Bidirectional Type Checking for Relational Properties

Ezgi Çiçek*
Facebook, UK

Weihao Qu
University at Buffalo, SUNY, USA

Gilles Barthe
Max Planck Institute for Security and Privacy, Germany and IMDEA Software Institute, Spain

Marco Gaboardi
University at Buffalo, SUNY, USA

Deepak Garg
Max Planck Institute for Software Systems, Germany

Abstract

Relational type systems have been designed for several applications including information flow, differential privacy, and cost analysis. In order to achieve the best results, these systems often use *relational refinements* and *relational effects* to maximally exploit the similarity in the structure of the two programs being compared. Relational type systems are appealing for relational properties because they deliver simpler and more precise verification than what could be derived from typing the two programs separately. However, relational type systems do not yet achieve the practical appeal of their non-relational counterpart, in part because of the lack of a general foundation for implementing them.

In this paper, we take a step in this direction by developing *bidirectional relational* type checking for systems with relational refinements and effects. Our approach achieves the benefits of bidirectional type checking, in a relational setting. In particular, it significantly reduces the need for typing annotations through the combination of type checking and type inference. In order to highlight the foundational nature of our approach, we develop bidirectional versions of several relational type systems which incrementally combine many different components needed for expressive relational analysis.

CCS Concepts • **Theory of computation → Type structures**; *Functional constructs.*

Keywords Bidirectional type-checking, relational type systems, refinement types, type-and-effect systems

*Work done while Ezgi Çiçek was a PhD student at the Max Planck Institute for Software Systems

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314603

ACM Reference Format:
Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional Type Checking for Relational Properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314603

1 Introduction

Type systems are a fundamental tool for proving program properties. They draw their success from their ability to enforce many desirable facts about programs. Bidirectional type checking [37] is a very successful method of implementing type systems through a combination of type inference and type checking [3, 8, 10, 33, 35]. The appeal of bidirectional type checking lies in its ability to minimize typing annotations—in most cases, type annotations are needed only on recursive functions, or on reducible expressions—while supporting disciplines that are too expressive to fall under the purview of type inference. Furthermore, bidirectional type systems offer a formal framework based on rules that resemble standard typing rules. This simplifies proofs of soundness and completeness of the algorithmic implementation relative to the declarative type system.

Standard type systems are primarily focused on program properties, i.e. reasoning about individual execution traces. In contrast, relational type systems [1, 4–7, 13, 15, 27, 38] aim to repeat the success of type systems, but for so-called relational properties, which consider pairs of execution traces. Typical examples of relational properties include noninterference in information flow systems, continuity and robustness analysis of programs, differential privacy, and relational cost analysis. The key difference of relational type systems is that they consider two expressions simultaneously, and maximally exploit structural similarities between them to achieve simpler and more precise verification than would be possible with unary analysis of the individual expressions. Similarities are exploited through two main ingredients: relational refinement types and relational effects.

Relational refinements types [7, 13, 15, 27, 38] relate two executions of two expressions and are akin to standard refinement types [44]. However, their interpretation is a relation between the values in the two executions. For example, in information flow control, a relational refinement is used to describe equivalence between the values that are observable at a specific security level.

Relational effects [7, 13, 15, 27, 38] are often of a quantitative nature and measure some quantitative difference between two executions of the two expressions. These relational effects are similar in spirit to their standard unary counterpart [12, 30, 32, 34] but their interpretation is a relation between the effects of the two executions. For example, in differential privacy, a relational effect is used to measure the level of indistinguishability between the observable outputs on two inputs differing in one data element.

While a lot of work cited above comes with implemented type checkers, there is, so far, no common understanding of the challenges and solutions for implementing relational type systems. Hence, the broad goal of our work is to investigate issues in *implementing* a type checker for relational type systems with relational refinements and relational effects. Bidirectional type checking is a natural starting point for the reasons mentioned above, and because it has been used for implementing refinement type systems [20, 23, 44] and subtyping [37], which are important common features in most of the type systems we are inspired from. However, bidirectional type checking has not been extensively applied to effect systems, although some examples exist [41], and it has not been applied to relational type systems at all.

Our contribution We present a study of *bidirectional* type checking for *relational type and effect systems*. We start with the study of a basic relational type system, named relSTLC, that includes judgments to only relate two expressions with the same top-level structure, with types to represent related and non-related boolean values, no relational refinements and no relational effects. This can be seen as the relational analogue of the simply typed lambda calculus over a base type with subtyping. For this system, bidirectional type checking works as expected and it delivers a sound and complete algorithm implementing the declarative system.

Next, we extend relSTLC in two steps inspired by the features of previously proposed relational type systems. Our first step, named RelRef, adds *relational refinement types* over lists (as an example of an inductive data type), and a comonadic type that represents *syntactic equality* of values. Our second step, named RelRefU, adds to RelRef the possibility to *relate arbitrary programs* of possibly dissimilar syntactic structure, by switching to a complementary unary type system. Both these extensions add intrinsic *nondeterminism* to the type system to allow a programmer flexibility in writing programs. The source of nondeterminism in both these systems is non-syntax-directed typing and subtyping rules. RelRef has such rules for relational refinement types and for subtyping, while RelRefU has such a rule for switching to unary typing and more such rules for subtyping.

To overcome the challenges introduced by nondeterminism, we introduce a two-step methodology. We first show that every well-typed program can be translated to a well-typed program in a core language with term constructors that resolve the nondeterminism. This translation is type derivation-directed; it introduces annotations to resolve the nondeterminism in applying the (non-syntax-directed) typing rules and eliminates relational subtyping by replacing it with explicit coercions defined within the core language. Next, we develop a bidirectional type system and prove it sound and complete with respect to the core system. It follows that every typeable program can be annotated to remove nondeterminism, and then bidirectionally type checked. Although this indirection via a core language does not directly lead to an implementation strategy, it makes a strong theoretical point, namely that the bidirectional type checking is complete modulo nondeterminism. We show that this methodology is applicable to both RelRef and RelRefU.

Our final step is to add *relational effects* to RelRefU. Specifically, we consider the type system RelCost [13]. This type system extends RelRefU with a relational effect to enable relational cost analysis. The objective of relational cost analysis is to establish a static upper bound on the cost of a program relative to another program: For two programs e_1 and e_2, relational cost analysis establishes an upper bound t such that $cost(e_1) - cost(e_2) \leq t$. t is called the relative cost of e_1 and e_2. It is described as a relational effect in the type system. Since RelCost extends RelRefU, it inherits the latter's many sources of nondeterminism. We resolve these using the same two-step approach that we described above, thus showing that the approach also extends to relational effects.

To show the effectiveness of bidirectional type checking for relational type systems, we have implemented a prototype for RelCost. (This prototype can also be used for the other type systems we describe, since RelCost extends them conservatively.) Our implementation handles the two steps of our approach simultaneously. To implement the first step, the translation of the source language to the core language, we rely on example-guided heuristics rather than programmer-specified annotations to eliminate nondeterminism. This reduces the programmer's annotation burden by compromising completeness to some degree. We explain these heuristics and our evaluation shows that they are effective for a large class of examples. For the second step, we implement the bidirectional typing rules. Both type checking and type inference generate constraints that capture arithmetic relationships between refinements (e.g., list sizes) of various subterms, relational refinements and relationships between unary and relational costs. Our constraints contain existentially quantified variables over integers and reals. Therefore, we design our own algorithm to eliminate existential variables by finding substitutions for them and use SMT solvers to discharge the substituted constraints.

Summing up, our contributions are:

- We present several bidirectional relational type systems that combine relational and non-relational typing, refinements, and unary and relational effects.
- We present a type-preserving, complete embedding of programs that are typeable in those systems into core type systems. This embedding eliminates nondeterminism in applying typing rules and eliminates relational subtyping. We use the embedding to theoretically argue that, modulo the nondeterminism, our bidirectional type checking is complete.
- We present an implementation of the largest of the type systems we consider (RelCost), using heuristics to get rid of the inherent non-determinism. We use the implementation to type-check several examples, including all examples from the original RelCost paper.

The rest of the paper is organized as follows. We start in Section 2 with relSTLC, our basic relational simply-typed calculus. In Sections 3 and 4, we extend it to RelRef and RelRefU. In Section 5, we add effects, finally reaching RelCost. In each of these sections, we describe a declarative type system, its bidirectional version and, where necessary, a core calculus that resolves nondeterminism of the declarative type system. In Section 6, we describe our implementation, heuristics to eliminate nondeterminism, and experimental results. Section 7 comments on extending the bidirectional approach to other features of relational type systems that are not considered explicitly in this paper. Section 8 presents related work. A supplementary online appendix available from the authors' homepages contains technical details omitted from this paper due to lack of space. All the code and examples can be found in a publicly accessible repository at https://github.com/ezgicicek/BiRelCost.

2 Relational STLC (relSTLC)

As an introduction to how relational reasoning works, we consider relSTLC, a rehash of the simply-typed lambda calculus (STLC) with relational reasoning. relSTLC has the following type and expression grammar:

Types $\tau ::= \text{bool}_r \mid \text{bool}_u \mid \tau_1 \to \tau_2$
Expr $e ::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \lambda x.e \mid e_1\ e_2$

A type τ is interpreted as a set of pairs of values. For instance, the primitive type bool_r ascribes pairs of *identical* booleans (the diagonal relation on booleans) whereas the type bool_u ascribes pairs of arbitrary booleans (the complete relation on booleans). In particular, $\text{bool}_r \sqsubseteq \text{bool}_u$. The function type $\tau_1 \to \tau_2$ relates pairs of functions that, given a pair of related arguments of type τ_1, return a pair of related computations of type τ_2. Even though relSTLC is quite primitive, it forms the basis of our development and we find it instructive to discuss challenges in its algorithmization.

Declarative typing The typing judgment $\Gamma \vdash e_1 \backsim e_2 : \tau$ ascribes the expressions e_1 and e_2 the relational type τ under the environment Γ. The typing rules and subtyping rules are standard. A selection is shown in Figure 1. Note how the rule **r-bool** relates two identical booleans at type bool_r, while **r-u-bool** relates two arbitrary booleans at type bool_u. This difference manifests in the rule **r-if**: If the branch condition of an if-then-else has type bool_r, then we only need to type the two "then" branches and the two "else" branches separately, but do not need to type a "then" and an "else" branch together. Finally, note that the calculus has (standard) subtyping induced by the base relation $\text{bool}_r \sqsubseteq \text{bool}_u$.

$\boxed{\Gamma \vdash e_1 \backsim e_2 : \tau}$

$$\frac{\mathsf{b} \in \{\text{true}, \text{false}\}}{\Gamma \vdash \mathsf{b} \backsim \mathsf{b} : \text{bool}_r}\ \textbf{r-bool}$$

$$\frac{\mathsf{b}_1, \mathsf{b}_2 \in \{\text{true}, \text{false}\}}{\Gamma \vdash \mathsf{b}_1 \backsim \mathsf{b}_2 : \text{bool}_u}\ \textbf{r-u-bool}$$

$$\frac{\Gamma \vdash e \backsim e' : \text{bool}_r \qquad \Gamma \vdash e_1 \backsim e_1' : \tau \qquad \Gamma \vdash e_2 \backsim e_2' : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \backsim \text{if } e' \text{ then } e_1' \text{ else } e_2' : \tau}\ \textbf{r-if}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_1 \backsim e_2 : \tau_2}{\Gamma \vdash \lambda x.e_1 \backsim \lambda x.e_2 : \tau_1 \to \tau_2}\ \textbf{r-lam}$$

$$\frac{\Gamma \vdash e_1 \backsim e_1' : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \backsim e_2' : \tau_1}{\Gamma \vdash e_1\ e_2 \backsim e_1'\ e_2' : \tau_2}\ \textbf{r-app}$$

$$\frac{\Gamma \vdash e_1 \backsim e_2 : \tau \qquad \models \tau \sqsubseteq \tau'}{\Gamma \vdash e_1 \backsim e_2 : \tau'}\ \textbf{r-}\sqsubseteq$$

$\boxed{\models \tau \sqsubseteq \tau'}$

$$\frac{}{\models \text{bool}_r \sqsubseteq \text{bool}_u}\ \textbf{bool} \qquad \frac{\models \tau_1' \sqsubseteq \tau_1 \qquad \models \tau_2 \sqsubseteq \tau_2'}{\models \tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'}\ \to$$

$$\frac{}{\models \tau \sqsubseteq \tau}\ \textbf{refl} \qquad \frac{\models \tau_1 \sqsubseteq \tau_2 \qquad \models \tau_2 \sqsubseteq \tau_3}{\models \tau_1 \sqsubseteq \tau_3}\ \textbf{trans}$$

Figure 1. relSTLC typing and subtyping rules

Algorithmic (bidirectional) typing The type system presented above is declarative, i.e. it doesn't prescribe an algorithm for building a typing derivation. In fact, two aspects of relSTLC make it difficult to straightforwardly algorithmize. First, the obvious approach of *inferring* the type of a term bottom-up by starting at the leaves (variables) does not work because relSTLC does not have type annotations on variable-bindings. The other obvious approach of *checking* a term against a given type top-down also runs into a problem, this time in the **r-app** rule, where the argument type τ_1 must be guessed. Second, the **trans** subtyping rule is also not syntax-directed (the type τ_2 must be guessed). Hence, the typing and subtyping rules of relSTLC cannot be directly interpreted as a typechecking algorithm.

A well-established way of making typing rules syntax-directed (hence obtaining a typechecking algorithm) is to make the rules *bidirectional* [37, 43, 44]. In comparison to fully type-annotating all bound variables, which could be tedious for a programmer, the main idea behind bidirectional typechecking is to only annotate programs at the top-level and at explicit β-redexes (which are usually rare) and infer all other types by combining top-down and bottom-up analysis.

In the case of relSTLC, bidirectional typechecking splits the usual typing judgment $\Gamma \vdash e_1 \backsim e_2 : \tau$ into two judgments: (1) the checking judgment $\Gamma \vdash e_1 \backsim e_2 \downarrow \tau$, where the type τ is an input (the type is checked), and (2) the inference judgment $\Gamma \vdash e_1 \backsim e_2 \uparrow \tau$, where the type τ is an output (the type is inferred). As a convention, we write all outputs in red and all inputs in black. Figure 2 shows selected algorithmic typing rules. We explain below the basic principles behind the bidirectional typing rules. These principles are completely standard for *unary* type systems [43, 44]; our observation thus far is simply that they apply as-is to relational type systems as well and, for relSTLC, they suffice to ensure completeness of bidirectional typechecking (this will cease to be the case for later type systems).

- Types of introduction forms are checked (e.g., rule **alg-r-lam**). Types of elimination forms are, in general, inferred (e.g., rule **alg-r-app**). However, types of case-like elimination forms (e.g., rule **alg-r-if**) are checked. In all cases, the type of an expression in an elimination position is inferred (e.g., the branch condition $e \backsim e'$ in rule **alg-r-if**). Types of variables and constants can always be inferred. However, here, we treat constants as constructors (introduction forms), so their types are checked (this is an arbitrary design choice).
- In checking mode, the rule **alg-r-**$\uparrow\downarrow$ allows switching to inference mode. The requirement is that the inferred type must be a subtype of the checked type.
- In inference mode, it is permissible to switch to the checking mode when an expression's type has been explicitly annotated by the programmer (rule **alg-r-anno-**$\uparrow$). It can be shown that, for completeness, it suffices to annotate only at explicit β-redexes (although there is no prohibition on annotating at other places).

Subtyping also has an algorithmic counterpart, $\models \tau_1 \leq \tau_2$, shown in Figure 2. We introduce two additional rules for reflexivity of base types (rules **alg-bl-u** and **alg-bl-r**). Importantly, it can be proved that reflexivity and transitivity of subtyping are *admissible*, so, in particular, there is no need for an explicit rule of transitivity, which, as mentioned, is difficult to use in an algorithm.

The bidirectional type system's rules, when read bottom-up, can be interpreted as a syntax-directed algorithm for typechecking. This algorithm is sound relative to the declarative type system in the following sense: If either $\Gamma \vdash e_1 \backsim e_2 \uparrow \tau$ or $\Gamma \vdash e_1 \backsim e_2 \downarrow \tau$, then $\Gamma \vdash |e_1| \backsim |e_2| : \tau$, where $|e|$ is obtained

$$\boxed{\Gamma \vdash e_1 \backsim e_2 \uparrow \tau, \Gamma \vdash e_1 \backsim e_2 \downarrow \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \backsim e' \uparrow \mathsf{bool}_r \\ \Gamma \vdash e_1 \backsim e_1' \downarrow \tau \qquad \Gamma \vdash e_2 \backsim e_2' \downarrow \tau\end{array}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 \backsim \text{if } e' \text{ then } e_1' \text{ else } e_2' \downarrow \tau}\ \textbf{alg-r-if}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_1 \backsim e_2 \downarrow \tau_2}{\Gamma \vdash \lambda x.e_1 \backsim \lambda x.e_2 \downarrow \tau_1 \to \tau_2}\ \textbf{alg-r-lam}$$

$$\frac{\Gamma \vdash e_1 \backsim e_1' \uparrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \backsim e_2' \downarrow \tau_1}{\Gamma \vdash e_1\ e_2 \backsim e_1'\ e_2' \uparrow \tau_2}\ \textbf{alg-r-app}$$

$$\frac{\Gamma \vdash e \backsim e' \uparrow \tau' \qquad \models \tau' \leq \tau}{\Gamma \vdash e \backsim e' \downarrow \tau}\ \textbf{alg-}\uparrow\downarrow$$

$$\frac{\Gamma \vdash e \backsim e' \downarrow \tau}{\Gamma \vdash (e : \tau) \backsim (e' : \tau) \uparrow \tau}\ \textbf{alg-r-anno-}\uparrow$$

$$\boxed{\models \tau \leq \tau'}$$

$$\frac{}{\models \mathsf{bool}_r \leq \mathsf{bool}_r}\ \textbf{alg-bl-r} \qquad \frac{}{\models \mathsf{bool}_u \leq \mathsf{bool}_u}\ \textbf{alg-bl-u}$$

$$\frac{}{\models \mathsf{bool}_r \leq \mathsf{bool}_u}\ \textbf{alg-bl} \qquad \frac{\begin{array}{c}\models \tau_1' \leq \tau_1 \\ \models \tau_2 \leq \tau_2'\end{array}}{\models \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\ \textbf{alg-}\to$$

Figure 2. relSTLC algorithmic typing and subtyping rules

by erasing type annotations from e. The bidirectional type system is also *complete* relative to the declarative type system: If $\Gamma \vdash e_1 \backsim e_2 : \tau$, then there are type-annotated variants, e_1', e_2' of e_1, e_2 such that $\Gamma \vdash e_1' \backsim e_2' \uparrow \tau$. These annotations can be limited to the top-level and any explicit β-redexes. The proofs of these statements are in the appendix.

3 RelRef

Next, we extend bidirectional typechecking to *relational refinements*. Relational refinements [7, 13, 15, 27, 38] express fine-grained relations between pairs of expressions. They have been used for many different purposes ranging from information flow control to differential privacy. We consider here a simple setting, which still suffices to bring out key challenges in applying bidirectional typechecking to relational refinements.

We extend relSTLC with primitive lists and a relational refinement type $\mathsf{list}[n]^\alpha\ \tau$, which ascribes a pair of lists, both of length n, that differ pointwise in at most α positions (n and α are natural numbers). $\mathsf{list}[n]^\alpha\ \tau$ *refines* the standard list type with n and α and the refinement is *relational* since α expresses a constraint on the two lists together. To construct lists of this type when $\alpha \neq n$, we also need a way to express that at least $(n - \alpha)$ elements are pointwise equal. To this end, we introduce the comonadic type $\Box\,\tau$, which ascribes

$$\frac{\begin{array}{c}\Delta;\Phi_a;\Gamma \vdash e_1 \backsim e_1' : \tau \\ \Delta;\Phi_a;\Gamma \vdash e_2 \backsim e_2' : \text{list}[n]^{\alpha}\,\tau\end{array}}{\Delta;\Phi_a;\Gamma \vdash \text{cons}(e_1,e_2) \backsim \text{cons}(e_1',e_2') : \text{list}[n+1]^{\alpha+1}\,\tau}\ \textbf{rr-cons1} \qquad \frac{\begin{array}{c}\Delta;\Phi_a;\Gamma \vdash e_1 \backsim e_1' : \Box\,\tau \\ \Delta;\Phi_a;\Gamma \vdash e_2 \backsim e_2' : \text{list}[n]^{\alpha}\,\tau\end{array}}{\Delta;\Phi_a;\Gamma \vdash \text{cons}(e_1,e_2) \backsim \text{cons}(e_1',e_2') : \text{list}[n+1]^{\alpha}\,\tau}\ \textbf{rr-cons2}$$

$$\frac{\begin{array}{c}\Delta;\Phi_a;\Gamma \vdash e \backsim e' : \text{list}[n]^{\alpha}\,\tau \qquad \Delta;\Phi_a \wedge n = 0;\Gamma \vdash e_1 \backsim e_1' : \tau' \\ i,\Delta;\Phi_a \wedge n = i+1; h : \Box\,\tau, tl : \text{list}[i]^{\alpha}\,\tau, \Gamma \vdash e_2 \backsim e_2' : \tau' \\ i,\beta,\Delta;\Phi_a \wedge n = i+1 \wedge \alpha = \beta+1; h : \tau, tl : \text{list}[i]^{\beta}\,\tau, \Gamma \vdash e_2 \backsim e_2' : \tau'\end{array}}{\Delta;\Phi_a;\Gamma \vdash \text{ case } e \text{ of nil} \to e_1 \mid h :: tl \to e_2 \backsim \text{ case } e' \text{ of nil} \to e_1' \mid h :: tl \to e_2' : \tau'}\ \textbf{rr-caseL}$$

$$\frac{\begin{array}{c}\Delta;\Phi_a \wedge C;\Gamma \vdash e_1 \backsim e_2 : \tau \\ \Delta;\Phi_a \wedge \neg C;\Gamma \vdash e_1 \backsim e_2 : \tau \qquad \Delta \vdash C\ \text{wf}\end{array}}{\Delta;\Phi_a;\Gamma \vdash e_1 \backsim e_2 : \tau}\ \textbf{rr-split} \qquad \frac{\begin{array}{c}\Delta;\Phi_a;\Gamma \vdash e \backsim e : \tau \\ \forall x \in dom(\Gamma).\ \Delta;\Phi_a \models \Gamma(x) \sqsubseteq \Box\,\Gamma(x)\end{array}}{\Delta;\Phi_a;\Gamma,\Gamma' \vdash e \backsim e : \Box\,\tau}\ \textbf{rr-nochange}$$

$$\frac{}{\Delta;\Phi_a \models \Box(\tau_1 \to \tau_2) \sqsubseteq \Box\tau_1 \to \Box\tau_2}\to\Box\textbf{diff} \qquad \frac{\Delta;\Phi_a \models n \doteq n' \qquad \Delta;\Phi_a \models \alpha \leq \alpha' \qquad \Delta;\Phi_a \models \tau \sqsubseteq \tau'}{\Delta;\Phi_a \models \text{list}[n]^{\alpha}\,\tau \sqsubseteq \text{list}[n']^{\alpha'}\,\tau'}\ \textbf{l1}$$

$$\frac{\Delta;\Phi_a \models \alpha \doteq 0}{\Delta;\Phi_a \models \text{list}[n]^{\alpha}\,\tau \sqsubseteq \text{list}[n]^{\alpha}\,\Box\,\tau}\ \textbf{l2} \qquad \frac{}{\Delta;\Phi_a \models \text{list}[n]^{\alpha}\,\Box\,\tau \sqsubseteq \Box\,(\text{list}[n]^{\alpha}\,\tau)}\ \textbf{l}\Box \qquad \frac{}{\Delta;\Phi_a \models \Box\,\tau \sqsubseteq \tau}\ \textbf{T}$$

Figure 3. RelRef typing and subtyping

$$\frac{\Delta;\Phi_a;x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash e_1 \backsim e_2 :^{c} \tau_2}{\Delta;\Phi_a;\Gamma \vdash \text{fix } f(x).e_1 \backsim \text{fix } f(x).e_2 :^{c} \tau_1 \to \tau_2}\ \textbf{c-r-fix} \qquad \frac{\Delta;\Phi_a;\Box\,\Gamma \vdash e \backsim e :^{c} \tau}{\Delta;\Phi_a;\Box\,\Gamma,\Gamma' \vdash \mathsf{NC}\ e \backsim \mathsf{NC}\ e :^{c} \Box\,\tau}\ \textbf{c-nochange}$$

$$\frac{\Delta;\Phi_a \wedge C;\Gamma \vdash e_1 \backsim e_2 :^{c} \tau \qquad \Delta;\Phi_a \wedge \neg C;\Gamma \vdash e_1' \backsim e_2' :^{c} \tau}{\Delta;\Phi_a;\Gamma \vdash \mathtt{split}\ (e_1,e_1')\ \text{with}\ C \backsim \mathtt{split}\ (e_2,e_2')\ \text{with}\ C :^{c} \tau}\ \textbf{c-s}$$

$$\frac{\Delta;\Phi_a;\Gamma \vdash e_1 \backsim e_1' :^{c} \tau \qquad \Delta;\Phi_a;\Gamma \vdash e_2 \backsim e_2' :^{c} \text{list}[n]^{\alpha}\,\tau}{\Delta;\Phi_a;\Gamma \vdash \text{cons}_C(e_1,e_2) \backsim \text{cons}_C(e_1',e_2') :^{c} \text{list}[n+1]^{\alpha+1}\,\tau}\ \textbf{c-cons1}$$

$$\frac{\Delta;\Phi_a;\Gamma \vdash e_1 \backsim e_1' :^{c} \Box\,\tau \qquad \Delta;\Phi_a;\Gamma \vdash e_2 \backsim e_2' :^{c} \text{list}[n]^{\alpha}\,\tau}{\Delta;\Phi_a;\Gamma \vdash \text{cons}_{NC}(e_1,e_2) \backsim \text{cons}_{NC}(e_1',e_2') :^{c} \text{list}[n+1]^{\alpha}\,\tau}\ \textbf{c-r-cons2} \qquad \frac{\Delta;\Phi_a;\Gamma \vdash e \backsim e' :^{c} \tau \qquad \Delta;\Phi_a \models \tau \equiv \tau'}{\Delta;\Phi_a;\Gamma \vdash e \backsim e' :^{c} \tau'}\ \textbf{c-r-}\equiv$$

Figure 4. RelRef Core typing rules

$$\frac{\begin{array}{c}i,\beta \in \text{fresh}(\mathbb{N}) \qquad \Delta;\psi_a;\Phi_a;\Gamma \vdash e_1 \ominus e_1' \downarrow \tau \Rightarrow \Phi_1 \\ \Delta;i,\beta,\psi_a;\Phi_a;\Gamma \vdash e_2 \ominus e_2' \downarrow \text{list}[i]^{\beta}\,\tau \Rightarrow \Phi_2 \qquad \Phi_2' = n \doteq (i+1) \wedge \exists\beta :: \mathbb{N}.\Phi_2 \wedge \alpha \doteq \beta+1\end{array}}{\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{cons}_C(e_1,e_2) \ominus \text{cons}_C(e_1',e_2') \downarrow \text{list}[n]^{\alpha}\,\tau \Rightarrow \Phi_1 \wedge \exists i :: \mathbb{N}.\Phi_2'}\ \textbf{alg-r-consC-}\downarrow$$

$$\frac{\begin{array}{c}i \in \text{fresh}(\mathbb{N}) \qquad \Delta;\psi_a;\Phi_a;\Gamma \vdash e_1 \ominus e_1' \downarrow \Box\,\tau \Rightarrow \Phi_1 \\ \Delta;i,\psi_a;\Phi_a;\Gamma \vdash e_2 \ominus e_2' \downarrow \text{list}[i]^{\alpha}\,\tau \Rightarrow \Phi_2 \qquad \Phi_2' = \Phi_2 \wedge n \doteq (i+1)\end{array}}{\Delta;\psi_a;\Phi_a;\Gamma \vdash \text{cons}_{NC}(e_1,e_2) \ominus \text{cons}_{NC}(e_1',e_2') \downarrow \text{list}[n]^{\alpha}\,\tau \Rightarrow \Phi_1 \wedge \exists i :: \mathbb{N}.\Phi_2'}\ \textbf{alg-r-consNC-}\downarrow$$

$$\frac{\Delta;\psi_a;C \wedge \Phi_a;\Gamma \vdash e_1 \ominus e_1' \downarrow \tau \Rightarrow \Phi_1 \qquad \Delta;\psi_a;\neg C \wedge \Phi_a;\Gamma \vdash e_2 \ominus e_2' \downarrow \tau \Rightarrow \Phi_2 \qquad \Delta \vdash C\ \text{wf}}{\Delta;\psi_a;\Phi_a;\Gamma \vdash \mathtt{split}\ (e_1,e_2)\ \text{with}\ C \ominus \mathtt{split}\ (e_1',e_2')\ \text{with}\ C \downarrow \tau \Rightarrow C \to \Phi_1 \wedge \neg C \to \Phi_2}\ \textbf{alg-r-split}\downarrow$$

$$\frac{\Delta;\psi_a;\Phi_a \models \tau \equiv \tau' \Rightarrow \Phi}{\Delta;\psi_a;\Phi_a \models \text{list}[n]^{\alpha}\,\tau \equiv \text{list}[n']^{\alpha'}\,\tau' \Rightarrow \Phi \wedge n \doteq n' \wedge \alpha \doteq \alpha'}\ \textbf{alg-r-list}$$

Figure 5. BiRelRef algorithmic typing and subtyping rules

pairs of expressions of type τ that are equal (i.e. the diagonal relation on τ). $\Box\,\tau$ generalizes the refinement r in bool_r to arbitrary types. Type-level terms like n and α are called index terms or indices, generically denoted I. The type system also supports quantification over such terms. To write recursive programs on lists, we also add a fixpoint operator, which poses no additional difficulty for bidirectional typechecking. The resulting system, called RelRef, has the following syntax.

$$\begin{array}{lll}
\text{Types } \tau ::= & \ldots \mid \text{list}[n]^{\alpha}\,\tau \mid \forall i{::}S.\,\tau \mid \exists i{::}S.\,\tau \\
& \mid \Box\,\tau \mid C \,\&\, \tau \mid C \supset \tau \\
\text{Expr } e ::= & \ldots \mid \text{fix } f(x).e \mid \text{nil} \mid \text{cons}(e_1, e_2) \\
& \mid (\text{case } e \text{ of nil} \rightarrow e_1 \mid h :: tl \rightarrow e_2) \mid \Lambda e \\
& \mid e[\,] \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{clet } e_1 \text{ as } x \text{ in } e_2 \\
& \mid \text{pack } e \mid \text{unpack } e_1 \text{ as } x \text{ in } e_2 \mid \text{celim } e \\
\text{Indices } I, n, \alpha ::= & i \mid 0 \mid I+1 \mid I_1 + I_2 \mid I_1 - I_2 \mid \frac{I_1}{I_2} \mid \\
& I_1 \cdot I_2 \mid \lceil I \rceil \mid \lfloor I \rfloor \mid \mathtt{min}(I_1, I_2) \mid \mathtt{max}(I_1, I_2)
\end{array}$$

Types can quantify over index variables, i, as in $\forall i{::}S.\,\tau$ and $\exists i{::}S.\,\tau$. The constructs pack e and unpack e_1 as x in e_2 are the introduction and elimination forms for existentially quantified types. The constructs $\Lambda.e$ and $e[\,]$ are the introduction and elimination forms for universally quantified types. To represent arithmetic relations over index variables, constraints denoted C, sets of predicates over index terms, appear in types as in $C \,\&\, \tau$ and $C \supset \tau$. The type $C \,\&\, \tau$ means the type τ and that C holds, while $C \supset \tau$ means that, if the constraint C holds, then the type is τ. The construct clet e_1 as x in e_2 is the elimination form for the constrained type $C \,\&\, \tau$. By design, index terms do not appear in RelRef expressions.

Example (map) As an example, we can write the standard list map function, and give it the following very informative relational type in RelRef (for any τ_1, τ_2).

$$\begin{array}{l}
\text{fix } \mathtt{map}(f).\Lambda.\Lambda.\lambda l.\ \text{case } l \text{ of nil} \rightarrow \text{nil} \\
\qquad\qquad\qquad \mid h :: tl \rightarrow \text{cons}(f\ h,\ \mathtt{map}\ f[\,][\,]\ tl)
\end{array}$$

$$\mathtt{map} : (\Box\,(\tau_1 \rightarrow \tau_2)) \rightarrow \forall n, \alpha.\text{list}[n]^{\alpha}\,\tau_1 \rightarrow \text{list}[n]^{\alpha}\,\tau_2$$

The type means that two runs of map with *equal* mapping functions and two lists that differ in at most α positions result in two lists with the same property. Notice how α is universally quantified in the type, and how $\Box$ represents that the mapping function be equal in the two runs.

Declarative typing RelRef's typing judgment has the form $\Delta; \Phi_a; \Gamma \vdash e_1 \backsim e_2 : \tau$ and means that e_1 and e_2 have the relational type τ if the constraints Φ_a hold. Δ is a (universally quantified) context of index variables and Γ, as usual, is the typing context for program variables. Figure 3 shows selected typing rules that use refinements and constraints in interesting ways, and make bidirectional typechecking difficult. To start, as a result of the relational refinement α in the list type, there are *two* rules for typing the list cons constructor. Rule **rr-cons1** applies when the head elements of the constructed lists may differ. Note how the relational refinement α changes to $\alpha+1$ from the premise to the conclusion. Rule **rr-cons2** applies when the head elements are equal, witnessed by the comonadic type $\Box\,\tau$. α does not change in this rule. Dually, the cons branch of list case analysis (rule **rr-caseL**) is typed *twice* with different index constraints—once for each of these two possible ways of constructing the cons-ed list. A consequence of this double typing of the same branch with different constraints is that expressions cannot contain index terms (else such typing may be impossible). The rule **rr-split** case-splits on an arbitrary constraint C in the context. This is useful for typing recursive functions [13, 15]. Finally, the rule for introducing the type $\Box\,\tau$, **rr-nochange**, is interesting. It says that if e relates to itself at type τ and all variables in Γ morally have $\Box$-ed types (checked via subtyping), then e also relates to itself at type $\Box\,\tau$.

Declarative subtyping RelRef subtyping $\tau \sqsubseteq \tau'$ is complex. Some of the rules are shown in Figure 3. First, subtyping is constraint-dependent, because it must, for instance, be able to show that $\text{list}[n]^{\alpha}\,\tau \sqsubseteq \text{list}[m]^{\alpha}\,\tau$ when $n = m$. Second, in RelRef, $\Box$'s comonadic properties manifest themselves via subtyping. This results in interactions between $\Box$ and other connectives as, for instance, in the rules $\rightarrow \Box_{\textbf{diff}}$, **l2** and **l**$\Box$.

We explain some of the subtyping rules. The rule **l1** allows α, the upper-bound on the number of elements that differ in the two lists, to be weakened covariantly. The rule **l2** allows two related lists with zero differences to be retyped as two related lists whose elements are in the diagonal relation. The rule **l**$\Box$ allows two related lists whose elements are equal to be retyped as two equal lists, represented by the outer $\Box$. The rule **T** coerces $\Box\,\tau$ to τ by forgetting that the two related elements are, in fact, equal.

Towards algorithmization An algorithm for typechecking RelRef faces two difficulties beyond those seen in relSTLC. Both difficulties arise due to RelRef's relational refinements. First, there is additional non-syntax-directedness in the rules: Rules **rr-cons1** and **rr-cons2** apply to expressions of the *same* shape (the rules differ only in their treatment of index terms), and rules **rr-split** and **rr-nochange** are not syntax-directed (their use overlaps with other rules). Second, owing to the interaction between $\Box$ and other type constructs, it is infeasible to re-define subtyping in a way that makes transitivity admissible. As a result of these two problems, bidirectional typing alone does not yield an *algorithm* for typechecking.

An obvious way to address the first of these problems is to force additional annotations in expressions to remove the non-syntax-directness. However, this will not address the problem with subtyping. Importantly, it also will not allow us to theoretically connect the algorithmic type system to the declarative type system (with its non-syntax-directedness).

Consequently, we follow a slightly different approach. First, we introduce a simpler core calculus RelRef Core, which annotates expressions to resolve the lack of syntax-directedness in typing rules. Additionally, RelRef Core features only *type equivalence*, not subtyping. We show that every RelRef expression can be *elaborated* to a semantically equivalent expression in RelRef Core by adding enough annotations and expressing subtyping as definable type coercions. Next, we build a bidirectional, algorithmic type system for RelRef Core and prove it relatively sound and complete. Although the elaboration to RelRef Core cannot be directly *implemented* without losing completeness or relying on programmer annotations, this approach does establish a strong *theoretical* point, end-to-end: There is a calculus (RelRef Core) that is as expressive as RelRef, and that is fully amenable to bidirectional typechecking. Our prototype implementation (Section 6) relies on sound heuristics to implement the elaboration.

RelRef Core syntax The core calculus RelRef Core is similar to RelRef but has explicit syntactic markers to indicate which typing rules to apply where, thus resolving the nondeterminism caused by the aforementioned typing rules such as **rr-split** and **rr-cons1**/**rr-cons2**. The expression syntax of RelRef Core is as follows.

$$\begin{array}{lll}\text{Expr } e ::= & \ldots \mid \texttt{split}\ (e_1, e_2)\ \text{with}\ C \mid \texttt{NC}\ e \mid \Lambda i.e \mid e[I] \mid \\ & \text{cons}_{NC}(e_1, e_2) \mid \text{cons}_{C}(e_1, e_2) \mid \\ & \left(\begin{array}{l}\text{case}\ e\ \text{of nil}\ \rightarrow e_1 \\ \mid h\ ::_{NC}\ tl\ \rightarrow e_2 \mid h ::_C\ tl\ \rightarrow e_3\end{array}\right)\end{array}$$

The list constructor cons is separated into two—cons_C and cons_{NC}—to disambiguate the rules **rr-cons1** and **rr-cons2**. Dually, the list-case construct now has three branches, one each for nil, cons_C and cons_{NC}. "$\texttt{split}\ (e_1, e_2)$ with C" indicates that rule **rr-split** must be applied and records the constraint C to split on. To write meaningful Cs, expressions now carry index terms. For instance, the elimination form for universally quantified types in RelRef Core is $e[I]$ as opposed to RelRef's $e[\,]$. The construct $\texttt{NC}\ e$ indicates an application of the rule **rr-nochange**.

RelRef Core typing rules Selected rules of RelRef Core's typing judgment $\Delta; \Phi_a; \Gamma \vdash e \backsim e' :^c \tau$ are shown in Figure 4. Φ_a and Δ are interpreted as in RelRef. RelRef Core's rules are similar to RelRef's, but there are important differences. First, rules are now syntax-directed. Second, there is no relational subtyping. Instead, there is type-equivalence, $\equiv$, which is much simpler than subtyping. It only lifts equality modulo constraints to types (e.g., $\text{list}[1+2]^\alpha\ \tau \equiv \text{list}[3]^\alpha\ \tau$) and it can be easily implemented algorithmically (modulo constraint solving). We omit its straightforward details.

Simulating RelRef's subtyping A key property of RelRef Core is that it can simulate RelRef's subtyping via explicit coercion functions, as formalized in the following lemma. Such elimination of subtyping is a common technique for simplifying typechecking in the unary setting [11, 17]; here, we lift the idea to the relational setting and to our comonad.

Lemma 1
If $\Delta; \Phi_a \models \tau \sqsubseteq \tau'$ in RelRef then there exists $e \in$ RelRef Core s.t. $\Delta; \Phi_a; \cdot \vdash e \backsim e :^c \tau \rightarrow \tau'$.

Proof. By induction on the subtyping derivation. □

Elaboration Given Lemma 1, we define a straightforward type derivation-directed embedding from RelRef to RelRef Core. Briefly, we use the RelRef typing derivation to insert additional syntactic annotations that RelRef Core needs and use Lemma 1 wherever subtyping appears in the RelRef derivation. The embedding preserves well-typedness (see the appendix for details). This shows that RelRef Core is as expressive as RelRef. (In fact, this is expressiveness in a strong sense, dubbed macro-expressiveness by Felleisen [25].)

Algorithmic (bidirectional) typechecking We now build an algorithmic, bidirectional type system for RelRef Core. We call this system BiRelRef. Selected rules of BiRelRef are shown in Figure 5. As before, BiRelRef has two typing judgments: one to check types and the other to infer them. The key addition over relSTLC is that BiRelRef's typing judgments output *constraints* between index terms, which must be verified for typing. The checking judgment has the form $\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \downarrow \tau \Rightarrow \Phi$. Here, the type τ is an input, but the constraints Φ are an output. The intuitive meaning is that *if constraints* Φ *hold* (assuming Φ_a), then $\Delta; \Phi_a; \Gamma \vdash e_1 \backsim e_2 :^c \tau$ holds in RelRef Core. (The reader may ignore the index variable context ψ_a; it contains variables universally quantified in Φ.) As an example, consider the rule **alg-r-consC-↓** for relating cons-ed lists at type $\text{list}[n]^\alpha\ \tau$ when the heads may differ. To do this, the rule relates the tails at type $\text{list}[i]^\beta\ \tau$ for some i and β. For this to be sound, $n \doteq i + 1$ and $\alpha \doteq \beta + 1$ must hold, so these appear as constraints in Φ. Note that Φ quantifies over i and β existentially. Constraint solvers cannot handle such existential quantification easily, a point to which we return in Section 6.

In the inference judgment $\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \uparrow \tau \Rightarrow \Phi$, both Φ and τ are outputs. The meaning of the judgment is similar: if Φ holds (assuming Φ_a), then $\Delta; \Phi_a; \Gamma \vdash e_1 \backsim e_2 :^c \tau$.

Like typing, the algorithmic type equivalence judgment $\Delta; \psi_a; \Phi_a \models \tau \equiv \tau' \Rightarrow \Phi$ also generates constraints.

Soundness and completeness We prove that BiRelRef is sound and complete w.r.t. RelRef Core's declarative type system. Soundness says that any inference or checking judgment provable in the algorithmic type system can be simulated in RelRef Core if the output constraints Φ are satisfied. Dually, completeness says that any pair of typeable RelRef Core programs can be sufficiently annotated with types to make their type checkable in BiRelRef, with satisfiable output constraints. As before, we write $|e|$ for the erasure of typing annotations from a BiRelRef expression e to yield a RelRef Core expression.

Theorem 2 **(Soundness)**

1. Assume $\Delta; \psi_a; \Phi_a; \Gamma \vdash e \ominus e' \downarrow \tau \Rightarrow \Phi$, $\mathrm{FIV}(\Phi_a, \Gamma, \tau) \subseteq dom(\Delta, \psi_a)$, and θ_a is a valid substitution for ψ_a s.t. $\Delta; \Phi_a[\theta_a] \models \Phi[\theta_a]$ holds.
   Then, $\Delta; \Phi_a[\theta_a]; \Gamma[\theta_a] \vdash |e| \backsim |e'| :^c \tau[\theta_a]$.
2. Assume $\Delta; \psi_a; \Phi_a; \Gamma \vdash e \ominus e' \uparrow \tau \Rightarrow \Phi$, $\mathrm{FIV}(\Phi_a, \Gamma) \subseteq dom(\Delta, \psi_a)$, and θ_a is a valid substitution for ψ_a s.t. $\Delta; \Phi_a[\theta_a] \models \Phi[\theta_a]$ holds.
   Then, $\Delta; \Phi_a[\theta_a]; \Gamma[\theta_a] \vdash |e| \backsim |e'| :^c \tau[\theta_a]$.

Proof. By simultaneous induction on the given BiRelRef derivations. □

Theorem 3 **(Completeness)**

1. Assume that $\Delta; \Phi_a; \Gamma \vdash e_1 \backsim e_2 :^c \tau$. Then, there exist e_1', e_2' such that $\Delta; \cdot; \Phi_a; \Gamma \vdash e_1' \ominus e_2' \downarrow \tau \Rightarrow \Phi$ and $\Delta; \Phi_a \models \Phi$ and $|e_1'| = e_1$ and $|e_2'| = e_2$.

Proof. By induction on the given RelRef Core typing derivation. □

4 RelRefU

In many cases, it is possible to prove a relation between two expressions by analyzing them *individually*. For example, if we can prove that e_1 and e_2 individually produce lists of length n with elements of type τ, then it is immediate that $\vdash e_1 \backsim e_2 : \mathrm{list}[n]^n\, \tau$. Falling back to such *unary* analysis during relational analysis is not allowed in RelRef. In this section, we extend RelRef to allow such fall back to unary analysis. We call the new system RelRefU.

RelRefU adds a new class of unary types, A, which ascribe individual expressions. These are the "standard" types from existing refinement systems like DML [44]. For example, the unary type for lists, $\mathrm{list}[n]\, A$, carries a refinement n, the length of the list. Unary types also include quantification over index variables, which we elide here for brevity. Importantly, we also add a new *relational* type $U\,(A_1, A_2)$ which ascribes any pair e_1, e_2, whose unary types are A_1 and A_2, respectively.

Unary types	A	$::=$	$\mathrm{bool} \mid A_1 \rightarrow A_2 \mid \mathrm{list}[n]\, A \mid \ldots$
Relational types	τ	$::=$	$\ldots \mid U\,(A_1, A_2)$

Declarative typing RelRefU has two typing judgments, unary and relational. The unary judgment's rules are exactly those of standard (unary) refinement type systems like DML [44], so we elide them here. The relational rules are those of RelRef and the following new rule, **r-switch**, which allows the use of unary typing in relational typing. Here, $|.|_i$ is a projection function that converts a relational type to its left ($i = 1$) or right ($i = 2$) unary type by forgetting relational refinements. For example, $|U\,(A_1, A_2)|_1 = A_1$ and $|\mathrm{list}[n]^\alpha\, \tau|_1 = \mathrm{list}[n]\,(|\tau|_1)$.

$$\frac{|\Gamma|_1 \vdash e_1 : A_1 \qquad |\Gamma|_2 \vdash e_2 : A_2}{\Gamma \vdash e_1 \backsim e_2 : U\,(A_1, A_2)}\ \textbf{r-switch}$$

Besides this rule, the second interesting aspect of RelRefU is subtyping for $U\,(A_1, A_2)$, which makes unary typing useful in relational reasoning. For instance, the example given at the beginning of this section is typed using **r-switch** and the subtyping rule $U\,(\mathrm{list}[n]\, A_1, \mathrm{list}[n]\, A_2) \sqsubseteq \mathrm{list}[n]^n\,(U\,(A_1, A_2))$.

Algorithmization Algorithmizing RelRefU faces new hurdles: The new rule **r-switch** is also not syntax-directed and subtyping for $U\,(A_1, A_2)$ is complex and cannot be re-defined to make transitivity admissible. Consequently, we follow the approach we used for RelInf. We define a core language, RelRefU Core, that has the syntactic markers of RelRef Core and a new construct `switch` e (which marks the rule **r-switch**) to resolve the ambiguity in typing rules. The language has simple type equivalence, not subtyping. We then define an elaboration of RelRefU into RelRefU Core. Finally, we define a bidirectional, algorithmic type system called BiRelRefU and prove it sound and complete relative to RelRefU Core. The entire development is not particularly more challenging than that of RelInf but is more tedious because we have to handle both unary and relational typing. Conceptually, the interesting aspect is that, in typing examples, we found it convenient to apply the **r-switch** rule in both checking and inference mode in the bidirectional type system, so this type system features two versions of the rule, one in each mode. This does not cause ambiguity in the rules since the mode is always uniquely known.

5 RelCost

As our last step, we add a relational effect, namely, relative cost to RelRefU. This results in the type system RelCost of Çiçek et al. [13]. RelCost allows establishing an upper bound on the *relative cost* of two expressions e_1 and e_2, i.e. on $\mathrm{cost}(e_1) - \mathrm{cost}(e_2)$. For this, RelCost extends RelRefU's types and judgments with cost effects. Expressions are syntactically those of RelRefU, but the evaluation of elimination forms like list-case and function application produces nontrivial cost in the operational semantics.

We start with the types. The relational function type $\tau_1 \rightarrow \tau_2$ is refined to $\tau_1 \xrightarrow{\mathrm{diff}(t)} \tau_2$, where t (an index term of sort real) is an upper-bound on the relative cost of the bodies of the two functions that the type ascribes. Similarly, the *unary* function type $A_1 \rightarrow A_2$ is refined to $A_1 \xrightarrow{\mathrm{exec}(k,t)} A_2$, where k and t are lower and upper bounds on the cost of the body of the function. Other than this, the types match those of RelRefU.

As an example, the list map function from Section 3 can be given the following more informative type in RelCost:

$$\forall t.(\Box\,(\tau_1 \xrightarrow{\text{diff}(t)} \tau_2)) \to \forall n, \alpha.\text{list}[n]^{\alpha}\ \tau_1 \xrightarrow{\text{diff}(t\cdot\alpha)} \text{list}[n]^{\alpha}\ \tau_2$$

This type says that if two runs of map are given the *same* mapping function whose body's cost may vary by at most t across inputs, and two lists that differ in no more than α elements, then the relative cost of the two runs is no more than $t \cdot \alpha$. Intuitively, this makes sense. The two runs differ only in applications of the mapping function to list elements that differ. Each such application results in a relative cost at most t and there are at most α such applications.

Declarative typing Like RelRefU, RelCost has two typing judgments, one unary and one relational. The difference from RelRefU is that these judgments now carry *cost effects*—upper and lower bounds on the cost of the expression being typed in the unary judgment and an upper bound on the relative cost of the two expressions in the relational judgment. The unary judgment, $\Delta; \Phi_a; \Omega \vdash_k^t e : A$, means that e has the unary type A and its cost is upper- and lower-bounded by t and k, respectively. The relational judgment, $\Delta; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$, means that e_1, e_2 have the relational type τ and their relative cost is upper-bounded by t. Here, we describe only on the relational judgment, since the unary judgment has simpler rules.

The rules of the relational judgment are obtained by augmenting the rules of RelRefU to track costs. Selected, interesting rules are shown in Figure 6. The rule **r-fix** types two recursive functions at type $\tau_1 \xrightarrow{\text{diff}(t)} \tau_2$ if their bodies have the relative cost t. Rule **r-app** relates $e_1\ e_2$ and $e_1'\ e_2'$ with a relative cost obtained by *adding* the relative costs of (e_1, e_1'), (e_2, e_2') and of the bodies of the applied functions (obtained from the function type in the first premise). Rule **nochange** says that if e relates to itself in a context that has only variables of $\Box$-ed types (i.e. they will get substituted by equal values in the two runs), then e's cost relative to itself is 0. Finally, rule **switch** allows a fallback to unary reasoning: If e_1's unary cost is upper-bounded by t_1 and e_2's unary cost is lower-bounded by k_2, then e_1, e_2's relative cost is (trivially) upper-bounded by $t_1 - k_2$.

Declarative subtyping RelCost's subtyping is directly based on RelRef and RelRefU, but the rules are additionally aware of costs. For example, the RelRef rule $\to \Box$**diff** (Figure 3) gets refined to: $\Box\,(\tau_1 \xrightarrow{\text{diff}(t)} \tau_2) \sqsubseteq \Box\,\tau_1 \xrightarrow{\text{diff}(0)} \Box\,\tau_2$, which intuitively means that two equal functions when given two equal arguments reduce with exactly the same cost.

Towards algorithmization RelCost inherits all the non syntax-directedness and subtyping complexity of RelRefU, and additionally adds costs. To build an algorithmic type system for RelCost, we follow the approach of (RelRef and) RelRefU. We first define a simpler core language, RelCostCore, which resolves all rule ambiguity and has type equivalence in place of subtyping, and elaborate RelCost into this core language. This step is not significantly harder than for RelRefU since RelCost does not add more rule-ambiguity. Hence, we do not describe this step further.

The interesting step is the second one—the bidirectional type system for RelCostCore. This bidirectional system uses constraints to relate not just type refinements but also costs of subexpressions, as we explain next.

Algorithmic (bidirectional) typechecking For RelCost, the bidirectional typechecking must not only check/infer the type but also the cost. Consequently, one might expect that, for each of unary and relational typing, one would need not two but four bidirectional judgments—one judgment for each combination of checking and inferring the type and checking and inferring the cost. However, after some experimentation with the design, we realized that the judgments where the type is checked and the cost is inferred or vice-versa are actually not required. In hindsight, this is because the cost can be viewed as an extension of the type even if the two are written separately for convenience, so the type and the cost always follow the same mode (checking or inference).

As a result, RelCost's bidirectional type system, called BiRelCost, uses two judgments – one for type checking and one for type inference – for each of unary and relational typing. We describe here only the two relational judgments as these are more complicated than the unary judgments. The relational checking judgment $\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \downarrow \tau, t \Rightarrow \Phi$ means that if Φ holds (assuming Φ_a), then $\Delta; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau$ is provable in RelCostCore. Like the type τ, the relative cost t is also an input in this judgment, i.e. it is also checked. The relational inference judgment $\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \uparrow \tau \Rightarrow [\psi], t, \Phi$ has a similar meaning, but both the cost t and the type τ are outputs, i.e. both are inferred. This judgment also has an additional output ψ. This is a set of existentially quantified cost variables generated by the rules. Substitutions for these variables must be found to satisfy Φ.

To understand the need for ψ, let us examine a few typing rules (Figure 7). Consider the rule **alg-r-app-**$\uparrow$ for function application. In this rule, the relative cost t_1 of the functions is inferred (first premise), but the cost of the arguments must be checked. Since this cost is not available upfront, the rule creates a fresh existential variable t_2 and "checks" the arguments against that. This results in appropriate constraints on t_2 getting added to Φ_2 in the second premise. t_2 is included in the total cost and, importantly, it is added to ψ in the conclusion to indicate that it is existentially quantified (any substitution for it that satisfies the final constraints is okay). This pattern of creating existential variables for the costs that need to be checked but aren't already known is

$$\dfrac{\Delta; \Phi_a; x : \tau_1, f : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2, \Gamma \vdash e_1 \ominus e_2 \lesssim t : \tau_2}{\Delta; \Phi_a; \Gamma \vdash \text{fix } f(x).e_1 \ominus \text{fix } f(x).e_2 \lesssim 0 : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2}\ \textbf{r-fix}$$

$$\dfrac{\Delta; \Phi_a; \Gamma \vdash e_1 \ominus e_1' \lesssim t_1 : \tau_1 \xrightarrow{\text{diff}(t)} \tau_2 \qquad \Delta; \Phi_a; \Gamma \vdash e_2 \ominus e_2' \lesssim t_2 : \tau_1}{\Delta; \Phi_a; \Gamma \vdash e_1\ e_2 \ominus e_1'\ e_2' \lesssim t_1 + t_2 + t : \tau_2}\ \textbf{r-app}$$

$$\dfrac{\Delta; \Phi_a; \Gamma \vdash e \ominus e \lesssim t : \tau \qquad \forall x \in dom(\Gamma).\ \Delta; \Phi_a \models \Gamma(x) \sqsubseteq \Box\Gamma(x)}{\Delta; \Phi_a; \Gamma, \Gamma'; \Omega \vdash e \ominus e \lesssim 0 : \Box\tau}\ \textbf{nochange}$$

$$\dfrac{\Delta; \Phi_a; |\Gamma| \vdash_{k_1}^{t_1} e_1 : A \qquad \Delta; \Phi_a; |\Gamma| \vdash_{k_2}^{t_2} e_2 : A}{\Delta; \Phi_a; \Gamma \vdash e_1 \ominus e_2 \lesssim t_1 - k_2 : U\,A}\ \textbf{switch}$$

Figure 6. RelCost relational typing rules

$$\dfrac{t' \in \text{fresh}(\mathbb{R}) \qquad \Delta; t', \psi_a; \Phi_a; \Box\Gamma \vdash e \ominus e \downarrow \tau, t' \Rightarrow \Phi}{\Delta; \psi_a; \Phi_a; \Gamma', \Box\Gamma \vdash \mathsf{NC}\ e \ominus \mathsf{NC}\ e \downarrow \Box\tau, t \Rightarrow 0 \doteq t \wedge (\exists t' :: \mathbb{R}.\Phi)}\ \textbf{alg-r-nochange-}\!\downarrow$$

$$\dfrac{\Delta; \psi_a; \Phi_a; \Gamma \vdash e \ominus e' \uparrow \tau' \Rightarrow [\psi], t', \Phi_1 \qquad \Delta; \psi, \psi_a; \Phi_a \models \tau' \equiv \tau \Rightarrow \Phi_2}{\Delta; \psi_a; \Phi_a; \Gamma \vdash e \ominus e' \downarrow \tau, t \Rightarrow \exists(\psi).\Phi_1 \wedge \Phi_2 \wedge t' \leq t}\ \textbf{alg-r-}\!\uparrow\downarrow$$

$$\dfrac{\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1 \ominus e_1' \uparrow \tau_1 \xrightarrow{\text{diff}(t_e)} \tau_2 \Rightarrow [\psi], t_1, \Phi_1 \qquad t_2 \in \text{fresh}(\mathbb{R}) \qquad \Delta; t_2, \psi, \psi_a; \Phi_a; \Gamma \vdash e_2 \ominus e_2' \downarrow \tau_1, t_2 \Rightarrow \Phi_2}{\Delta; \psi_a; \Phi_a; \Gamma \vdash e_1\ e_2 \ominus e_1'\ e_2' \uparrow \tau_2 \Rightarrow [t_2, \psi], t_1 + t_2 + t_e, \Phi_1 \wedge \Phi_2}\ \textbf{alg-r-app-}\!\uparrow$$

Figure 7. BiRelCost algorithmic typing rules

pervasive in the rules, and is the key technical increment in BiRelCost relative to BiRelRefU.

We comment on two other interesting rules. The rule **alg-r-↑↓**, which switches from inference to checking mode when read top-down, closes the existential variables ψ in the premise by explicitly introducing an existential quantifier over them in the conclusion. In the rule **alg-r-nochange-↓** (which implements the rule **nochange** from Figure 6), the final cost must be 0. So, a constraint equating the given cost t to 0 is generated.

BiRelCost is sound and complete relative to RelCostCore in the sense of Theorems 2 and 3 (appropriately adapted to the judgments of RelCost). We defer the details to the appendix.

Summary Since bidirectional typechecking for effects has received relatively little attention even in the context of unary analysis, we briefly recapitulate the insights we gained from designing BiRelCost. First, bidirectional typechecking extends very well to type systems with effects, even when combined with refinements and relational reasoning. Second, the mode of the effect (cost in our case) seems to mirror the mode of the type: The checking judgment checks both the type and the cost, while the inference judgment infers both. We did not find the need for a judgment that checks one but infers the other. Finally, bidirectional typechecking generates more existential variables than it would without effects, but effects do not complicate the meta-theory (soundness and completeness) substantially. The creation of additional existential variables has consequences for constraint solving, since SMT solvers do not handle such variables well, an issue to which we return in the next section.

6 Implementation

We have implemented a bidirectional typechecker for RelCost in OCaml. The typechecker implements the checking and inference judgments of BiRelCost but, to avoid overburdening the programmer, it works on the compact terms (expressions) of RelCost rather than the elaborate, annotated terms of RelCostCore. Hence, the terms do not resolve all the ambiguities of which (sub)typing rules to apply when. For this, the typechecker uses heuristics that we designed carefully by looking at a variety of examples. Conceptually, these heuristics are a sound but incomplete implementation of the elaboration (embedding) from RelCost to RelCostCore. As an alternative to these heuristics, we could have relied on additional programmer-provided annotations to guide the elaboration, but the heuristic approach is obviously easier for the programmer.

The constraints output by the bidirectional rules are solved using a combination of a custom procedure to eliminate existential variables and an off-the-shelf SMT solver. We describe both the heuristics and the constraint solving below.

Note that RelCost is a conservative extension of RelRef and RelRefU. Any derivation in RelRef or RelRefU can be simulated in RelCost by adding the trivial cost upper-bound of ∞. As a result, our implementation can also be used for RelRef and RelRefU.

Heuristics We list below the main heuristics we use to reduce the nondeterminism in picking (sub)typing rules.

1. To type a pair of cons-ed lists, we apply the bidirectional analogues of both the rules **rr-cons1** and **rr-cons2** (Figure 3) and combine the resulting constraints via disjunction.

2. When typing a function that takes an argument of type $\text{list}[n]^{\alpha}\,\tau$, we immediately apply the algorithmic analogue of the rule **rr-split** (Figure 3) with $C = (\alpha \doteq 0)$. For the case $\alpha \doteq 0$, we first try to complete the typing by invoking the rule **alg-r-nochange-↓** (Figure 7). This is because we found experimentally that many recursive list programs require this analysis. Moreover, the rule **rr-split** is *invertible*: Applying the rule cannot cause backtracking during search.

3. Subtyping is only invoked in three places: (a) for switching from checking to inference mode (rule **alg-r-↑↓** in Figure 7), (b) for the algorithmic version of the **nochange** rule (Figure 6), which checks subtyping on all variables in the context, and (c) as mentioned in the next point.

4. Relational subtyping rules that mention $\square$ are applied lazily at specific elimination points. For instance, in typing a function application, if the applied expression's inferred type is $\square\,(\tau_1 \xrightarrow{\text{diff}(k)} \tau_2)$, we try to complete the typing by subtyping to $\square\,\tau_1 \xrightarrow{\text{diff}(0)} \square\,\tau_2$ and $\tau_1 \xrightarrow{\text{diff}(k)} \tau_2$, in that order.

5. We switch to the unary reasoning (algorithmic analogue of rule **switch** from Figure 6) only when necessary i.e. when (a) eliminating expressions of the type $U\,(A_1, A_2)$, (b) checking related expressions at type $U\,(A_1, A_2)$, and (c) no other relational rules apply.

These heuristics suffice for all the examples we have seen so far. We list some of these examples later.

Constraint solving In principle, we could pass the constraints Φ output by BiRelCost's checking and inference judgments to an SMT solver that understands the domain of integers (for sizes) and real numbers (for costs). However, the constraint typically contains many existentially quantified variables and current SMT solvers do not eliminate such variables well. To solve this problem, we wrote a simple pre-processing pass that finds candidate substitutions for existentially quantified variables. For any such variable n, we look for constraints of the form $n = I$ and $n \leq I$. In either case, we consider I a candidate substitution for n. In this way, we generate a set of candidate substitutions for all existentially quantified variables. For each such substitution, we try to check the constraint's satisfiability using an SMT solver. (Our implementation actually does this lazily: It generates a candidate substitution; calls SMT; if that fails, generates the next substitution, and so on.)

To check the satisfiability of existential-free constraints, we invoke an SMT solver. Specifically, we use Why3 [26], a common front-end for many SMT solvers. Empirically, we have observed that only one SMT solver, Alt-Ergo [9], can handle our constraints and, so, our implementation uses this solver behind Why3. Why3 provides libraries of lemmas for exponentiation, logarithms and iterated sums, which we use in some of the examples. For typing programs that use divide-and-conquer over lists (e.g., merge sort), we have to provide as an axiom one additional lemma that solves a general recurrence related to costs. This lemma was proved in prior work (Lemma 2 in the appendix of [14]).

Experimental evaluation We have used our implementation to typecheck a variety of examples, including all the examples from the RelCost paper. Some of the examples, such as the relational analysis of merge sort (`msort`), have rather complex paper proofs. However, in all cases, the total typechecking time (including existential elimination and SMT solving) is less than 1s, suggesting that the approach is practical. Table 1 shows the experimental results over a subset of our example programs (our appendix lists all our example programs, including their code and experimental results). A "-" indicates a negligible value. Our experiments were performed on a 3.10GHz 2-core Intel Core i5 processor with 8 GB of RAM.

We briefly describe some of the example programs in Table 1 to highlight their diversity. The sizes of the programs vary from 7 to 81 LoC. The program `map` is the list map function of Section 3, typed with τ_1 = int and τ_2 = bool. The program `comp` is a constant-time (0 relative cost) comparison function that checks the equality of two passwords, represented as lists of bits. The program `sam` (square-and-multiply) computes the positive powers of a number, represented as a list of bits. The experiment `find` compares two functions that find a given element by scanning a list from head to tail and tail to head, respectively. The program `2Dcount` counts the number of rows of a matrix, represented as a list of lists in row-major form, that satisfy a predicate p and contain a key x. The program `bsplit` splits a list into two nearly equal length lists. The program `merge` merges two sorted lists and the program `msort` is the standard merge sort function. The program `ssort` is the standard selection sort function. We also created two slightly larger composite programs just to understand how well type-checking scales. The program, `multi_sort`, combines `msort` and `ssort`. It first sorts a list using `ssort`, then reverses the list using `rev` (which reverses a list), and then re-sorts the reversed list using `msort`. The program `ssort_list` first sorts a list, then appends the sorted list to the original unsorted list, and sorts the resulting longer list. Both sorts are done using `ssort`.

In all cases except `find`, the goal of the analysis is to find an upper bound on the relative cost of the function across *all* pairs of inputs that satisfy the function's input type. In all cases, the cost bounds we check are asymptotically tight.

Table 1. Statistics from BiRelCost examples. All times are in seconds. A "-" indicates a negligible value.

Benchmark	Total time(s)	Type-checking(s)	Existential elim.(s)	Constraint solving(s)	LoC	# of Annotations
`filter`	0.167	0.005	-	0.162	8	1
`append`	0.173	0.005	-	0.167	15	1
`rev`	0.170	0.005	-	0.164	12	1
`map`	0.172	0.005	-	0.167	7	1
`comp`	0.171	0.005	-	0.166	16	3
`sam`	0.174	0.005	-	0.169	13	1
`find`	0.172	0.005	-	0.167	21	3
`2Dcount`	0.168	0.005	-	0.163	17	4
`ssort`	0.181	0.005	-	0.175	26	3
`bsplit`	0.180	0.005	-	0.175	15	1
`flatten`	0.173	0.005	-	0.167	16	2
`appSum`	0.171	0.005	-	0.166	17	3
`merge`	0.207	0.005	-	0.201	26	3
`zip`	0.187	0.005	-	0.181	11	1
`msort`	0.384	0.008	0.003	0.373	29	3
`bfold`	0.216	0.006	0.001	0.208	22	2
`multi_sort`	0.958	0.012	0.015	0.930	81	9
`ssort_list`	0.207	0.006	0.004	0.197	72	9

Annotation effort In a traditional bidirectional type system, the programmer's annotation effort is limited to providing the eliminated type at every explicit β-redex and the type of every top-level function definition in the program. In our setting, the burden is similar, except that type annotations on functions also include a cost (on the arrow). In all but one of the examples we have tried, annotations are only necessary at each top-level function. One example has an explicit β-redex (in the form of a let-binding) and needs one additional annotation. Table 1 also shows the number of typing annotations we added to each example. This number is low across all examples except `multi_sort` and `ssort_list`. These two examples use a large number of let-bindings, which are explicit β-redexes. Overall, this shows that the bidirectional approach extends to the relational setting (even with refinements and effects), without adding significant annotation burden.

Heuristics illustrated with merge sort As an illustration of our heuristics, we explain how our implementation types the standard merge sort function relationally. The merge sort function, `msort`, splits a list into two nearly equal-sized sublists using an auxiliary function we call `bsplit`, recursively sorts each sublist and then merges the two sorted sublists using another function `merge`. In their paper on RelCost, Çiçek et al. [13] show that the relative cost of two runs of `msort` with two input lists of length n that differ in at most α positions is upper-bounded by $Q(n,\alpha) = \sum_{i=0}^{H} h(\left\lceil \frac{2^i}{2} \right\rceil)\cdot \mathtt{min}(\alpha, 2^{H-i})$, where $H = \lceil \mathtt{log}_2(n) \rceil$ and h is a specific linear function. This open-form expression lies in $O(n \cdot (1 + \mathtt{log}_2(\alpha)))$.[1] Next, we explain at a high-level how this relative cost is typechecked bidirectionally. We show below the code of the top-level merge sort function `msort`.

$\text{fix } \mathtt{msort}(\_).\Lambda.\Lambda.\lambda l.\text{case } l \text{ of nil } \rightarrow \text{nil}$
$\mid h_1 :: tl_1 \rightarrow \text{ case } tl_1 \text{ of nil } \rightarrow \text{cons}(h_1, \text{nil})$
$\quad \mid \_ :: \_ \rightarrow \text{let } r = \mathtt{bsplit}\ ()[\,][\,]\ l \text{ in}$
$\qquad\qquad \text{unpack } r \text{ as } r' \text{ in clet } r' \text{ as } (z_1, z_2)$
$\quad \text{in } \mathtt{merge}\ ()[\,][\,]\ (\mathtt{msort}\ ()[\,][\,]\ z_1, \mathtt{msort}\ ()[\,][\,]\ z_2)$

We do not show the code of the helper functions `bsplit` and `merge`, but they have the following types (these types are also checked with BiRelCost; we omit those details here):

$$\mathtt{bsplit} : \square\,(\text{unit}_r \rightarrow \forall n, \alpha{::}\mathbb{N}.\ \text{list}[n]^{\alpha}\ \tau \rightarrow \exists \beta{::}\mathbb{N}.\ \beta \leq \alpha\ \&\ (\text{list}[\lceil \tfrac{n}{2} \rceil]^{\beta}\ \tau \times \text{list}[\lfloor \tfrac{n}{2} \rfloor]^{\alpha-\beta}\ \tau))$$

$$\mathtt{merge} : \square\,(U\ (\text{unit} \rightarrow \forall n, m{::}\mathbb{N}.\ (\text{list}[n]\ \text{int} \times \text{list}[m]\ \text{int}) \xrightarrow{\text{exec}(h(\mathtt{min}(n,m)),\, h(n+m))} \text{list}[n+m]\ \text{int}))$$

Note the $\square$-es outside the types; their significance will be clear soon. Our aim is to typecheck `msort` relative to itself at the following type:

$$\square\,(\text{unit}_r \rightarrow \forall n, \alpha{::}\mathbb{N}.\ \text{list}[n]^{\alpha}\ (U\ \text{int}) \xrightarrow{\text{diff}(Q(n,\alpha))} U\ (\text{list}[n]\ \text{int}))$$

Here, $Q(n,\alpha)$ is the cost function defined above. We focus on the most interesting part where we call `merge` on the results of the two recursive calls to `msort`. At this point, we

[1]This relative cost $O(n \cdot (1 + \mathtt{log}_2(\alpha)))$ is asymptotically better than the relative cost $O(n \cdot \mathtt{log}_2(n))$ that can be established non-relationally. Also, this relative cost bound is tight.

have $z_1 : \text{list}[\lceil \frac{n}{2} \rceil]^{\beta} (U \text{ int})$ and $z_2 : \text{list}[\lfloor \frac{n}{2} \rfloor]^{\alpha-\beta} (U \text{ int})$ in the context (from the type of `bsplit`). Considering that only the calls to `merge` and `msort` incur additional costs (all the remaining operations occur synchronously on both sides and the relative cost of `bsplit` is 0 from its type), if we were to naively establish the bound $Q(n, \alpha)$, we would have to show the following inequality:

$$h(\left\lceil \frac{n}{2} \right\rceil) + Q(\left\lceil \frac{n}{2} \right\rceil, \beta) + Q(\left\lfloor \frac{n}{2} \right\rfloor, \alpha - \beta) \leq Q(n, \alpha) \qquad (1)$$

where the cost $h(\lceil \frac{n}{2} \rceil) = h(n) - h(min(\lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor))$ comes from the relative cost of `merge`. However, as observed in the RelCost paper, this inequality holds only when $\alpha > 0$. When $\alpha = 0$, the right hand side is 0 whereas the left hand side is $h(\lceil \frac{n}{2} \rceil)$. Nevertheless, when $\alpha = 0$, the two input lists do not differ at all, so the relative cost of `merge` can be trivially established as 0 using the **nochange** rule.

Consequently, the verification of `merge`'s body differs based on whether $\alpha \doteq 0$ or not. In our implementation, this case analysis is provided for by heuristic (2). As soon as the list l is introduced into the context, we apply the algorithmic analogue of the rule **rr-split**, introducing the two cases $\alpha \doteq 0$ and $\alpha > 0$. For the case $\alpha \doteq 0$, the heuristic immediately invokes the rule **alg-r-nochange-↓**, where we must show that all the free variables in the context have $\square$-ed types. Since we know that the functions `merge`, `bsplit` and `msort` all have $\square$-ed types, it remains to be shown that $\text{list}[n]^{\alpha}\ \tau_1 \sqsubseteq \square\,(\text{list}[n]^{\alpha}\ \tau_1)$. At this point, heuristic (3) kicks in and uses the subtyping rules **l2**, **l$\square$** (Figure 3) and the case-constraint $\alpha \doteq 0$ to establish this property. For the other case ($\alpha > 0$), we type the function body following its syntax. This eventually generates the satisfiable constraint (1) that is discharged by the constraint solver. The verification also uses heuristic (4) to typecheck the applications of `bsplit` and `merge` despite their $\square$-ed types.

7 Discussion

Our observation so far is that, to a large extent, bidirectional relational type systems follow the same broad principles as (the well-studied) bidirectional unary type systems. However, relational type systems usually have more nondeterminism in both typing and subtyping rules. This nondeterminism can be resolved by additional annotations, but an alternate approach is to use the annotations (and an elaboration to programs with these annotations) only as a theoretical tool in proving completeness of the bidirectional type system relative to a standard declarative one, and to use heuristics to resolve the nondeterminism in an implementation. This is the approach we follow here. Bidirectionality also extends to type-and-effect systems, with the general principle that the mode of the effect follows the mode of the type. In the following, we explain the extent to which some other features of relational type systems that we have not considered so far can be brought under the purview of bidirectional typing.

First, in the type systems we have considered so far, the *same* variable is used for related inputs in the two typed expressions. A more expressive alternative is to allow the two expressions to have different free variables and type them relative to assumptions about relations between the variables. Bidirectionality extends to this setup fairly easily, without any new principles. The rule for typing variables still remains in inference mode, but the variables on the two sides do not have to be the same.

Second, many relational type systems feature *asynchronous* rules that allow analysis of only one expression, while maintaining relational reasoning, e.g., [2, Section 3][5]. In fact, RelCost also has such rules [13, Section 3.4], which we did not show due to lack of space (our appendix shows these rules under the headings "Asynchronous typing"). These rules require care in a bidirectional type system. As a simple example, suppose we add to RelRefU the following rule, which allows relating $e_1\ e_2$ to e_2' by relating e_2 to e_2'.

$$\frac{|\Gamma|_1 \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 \backsim e_2' : U\,(A_1, A_2')}{\Gamma \vdash e_1\ e_2 \backsim e_2' : U\,(A_2, A_2')}$$

The question is what modes – inference or checking – the two premises and the conclusion of the rule's bidirectional version should follow? The usual principle for function application says that the first premise and the conclusion should have inferred types, and the second premise should have a checked type. However, with this choice, the type A_2' must be guessed in an implementation. Consequently, the standard principle does not work for this (and other) asynchronous rules. Here, one possible way is to infer the type in the first premise and check the types in the second premise and the conclusion. This deviation from the usual bidirectional principle for function application arises because the types of the two sides (A_2, A_2') are coupled in a single type in the conclusion, while the (asynchronous) rule analyzes the two sides differently. This suggests possible future work on bidirectional typing where the different components of a *single* type can have different modes. Here, for instance, we may say that in $U\,(A_2, A_2')$, A_2 will be inferred, while A_2' will be checked. Then, one could stay with the usual bidirectional principle for function application on the left side.

Finally, we note that for type systems with refinements or dependencies, any algorithmization (bidirectional or not) is limited by the tractability of the underlying logic of assertions. There are many relational refinement type systems that are designed for manual proofs and rely on very powerful assertion logics, in some cases as expressive as HOL or CiC [5, 31]. While bidirectional principles can be used to direct the generation of constraints in these cases, solving the constraints automatically is fundamentally intractable.

A particular case of the previous point is that in type systems with even simple quantitative refinements like list lengths and costs, the verification constraints often have existential quantifiers, which SMT solvers have significant difficulty handling. In our prototype implementation of BiRelCost, we use a custom algorithm to eliminate these quantifiers, which works well. An alternative approach, used for example in liquid types [42] and by Dunfield and Krishnaswami for GADTs [23], is to engineer or restrict the type system so that generated constraints do not have existential quantifiers. In the context of bidirectional type checking, this seems fundamentally difficult for quantitative *effects* like costs. The problem, as explained earlier, is that in trying to check a composite expression, the expression's given cost must be nondeterministically split between the subexpressions. Other (co)effect type systems [18, 19, 21, 27] use bounded exponentials $!_n\ \tau$ (n copies of τ) from bounded linear logic to express quantitative (co)effects. The difficulty for bidirectionality is similar: Now, in a rule with two or more premises, one must nondeterministically split the context. Further work is needed to understand whether these type systems can be re-designed to not generate existential quantifiers.

8 Related Work

There is a lot of literature on implementing various combinations of refinement types, effect systems, modal types, and subtyping. A distinctive feature of our work is that it combines all these aspects in a relational setting.

The idea of bidirectional type systems appeared in literature early on. However, the idea was popularized only more recently by Pierce and Turner [37]. The technique has shown great applicability—it has been used for dependent types [16], indexed refinement types [43, 44], intersection and union types [20, 24], higher-rank polymorphism [22, 23, 35], contextual modal types [36], algebraic effect handlers [29] and gradual typing [41]. Our approach is inspired by many of these papers, in particular DML [43, 44], but departs in the technical design of the algorithmic type system due to new challenges offered by relational and modal types, and costs. In particular, in these works is *unary*, i.e. a single program is checked (inferred) in isolation. Moreover, none of these works consider effects explicitly, i.e. as a type and effect system. One exception is the bidirectional effect system by [41], which uses bidirectional typechecking for gradual unary effects. However, their end goal is different since they infer minimal effects at compile time and then check dynamic effects at runtime.

Numerous other systems use lightweight dependent types for program verification including, for instance, F* [39, 40] and LiquidHaskell [42]. However, these developments also do not consider comonadic types and costs. The DML approach has also been used in combination with linear types for asymptotic complexity analysis [18, 19] and for reasoning about differential privacy [21, 27]. Besides lightweight dependent types, these papers also consider the comonadic modality of linear logic. This modality's structural properties are quite different from those of the comonadic $\square$ we consider here. Another way to extend dependent types with cost information is to index a monad with the execution cost. Gundry considers this approach in a unary setting for a subset of Haskell with support for bidirectional typechecking [28]. None of these papers consider relational typing.

Some other type systems establish relational properties of programs. For instance, Barthe et al. [6] consider a relational variant of a fragment of F* for the verification of cryptographic implementations, and similarly Barthe et al. [7] consider a relational refinement type system for differential privacy. However, some of the key technical challenges of our system, including those that arise from the interaction between unary and relational typing, as well as costs, do not show up in these settings. In the realm of incremental computing, some work [14, 15] has proposed declarative type systems for reasoning about update costs of incremental programs. These systems share similarities with the systems we considered here and we believe that the ideas developed in this paper can be applied to obtain algorithmic versions of these type systems as well.

Prior work has also studied methods of eliminating subtyping as a way of simplifying type checking, e.g. [11, 17]. While our approach is similar in motivation, our technical challenges are quite different. Main difficulties in simplifying subtyping in our work arise from the interaction of the modalities $\square$ and U with other connectives.

9 Conclusion

This paper presented a theoretical study and a concrete implementation of bidirectional type checking in a setting that combines relational refinements, comonadic types and relational effects. This rich setting poses unique challenges: The typing rules are not syntax-directed due to relational refinements; switching from relational to unary reasoning adds to the ambiguity; subtyping for (relational) comonads poses additional problems, as do the relational effects. We resolve these challenges through a process of elaboration and subtyping-elimination in the theory and using example-guided heuristics in the implementation. We validate experimentally that this approach is practical—it works for many different kinds of programs, has little annotation burden and typechecking is quick. Although we have focused here on a specific line of type systems with the features mentioned above, we believe that our work will help future designers of other relational type (and effect) systems as well.

Acknowledgments

We thank the anonymous referees for helpful comments and suggestions. This article is based on research that has been supported, in part, by the NSF under grant 1718220.

References

[1] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal Parametric Polymorphism. In *Proc. POPL*. 157–170.

[2] Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012).

[3] Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *PACMPL* 1, ICFP (2017), 33:1–33:30.

[4] Alejandro Aguirre, Gilles Barthe, Lars Birkedal, Ales Bizjak, Marco Gaboardi, and Deepak Garg. 2018. Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. In *Proc. ESOP*. 214–241.

[5] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *PACMPL* 1, ICFP (2017), 21:1–21:29.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. 2014. Probabilistic relational verification for cryptographic implementations. In *Proc. POPL*. 193–206.

[7] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proc. POPL*. 55–68.

[8] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2007. Lost in translation: Formalizing proposed extensions to C#. In *Proc. OOPSLA*. 479–498.

[9] François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. 2013. The Alt-Ergo automated theorem prover. http://alt-ergo.lri.fr/

[10] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proc. OOPSLA*. 183–200.

[11] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance As Implicit Coercion. *Inf. Comput.* 93, 1 (July 1991), 172–221.

[12] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Proc. ESOP*. 351–370.

[13] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proc. POPL*. 316–329.

[14] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Proc. ESOP*. 406–431.

[15] Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity With Control Flow Changes. In *Proc. ICFP*. 132–145.

[16] Thierry Coquand. 1996. An algorithm for type-checking dependent types. *Science of Computer Programming* 26, 1 (1996), 167 – 177.

[17] Karl Crary. 2000. Typed Compilation of Inclusive Subtyping. In *Proc. ICFP*. 68–81.

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

[19] Ugo Dal Lago and Barbara Petit. 2013. The Geometry of Types. In *Proc. POPL*. 167–178.

[20] Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proc. ICFP*. 198–208.

[21] Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proc. International Symposium on Implementation and Application of Functional Languages (IFL)*. 5:1–5:12.

[22] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *Proc. ICFP*. 429–442.

[23] Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *PACMPL* 3, POPL (2019), 9:1–9:28.

[24] Joshua Dunfield and Frank Pfenning. 2003. Type Assignment for Intersections and Unions in Call-by-value Languages. In *Proc. FOSSACS*. 250–266.

[25] Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Science of Computer Programming* 17, 1-3 (1991), 35–75.

[26] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proc. ESOP*. 125–128.

[27] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proc. POPL*. 357–370.

[28] Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

[29] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proc. POPL*. 500–514.

[30] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proc. POPL*. 47–57.

[31] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2011. Verification of Information Flow and Access Control Policies with Dependent Types. In *Proc. IEEE Symposium on Security and Privacy (S&P)*. 165–179.

[32] Flemming Nielson and HanneRiis Nielson. 1999. Type and Effect Systems. In *Correct System Design*. Lecture Notes in Computer Science, Vol. 1710. Springer-Verlag, 114–136.

[33] Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored Local Type Inference. In *Proc. POPL*. 41–53.

[34] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Proc. ICALP*. 385–397.

[35] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Funct. Program.* 17, 1 (2007), 1–82.

[36] Brigitte Pientka. 2008. A Type-theoretic Foundation for Programming with Higher-order Abstract Syntax and First-class Substitutions. In *Proc. POPL*. 371–382.

[37] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.

[38] François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Prog. Lang. Sys.* 25, 1 (Jan. 2003), 117–158.

[39] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proc. ICFP*. 266–278.

[40] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F^*. In *Proc. POPL*. 256–270.

[41] Matías Toro and Éric Tanter. 2015. Customizable Gradual Polymorphic Effects for Scala. In *Proc. OOPSLA*. 935–953.

[42] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with refinement types in the real world. In *Proc. ACM Symposium on Haskell*. 39–51.

[43] Hongwei Xi. 1998. *Dependent Types in Practical Programming*. Ph.D. Dissertation. Carnegie Mellon University.

[44] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proc. POPL*. 214–227.