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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART36
https://doi.org/10.1145/3158124
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 \( \Gamma \) is a simply typed context, \( \tau_1 \) and \( \tau_2 \) are the simple types of \( e_1 \) and \( e_2 \), \( \Psi \) is a set of assumed assertions about free variables and \( \phi \) is a HOL assertion about \( e_1, e_2 \).\(^1\) \( \phi \) can be read either as a postcondition for \( e_1, e_2 \) or as a relational refinement for the types \( \tau_1, \tau_2 \). 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

\[
\Gamma ; \Psi \vdash m_1 \wedge m_2 \mid n \mid \phi
\]

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 \( \phi \) 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 (\( \phi \)) 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 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 $\mathbb{R}^C$ great expressiveness—in fact, we show that its expressiveness equals that of HOL. Finally, $\mathbb{R}^C$ 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 $\mathbb{R}^C/\mathbb{U}^C$ is simple; we prove the framework sound in a set-theoretic model through a cost-passing interpretation of the monadic expressions. We illustrate $\mathbb{R}^C/\mathbb{U}^C$’s working on several examples that were out of reach of previous systems. Moreover, we demonstrate that $\mathbb{R}^C/\mathbb{U}^C$ can be used for embedding other existing cost analyses.

To summarize, the contributions of this work are:

- We present the logic/refinement frameworks $\mathbb{R}^C$ and $\mathbb{U}^C$ 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 $\mathbb{R}^C$ and $\mathbb{U}^C$.
- We demonstrate that $\mathbb{R}^C$ and $\mathbb{U}^C$ 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 implementation of $\mathbb{R}^C$ and $\mathbb{U}^C$ is out of the scope of this paper. Nonetheless, since $\mathbb{R}^C$ and $\mathbb{U}^C$ separate refinements from typing syntactically, an implementation of interactive proofs based on constraint solving seems feasible.

The programming language underlying $\mathbb{R}^C$ and $\mathbb{U}^C$ 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 $\mathbb{R}^C$ and $\mathbb{U}^C$ 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 $\lambda$-calculus that syntactically separates pure, cost-free expressions, $e$, from monadic expressions, $m$, that have cost. The syntax is shown below.

Types

| $\tau$ | ::= $b \mid \theta \mid \tau \times \tau \mid \tau + \tau \mid \tau \rightarrow \tau \mid \mathbb{C}(\tau)$ |

(Pure) expr. $e, n, k, \ell$ | ::= $x \mid \langle e, e \rangle \mid \pi_i e \mid \text{inj}_i e \mid \text{case } e \text{ of } e; e \mid e \mid \lambda x.e \mid c \mid \mathbb{C}(\bar{e})$

| | | | | | | | | | $\mid \text{match } e \text{ with } \overline{\mathbb{C}i} \mapsto e_i \mid \text{rec } f(x).e \mid \{m\}$ |

Monadic expr. $m$ | ::= $\text{cret}(e) \mid \text{cbind}(e_1, \{x\}.m_2) \mid \text{cstep}_n(m)$ |

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 $\ell$ 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 $\tau$ when forced. $\theta$ 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 $e = \text{inj}_j e'$, then (case $e$ of $e_1; e_2$) reduces to $(e_j e')$. $\mathbb{C}(\bar{e})$ constructs an expression of a data type by applying the constructor $C$ to the terms $\bar{e}$. The expression (match $e$ with $\overline{\mathbb{C}i} \mapsto e_i$) is the corresponding case analysis. If $e = C_j(e_1', \ldots, e_k')$, then
(match e with \( \overline{C_i \mapsto e_i} \)) reduces to \( (e_j e' \ldots e'_l) \). \( \text{rec } f(x).e \) 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 \( n \).

\[
\begin{array}{c}
\text{cret}(e) \Downarrow^0 e \\
\text{cbind}(e_1, \{x\}.m_2) \Downarrow^{n+n_2} e_2 \\
\text{cstep}_n(m) \Downarrow^{n+n'} e
\end{array}
\]

\( \text{cret}(e) \), when forced, returns \( e \) immediately with 0 cost. This is the usual “return” or “unit” of the monad. Forcing \( \text{cbind}(e_1, \{x\}.m_2) \) first evaluates \( e_1 \) purely to some \( \{ m_1 \} \) (\( \rightarrow \) 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 \( m_2[e'_1/x] \). \text{cbind} is the usual “bind” of the monad. \( \text{cstep}_n(m) \) 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 \( \theta = C_1(\sigma_{1,1} \times \ldots \times \sigma_{1,a_1}) + \ldots + C_n(\sigma_{n,1} \times \ldots \times \sigma_{n,a_n}) \). This defines a data type \( \theta \) with \( n \) distinct constructors \( C_1, \ldots, C_n \). The types of the arguments of the constructors, denoted \( \sigma \), 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 \( \text{list} = \text{nil}() + \text{cons}(\mathbb{Z} \times \text{list}) \). All data type definitions are assumed to be collected in a context \( \Theta \), which we leave implicit in judgments.

A typing environment \( \Gamma \) assigns types to variables, as usual. We define two typing judgments, \( \Gamma \vdash e : \tau \) and \( \Gamma \vdash m \Downarrow \tau \), for pure and monadic expressions, respectively. In the second typing judgment, \( \tau \) 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 \( \text{rec } f(x).e \). This rule requires that the body of the recursive definition satisfy a predicate \( \text{Def}(f, x, e) \), which ensures that all recursive calls are performed on smaller arguments. The language has standard metatheoretic properties like subject reduction.

### \( \beta \)-reduction and equational theory

The equality relation \( \Downarrow \) defines when two pure or monadic expressions are equal to each other. As usual, we define \( \Downarrow \) as the congruence closure of small-step \( \beta \)-reduction. On pure expressions, the rules for \( \Downarrow \) are the expected ones. As an example, \( (\text{rec } f(x).e) e' \Downarrow (\text{rec } f(x).e)[e'/x] \). 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 for the \text{cstep} construct. The main rules are listed below.

\[\text{cstep}_n(m) \rightarrow m \quad \text{cstep}_{n_1}(\text{cstep}_{n_2}(m)) \rightarrow \text{cstep}_{n_1+n_2}(m) \quad \text{cbind}(\{m_1\}, \{x\}.m_2) \rightarrow \|m_1/x\|m_2\]

\(^3\)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.
\[
C(\sigma_1 \times \cdots \times \sigma_n) \in \Theta(\theta) \quad \Gamma \vdash e_i : \sigma_i \text{ for all } 1 \leq i \leq n
\]
\[
\Gamma \vdash C(e_1, \ldots, e_n) : \theta
\]
\[
\Gamma \vdash e : \theta \quad \theta = C_1(\sigma_1, \ldots, \sigma_{a_1}) + \cdots + C_n(\sigma_{a_1}, \ldots, \sigma_{a_n}) \in \Theta
\]
\[
\Gamma \vdash e_i : \sigma_{a_1} \rightarrow \cdots \rightarrow \sigma_{a_n} \rightarrow \tau \text{ for all } 1 \leq i \leq n
\]
\[
\Gamma \vdash \text{match } e \text{ with } e_1 \mapsto \cdots ; e_n \mapsto e_n : \tau
\]

The new substitution \(\|m_1/m_2\|_m_2\) is defined by induction on \(m_1\) (not \(m_2\)):
\[
\|\text{cret}(e)/x\|_m_2 \triangleq m_2[e/x] \quad \|\text{cbind}(e, \{y\}m)/x\|_m_2 \triangleq \text{cbind}(e, \{y\}|m/x/m_2)
\]
\[
\|\text{cstep}_n(m)/x\|_m_2 \triangleq \text{cstep}_n(\|m/x/m_2\|)
\]

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: \(\text{cbind}(\text{cstep}_n(\text{cret}(e_1))), \{x\}.\text{cstep}_n(\text{cret}(e_2))) \rightarrow \text{cstep}_n(\text{cret}(\{e_2[e_1/x]\})) \rightarrow \text{cstep}_{n+n_2}(\text{cret}(e_2[e_1/x]))\). This suggests that every closed, well-typed monadic expression has a “normal form” of the shape cstep\(_n\)(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 \(m \| e\), then \(m \triangleq \text{cstep}_n(\text{cret}(e))\).

**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., \([\mathbb{R}^\omega] \triangleq \mathbb{R} \cup \{-\infty, \infty\}\); products, sums and arrows map homomorphically to their set-theoretic analogues, e.g., \([\tau_1 \rightarrow \tau_2] \triangleq [\tau_1] \rightarrow [\tau_2]\); data types \(\theta\) map to initial (tree) algebras. The interpretation of \(C(\tau)\) is more interesting: \([C(\tau)] \triangleq [\tau] \times [\mathbb{R}^\omega]\), representing both the returned pure expression and the cost. (Technically, this makes \(C(\tau)\) a specific writer monad.)

The interpretation \(\langle \cdot \rangle_\rho\) of expressions is indexed by a valuation \(\rho\) for free variables. Pure expressions have the expected interpretations, e.g., \(\langle \langle e_1, e_2 \rangle \rangle_\rho \triangleq \langle \langle e_1 \rangle_\rho, \langle e_2 \rangle_\rho \rangle\). The recursive function definition is interpreted via a fixpoint: \(\langle \text{rec } f(x).e \rangle_\rho \triangleq \text{fix}(\lambda F. \lambda x. \langle e \rangle_\rho.f \rightarrow F)\). 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 \([\tau] \times [\mathbb{R}^\omega]\), representing the returned expression and the cost. The cost is accumulated over binds using the addition operator \(+\) on \([\mathbb{R} \cup \{-\infty, \infty\}\]. This interpretation is reminiscent of Danner et al. [2015] and Grobauer [2001].

\[
\langle \langle m \rangle \rangle_\rho \triangleq \langle m \rangle_\rho \quad \langle \text{cret}(e) \rangle_\rho \triangleq \langle \langle e \rangle_\rho, 0 \rangle \quad \langle \text{cstep}_n(m) \rangle_\rho \triangleq \left( \pi_1 \langle m \rangle_\rho, \langle m \rangle_\rho + \pi_2 \langle m \rangle_\rho \right)
\]
\[
\langle \text{cbind}(e, \{x\}m) \rangle_\rho \triangleq \text{let } y \leftarrow \langle e \rangle_\rho \text{ in let } x \leftarrow \pi_1 y \text{ in let } z \leftarrow \langle m \rangle_\rho \text{ in } \left( \pi_1 z, \pi_2 y + \pi_2 z \right)
\]
General rules for equality. Here, $u ::= e \mid m$.

- $u_1 \rightarrow u_2$ \quad \text{BETA} \\
  \frac{}{\Gamma; \Psi \vdash \Gamma C \ u_1 \equiv u_2} \quad \frac{}{\Gamma; \Psi \vdash \Gamma C \ u \equiv u} \quad \frac{}{\Gamma; \Psi \vdash \Gamma C \ \phi[u_1/x]} \quad \frac{}{\Gamma; \Psi \vdash \Gamma C \ \phi[u_2/x]} \quad \text{SUBST}

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

- \text{Axioms specific to monadic expression equality}

  $\Gamma; \Psi \vdash \Gamma C \ \phi$ if $\phi$ is one of:

  $\forall x, y : \tau. \ |x| \leq |y| \Rightarrow x \equiv y$  \\
  $\forall x : \mathbb{C}(\tau). \ \exists y : \tau. \ x \equiv |y|$  \\
  $\forall x : \tau. \ \exists y : \mathbb{R}^\infty, z : \tau. \ x \equiv \text{cstep}_y(\text{cret}(z))$

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

- \text{Rules for data types}

  $\theta = C_1(\sigma_{1,1} \times \cdots \times \sigma_{1,a_1}) + \cdots + C_n(\sigma_{n,1} \times \cdots \times \sigma_{n,a_n}) \in \Theta$

  $\Gamma \vdash e : \theta$  \\
  $\Gamma, x_1 : \sigma_{1,1}, \ldots, x_{a_1} : \sigma_{1,a_1}; \Psi, e \vdash C_1(x_1, \ldots, x_{a_1}) \vdash \Gamma C \ \phi$

  for all $1 \leq i \leq n$  \\
  where $x_1, \ldots, x_{a_1} \notin \phi$ \quad \text{ELIM}

  $\theta \in \Theta$  \\
  $\Gamma, x : \theta; \Psi, \forall y : \theta. \ |y| < |x| \Rightarrow \phi[y/x] \vdash \Gamma C \ \phi$

  \quad \text{IND}

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

\textbf{Theorem 2.2 (Soundness).} Let $\rho \vdash \Gamma$ mean that for each $x \in \text{dom}(\Gamma)$, $\rho(x) \in [\Gamma(x)]$. Then: (1) If $\Gamma \vdash e : \tau$ and $\rho \vdash \Gamma$, then $[e]_{\rho} \in [\tau]$. (2) If $\Gamma \vdash m : \tau$ 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 $\text{cstep}_n$. Consequently, a program’s cost analysis is correct only to the extent that it carries correct $\text{cstep}_n$ annotations. Danielsson [2008] calls this style of analysis “semi-formal”. We argue that this style aids expressiveness since $\text{cstep}_n$ 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.

\section{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 $\phi$. The letter $u$ denotes either a pure or a monadic expression ($u ::= e \mid m$).

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

$P$ denotes an atomic predicate. Predicates are defined through axioms. We pre-define expression equality, $u_1 \equiv u_2$, and add more predicates for typing examples as needed. The remaining constructs are standard. Variables $x : \tau$ and $x \vdash \tau$ represent pure and monadic expressions of type $\tau$, respectively.
The logic establishes judgments of the form $\Gamma; \Psi \vdash_{C} \phi$ where $\Gamma$ is a typing context with assumptions of the forms $x : \tau$ and $x \vdash \tau$, and $\Psi$ 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 $\text{BETA}$ subsumes reduction $\to$ into the logic’s equality $\equiv$. The axioms in the middle of the figure specify important properties of $\equiv$ 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, $\text{cstep}_n(m) \equiv \text{cstep}_n'(\text{cret}(e)) \Rightarrow \exists n'. n' \equiv n + n'' \wedge m \equiv \text{cstep}_n'(\text{cret}(e))$.

The rules $\text{ELIM}$ and $\text{IND}$ allow case analysis and induction on data types. In the rule $\text{IND}$, the notation $|e|$ stands for the depth of $e$ (which must be of some data type $\theta$). 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 $\equiv$ 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 $\Gamma; \Psi \vdash_{C} \phi, \rho \models \Gamma$ and $\wedge_{\phi' \in \Psi}(\phi')^{\rho}$, 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:

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

The first judgment, also called the pure judgment, means that under assumptions $\Psi$, $e$ (of simple type $\tau$) satisfies $\phi[e/\tau]$, where $\tau$ is a distinguished variable in $\phi$ representing the result of $e$. $\phi$ can be viewed as either a postcondition for $e$ or a refinement for the type $\tau$. 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 $\phi[e/\tau]$ and $k$ and $\ell$ are expressions that denote, respectively, lower and upper bounds on the cost of $m$. In verifying programs, we use $\phi$ to represent functional properties of the output, and use $k$ and $\ell$ 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 $\phi$ for a variable $x$ if we can show $\phi[x/\tau]$ from the assumptions $\Psi$ in the assertion logic (rule $U^C\text{-VAR}$). For function types $\tau \rightarrow \tau'$, the refinement has the shape $\forall x.\phi \Rightarrow \phi'$, where $\phi$ is a refinement on the argument $x$ and $\phi'$ is a refinement on the result. In the rule $U^C\text{-LETREC}$ for $\text{rec}(f)(x).e$, we make the assumption that the refinement holds for the function $f$ for all arguments $y$ with $|y| < |x|$, and show that $e$ has the postcondition $\phi'$. Rule $U^C\text{-APP}$ allows applying a function if the argument satisfies the pre-condition.

Rule $U^C\text{-CONS}$ says that a data type constructor $C(e_1, \ldots, e_n)$ can be typed with refinement $\phi$ if the constructor transforms arguments with refinements $\{\phi_i\}_{i=1}^n$ to $\phi$ and the arguments actually have these refinements. Dually, $U^C\text{-MATCH}$ allows establishing refinement $\phi$ for (match $e$ with $C_1 \mapsto$

4The cost bounds $k$ and $\ell$ are drawn from $\mathbb{R}^\infty$ and can be $-\infty$ and $\infty$. 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.
Fig. 3. U^C rules
e_1; \ldots; C_n \mapsto e_n) when e has some refinement \( \phi' \) and each function \( e_i \) maps arguments \( x_1, \ldots, x_{a_i} \) satisfying \( \phi'[C[x_1, \ldots, x_{a_i}]/r] \) to a result satisfying \( \phi \).

Next, we define a refinement \( \text{C}_U(e, k, \ell, x, \phi) \) for the monadic type \( \text{C}(\tau) \).

\[
\text{C}_U(e, k, \ell, x, \phi) \triangleq \exists n, y. (e \equiv \{\text{cstep}_n(\text{cret}(y))\}) \land (k \leq n \leq \ell) \land \phi[y/x]
\]

Note that \( x \) is locally bound in \( \phi \). In words, the refinement means that \( e \) is (up to \( \equiv \)) \{\text{cstep}_n(\text{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 \( \ell \), respectively, and that \( y \) satisfies \( \phi \). Due to Lemma 2.1, a consequence is that if \( e \rightarrow^* \{m\} \), then \( m \), when forced, returns a pure expression (named \( y \)) that satisfies \( \phi \) and the cost of this forcing is lower- and upper-bounded by \( k \) and \( \ell \), respectively.

Rule \( U \text{-MONAD} \) 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 \( \ell \) and some postcondition \( \phi \). Then, the postcondition of \( \{m\} \) is \( \text{C}_U(r, k, \ell, x, \phi[x/r]) \).

Monadic expressions. The crux of \( U^C \) are the rules for typing monadic expressions. Rule \( U \text{-RET} \) says that \( \text{cret}(e) \) has postcondition \( \phi \) if \( e \) has postcondition \( \phi \), and that the cost of \( \text{cret}(e) \) is both lower- and upper-bounded by \( 0 \). This represents the fact that \( \text{cret}(e) \) forces to \( e \) with \( 0 \) cost. Rule \( U \text{-STEP} \) says that \( \text{cstep}_n(m) \) has cost bounds \( k + n \) and \( \ell + n \), and postcondition \( \phi \) if \( m \) has cost bounds \( k \) and \( \ell \) and the same postcondition \( \phi \). This represents the fact that \( \text{cstep}_n(m) \) forces like \( m \), but with additional cost \( n \). Finally, to type \( \text{cbind}(e_1, \{x\}.m_2) \), we first verify \( e_1 \) (which has a monadic type) purely with a refinement \( \text{C}_U(r, k', \ell', x, \phi_1) \), then verify \( m_2 \) assuming that \( x \) satisfies \( \phi_1 \). The bounds on the cost of \( \text{cbind}(e_1, \{x\}.m_2) \) are obtained by adding \( k' \) and \( \ell' \) to the bounds of \( m_2 \), and the postcondition is the same as that of \( m_2 \). Again, this directly reflects how \( \text{cbind}(e_1, \{x\}.m_2) \) 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 \( U \text{-SUB} \) weakens the postcondition of a pure expression. The rule \( U \text{-SUBC} \) 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 (\( U \text{-SUBM1} \) and \( U \text{-SUBM2} \)). The admissible rules \( U \text{-EQ-PURE} \) and \( U \text{-EQ-MONADIC} \) show that \( U^C \)'s judgments are closed under \( \equiv \).

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. \( \Gamma; \Psi \vdash e : \tau | \phi \text{ if and only if } \Gamma; \Psi \vdash L^C e[r] \text{ (and } \Gamma \vdash e : \tau) \).
2. \( \Gamma; \Psi \vdash m \vdash \tau | k | \ell | \phi \text{ if and only if } \Gamma; \Psi \vdash L^C \exists y, n. (m \equiv \text{cstep}_n(\text{cret}(y))) \land (k \leq n \leq \ell) \land \phi[y/r] \text{ (and } \Gamma \vdash m \vdash \tau) \).

**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 \( T \), and then using rule \( U \text{-SUB} \) 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\( \vdash m \parallel \tau \mid k \mid \ell \mid \phi \) and \( m \downarrow^n e \), then \( \vdash \ell \mid k \leq n \leq \ell \) 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.

\[
\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 \mid \phi \quad \Gamma; \Psi \vdash m_1 \equiv m_2 : \tau_1 \sim m_2 \equiv \tau_2 \mid n \mid \phi
\]

The first judgment, also called the pure judgment, is based on RHOL [Aguirre et al. 2017b] and means that under assumptions \( \Psi, e_1 \) (of simple type \( \tau_1 \)) and \( e_2 \) (of simple type \( \tau_2 \)) satisfy \( \phi[e_1/r_1][e_2/r_2] \). Here, \( r_1 \) and \( r_2 \) are distinguished variables in the relational assertion \( \phi \) 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 \( e_2 \), \( \phi[e_1/r_1][e_2/r_2] \) 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 \( \phi \) 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 \( R-\text{VAR} \) relates the variables \( x_1, x_2 \) at \( \phi \) when \( \phi[x_1/r_1][x_2/r_2] \) holds. The one-sided rule \( R-\text{VAR-L} \) relates \( x_1 \) to an arbitrary \( e_2 \) when \( \phi[x_1/r_1] \) holds and \( r_2 \) does not appear in \( \phi \) (however, \( \phi \) may contain \( e_2 \)). The two-sided rule \( R-\text{LETREC} \) applies to two recursive function definitions \( \text{rec} \ f_1(x_1).e_1 \) and \( \text{rec} \ f_2(x_2).e_2 \). 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 \( R-\text{LETREC-L} \) allows unfolding a recursive function definition on the left side only. The application rules \( R-\text{APP} \) and \( R-\text{APP-L} \) are dual. The rule \( R-\text{MATCH} \) relates two case-analyses on data types. We show here only a simplified version of the rule where the analyzed data type \( \theta \) 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 \( C_{\tau}(e_1, e_2, n, x_1, x_2, \phi) \) for the monadic type \( C(\tau) \):

\[
\exists y_1, n_1, y_2, n_2. \ e_1 \equiv \{ \text{cstep}_{n_1}(\text{cret}(y_1)) \} \land e_2 \equiv \{ \text{cstep}_{n_2}(\text{cret}(y_2)) \} \land \phi[y_1/x_1][y_2/x_2] \land n_1 - n_2 \leq n
\]

Here, \( x_1, x_2 \) are locally bound in \( \phi \). The relational refinement means that \( e_1 \) and \( e_2 \) are (up to \( \equiv \)) \( \{ \text{cstep}_{n_1}(\text{cret}(y_1)) \} \) and \( \{ \text{cstep}_{n_2}(\text{cret}(y_2)) \} \), 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

\(^5\)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 $\Gamma; \Psi \vdash e_1 : r_1 \sim e_2 : r_2 \mid \phi$ (selected)

\[
\begin{align*}
\Gamma \vdash x_1 : r_1 & \quad \Gamma \vdash x_2 : r_2 & \quad \Gamma; \Psi \vdash \phi[x_1/r_1][x_2/r_2] & \quad \text{R-VAR} \\
\Gamma; \Psi \vdash x_1 : r_1 \sim x_2 : r_2 & \quad \phi & \quad \text{R-LETREC} \\
\end{align*}
\]

$\Phi \equiv \{f_1(x_1, e_1), f_2(x_2, e_2)\}
\Psi, \phi, \forall y_1 y_2. (|y_1|, |y_2|) < (|x_1|, |x_2|) \implies \phi[y_1/x_1][y_2/x_2] \implies \phi'[y_1/x_1][y_2/x_2][f_1 y_1 r_1][f_2 y_2 r_2]$

\[\Gamma; \Psi + \text{rec}\ f_1(x_1). e_1 : \theta_1 \rightarrow r_1 \sim \text{rec}\ f_2(x_2). e_2 : \theta_2 \rightarrow r_2 \mid \forall x_1 x_2. \phi \implies \phi'[r_1 x_1 r_1][r_2 x_2 r_2] \quad \text{R-APP}
\]

\[\begin{align*}
\phi & = C_1(\sigma_{1,1} \times \ldots \times \sigma_{1,a_1}) + \ldots + C_n(\sigma_{n,1} \times \ldots \times \sigma_{n,a_n}) \in \Theta \\
\Gamma; \Psi + e : \theta \sim e' : \theta' \mid \phi'
\end{align*}
\]

For all $1 \leq i, j \leq n$: $\Gamma; \Psi + e_i : \sigma_{i,1} \rightarrow \ldots \rightarrow \sigma_{i,a_i} \rightarrow r_1 \sim e_j' : \sigma_{j,1} \rightarrow \ldots \rightarrow \sigma_{j,a_j} \rightarrow r_2 \mid \phi'_{i,j}$ where $\phi'_{i,j} \equiv \forall x_1 : \sigma_{1,1}, \ldots, x_{a_1} : \sigma_{1,a_1}, y_1 : \sigma_{j,1}, \ldots, y_{a_j} : \sigma_{j,a_j}. C_i(x_1, \ldots, x_{a_1})/r_1][C_j(y_1, \ldots, y_{a_j})/r_2]

\[\begin{align*}
\Gamma; \Psi + e_1 : r_1 \sim e_2 : r_2 \mid \phi & \quad \text{R-MATCH} \\
\Gamma; \Psi + m_1 + r_1 \sim m_2 + r_2 \mid n \mid \phi & \quad \text{R-MONAD} \\
\end{align*}
\]

Two-sided rules for the monadic judgment $\Gamma; \Psi + m_1 + r_1 \sim m_2 + r_2 \mid n \mid \phi$

\[
\begin{align*}
\Gamma; \Psi \vdash e_1 : r_1 \sim e_2 : r_2 & \quad \phi & \quad \text{R-RET} \\
\Gamma; \Psi \vdash \text{cret}(e_1) + r_1 \sim \text{cret}(e_2) + r_2 & \quad 0 \mid \phi \\
\Gamma + n_1 : \mathbb{R}^\infty & \quad \Gamma + n_2 : \mathbb{R}^\infty & \quad \Gamma; \Psi + m_1 + r_1 \sim m_2 + r_2 \mid n \mid \phi & \quad \text{R-STEP} \\
\Gamma; \Psi \vdash \text{cstep}_{n_1}(m_1) + r_1 \sim \text{cstep}_{n_2}(m_2) + r_2 \mid n + n_1 - n_2 \mid \phi \\
\Gamma; \Psi + e_1 : r'_1 \sim e_2 : r'_2 \mid \text{C}_1(r_1, r_2, n', x_1, x_2, \phi') & \quad \Gamma, x_1 : r'_1, x_2 : r'_2, \Psi, \phi', m_1 + r_1 \sim m_2 + r_2 \mid \phi \quad x_1, x_2 \not\in n, \phi & \quad \text{R-BIND} \\
\Gamma; \Psi \vdash \text{cbind}(e_1, \{x_1\}, m_1) + r_1 \sim \text{cbind}(e_2, \{x_2\}, m_2) + r_2 \mid n' + n \mid \phi & \quad \text{R-MONAD}
\end{align*}
\]

Fig. 4. R^c two-sided rules

$n$, and that $y_1$ and $y_2$ satisfy the relational assertion $\phi$. Lemma 2.1 then implies that if $e_1 \rightarrow^* \{m_1\}$ and $e_2 \rightarrow^* \{m_2\}$, then $m_1$ and $m_2$, when forced, return pure expressions (named $y_1$ and $y_2$) that satisfy $\phi$ 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 $\mathbb{C}_r(r_1, r_2, n, r_1, r_2, \phi)$, we should verify, using the monadic judgment, that the monadic expressions $m_1$ and $m_2$ are related at the assertion $\phi$ and that their relative cost is at most $n$.

Monadic expressions. Rule R-RET says that we can relate $\text{cret}(e_1)$ and $\text{cret}(e_2)$ at $\phi$ if $e_1$ and $e_2$ are related at $\phi$, and that the relative cost is upper-bounded by 0. This corresponds to the fact that $\text{cret}(e)$ forces to $e$ with 0 cost. The one-sided rule R-RET-L relates $\text{cret}(e_1)$ to an arbitrary $m_2$. The
relative cost is \(-k\), where \(k\) is a lower bound on the unary cost of forcing \(m\). Rule \textsc{r-step} relates two expressions \(\text{cstep}_{n_1}(m_1)\) and \(\text{cstep}_{n_2}(m_2)\) at \(\phi\) if \(m_1\) and \(m_2\) are related at \(\phi\). 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 \(\text{cstep}_n(m)\) forces like \(m\), but with additional cost \(n\). The one-sided rule \textsc{r-step-l} analyzes only a left expression of
the shape \text{cstep}_n_1(m_1): it increases the relative cost by \(n_1\). Finally, the rule \text{R-BIND} can be used to relate \text{cbind}(e_1, \{x_1\}, m_1) and \text{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 \text{R-BIND-L} performs a unary analysis of \(e_1\).

**Structural and admissible rules.** Figure 6 shows structural and admissible rules. The rule \text{R-SUB} weakens the postcondition of two pure expressions. The rule \text{R-SUBC} weakens the relative cost of two monadic expressions. The admissible rules \text{R-SUBM1} and \text{R-SUBM2} allow weakening of relational postconditions. The admissible rule \text{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. \(\Gamma; \Psi \vdash e_1 : \tau_1 \sim e_2 : \tau_2 | \phi \text{ if and only if } \Gamma; \Psi \vdash C[e_1/r_1][e_2/r_2] (\text{and } \Gamma \vdash e_1 : \tau_1 \text{ and } \Gamma \vdash e_2 : \tau_2).\)
2. \(\Gamma; \Psi \vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 | n \text{ if and only if } \Gamma; \Psi \vdash _{L^C} \exists e_1, e_2, n_1, n_2, m_1 \equiv \text{cstep}_{n_1} (\text{cret}(e_1)) \land m_2 \equiv \text{cstep}_{n_2} (\text{cret}(e_2)) \land \phi[e_1/r_1][e_2/r_2] \land n_1 - n_2 \leq n (\text{and } \Gamma \vdash m_1 \div \tau_1 \text{ and } \Gamma \vdash m_2 \div \tau_2).\)

**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\(\vdash m_1 \div \tau_1 \sim m_2 \div \tau_2 | n \mid \phi \text{ and } m_1 \parallel_{n_1} e_1 \text{ and } m_2 \parallel_{n_2} e_2, \text{ then } \vdash e_1 : \tau_1 \sim e_2 : \tau_2 | \phi.\)

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 nonstandard 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 \text{cret}, writing \(e\) in place of \text{cret}(e). We use \(\uparrow^n m\) as alternate notation for \text{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 \text{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 \(\ell\), where the list type is
defined as usual with the equation \(\text{list}_\mathbb{N} = \text{nil}() + \text{cons}(\mathbb{N} \times \text{list}_\mathbb{N})\).

\[
insert \triangleq \lambda x. \text{rec } f(\ell).
\]

match \(\ell\) with \(\text{nil} \mapsto \{x :: \text{nil}\};
\]

\[
\text{cons} \mapsto \lambda h, t. \text{match } x \leq h \text{ with } \text{tt} \mapsto \{x :: (h :: t)\};
\]

\[
\text{ff} \mapsto \{t' \leftarrow f \cdot t; \uparrow^1 h :: t'\}
\]

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

\[
\forall x, \ell, n. \text{LargerThan}(x, \ell, n) \Leftrightarrow (n \equiv 0 \land \ell \equiv \text{nil})
\]

\[
\lor (\exists h, t. \ell \equiv h :: t \land x \leq h \land \text{LargerThan}(x, t, n))
\]

\[
\lor (\exists h, t, n'. \ell \equiv h :: t \land x \not< h \land \text{LargerThan}(x, t, n') \land n \equiv n' + 1)
\]

\[
\forall \ell, n. \text{Unsorted}(\ell, n) \Leftrightarrow (n \equiv 0 \land \ell \equiv \text{nil})
\]

\[
\lor (\exists h, t, n_1, n_2. \ell \equiv \text{cons}(h, t) \land \text{LargerThan}(h, t, n_1) \land \text{Unsorted}(t, n_2) \land n \equiv n_1 + n_2)
\]

\[
\forall l. \text{Sorted}(\ell) \Leftrightarrow \text{Unsorted}(\ell, 0)
\]

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

We show two additional properties of \(\text{insert}\), for later use in our verification of insertion sort:

(1) The list output by \(\text{insert}\) 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 \(L^C\) that:

\[
\vdash \text{insert} : \mathbb{N} \to \text{list}_\mathbb{N} \to \bigcirc(\text{list}_\mathbb{N}) \mid \forall x, \ell. \text{Sorted}(\ell) \Rightarrow \forall n. \text{LargerThan}(x, \ell, n) \Rightarrow \bigcirc_U(r \times \ell, n, n, r.\phi)
\]

where \(\phi \triangleq \text{Sorted}(r) \land \forall y, q. \text{LargerThan}(y, n, n, r, q) \Rightarrow \text{LargerThan}(y, r, q)\).

Following the syntax of \(\text{insert}\), the proof first applies a \(U\) rule for \(\lambda\)-abstraction, which is similar to \(U\)-\textsc{letrec}, but without the inductive hypothesis. Next we apply \(U\)-\textsc{letrec} to get the inductive hypothesis (IH): \(\forall m. |m| < |\ell| \Rightarrow \text{Sorted}(m) \Rightarrow \forall n. \text{LargerThan}(x, m, n) \Rightarrow \bigcirc_U(r \times m, n, n, r.\phi[m/f])\).

Next, by rule \(U\)-\textsc{match} we need to consider two cases. For the case \(\ell \equiv \text{nil}\) we need to show

\[
\vdash \{x :: \text{nil}\} : C(\text{list}_{\mathbb{N}}) \mid \bigcirc_U(r, n, n, n, r.\phi)
\]

The assumptions \(\ell \equiv \text{nil}\) and \(\text{LargerThan}(\ell, x, n)\) force \(n \equiv 0\). The cost part follows trivially by the rules \(U\)-\textsc{monad} and \(U\)-\textsc{ret}. The proof of \(\phi\) is also easy.

For the case when \(\ell \equiv \text{cons}(h, t)\) we further apply the rule for \(\lambda\) twice and the rule \(U\)-\textsc{match}. This yields two sub-cases: \(x \leq h\) and \(x \not< h\) (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

\[
\vdash \{t' \leftarrow f \cdot t; \uparrow^1 h :: t'\} : C(\text{list}_{\mathbb{N}}) \mid \bigcirc_U(r, n, n, n, r.\phi)
\]

From \(\ell \equiv \text{cons}(h, t)\), \(\text{LargerThan}(x, \ell, n)\) and \(x \not< h\), it follows \(\text{LargerThan}(x, t, n')\) for some \(n' \equiv \mathbb{N}\) s.t. \(n \equiv n' + 1\). From \(\text{Sorted}(\ell)\), it follows that \(\text{Sorted}(t)\). Then, by applying the IH and using Theorem 4.1, we get

\[
\vdash \{t' \leftarrow f \cdot t; \uparrow^1 h :: t'\} : C(\text{list}_{\mathbb{N}}) \mid \bigcirc_U(r, n', n', t'.\phi[t/\ell][t'/r])
\]

Using the rules \(U\)-\textsc{bind} and \(\text{subc}\) (with \(n' + 1 \equiv n\)) it suffices to show

\[
\vdash \uparrow^1 h :: t' \equiv \text{list}_{\mathbb{N}} \mid 1 | 1 | \phi
\]

The cost part follows from the rules \(U\)-\textsc{step} and \(U\)-\textsc{ret}. Proving \(\phi\) 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 \(\equiv\) ff, 1 \(\equiv\) 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.

\[
\begin{align*}
\text{bool} & \equiv \text{tt}() + \text{ff}() \\
\text{list} & \equiv \text{nil}() + \text{cons}(\text{bool} \times \text{list}) \\
\text{inc} & \equiv \text{rec}\ f_1(\ell_1). \text{match}\ \ell_1 \text{ with} \\
& \quad \text{nil} \mapsto \{\text{nil}\}; \\
& \quad \text{cons} \mapsto \lambda x_1, t_1. \text{match}\ x_1 \text{ with} \quad \text{ff} \mapsto \{\uparrow^1 \text{tt} :: t_1\}; \quad \text{tt} \mapsto \{t'_1 \leftarrow f_1 \ t_1; \ \uparrow^1 \text{ff} :: t'_1\}
\end{align*}
\]

\[
\begin{align*}
\text{dec} & \equiv \text{rec}\ f_2(\ell_2). \text{match}\ \ell_2 \text{ with} \\
& \quad \text{nil} \mapsto \{\text{nil}\}; \\
& \quad \text{cons} \mapsto \lambda x_2, t_2. \text{match}\ x_2 \text{ with} \quad \text{ff} \mapsto \{t'_2 \leftarrow f_2 \ t_2; \ \uparrow^1 \text{tt} :: t'_2\}; \quad \text{tt} \mapsto \{\uparrow^1 \text{ff} :: t_2\}
\end{align*}
\]

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 \(\text{dual}(\ell_1, \ell_2)\) on lists of booleans, which says that \(\ell_1\) and \(\ell_2\) have the same length and are pointwise dual.

\[
\forall \ell_1, \ell_2. \text{dual}(\ell_1, \ell_2) \iff (\ell_1 \equiv \ell_2 \equiv \text{nil}) \lor \quad (\exists x_1, t_1, x_2, t_2. \ \ell_1 \equiv \text{cons}(x_1, t_1) \land \ell_2 \equiv \text{cons}(x_2, t_2) \land x_1 \neq x_2 \land \text{dual}(t_1, t_2))
\]

We can then state our invariant as an \(\text{R}^C\) judgment:

\[
\vdash \text{inc} : \text{list} \to \text{C(list)} \sim \text{dec} : \text{list} \to \text{C(list)} \mid \forall \ell_1, \ell_2. \text{dual}(\ell_1, \ell_2) \Rightarrow \text{C}(\ell_1, \ell_2, 0, r_1, r_2. \text{dual}(r_1, r_2))
\]

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

When \(x_1 \equiv \text{ff}, x_2 \equiv \text{tt}\), our goal reduces to proving \(\vdash \{\uparrow^1 \text{tt} :: t_1\} : \text{C}(\text{list}) \sim \{\uparrow^1 \text{ff} :: t_2\} : \text{C}(\text{list}) \mid \text{C}(r_1, r_2, 0, r_1, r_2. \text{dual}(r_1, r_2))\). Applying the rules \text{R-MONAD}, \text{R-STEP} and \text{R-RET}, this reduces to \(\vdash \text{ff} :: t_1 : \text{list} \sim \text{tt} :: t_2 : \text{list} \mid \text{dual}(r_1, r_2)\), which follows immediately by switching to \(\text{L}^C\) since \(\text{dual}(t_1, t_2)\).

When \(x_1 \equiv \text{tt}, x_2 \equiv \text{ff}\), our goal is \(\vdash \{t'_1 \leftarrow f_1 \ t_1; \ \uparrow^1 \text{ff} :: t'_1\} : \text{C}(\text{list}) \sim \{t'_2 \leftarrow f_2 \ t_2; \ \uparrow^1 \text{tt} :: t_2\} : \text{C}(\text{list}) \mid \text{C}(r_1, r_2, 0, r_1, r_2. \text{dual}(r_1, r_2))\). Using \text{R-MONAD}, this reduces to \(\vdash t'_1 \leftarrow f_1 \ t_1; \ \uparrow^1 \text{ff} :: t'_1 \sim \text{list} \sim t'_2 \leftarrow f_2 \ t_2; \ \uparrow^1 \text{tt} :: t_2 \sim \text{list} \mid 0 \mid \text{dual}(r_1, r_2)\). From the IH, we derive (via \(\text{L}^C\)) that \(\vdash f_1 \ t_1 : \text{C}(\text{list}) \sim f_2 \ t_2 : \text{C}(\text{list}) \mid \text{C}(r_1, r_2, 0, r_1, r_2. \text{dual}(r_1, r_2))\). Hence, by rule \text{R-BIND}, it suffices to prove that
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 \( \text{and}(e_1, e_2) \) when \( e_1 \) evaluates to \( \text{ff} \) (and analogously for \( \text{or} \)).

\[
\begin{align*}
\text{eval}_1 & \triangleq \text{rec } f_1(e_1).\text{match } e_1 \text{ with} \\
& \quad \text{const } \mapsto \lambda b'.\{b'\} \\
& \quad \text{and } \mapsto \lambda e', e''.\{x_1 \leftarrow f_1 e';
\quad y_1 \leftarrow f_1 e''; \quad \uparrow^1 \text{match } x_1 \text{ with } \text{tt } \mapsto y_1; \text{ff } \mapsto \text{ff}\} \\
& \quad \text{or } \mapsto \lambda e', e''.\{x_1 \leftarrow f_1 e';
\quad y_1 \leftarrow f_1 e''; \quad \uparrow^1 \text{match } x_1 \text{ with } \text{tt } \mapsto \text{tt}; \text{ff } \mapsto y_1\}
\end{align*}
\]

\[
\begin{align*}
\text{eval}_2 & \triangleq \text{rec } f_2(e_2).\text{match } e_2 \text{ with} \\
& \quad \text{const } \mapsto \lambda b'.\{b'\} \\
& \quad \text{and } \mapsto \lambda e', e''.\{x_2 \leftarrow f_2 e';
\quad y_2 \leftarrow (\text{match } x_2 \text{ with } \text{tt } \mapsto f_2 e''; \text{ff } \mapsto \{\text{ff}\}); \quad \uparrow^1 y_2\} \\
& \quad \text{or } \mapsto \lambda e', e''.\{x_2 \leftarrow f_2 e';
\quad y_2 \leftarrow (\text{match } x_2 \text{ with } \text{tt } \mapsto \{\text{tt}\}; \text{ff } \mapsto f_2 e''); \quad \uparrow^1 y_2\}
\end{align*}
\]

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 eval1 is no less than the cost of eval2. This property can be established trivially in \( R^C \). Here, we define a refinement noshort on bexprs, which ensures that the costs of eval1 and eval2 are equal and show that this is actually the case.

\[
\forall e, b. \text{noshort}(e, b) \Leftrightarrow (\exists b'. e \equiv \text{const}(b') \land b \equiv b') \lor
\]

\[
(\exists e_1, e_2, b. e \equiv \text{and}(e_1, e_2) \land \text{noshort}(e_1, \text{tt}) \land \text{noshort}(e_2, b)) \lor
\]

\[
(\exists e_1, e_2, b. e \equiv \text{or}(e_1, e_2) \land \text{noshort}(e_1, \text{ff}) \land \text{noshort}(e_2, b))
\]

In words, 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 \( \text{tt} \), and the left sub-expression of every nested “or” evaluates to \( \text{ff} \).

We want to show that eval1 and eval2, when applied to the same bexpr \( e \) satisfying noshort\((e, b)\), have relative cost 0, and the result in any case is \( b \). Formally:

\[
\vdash \text{eval}_1 : \text{bexpr} \rightarrow C(\text{bool}) \sim \text{eval}_2 : \text{bexpr} \rightarrow C(\text{bool}) \mid \forall e_1, e_2. e_1 \equiv e_2 \Rightarrow \forall b. \text{noshort}(e_1, b) \\
\Rightarrow C_\Gamma(r_1 e_1, r_2 e_2, 0, r_1, r_2, r_1 \equiv r_2 \equiv b)
\]

The proof is mostly synchronous and follows the structure of eval1 and eval2. It starts by applying the \( R^C \) rules \( R\text{-LETREC} \) and \( R\text{-MATCH} \). The latter requires considering 9 cases (all possible pairs of bexpr constructors), but because \( e_1 \equiv e_2 \) is assumed, we immediately reduce to only 3 cases, where the constructors on the two sides are the same. In the case \( e_1 \equiv e_2 \equiv \text{const}(b') \), we get from the assumption noshort\((e_1, b)\) that \( b \equiv b' \). We need to show \( \vdash \{b' \} : C(\text{bool}) \sim \{b' \} : C(\text{bool}) \mid C_\Gamma(r_1, r_2, 0, r_1, r_2, r_1 \equiv r_2 \equiv b) \), which follows immediately by applying the rules \( R\text{-MONAD} \) and \( R\text{-RET} \) and then switching to \( L^C \) to show \( b \equiv b \equiv b' \).
In the case \( e_1 \neq e_2 \neq \text{and}(e', e'') \), we have \( \text{noshort}(e', \text{tt}) \) and \( \text{noshort}(e'', b) \). By the IH on \( e' \) followed by Theorem 5.1, we get \( \vdash f_1 \ e' : \wp(\text{bool}) \sim f_2 \ e' : \wp(\text{bool}) | C_f(r_1, r_2, 0, x_1, x_2, x_1 \neq x_2 \neq \text{tt}) \).

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 \( \vdash (y_1 \leftarrow f_1 \ e''; \ ↑1 \ y_1) \sim (y_2 \leftarrow f_2 \ e''; \ ↑1 \ y_2) \sim \text{bool} | 0 \ | r_1 \neq r_2 \neq b \). This follows by the IH on \( e'' \). The case \( e_1 \neq e_2 \neq \text{or}(e', e'') \) 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, \( \text{length}_1 \) and \( \text{length}_2 \), of the list length function. \( \text{length}_1 \) is tail-recursive, and uses a helper function \( \text{length}_h \), while \( \text{length}_2 \) is the standard recursive implementation.

\[
\begin{align*}
\text{length}_h & \triangleq \text{rec}_f(\ell_1). \lambda n. \text{match} \ell_1 \text{ with nil} \Rightarrow \{ n \}; \text{cons} \Rightarrow \lambda \ldots, t_1. \{ x_1 \leftarrow f_1 \ t_1 \ (n + 1); x_1 \} \\
\text{length}_1 & \triangleq \lambda \ell. \text{length}_h \ell \ 0 \\
\text{length}_2 & \triangleq \text{rec}_f(\ell_2). \text{match} \ell_2 \text{ with nil} \Rightarrow \{ 0 \}; \text{cons} \Rightarrow \lambda \ldots, t_2. \{ x_2 \leftarrow f_2 \ t_2; \ ↑1 \ x_2 + 1 \}
\end{align*}
\]

\( \text{length}_2 \) incurs a unit cost on every recursive call, while there is no such cost in \( \text{length}_h \); the intent is to model the number of allocated stack-frames. We want to show that the relative cost of \( \text{length}_1 \) and \( \text{length}_2 \) 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:

\[
\forall \ell. \text{Len}(\ell, 0) \Leftrightarrow \ell \triangleq \text{nil} \\
\forall \ell, n. \text{Len}(\ell, n + 1) \Leftrightarrow \exists h. t. \ell \triangleq \text{cons}(h, t) \land \text{Len}(t, n)
\]

Then, formally, we want to show:

\[
\vdash \text{length}_1 : \text{list} \rightarrow \wp(\text{N}) \sim \text{length}_2 : \text{list} \rightarrow \wp(\text{N}) | \forall \ell_1, \ell_2. \ell_1 \neq \ell_2 \Rightarrow \forall m. \text{Len}(\ell_1, m) \\
\Rightarrow C_m(r_1, \ell_1, r_2, \ell_2, -m, r_1, r_2, r_1 \neq r_2 \neq 1 \neq m + n)
\]

The cost part of this property means that the cost of \( \text{length}_1 \) minus the cost of \( \text{length}_2 \) is upper-bounded by \( -m \) or, equivalently, the cost of \( \text{length}_2 \) is lower-bounded by the cost of \( \text{length}_1 \) plus \( m \) (where \( m \) is the length of the input list). Since \( \text{length}_1 \) merely calls \( \text{length}_h \) with second argument 0, this can be easily reduced to showing:

\[
\vdash \text{length}_h : \text{list} \rightarrow \text{N} \rightarrow \wp(\text{N}) \sim \text{length}_2 : \text{list} \rightarrow \wp(\text{N}) | \forall \ell_1, \ell_2, n. \ell_1 \neq \ell_2 \Rightarrow \forall m. \text{Len}(\ell_1, m) \\
\Rightarrow C_m(r_1, \ell_1, n, r_2, \ell_2, -m, r_1, r_2, r_1 \neq r_2 + n \neq m + n)
\]

Note that \( \text{length}_h \) and \( \text{length}_2 \) have different types. We first apply the \( R^C \) rule \( R\text{-LETREC} \), and then the one-sided rule for \( \lambda \) (since \( \text{length}_h \) has an extra \( \lambda \)). The latter rule is similar to \( R\text{-LETREC}-\lambda \), but without the IH. Next, we apply the two-sided rule \( R\text{-MATCH} \). Since \( \ell_1 \neq \ell_2 \) is assumed, we get two cases in the proof. In the case \( \ell_1 \neq \ell_2 \triangleq \text{nil} \), \( \text{Len}(\ell_1, m) \) forces \( m \neq 0 \). We need to show \( \vdash \{ n \} : \wp(\text{N}) \sim \{ 0 \} : \wp(\text{N}) | C_m(r_1, r_1, 0, r_1, r_2, r_1 \neq r_2 + n \neq 0 + n) \), which follows from the rules \( R\text{-MONAD}, R\text{-RET} \) and reasoning in \( L^C \).

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

\[
\vdash f_1 \ t_1 \ (n + 1) : \wp(\text{N}) \sim f_2 \ t_2 : \wp(\text{N}) | C_m(r_1, r_2, -m', x_1, x_2, x_1 \neq x_2 + (n + 1) \equiv m' + (n + 1))
\]

Hence, by the rule \( R\text{-BIND} \), it remains to show (under the assumption \( x_1 \neq x_2 + (n + 1) \equiv m' + (n + 1) \)):

\[
\vdash x_1 \div N \sim ↑1 x_2 + 1 \div N | -1 \ | r_1 \neq r_2 + n \neq m + n
\]

At this point, we apply the one-sided rule \( R\text{-STEP-R} \) (which is completely analogous to \( R\text{-STEP-L} \)) to reduce the goal to \( \vdash x_1 \div N \sim x_2 + 1 \div N | 0 \ | r_1 \neq r_2 + n \neq m + n \). This follows immediately by the rule \( R\text{-RET} \) and reasoning in \( L^C \).
Insertion sort. We perform a precise relational analysis of insertion sort. Insertion sort calls the \textit{insert} function for which we proved a \textit{unary} property earlier. We use that property now. Hence, this example demonstrates an interaction between relational and unary analysis in R\textsuperscript{C}. The analysis also relies on nontrivial functional properties of insertion sort itself.

The insertion sort function, \textit{isort}, is defined below:

\[
\text{isort} \triangleq \text{rec } f (\ell). \text{match } \ell \text{ with } \text{nil} \mapsto \text{nil}; \text{cons} \mapsto \lambda h. t. \{ t' \leftarrow f \ t; z \leftarrow \text{insert } h \ t' \}; \uparrow \}
\]

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

\[\forall \ell_1, \ell_2, n. \text{UnsortedDiff}(\ell_1, \ell_2, n) \equiv \exists u_1, u_2, m. \text{Unsorted}(\ell_1, u_1) \land \text{Unsorted}(\ell_2, u_2) \land n = u_1 - u_2 \land \text{Len}(\ell_1, m) \land \text{Len}(\ell_2, m)\]

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

\[\vdash \text{isort} : \text{list} \rightarrow \mathbb{C}(\text{list}) \sim \text{isort} : \text{list} \rightarrow \mathbb{C}(\text{list}) \mid \forall \ell_1, \ell_2, n. \text{UnsortedDiff}(\ell_1, \ell_2, n) \Rightarrow \mathbb{C}_R(\ell_1, \ell_2, n, t_1, t_2, \phi)\]

where \( \phi \triangleq \text{Sorted}(r_1) \land \text{Sorted}(r_2) \land (\forall y, q. \text{LargerThan}(y, \ell_1, q) \Rightarrow \text{LargerThan}(y, r_1, q)) \land (\forall y, q. \text{LargerThan}(y, \ell_2, q) \Rightarrow \text{LargerThan}(y, r_2, q))\)

The proof starts by applying the R\textsuperscript{C} rule R\textsuperscript{LETrec}, which introduces the induction hypothesis, and then R\textsuperscript{MATCH}, which causes a case analysis on the input lists. Since the lists have the same length (by assumption 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, \( \ell_1 \models \text{cons}(h_1, t_1) \) and \( \ell_2 \models \text{cons}(h_2, t_2) \). The definition of UnsortedDiff yields \( u_1, u_2 \) such that \( n = u_1 - u_2 \). Unsorted(\ell_1, u_1) and Unsorted(\ell_2, u_2). The definition of Unsorted now yields \( u_1 \triangleq n_1' + u_1', u_2 \triangleq n_2' + u_2' \), LargerThan(h_1, t_1, n_1'), Unsorted(t_1, u'_1), LargerThan(h_2, t_2, n_2'), and Unsorted(t_2, u'_2) for some \( u'_1, n'_1, u'_2, n'_2 \). Further, we also have UnsortedDiff(t_1, t_2, n), s.t. \( n'' \triangleq u'_1 - u'_2 \), and Len(t_1, m') and Len(t_2, m'), where \( m \equiv m' + 1 \). Hence, from the IH we get:

\[\vdash \text{f}_1 t_1 : \mathbb{C}(\text{list}) \sim f_2 t_2 : \mathbb{C}(\text{list}) \mid \mathbb{C}_R(r_1, r_2, n', t'_1, t'_2, \phi')\]

for \( \phi' \triangleq \phi[t_1/\ell_1][t_2/\ell_2][t'_1/r_1][t'_2/r_2] \). From this \( \phi' \) we have Sorted(t'_1) and LargerThan(h_1, t'_1, n'_1). Applying these to the unary property we proved for \textit{insert} earlier, we get:

\[\vdash \text{insert } h_1 t'_1 : \mathbb{C}(\text{list}) \mid \mathbb{C}_U(r, n'_1, z, \text{Sorted}(z) \land \forall y, q. \text{LargerThan}(y, \text{cons}(h_1, t'_1), q) \Rightarrow \text{LargerThan}(y, z, q))\]

and a similar property for \textit{insert} h_2 t'_2. Next, we combine these two unary properties of \textit{insert} into a relational property using the admissible R\textsuperscript{C} rule R\textsuperscript{SPLIT}:

\[\vdash \text{insert } h_1 t'_1 : \mathbb{C}(\text{list}) \sim \text{insert } h_2 t'_2 : \mathbb{C}(\text{list}) \mid \mathbb{C}_R(r_1, r_2, n'_1 - n'_2, z_1, z_2, \phi'')\]

where \( \phi'' \triangleq \text{Sorted}(z_1) \land \text{Sorted}(z_2) \land \forall y, q. \text{LargerThan}(y, \text{cons}(h_1, t'_1), q) \Rightarrow \text{LargerThan}(y, z_1, q) \land \forall y, q. \text{LargerThan}(y, \text{cons}(h_1, t'_2), q) \Rightarrow \text{LargerThan}(y, z_2, q)\).

Note that the additional property \( \phi' \) obtained for (a recursive call to) \textit{isort} enabled us to derive the results for \textit{insert}. Without this, we would not have been able to derive Sorted(t'_1) and LargerThan(h_1, t'_i, n'_i) from Sorted(t_i) and LargerThan(h_1, t_i, n_i) (for \( i \in \{1, 2\} \)). In turn, the additional property \( \phi'' \) of \textit{insert} obtained above is required to show \( \phi \).
Monadic refinements for relational cost analysis  

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

7 Embedding of RelCost

Next, we show that \( R^C / L^C \) 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 \( U^C / R^C \), 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 \( \tau \) (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 \( \ell \) 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 \( \text{int}_\tau \) 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 \( \alpha \) 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 \( \tau \). 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 \( (\Delta; \Phi; \Omega \vdash e : A) \) and one relational \( (\Delta; \Phi; \Gamma \vdash e_1 \otimes e_2 \lesssim n : \tau) \). In both judgments, \( \Delta \) is an environment for index variables, and \( \Phi \) are assumed constraints over the index variables. The unary judgment states that, under a unary typing environment \( \Omega \), \( e \) has type \( A \), and its execution cost is lower- and upper-bounded by \( k \) and \( \ell \) respectively. The relational judgment states that, under a relational typing environment \( \Gamma \), expressions \( e_1 \) and \( e_2 \) have the relational type \( \tau \), and their relative cost is at most \( n \). The function \( \tau \) 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 \( e_{\text{app}} \) 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 \( |A_1| \rightarrow C(|A_2|) \). We assume that our language has a type unit and a family of list types, \( \text{list}_\sigma \), 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 \( C(|A|) \). To capture
Unary typing judgment

\[ \frac{\Delta; \Phi; \Omega \vdash e : A}{\Delta; \Phi; \Omega \vdash x : A} \]

Relational typing judgment

\[ \frac{\Gamma(x) = \tau}{\Delta; \Phi; \Gamma \vdash \overline{x} \otimes \overline{x} \subseteq 0 : \tau} \]

\[ \Delta; \Phi; \Gamma \vdash e_1 \otimes e_2 \subseteq n : \tau \]

\[ \quad \quad \forall x \in \text{dom}(\Gamma). \Delta; \Phi \vdash \Gamma(x) \subseteq \Box \Gamma(x) \]

We denote this translation \( \langle \cdot \rangle \). Representative clauses are shown in Figure 8.

**Unary refinements.** To capture RelCost’s unary refinements, we define a translation \( [\cdot] \) from unary types into \( L^C \) assertions. \( [A] \) 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 \( \text{list} e \) has length \( n \). The interesting clause is that for function types \( A_1 \xrightarrow{\text{exec}(k, \ell)} A_2 \), where the cost bounds \( k, \ell \) are represented through a monadic refinement \( C_U(x, y, k, \ell, r, \_\_) \).

**Relational refinements.** Like unary refinements, we capture relational refinements by a translation \( \lfloor \cdot \rfloor \) from relational types to \( L^C \) assertions. \( \lfloor \tau \rfloor \) is a binary relation on expressions that holds at \( (e_1, e_2) \) when this pair of expressions satisfies the relational refinement inherent in \( \tau \). 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

\[
\begin{array}{c}
\| \text{int} \| \triangleq \| \text{int}_r \| \triangleq \mathbb{Z} \\
\| \text{list}[n] \ A \| \triangleq \| \text{list}_{\{A\}} \| \\
\| \text{list}[n]^\alpha \ \tau \| \triangleq \| \text{list}_{\{\tau\}} \|
\end{array}
\]

\[
\begin{array}{c}
|A_1 \xrightarrow{\text{exec}(k, \ell)} A_2| \triangleq |A_1| \rightarrow C(|A_2|) \\
\| \tau_1 \xrightarrow{\text{diff}(n)} \tau_2\| \triangleq \| \tau_1\| \rightarrow C(|\tau_2|) \\
|UA| \triangleq |A|
\end{array}
\]

\[
\begin{array}{c}
\forall \ell \cdot \text{listU}_A(\ell, 0) \Leftrightarrow \ell \equiv \text{nil} \\
\forall \ell, n \cdot \text{listU}_A(\ell, n + 1) \Leftrightarrow \exists h, t, \ell \equiv \text{cons}(h, t) \land |A|(h) \land \text{listU}_A(t, n) \\
|\text{int}(x)| \triangleq \top \\
|\text{list}[n] \ A(x)| \triangleq \text{listU}_A(x, n) \\
|A_1 \xrightarrow{\text{exec}(k, \ell)} A_2|(x) \triangleq \forall y \cdot |A_1|(y) \Rightarrow C_u(x \ y \ k \ \ell \ \tau, |A_2|)(r)) \\
|x_1 : A_1, \ldots, x_n : A_n| \triangleq |A_1|(x_1), \ldots, |A_n|(x_n)
\end{array}
\]

Translation of relational refinements (selected clauses)

\[
\begin{array}{c}
\forall \ell_1, \ell_2, a \cdot \text{listR}(\ell_1, \ell_2, 0, a) \Leftrightarrow \ell_1 \equiv \ell_2 \equiv \text{nil} \\
\forall \ell_1, \ell_2, n, a \cdot \text{listR}(\ell_1, \ell_2, n + 1, a) \Leftrightarrow \exists h_1, h_2, t_1, t_2, \ell_1 \equiv \left( \text{cons}(h_1, t_1) \land \ell_2 \equiv \text{cons}(h_2, t_2) \land \| \tau \|(h_1, h_2) \land \right. \\
\left. \left( \text{listR}(t_1, t_2, n, a) \lor \left( a > 0 \land \exists b \cdot a = b + 1 \land \text{listR}(t_1, t_2, n, b) \right) \right) \right)
\end{array}
\]

\[
\begin{array}{c}
\| \text{int}_r \|\ (x, y) \triangleq x \equiv y \\
\| \text{list}[n]^{\alpha} \ \tau\|\ (x, y) \triangleq \text{listR}_r(x, y, n, \alpha)
\end{array}
\]

\[
\begin{array}{c}
\| \tau_1 \xrightarrow{\text{diff}(n)} \tau_2\|\ (x, y) \triangleq \| \tau_1 \rightarrow \tau_2 \|\ (x, y) \land \\
\left( \forall x_1, x_2 \cdot \| \tau_1\|\ (x_1, x_2) \Rightarrow C_r(x \ x_1 \ y \ x_2, n, \tau_1, \tau_2, \| \tau_2\|)(r_1, r_2) \right)
\end{array}
\]

\[
\begin{array}{c}
|UA|(x, y) \triangleq |A|(x) \land |A|(y) \\
\| \tau\|\ (x, y) \triangleq x \equiv y \land \| \tau\|\ (x, y) \\
|x_1 : \tau_1, \ldots, x_n : \tau_n| \triangleq \| \tau_1\|\ (x_1, x_1^2), \ldots, \| \tau_n\|\ (x_n^1, x_n^2)
\end{array}
\]

Fig. 8. Embedding of RelCost in U^C/R^C
amortically. The predicate means that the lists \( e_1, e_2 \) each have length \( n \) and their Hamming distance is at most \( a \).

The translations \( \| \cdot \|, \| \cdot \|, [\cdot] \) 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 \( \langle e \rangle \) in our language are equal.

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

1. If \( \Delta ; \Phi; \Omega \vdash e : A \) in RelCost, then \( \| \Omega \|, \Delta ; \Phi; \| \Omega \| \vdash \langle e \rangle : C(\| A \|) \) \( \|
\begin{array}{c}
C_L(\| k \|, \| f \|, \| \cdot \|) ) \in U^C.
\end{array}
\)

2. If \( \Delta ; \Phi; \Gamma \vdash e_1 \otimes e_2 \subseteq n : \tau \) in RelCost, then \( \| \tau \|, \Delta ; \Phi; \| \tau \| \vdash \langle e_1 \rangle_1 : \| \tau \| \) \( \|
\begin{array}{c}
C_R(\tau_1, \tau_2, n, \tau_1 . \tau_2, \| \tau \|) ) \) in \( R^C \), where \( \langle e_1 \rangle_1 \) is a copy of \( \langle e_1 \rangle \) 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 variables 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 \( F := A_1 \to A_2 \to A_2 \), 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 RAME 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.
but they can be duplicated explicitly using the rule of an affine sum of the potentials of values in the range of \( \gamma \) for any closing substitution \( \gamma \).

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 \( L_{\text{Share}} \). Such duplication splits the potential of...
we axiomatically define a predicate

\[ \Phi_A(e, p) \] that encodes potentials

\[
\forall x, p. \Phi_{\text{int}}(x, p) \iff p = 0 \\
\forall x, p. \Phi_{L^q(A)}(x, p) \iff (3h, t, p_h, p_t \cdot x \equiv \text{cons}(h, t) \land \Phi_A(h, p_h) \\
\land \Phi_{L^q(A)}(t, p_t) \land p \equiv q + p_h + p_t)
\]

Translations for contexts

| \[ x_1 : A_1, \ldots, x_n : A_n \] | \( \equiv \) | \[ x_1 : |A_1|, x_1^p : \mathbb{R}^\infty, \ldots, x_n : |A_n|, x_n^p : \mathbb{R}^\infty \] |
| \[ \Sigma \] | \( \equiv \) | \{ f : |F| \mid \text{for all } f \in \text{dom}(\Sigma) \text{ and all } F \in \Sigma(f) \} |

Predicate over function types, that relates potentials to costs

\[
[A_1 \xrightarrow{q/q'} A_2](f) \equiv \forall y : |A_1|. \top \Rightarrow \forall y^p : \mathbb{R}^+. \Phi_{A_1}(y, y^p) \Rightarrow \\
\exists p_r. \mathbb{C}_U(f, y, 0, y^p + q - q' - p_r, r.\Phi_{A_1}(r, p_r))
\]

Fig. 10. Embedding of RAML in \( U^C \)

the duplicated variable among the duplicates (using the relation \( \gamma \)). 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 \(|\cdot|\) 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 \( \Phi_A(e, p) \) 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 \( \Phi \) 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 \( \Phi_A(x, x^p) \). Finally, we define a
predicate $[A_1 \overset{q/q'}{\rightarrow} A_2](f)$ 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 $\Sigma; \Gamma \vdash q e : A$ in RAML with additive costs only, then $[\Sigma, [\Gamma]; [\Sigma, [\Gamma] + (e) : C(A)] \vdash_\Phi \Phi_\Gamma(r, 0, \Phi_\Gamma) + q - q' - p_r, r, \Phi_A(r, p_r))$ in $U^C$, where $\Phi_\Gamma \triangleq \sum_{x \in \text{dom}(\Gamma)} x^p$ 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 $(\Phi_\Gamma + q)$ 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 nat $= 0() + S(\text{nat})$) by iterating $incg$ $k$ times.

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

$$\text{incg} : \text{list} \rightarrow C(\text{list})$$

$$\text{incg} \triangleq \text{rec } f(\ell). \text{match } \ell \text{ with}$$

$$\text{nil} \mapsto \{\text{nil}\}; \quad \text{cons} \mapsto \lambda x, t. \text{ if } x < D - 1 \text{ then } \{\uparrow^1(x + 1) :: t\} \text{ else } \{t' \leftarrow f t; \uparrow^1 0 :: t'\}$$

$$\text{rinc} : \text{nat} \rightarrow \text{list} \rightarrow C(\text{list})$$

$$\text{rinc} \triangleq \text{rec } f(k). \lambda \ell. \text{match } k \text{ with } 0 \mapsto \{\ell\}; \quad S \mapsto \lambda k'. \{\ell' \leftarrow \text{incg } \ell; \ell'' \leftarrow f k' \ell'; \ell''\}$$

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 $\frac{D}{D-1}$ 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 $\ell$ has total potential $n$:

$$\Phi(\ell, n) \iff (\ell \equiv n \land \ell \equiv n \lor \exists t, i, n'. \ell \equiv \text{cons}(i, t) \land \Phi(t, n') \land n \equiv n' + \frac{l}{D-1} \land i \leq D - 1)$$

$^8$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 $\text{Three} = \text{zero} \mid \text{one} \mid \text{two}$ 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 $\mathbb{R}^\omega$).

\[
\begin{align*}
& \vdash \text{inc} : \text{list} \to C(\text{list}) \mid \forall \ell, n. \Phi(\ell, n) \Rightarrow \exists n'. \ C_\ell (r \ell, 0, n + \frac{D}{D-1} - n', r \Phi(r, n')) \\
& \vdash rinc : \text{nat} \to \text{list} \to C(\text{list}) \mid \forall k, n, \ell. \Phi(\ell, n) \Rightarrow \exists n'. \ C_\ell (r k \ell, 0, \frac{D}{D-1} k + n - n', r \Phi(r, n'))
\end{align*}
\]

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 $\ell \triangleq x :: t$ (the case $\ell \triangleq \text{nil}$ is trivial). When $x < D - 1$, the potential of $\ell$ 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 $x \geq D - 1$, the potential of $\ell$ 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 $\mathbb{R}^\omega$ 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 $\mathbb{U}^c$ can be generalized to reason about unary non-additive costs by noting that its rules work for any monoid, not just $(\mathbb{R}^\omega, 0, +)$. In particular, it should be feasible to replace $(\mathbb{R}^\omega, 0, +)$ 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 \(\text{U-SUBM1}\)), and in the definition of \(\mathcal{C}_U\). This relation can be defined as \((q, q') \leq (p, p') \triangleq (q \leq p) \land (q - q') \leq (p - p')\). 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 \(+, -\)), 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 \(\mathbb{S}(\tau) \triangleq S \rightarrow (\tau \times S)\), where \(S\) is the type of the mutable state. The standard Hoare triple \(\{\Theta\}e\{\Theta'(x)\}\) could be represented as the logical refinement assertion \(\forall s. \Theta(s) \Rightarrow \Theta'(\pi_1(e\ s))\). Here, \(x\) (of type \(\tau\)) represents the expression returned by the monadic computation, and \(\Theta\) 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 \(\Theta\) and \(\Theta'\) 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 \(\Theta\) and \(\Theta'\) 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 \(\mathbb{S}(\tau)\) that includes both state and cost. Briefly, we could define \(\mathbb{S}(\tau) \triangleq S \rightarrow \mathbb{C}(\tau \times S)\)—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 \(\forall s. p. (\Theta(s) \land P(s) \Rightarrow q) \Rightarrow \exists q. \mathcal{C}_U(e\ s, 0, p - q, x. (\Theta'(\pi_2(x)) \land Q(\pi_2(x)) \Rightarrow q))\) 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 \(\lambda\)-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. UC supports similar unary analysis over lazy data structures, and RC 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, some programs such as our insertion sort example can be quite difficult to verify with DML-like refinements, since DML’s refinements do not apply to arbitrary program expressions.

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 RC shows that RC is more expressive, but RC 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 bool = ff + 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.

Carbonneau 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 Carbonneau et al. [2015] can be ported to UC and RC 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 UC. 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 UC/RC.

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 RC. 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.

RC 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 RC (UC) 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 LC, UC and RC) and the proofs of soundness and completeness of UC/RC with respect to LC 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


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


