \ell_1 - k_2 : UA} \qquad \frac{i, \Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim n : \tau}{\Delta; \Phi; \Gamma \vdash \Lambda e_1 \ominus \Lambda e_2 \lesssim 0 : \forall i \overset{\text{diff}(n)}{::} S.\ \tau}$$

$$\frac{\Delta; \Phi; \Gamma \vdash e \ominus e \lesssim n : \tau \qquad \forall x \in dom(\Gamma).\ \Delta; \Phi \vDash \Gamma(x) \sqsubseteq \Box\Gamma(x)}{\Delta; \Phi; \Gamma, \Gamma' \vdash e \ominus e \lesssim 0 : \Box\tau}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash e \ominus e' \lesssim m : \text{list}_{\tau'}[n]^{\alpha} \qquad \Delta; \Phi \wedge n = 0; \Gamma \vdash e_1 \ominus e_1' \lesssim m' : \tau \\ i, \Delta; \Phi \wedge n = i+1; \Gamma, h : \Box\tau', t : \text{list}_{\tau'}[i]^{\alpha} \vdash e_2 \ominus e_2' \lesssim m' : \tau \\ i, \beta, \Delta; \Phi \wedge n = i+1 \wedge \alpha = \beta+1; \Gamma, h : \Box\tau', t : \text{list}_{\tau'}[i]^{\beta} \vdash e_2 \ominus e_2' \lesssim m' : \tau\end{array}}{\Delta; \Phi; \Gamma \vdash \text{case } e \text{ of nil} \to e_1 \mid h :: t \to e_2 \ominus \text{case } e' \text{ of nil} \to e_1 \mid h :: t \to e_2 \lesssim m + m' : \tau}$$

Fig. 7. RelCost: Types and selected typing rules

costs from RelCost's semantics, cstep annotations are added to elimination constructs. We denote this translation $(\!|\cdot|\!)$. Representative clauses are shown in Figure 8.

Unary refinements. To capture RelCost's unary refinements, we define a translation $\lfloor\cdot\rfloor$ from unary types into L^{C} assertions. $\lfloor A \rfloor$ is a predicate on expressions, which holds at e when e satisfies the refinement inherent in A. To handle list lengths, we axiomatically define the predicate $\text{listU}_A(e, n)$ which means that list e has length n. The interesting clause is that for function types $A_1 \xrightarrow{\text{exec}(k,\ell)} A_2$, where the cost bounds k, ℓ are represented through a monadic refinement $\mathbb{C}_{\mathbf{u}}(x\ y, k, \ell, \mathbf{r}.\_)$.

Relational refinements. Like unary refinements, we capture relational refinements by a translation $\|\cdot\|$ from relational types to L^{C} assertions. $\|\tau\|$ is a binary relation on expressions that holds at (e_1, e_2) when this pair of expressions satisfies the relational refinement inherent in τ. To capture the list refinements for length and Hamming distance, we define a predicate $\text{listR}_{\tau}(e_1, e_2, n, a)$

Erasing translation from RelCost types to simple types

$$|\text{int}| \triangleq \|\text{int}_r\| \triangleq \mathbb{Z} \qquad |\text{list}[n]\ A| \triangleq \text{list}_{|A|} \qquad \|\text{list}[n]^{\alpha}\ \tau\| \triangleq \text{list}_{\|\tau\|}$$

$$|A_1 \xrightarrow{\text{exec}(k,\ell)} A_2| \triangleq |A_1| \to \mathbb{C}(|A_2|) \qquad \|\tau_1 \xrightarrow{\text{diff}(n)} \tau_2\| \triangleq \|\tau_1\| \to \mathbb{C}(\|\tau_2\|) \qquad \|UA\| \triangleq |A|$$

$$|\forall i \overset{\text{exec}(k,\ell)}{::} S.\ A| \triangleq \text{unit} \to \mathbb{C}(|A|) \qquad \|\forall i \overset{\text{diff}(n)}{::} S.\ \tau\| \triangleq \text{unit} \to \mathbb{C}(\|\tau\|) \qquad \|\Box\tau\| \triangleq \|\tau\|$$

$$|x_1 : A_1, \ldots, x_n : A_n| \triangleq x_1 : |A_1|, \ldots, x_n : |A_n|$$

$$\|x_1 : \tau_1, \ldots, x_n : \tau_n\| \triangleq x_1^1 : \|\tau_1\|, x_1^2 : \|\tau_1\| \ldots, x_n^2 : \|\tau_n\|, x_n^2 : \|\tau_n\|$$

Expression translation (selected clauses)

$$(\!|x|\!) \triangleq \{\text{cret}(x)\} \qquad (\!|\lambda x.\ e|\!) \triangleq \{\text{cret}(\lambda x.\ (\!|e|\!))\}$$

$$(\!|e_1\ e_2|\!) \triangleq \{\text{cbind}((\!|e_1|\!), \{x\}.\ \text{cbind}((\!|e_2|\!), \{y\}.\ \text{cbind}(x\ y, \{z\}.\ \text{cstep}_{c_{app}}(\text{cret}(z)))))\}$$

Translation of unary refinements (selected clauses)

$$\forall \ell.\ \text{listU}_A(\ell, 0) \Leftrightarrow \ell \doteq \text{nil}$$

$$\forall \ell, n.\ \text{listU}_A(\ell, n+1) \Leftrightarrow \exists h, t.\ \ell \doteq \text{cons}(h, t) \wedge \lfloor A \rfloor(h) \wedge \text{listU}_A(t, n)$$

$$\lfloor \text{int} \rfloor(x) \triangleq \top \qquad \lfloor \text{list}[n]\ A \rfloor(x) \triangleq \text{listU}_A(x, n)$$

$$\lfloor A_1 \xrightarrow{\text{exec}(k,\ell)} A_2 \rfloor(x) \triangleq \forall y.\ \lfloor A_1 \rfloor(y) \Rightarrow \mathbb{C}_{\text{U}}(x\ y, k, \ell, \mathbf{r}.\lfloor A_2 \rfloor(\mathbf{r}))$$

$$\lfloor x_1 : A_1, \ldots, x_n : A_n \rfloor \triangleq \lfloor A_1 \rfloor(x_1), \ldots, \lfloor A_n \rfloor(x_n)$$

Translation of relational refinements (selected clauses)

$$\forall \ell_1, \ell_2, a.\ \text{listR}_\tau(\ell_1, \ell_2, 0, a) \Leftrightarrow \ell_1 \doteq \ell_2 \doteq \text{nil}$$

$$\begin{aligned}\forall \ell_1, \ell_2, n, a.\ \text{listR}_\tau(\ell_1, \ell_2, n+1, a) \Leftrightarrow \exists h_1, h_2, t_1, t_2.\ & \ell_1 \doteq \text{cons}(h_1, t_1) \wedge \ell_2 \doteq \text{cons}(h_2, t_2) \\ & \wedge \llbracket \tau \rrbracket(h_1, h_2) \wedge ((h_1 \doteq h_2 \wedge \text{listR}_\tau(t_1, t_2, n, a)) \\ & \vee (a > 0 \wedge \exists b.\ a = b + 1 \wedge \text{listR}_\tau(t_1, t_2, n, b)))\end{aligned}$$

$$\llbracket \text{int}_r \rrbracket(x, y) \triangleq x \doteq y \qquad \llbracket \text{list}[n]^{\alpha}\ \tau \rrbracket(x, y) \triangleq \text{listR}_\tau(x, y, n, \alpha)$$

$$\begin{aligned}\llbracket \tau_1 \xrightarrow{\text{diff}(n)} \tau_2 \rrbracket(x, y) \triangleq\ & \lfloor \overline{\tau_1} \xrightarrow{\text{exec}(0,\infty)} \overline{\tau_2} \rfloor(x) \wedge \lfloor \overline{\tau_1} \xrightarrow{\text{exec}(0,\infty)} \overline{\tau_2} \rfloor(y) \wedge \\ & (\forall x_1, x_2.\ \llbracket \tau_1 \rrbracket(x_1, x_2) \Rightarrow \mathbb{C}_{\text{R}}(x\ x_1, y\ x_2, n, \mathbf{r}_1.\mathbf{r}_2.\llbracket \tau_2 \rrbracket(\mathbf{r}_1, \mathbf{r}_2)))\end{aligned}$$

$$\llbracket UA \rrbracket(x, y) \triangleq \lfloor A \rfloor(x) \wedge \lfloor A \rfloor(y) \qquad \llbracket \Box\tau \rrbracket(x, y) \triangleq x \doteq y \wedge \llbracket \tau \rrbracket(x, y)$$

$$\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n \rrbracket \triangleq \llbracket \tau_1 \rrbracket(x_1^1, x_1^2), \ldots, \llbracket \tau_n \rrbracket(x_n^1, x_n^2)$$

Fig. 8. Embedding of RelCost in $\text{U}^{\text{C}}/\text{R}^{\text{C}}$

axiomatically. The predicate means that the lists e_1, e_2 each have length n and their Hamming distance is at most a.

The translations $|\cdot|$, $\|\cdot\|$, $\lfloor\cdot\rfloor$ and $\|\!|\cdot|\!\|$ lift to contexts straightforwardly (see Figure 8). The relational translations $\|\cdot\|$ and $\|\!|\cdot|\!\|$ actually duplicate the variables in the context by systematic renaming—each variable x is replaced by x^1 and x^2. This cosmetic change is necessary because in RelCost's relational judgment, the related expressions share free variables, while this is not the case in R^{C}.

Our main theorem is that this translation is sound. The translation also captures the *intent* of RelCost's type system—RelCost's soundness theorems can be derived as corollaries to this theorem by first showing that the costs of evaluating an expression e in RelCost and forcing $(\!|e|\!)$ in our language are equal.

Theorem 7.1 (Soundness). *The following hold.*

(1) If $\Delta; \Phi; \Omega \vdash^{\ell}_{k} e : A$ *in RelCost, then* $|\Omega|, \Delta; \Phi, \lfloor\Omega\rfloor \vdash (\!|e|\!) : \mathbb{C}(|A|) \mid \mathbb{C}_{\boldsymbol{u}}(\mathbf{r}, k, \ell, \mathbf{r}.\lfloor A\rfloor(\mathbf{r}))$ *in* U^{C}.

(2) If $\Delta; \Phi; \Gamma \vdash e_1 \ominus e_2 \lesssim n : \tau$ *in RelCost, then* $\|\Gamma\|, \Delta; \Phi, \|\!|\Gamma|\!\| \vdash (\!|e_1|\!)_1 : \|\tau\| \sim (\!|e_2|\!)_2 : \|\tau\| \mid \mathbb{C}_{r}(\mathbf{r}_1, \mathbf{r}_2, n, \mathbf{r}_1.\mathbf{r}_2.\|\!|\tau|\!\|(\mathbf{r}_1, \mathbf{r}_2))$ *in* R^{C}, *where* $(\!|e_i|\!)_i$ *is a copy of* $(\!|e_i|\!)$ *where each variable* x *is replaced by a variable* x^i, *for* $i \in \{1, 2\}$.

Remark. Aguirre et al. [2017b] present a translation of RelCost directly into RHOL. However, since RHOL lacks a specific treatment of cost, that translation encodes an expression's cost as a second output of the expression, resulting in a substantially more complex encoding and one in which the cost is an ordinary value, not an effect. Our translation keeps costs and program values separate and it is much simpler. Nonetheless, our method of translating simple types and refinements separately from each other owes lineage to this work.

8 EMBEDDING OF AMORTIZED COST ANALYSIS

Amortized cost analysis is a specific style of cost verification that proceeds by associating *potentials* with a program's inputs and paying for the program's costs from these potentials. The cost of a program is upper-bounded by the difference between the potentials associated with its inputs and the (remaining) potential associated with its outputs. Static, *unary* amortized analysis has been implemented in Resource-aware ML (RAML) [Hoffmann 2011; Hoffmann et al. 2012]. Here, we describe an embedding of a core calculus behind RAML (a fragment of the calculus of Hoffmann [2011]) into U^{C}. This translation is particularly interesting because RAML is an *affine* type system, while U^{C} has no support for counting variable use. Consequently, our embedding enforces affineness through refinements. We limit ourselves to the analysis of additive costs (since U^{C} supports only those[6]) and to structurally recursive functions. To simplify the presentation, we consider only one data structure—lists, and we consider only so-called linear potentials, where the potential associated with every element of a list is a constant.[7] This fragment suffices to explain the key ideas.

Figure 9 shows the syntax and typing rules of the fragment of RAML we consider. A program P is a list of (first-order) functions $f_1, \ldots, f_n$, each with a body e_{f_i} and a formal parameter y_{f_i}. We allow e_{f_i} to apply f_i (recursion) but on arguments strictly smaller than y_{f_i}. Expressions e are constants, function applications, and list operations (nil, cons, match ... with ...). Types A are either int or $L^q(A)$, which ascribes lists over A, with potential q associated to every element. In function types $F ::= A_1 \xrightarrow{q/q'} A_2$, the annotation q is a constant potential before a call to the function and q' is a constant potential after the function ends. The language has standard call-by-value semantics.

[6] RAML supports the analysis of non-additive costs, e.g., the space needed to run a program. Section 9 explains how U^{C} and the embedding described here can be extended to such costs.

[7] This restriction is not fundamental. Our translation can be extended to RAML's tree types and its polynomial and multi-variate potentials.

Expressions and types

Expressions	e	::=	$x \mid n \mid f(x) \mid \text{let } x \leftarrow e_1 \text{ in } e_2 \mid \text{nil} \mid$ $\text{cons}(x_h, x_t) \mid \text{match } x \text{ with nil} \mapsto e_1 \mid \text{cons}(x_h, x_t) \mapsto e_2$
Programs	P	::=	$(e_{f_1}, y_{f_1}), \ldots, (e_{f_n}, y_{f_n})$
Types	A	::=	$\text{int} \mid L^q(A)$
Function types	F	::=	$A_1 \xrightarrow{q/q'} A_2$

Typing rules (selected)

$$\frac{}{\Sigma; x : B \vdash^{q+K^{var}}_{q} x : B}\ \text{L:Var} \qquad \frac{\Sigma; \Gamma_1 \vdash^{q-K_1^{let}}_{p} e_1 : A \qquad \Sigma; \Gamma_2 \vdash^{p-K_2^{let}}_{q'+K_3^{let}} e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \vdash^{q}_{q'} \text{let } x \leftarrow e_1 \text{ in } e_2 : B}\ \text{L:Let}$$

$$\frac{A \xrightarrow{q/q'} B \in \Sigma(f)}{\Sigma; x : A \vdash^{q+K_1^{app}}_{q'-K_2^{app}} f(x) : B}\ \text{L:App} \qquad \frac{}{\Sigma; x_h : A, h_t : L^p(A) \vdash^{q+p+K^{cons}}_{q} \text{cons}(x_h, x_t) : L^p(A)}\ \text{L:Cons}$$

$$\frac{\Sigma; \Gamma \vdash^{q-K_1^{matN}}_{q'+K_2^{matN}} e_1 : B \qquad \Sigma; \Gamma, x_h : A, x_t : L^p(A) \vdash^{q+p-K_1^{matC}}_{q'+K_2^{matC}} e_2 : B}{\Sigma; \Gamma, x : L^p(A) \vdash^{q}_{q'} \text{match } x \text{ with nil} \mapsto e_1 \mid \text{cons}(x_h, x_t) \mapsto e_2 : B}\ \text{L:MatL}$$

$$\frac{\Sigma; \Gamma, x : A_1, y : A_2 \vdash^{q}_{q'} e : B \qquad A \not\curlyvee (A_1, A_2)}{\Sigma; \Gamma, z : A \vdash^{q}_{q'} e[z/x][z/y] : B}\ \text{L:Share} \qquad \frac{\Sigma; \Gamma \vdash^{p}_{p'} e : B \qquad q \geq p \qquad q - p \geq q' - p'}{\Sigma; \Gamma \vdash^{q}_{q'} e : B}\ \text{L:Relax}$$

Fig. 9. RAML syntax and typing rules

The language also has a cost semantics where every operation is assumed to incur some cost. For example, evaluation of a variable (substituted by a value) incurs cost K^{var}, while evaluation of let $x \leftarrow e_1$ in e_2 incurs cost K_1^{let} before starting e_1, cost K_2^{let} between e_1 and e_2 and cost K_3^{let} after e_2, for a total cost of $K_1^{let} + K_2^{let} + K_3^{let}$.

The potential of a value a of type A, denoted $\Phi(a : A)$ is defined as follows:

$$\Phi(n : \text{int}) \triangleq 0 \qquad \Phi([a_1; \cdots; a_n] : L^q(A)) \triangleq q \cdot n + \sum_{1 \leq i \leq n} \Phi(a_i : A)$$

RAML's main typing judgment is $\Sigma; \Gamma \vdash^{q}_{q'} e : A$. Here, Σ assigns nonempty sets of function types F to functions identifiers f (each f can have many function types, but they can differ only in the potential annotations q/q') and Γ assigns types A to variables x. The judgment informally says that for any closing substitution γ for Γ, $\Phi(\gamma : \Gamma) + q$ units of potential are enough to evaluate $e\gamma$ and if $e\gamma$ evaluates to a value a, then $q' + \Phi(a : A)$ potential will be left. Here, $\Phi(\gamma : \Gamma)$ is defined as the sum of the potentials of values in the range of γ.

Some interesting typing rules are shown in Figure 9. For space reasons, we do not explain the rules in detail here, but we note that the type system is *affine*—variables must be used at most once, but they can be duplicated explicitly using the rule L:Share. Such duplication *splits* the potential of

Erasing translation from RAML types to simple types

$$|\text{int}| \triangleq \text{int} \qquad |L^q(A)| \triangleq \text{list}_{|A|} \qquad |A_1 \xrightarrow{q/q'} A_2| \triangleq |A_1| \to \mathbb{C}(|A_2|)$$

Expression translation (selected)

$$(\!|x|\!) \triangleq \{\text{cstep}_{K^{var}}(\text{cret}(x))\} \qquad (\!|f(x)|\!) \triangleq \{\text{cbind}(f\ x, \{r\}.\ \text{cstep}_{K_1^{app}+K_2^{app}}(\text{cret}(r)))\}$$

$$(\!|\text{let } x \leftarrow e_1 \text{ in } e_2|\!) \triangleq \{\text{cbind}((\!|e_1|\!), \{x\}.\ \text{cbind}((\!|e_2|\!), \{r\}.\ \text{cstep}_{K_1^{let}+K_2^{let}+K^{let_3}}(\text{cret}(r))))\}$$

Predicate $\tilde{\Phi}_A(e,p)$ that encodes potentials

$$\forall x, p.\ \tilde{\Phi}_{\text{int}}(x,p) \Leftrightarrow p \doteq 0$$

$$\forall x, p.\ \tilde{\Phi}_{L^q(A)}(x,p) \Leftrightarrow \begin{array}{l}(p \doteq 0 \wedge x \doteq \text{nil}) \vee \\ (\exists h, t, p_h, p_t.\ x \doteq \text{cons}(h,t) \wedge \tilde{\Phi}_A(h,p_h) \\ \quad \wedge \tilde{\Phi}_{L^q(A)}(t,p_t) \wedge p \doteq q + p_h + p_t)\end{array}$$

Translations for contexts

$$|x_1 : A_1, \ldots, x_n : A_n| \triangleq x_1 : |A_1|, x_1^p : \mathbb{R}^\infty, \ldots, x_n : |A_n|, x_n^p : \mathbb{R}^\infty$$

$$\lfloor x_1 : A_1, \ldots, x_n : A_n \rfloor \triangleq \tilde{\Phi}_{A_1}(x_1, x_1^p), \ldots, \tilde{\Phi}_{A_n}(x_n, x_n^p)$$

$$|\Sigma| \triangleq \{f : |F| \mid \text{for all } f \in dom(\Sigma) \text{ and some } F \in \Sigma(f)\}$$

$$\lfloor \Sigma \rfloor \triangleq \{\lfloor F \rfloor(f) \mid \text{for all } f \in dom(\Sigma) \text{ and all } F \in \Sigma(f)\}$$

Predicate over function types, that relates potentials to costs

$$\lfloor A_1 \xrightarrow{q/q'} A_2 \rfloor(f) \triangleq \forall y : |A_1|.\ \top \Rightarrow \forall y^p : \mathbb{R}^+.\ \tilde{\Phi}_{A_1}(y, y^p) \Rightarrow \exists p_r.\ \mathbb{C}\mathrm{u}(f\ y, 0, y^p + q - q' - p_r, \mathbf{r}.\tilde{\Phi}_{A_2}(\mathbf{r}, p_r))$$

Fig. 10. Embedding of RAML in U^{C}

the duplicated variable among the duplicates (using the relation $\curlyvee$). This affineness is essential, since re-use of variables would increase input potential, which would make the analysis unsound.

Translation. Figure 10 summarizes our translation of RAML. The translation is quite similar to RelCost's translation in that it also follows the standard idea of embedding call-by-value in a monadic type system. We first define an erasing translation $|\cdot|$ from RAML types to simple types. This translation maps $A_1 \xrightarrow{q/q'} A_2$ to $|A_1| \to \mathbb{C}(|A_2|)$. Next, we define a translation $(\!|e|\!)$ of expressions that maps an expression of type A to a pure expression of type $\mathbb{C}(|A|)$. This translation adds appropriate cstep annotations to induce costs according to RAML's cost semantics. RAML programs, which are lists of function definitions, compile to lists of functions in the obvious way.

The key novelty lies in how we encode potentials and relate them to costs. To encode potentials, we axiomatically define a predicate $\tilde{\Phi}_A(e,p)$ in L^{C}. This predicate means that e (of type $|A|$) has potential p. The definition is straightforward and follows the definition of RAML's potential function Φ shown earlier. Using this predicate, we can define translations of contexts. The key idea is that for every variable $x : A$ in the RAML context, we introduce a new variable x^p of type $\mathbb{R}^\infty$ that represents x's potential, and assume that the two are related by $\tilde{\Phi}_A(x, x^p)$. Finally, we define a

predicate $\lfloor A_1 \xrightarrow{q/q'} A_2 \rfloor(f)$ on *functions* that captures the relation between potentials and the cost of f's body. This predicate can be best understood as an internalization of the soundness property of our translation, which we show next.

Theorem 8.1 (Soundness). *If* $\Sigma; \Gamma \vdash^{q}_{q'} e : A$ *in RAML with additive costs only, then* $|\Sigma|, |\Gamma|; \lfloor\Sigma\rfloor, \lfloor\Gamma\rfloor \vdash (\!|e|\!) : \mathbb{C}(|A|) \mid \exists p_r.\ \mathbb{C}_{\mathbf{U}}(\mathbf{r}, 0, \tilde{\Phi}(\Gamma) + q - q' - p_r, \mathbf{r}.\tilde{\Phi}_A(\mathbf{r}, p_r))$ *in* U^C, *where* $\tilde{\Phi}(\Gamma) \triangleq \sum_{x \in dom(\Gamma)} x^p$ *is the sum of all potential variables in the context.*

The theorem states that the cost of the monadic expression in $(\!|e|\!)$ is upper-bounded by the difference in the input potential ($\tilde{\Phi}(\Gamma) + q$) and the output potential ($p_r + q'$). This is exactly the intuition behind RAML's typing judgment. In fact, the soundness of RAML's typing judgment can be derived as a corollary to this theorem and the set-theoretic soundness of U^{C}.

Value-dependent potentials. Many examples of amortized analysis require the potential associated with an element to depend on its value. RAML cannot handle many such examples since it does not have refinements. However, such examples can be analyzed in U^{C} using a coding of potentials similar to that in the embedding of RAML. We show here one such example, which generalizes the binary counter of Section 6. Consider a fixed-width counter that counts in an arbitrary base $D \geq 2$ (D is a variable parameter, not a fixed constant). The counter is represented as a list of primitive integers (type $\mathbb{Z}$), representing individual digits of the counter with the least significant digit at the head. It is an invariant that the integers range from 0 to $D-1$ only. We define a function *incg* that increments the counter once and a function *rinc* that increments the counter k times, where k is an input of type nat (which is defined as $\text{nat} = 0() + S(\text{nat})$) by iterating *incg* k times.

$$\text{list} = \text{nil}() + \text{cons}(\mathbb{Z} \times \text{list})$$

$$\begin{aligned}
&\mathit{incg} : \text{list} \to \mathbb{C}(\text{list})\\
&\mathit{incg} \triangleq \text{rec}\, f(\ell).\ \text{match}\ \ell\ \text{with}\\
&\qquad\qquad \text{nil} \mapsto \{\text{nil}\};\\
&\qquad\qquad \text{cons} \mapsto \lambda x, t.\ \text{if}\ x < D-1\ \text{then}\ \{\uparrow^1(x+1) :: t\}\ \text{else}\ \{t' \leftarrow f\ t;\ \uparrow^1 0 :: t'\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{rinc} : \text{nat} \to \text{list} \to \mathbb{C}(\text{list})\\
&\mathit{rinc} \triangleq \text{rec}\, f(k).\lambda\ell.\ \text{match}\ k\ \text{with}\ 0 \mapsto \{\ell\};\ S \mapsto \lambda k'.\ \{\ell' \leftarrow \mathit{incg}\ \ell;\ \ell'' \leftarrow f\ k'\ \ell';\ \ell''\}
\end{aligned}$$

The cost of interest is the number of changes to digits, indicated by a unit cost $\uparrow^1$ whenever we change a digit in *incg*. Our goal is to show that, assuming the counter starts with all digits 0, the cost of (*rinc* k) is no more than $\frac{D}{D-1}k$. Informally, the result follows from the observation that the least significant digit changes at every increment, the second digit changes every D increments and so on. So, the total cost is no more than $k + \frac{k}{D} + \frac{k}{D^2} + \ldots = \frac{D}{D-1}k$. Formally, this can be established by associating a potential of $\frac{i}{D-1}$ to a digit if its current value is i. Note that this potential is value-sensitive—it depends on i. If we were unable to capture this value-sensitivity in the potential (as would be the case in RAML), then the best bound we would obtain is $O(l \cdot k)$, where l is the width of the counter.[8]

We define the predicate $\Phi(\ell, n)$ to mean that the counter ℓ has total potential n:

$$\Phi(\ell, n) \Leftrightarrow (\ell \doteq \text{nil} \wedge n \doteq 0) \vee (\exists t, i, n'.\ \ell \doteq \text{cons}(i, t) \wedge \Phi(t, n') \wedge n \doteq n' + \tfrac{i}{D-1} \wedge i \leq D-1)$$

[8] If D is not a parameter but a hard-coded number, then this example can be coded in RAML by defining a datatype that explicitly enumerates all D values of the digit. For example, for $D = 3$, one defines a datatype Three = zero | one | two and then uses a list over the type Three to represent the counter. RAML is then able to perform a precise analysis.

Then, we show the following two claims (we assume an implicit coercion from nat to $\mathbb{R}^\infty$).

$\vdash \mathit{inc} : \text{list} \to \mathbb{C}(\text{list}) \mid \forall \ell, n.\ \Phi(\ell, n) \Rightarrow \exists n'.\ \mathbb{C}_{\mathbf{u}}(\mathbf{r}\ \ell, 0, n + \frac{D}{D-1} - n', \mathbf{r}.\Phi(\mathbf{r}, n'))$
$\vdash \mathit{rinc} : \text{nat} \to \text{list} \to \mathbb{C}(\text{list}) \mid \forall k, n, \ell.\ \Phi(\ell, n) \Rightarrow \exists n'.\ \mathbb{C}_{\mathbf{u}}(\mathbf{r}\ k\ \ell, 0, \frac{D}{D-1}k + n - n', \mathbf{r}.\Phi(\mathbf{r}, n'))$

In words, the claims state that a single increment has cost at most $n + \frac{D}{D-1} - n'$, and k increments have cost at most $\frac{D}{D-1}k + n - n'$, where n is the potential of the input counter, and n' is the potential of the output counter. If the counter starts from 0, then $n = 0$, so the latter also implies that the cost of k increments is at most $\frac{D}{D-1}k - n' \leq \frac{D}{D-1}k$, as required. (The lower bound is 0 here only for simplicity; we can also show a tighter bound.)

While we defer a full proof to the appendix, we sketch here an informal proof of *incg* when $\ell \doteq x :: t$ (the case $\ell \doteq \mathsf{nil}$ is trivial). When $x < D - 1$, the potential of ℓ is n (by assumption), that of t is $n - \frac{x}{D-1}$, and that of the resulting list $(x + 1) :: t$ is $n - \frac{x}{D-1} + \frac{x+1}{D-1} = n + \frac{1}{D-1}$. Hence, the established cost is $n + \frac{D}{D-1} - (n + \frac{1}{D-1}) = 1$, which is exactly the number of digit changes. When $x \doteq D - 1$, the potential of ℓ is n (by assumption), and that of t is $n - \frac{D-1}{D-1} = n - 1$. Then, by the inductive hypothesis, incrementing t has cost $n - 1 + \frac{D}{D-1} - n' = n + \frac{1}{D-1} - n'$, where n' is the recursive call's result potential, which is also the potential of the overall result (since the result's leading digit is 0). The total cost is $(n + \frac{1}{D-1} - n') + 1 = n + \frac{D}{D-1} - n'$, as required.

9 POSSIBLE EXTENSIONS

We discuss preliminary ideas for extensions to our framework that we plan to work on in the future. We expect these extensions to be significant and technically nontrivial in the details, and all claims in this section are currently conjectural.

Non-additive costs. Currently, our development draws costs from $\mathbb{R}^\infty$ and our forcing semantics *add* costs along sequential execution. Some resources like memory can be re-used and do not fit this additive model. For example, if an expression frees some of the memory that it uses, then the next expression can re-use that freed memory. So the memory needed for the first expression should not simply be added to the memory needed for the second expression, as this would result in a very pessimistic upper-bound on the total amount of needed memory. Our current development cannot represent such *non-additive* costs precisely.

RAML [Hoffmann 2011; Hoffmann et al. 2012] includes an elegant, compositional way of handling non-additive costs via the method of potentials. This generalizes the additive fragment of RAML we covered in Section 8. Briefly, the cost of a program expression is represented by a pair of non-negative numbers (q, q'), where q is the incoming potential (e.g., the amount of available memory) and q' is the outgoing potential. $q - q'$ represents the net consumption of resources by the expression. This number may be negative when the expression frees more resources than it consumes. q itself is an upper bound on the resources needed to run the expression, akin to the upper bound on the cost in our current model. The remarkable fact about such costs (q, q') is that they form a monoid where the monoid operation $\cdot$ is defined as:

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q', p') & \text{if } q' \leq p \\ (q, q' - p + p') & \text{if } q' > p \end{cases}$$

RAML's cost semantics counts costs by applying $\cdot$ along the program's sequential execution (much as our forcing semantics counts costs by applying + along monadic binds).

We expect that U^{C} can be generalized to reason about unary non-additive costs by noting that its rules work for *any monoid*, not just $(\mathbb{R}^\infty, 0, +)$. In particular, it should be feasible to replace $(\mathbb{R}^\infty, 0, +)$ with the RAML monoid $M = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, (0, 0), \cdot)$ defined above. ($\mathbb{Q}_0^+$ is the set of non-negative rationals, which RAML uses to represent potentials.) We would also need an ordering

relation $(q, q') \leq (p, p')$ on the monoid M for use in weakening/subsumption of costs (e.g., the rule U-SUBM1), and in the definition of $\mathbb{C}_{\mathrm{u}}$. This relation can be defined as $(q, q') \leq (p, p') \triangleq (q \leq p) \wedge (q - q') \leq (p - p')$. With this change of monoid, we should be able to reason about non-additive costs in U^{C}. Further, we conjecture that our embedding of RAML with only additive costs from Section 8 will extend to RAML with non-additive costs using this monoid.

In the context of *relational* cost analysis (i.e., R^{C}), good reasoning principles for non-additive costs are unclear. The key difficulty is that a precise relational cost analysis—one that exploits program/input similarity—seems to require a notion of difference of costs (not just accumulation of costs via + or ·), but it is unclear how difference can be defined for the monoid M above. We believe that fundamentally new ideas will be needed in this space.

Mutable state. Our current framework, like the framework RHOL/UHOL [Aguirre et al. 2017b] it builds on, does not support mutable state. However, we believe that it is possible to extend both the underlying framework (without costs) and our development (with costs) to stateful programs.

For instance, the underlying unary framework UHOL (without costs) could be extended to state by defining a state-passing monad $\mathbb{S}(\tau)$ within the framework as $\mathbb{S}(\tau) \triangleq S \to (\tau \times S)$, where S is the type of the mutable state. The standard Hoare triple $\{\Theta\}e\{x.\Theta'(x)\}$ could be represented as the logical refinement assertion $\forall s : S.\ \Theta(s) \Rightarrow \Theta'(\pi_1(e\ s))(\pi_2(e\ s))$. Here, x (of type τ) represents the expression returned by the monadic computation, and Θ and $\Theta'(x)$ are assertions on the state s, representing the pre-condition and the post-condition of e, respectively. We conjecture that the standard rules of Hoare logic should then be derivable within UHOL and that this idea can be further generalized to the relational framework RHOL by replacing Θ and Θ' with relational assertions on pairs of states as in Relational Hoare Type Theory (RHTT) [Nanevski et al. 2013].

The same state-passing monad, when defined in U^{C} and R^{C}, should allow reasoning about costs of stateful programs, as long as the reasoning does not require predicates that range over both cost and state (since they would be in separate monads). However, Carbonneaux et al. [2015] show that predicates that range over both cost and state are, in fact, quite useful. They present a logic (for a C-like language) where potentials can be associated with the current state. Their triples have the form $\{\Theta; P\}e\{\Theta'; Q\}$, where Θ and Θ' are the standard pre- and post-conditions of e, and P and Q are functions from the initial and final states to the initial and final potentials, respectively. The unary cost of e is, as usual, upper-bounded by $P(s) - Q(s')$, where s and s' are the initial and final states, respectively. Note that, here, P and Q relate state and cost (potentials) to each other. We believe that the fragment of this logic with only additive costs and only structured control flow can be embedded in U^{C} by defining a different monad $\mathbb{SC}(\tau)$ that includes both state and cost. Briefly, we could define $\mathbb{SC}(\tau) \triangleq S \to \mathbb{C}(\tau \times S)$—computations that take a state, and return a result and a state, and a cost on the side. We conjecture that the triple $\{\Theta; P\}e\{\Theta'; Q\}$ should then be representable as the logical assertion $\forall s\, p.\ (\Theta(s) \wedge P(s) \doteq p) \Rightarrow \exists q.\ \mathbb{C}_{\mathrm{u}}(e\ s, 0, p - q, x.(\Theta'(\pi_2(x)) \wedge Q(\pi_2(x)) \doteq q))$ and that the rules of Carbonneaux et al. [2015]'s logic pertaining to state should be derivable in U^{C}. (Carbonneaux et al. [2015] also consider loop breaks and return-in-the-middle of functions. Encoding these constructs in a functional language would need additional ideas.) Further, it might be feasible to generalize the development to non-additive costs using the RAML monoid described earlier, and to extend the development to the relational cost analysis of stateful programs in R^{C}.

Nonterminating programs. Another limitation we inherit from RHOL/UHOL is that all expressions must be terminating. This is needed to give RHOL/UHOL (and our cost monad) a simple semantics in set theory. However, this limitation does not seem to be fundamental. In recent work [Aguirre et al. 2017a], a subset of the authors of this paper (and others) have re-worked a version of RHOL/UHOL based on the guarded λ-calculus [Clouston et al. 2016] and a model in the topos of trees, which

generalizes sets. This version supports infinite computations, including computations over infinite streams, and allows proving properties of all finite prefixes of the computations. Even though the syntax and the model are different, the proof rules are very similar to those of RHOL/UHOL over set theory. We believe that adding a cost monad to this modified framework should be feasible and should allow proving upper bounds on all finite approximations of an infinite computation.

10 RELATED WORK

Static cost analysis, using type systems or other methods, is a very widely studied topic. Danielsson [2008] performs unary cost analysis by embedding a cost monad in Agda. In principle, this approach can exploit Agda's rich dependent types for cost analysis, much as we exploit logical assertions. However, Danielsson's focus is exclusively on lazy data structures in a sharing semantics and the design is limited to unary cost analysis. U^{C} supports similar unary analysis over lazy data structures, and R^{C} additionally supports analysis of relational lazy costs—our appendix has examples of both.

Grobauer [2001] interprets a cost monad in the refinement type system DML [Xi and Pfenning 1999] by cost passing, much as we interpret our language in set-theory, and shows how to extract recurrence relations for unary costs of recursive functions. TiML [Wang et al. 2017] is a DML-based type-and-effect system for unary cost analysis. TiML has been used to verify examples where cost depends on data structure-size invariants. However, neither Grobauer [2001] nor TiML consider relational cost analysis. Moreover, it is unclear to us how some predicates like $\mathrm{LargerThan}(x, \ell, n)$ (that is central to the analysis of *insert*'s precise cost) can be defined using DML-style refinements.

Amortized analysis [Hofmann and Jost 2003; Jost et al. 2010] establishes cost using the method of potentials. The technique can be fully automated for polynomial bounds, as in Resource Aware ML (RAML) [Hoffmann et al. 2012, 2017]. Our embedding of RAML in R^{C} shows that R^{C} is more expressive, but R^{C} is a proof framework, not an automated system. Jost et al. [2017] extend amortized analysis to lazy semantics (Haskell). They specifically focus on co-inductive definitions. Despite the extensive development, work on amortized analysis has, so far, not been combined with functional properties or value-dependence. Nonetheless, RAML can implicitly handle value-dependence when a type's constructors are singletons. This happens, for instance, with enumerated types like $\mathrm{bool} = \mathrm{ff} + \mathrm{tt}$. Our example from Section 8 shows that a more general combination of value-dependence and amortized analysis is interesting. In recent work, Ngo et al. [2017] extend amortized analysis to the verification of constant-resource usage behavior in the context of preventing side-channel leaks of information. Although this has a flavor of relational analysis, it is technically based on unary analysis (it works by showing that the lower and upper cost bounds coincide). Çiçek et al. [2017] argue that this approach is insufficient for relational cost analysis in general.

Carbonneaux et al. [2015] present a quantitative program logic that can perform amortized cost analysis of programs in Clight, the first intermediate language of the CompCert C compiler. The settings of that work and ours are quite different: They consider automatic analysis for first-order imperative programs with mutable state and semi-structured control flow (break, return-in-the-middle of a function) in the unary setting, whereas we consider a proof system for higher-order functional programs without mutable state in both the unary and the relational settings. Nonetheless, as explained in Section 9, it seems to us that some of the key technical ideas developed by Carbonneaux et al. [2015] can be ported to U^{C} and R^{C} by defining a specific state and cost monad, and deriving their logic's rules as theorems, thus obtaining a proof framework that marries the best of both sides.

A lot of work for cost analysis relies on size types [Avanzini and Dal Lago 2017; Crary and Weirich 2000; Danner et al. 2015; Serrano et al. 2013]. Size types only specify the sizes of data structures, so they must be combined with other techniques to reason about costs. One approach is to use a cost-passing encoding to make cost an explicit output and to reason about its size [Avanzini

and Dal Lago 2017; Danner et al. 2015]. Crary and Weirich [2000] take a different approach that resembles a type-and-effect system. To the best of our knowledge, size types have not been used in the context of relational cost analysis, or for cost analysis in combination with expressive refinements, so all examples in this paper would be outside of their purview. We believe, but have not yet shown, that analysis based on size types can be embedded in U^C. Dal Lago and Gaboardi [2011]; Dal Lago and Petit [2013] follow a related approach, where cost is established by counting the number of uses of a term through a linear dependent type system. It is unclear to us whether this approach can be embedded in U^C/R^C.

RelCost [Çiçek et al. 2017] is a type-and-effect system that uses DML-like refinements for relational and unary cost analysis. The expressiveness of these refinements is limited and RelCost cannot handle any of the examples in this paper. In Section 7, we showed an embedding of RelCost into R^C. Nonetheless, some of our rules, e.g., R-SPLIT are inspired by similar rules in RelCost.

Going beyond type/logic-based systems, there is a significant amount of work on resource bound analysis for imperative programs. The focus is on fully automated techniques and a variety of different approaches to resource bound analysis have been developed, e.g., based on recurrence equations [Albert et al. 2012; Debray et al. 1990; Flores-Montoya and Hähnle 2014], template constraints [Carbonneaux et al. 2015], term-rewriting systems [Avanzini et al. 2015; Brockschmidt et al. 2016], ranking functions [Alias et al. 2010], abstract-interpretation [Gulwani et al. 2009; Gulwani and Zuleger 2010; Hermenegildo et al. 2005], abstract program models [Sinn et al. 2014, 2017; Zuleger et al. 2011] and interactive verification [Madhavan et al. 2017]. These approaches can compute some resource bounds that are value-dependent; however, complicated value-dependence—such as in the examples of this paper—is out of reach for these automated techniques.

R^C builds on two basic ideas—relational analysis for higher-order programs and monads. Relational analysis for higher-order programs has been studied extensively. Some of the work is based on higher-order relational refinements [Barthe et al. 2014, 2015], but the discipline of refinement types imposes strong limitations; in particular, reasoning about structurally different programs is restricted. Relational Higher-Order Logic (RHOL), and its unary counterpart UHOL, constitute an alternative approach that separates typing from logical reasoning, and supports reasoning about structurally different programs [Aguirre et al. 2017b]. Our work builds directly on RHOL and UHOL. Specifically, the pure fragment of R^C (U^C) is almost exactly RHOL (UHOL), with the difference that we added inductive datatypes to support more interesting programs. This change required us to rework parts of the soundness theorems of RHOL and UHOL. The cost monad, the monadic judgments, the rules for reasoning about costs (in L^C, U^C and R^C) and the proofs of soundness and completeness of U^C/R^C with respect to L^C are completely new to our work, and constitute our key technical contribution. At a conceptual level, our work shows how to extend the RHOL framework with a side-effect (cost) and prove properties of the side-effect that depend on nontrivial functional invariants. The original RHOL paper also includes some examples of cost analysis in RHOL directly, but these examples encode cost as an explicit program output and reason about cost as an ordinary value, without developing any specific reasoning principles pertaining to cost as an effect. Our development, on the other hand, treats cost as an effect, encapsulated in a monad with dedicated rules for verification, thus improving clarity in proofs.

Monads are widely used to represent and isolate effects (see, e.g., Wadler and Thiemann [2003]). The specific presentation of monads we follow is due to Pfenning and Davies [2001, Section 8], who introduce the idea of what we call the pure and monadic judgments (but without refinements and without any specific effect). This separation not only simplifies the equational theory (an aspect we did not highlight here) but also aids the design of monadic refinements in our setting.

Going beyond cost analysis, there is a lot of existing work on combining monads with refinements and dependent types. As examples, we mention HTT and RHTT, which define a state monad

with refinements for Hoare-style pre- and post-conditions in the unary and relational settings, respectively [Nanevski et al. 2013, 2008]. The new F* language includes monads for state and exceptions in a setting of rich (unary) refinements [Swamy et al. 2016].

11 SUMMARY

We have presented two frameworks—R^C and U^C—that combine cost analysis with program logics in the relational and unary settings, respectively. The combination is both theoretically simple and expressive. We are able to verify several new examples that highlight the importance of value-dependence, nonstandard refinements and functional correctness for cost analysis. As further evidence of expressiveness, we embed existing systems for cost analysis in R^C and U^C.

REFERENCES

Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Aleš Bizjak, Marco Gaboardi, and Deepak Garg. 2017a. Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. (2017). In preparation.

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017b. A Relational Logic for Higher-Order Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. http://arxiv.org/abs/1703.05042

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

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2 (2011).

Martin Avanzini and Ugo Dal Lago. 2017. Automating sized type inference for complexity analysis. In *Proceedings of DICE-FOPARA*. https://arxiv.org/abs/1704.05585.

Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2015. Analysing the complexity of functional programs: higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 152–164. https://doi.org/10.1145/2784731.2784753

Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'14*, Suresh Jagannathan and Peter Sewell (Eds.). 193–206.

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). 55–68.

Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. 2011. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.* 412, 25 (2011), 2776–2796.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50. http://dl.acm.org/citation.cfm?id=2866575

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2015*. 467–478. https://doi.org/10.1145/2737924.2737955

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 316–329.

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2016. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Logical Methods in Computer Science* 12, 3 (2016).

Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. 133–142.

Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages (POPL '13)*. 167–178.

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proceedings of the 35th Symposium on Principles of Programming Languages (POPL)*.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. 140–151.

S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. 1990. Task Granularity Analysis in Logic Programs. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*. 174–188.

Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*. 275–295. https://doi.org/10.1007/978-3-319-12736-1_15

Bernd Grobauer. 2001. Cost recurrences for DML programs. In *Proceedings of the 6th International Conference on Functional Programming (ICFP)*.

Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual Symposium on Principles of Programming Languages (POPL '09)*. 127–139.

Sumit Gulwani and Florian Zuleger. 2010. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 292–304. https://doi.org/10.1145/1806596.1806630

M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* 58, 1–2 (October 2005), 115–140.

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

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *24rd Int. Conf. on Computer Aided Verification (CAV'12)*.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Bart Jacobs. 1999. *Categorical Logic and Type Theory*. Elsevier. Studies in Logic and the Foundations of Mathematics 141.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *Journal of Automated Reasoning* 59, 1 (2017), 87–120.

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 330–343. http://dl.acm.org/citation.cfm?id=3009874

Eugenio Moggi. 1991. Notions of computations and monads. 93, 1 (1991), 55–92.

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.* 35, 2 (2013), 6:1–6:41.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911.

Van Chan Ngo, Mario Dehesa-Azuara, Matt Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security & Privacy*.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

A. Serrano, P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. 2013. Sized Type Analysis for Logic Programs. In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement*, Vol. 13. Cambridge U. Press, 1–14.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 745–761. https://doi.org/10.1007/978-3-319-08867-9_50

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *J. Autom. Reasoning* 59, 1 (2017), 3–45. https://doi.org/10.1007/s10817-016-9402-4

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 256–270.

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Trans. Comput. Log.* 4, 1 (2003), 1–32.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), part of SPLASH 2017*.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th Symposium on Principles of Programming Languages (POPL '99)*. 214–227.

Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. 280–297. https://doi.org/10.1007/978-3-642-23702-7_22