A Relational Logic for Higher-Order Programs

ALEJANDRO AGUIRRE, IMDEA Software Institute, Spain
GILLES BARTHE, IMDEA Software Institute, Spain
MARCO GABOARDI, University at Buffalo, SUNY, USA
DEEPAK GARG, MPI-SWS, Germany
PIERRE-YVES STRUB, École Polytechnique, France

Relational program verification is a variant of program verification where one can reason about two programs and as a special case about two executions of a single program on different inputs. Relational program verification can be used for reasoning about a broad range of properties, including equivalence and refinement, and specialized notions such as continuity, information flow security or relative cost. In a higher-order setting, relational program verification can be achieved using relational refinement type systems, a form of refinement types where assertions have a relational interpretation. Relational refinement type systems excel at relating structurally equivalent terms but provide limited support for relating terms with very different structures.

We present a logic, called Relational Higher Order Logic (RHOL), for proving relational properties of a simply typed λ-calculus with inductive types and recursive definitions. RHOL retains the type-directed flavour of relational refinement type systems but achieves greater expressivity through rules which simultaneously reason about the two terms as well as rules which only contemplate one of the two terms. We show that RHOL has strong foundations, by proving an equivalence with higher-order logic (HOL), and leverage this equivalence to derive key meta-theoretical properties: subject reduction, admissibility of a transitivity rule and set-theoretical soundness. Moreover, we define sound embeddings for several existing relational type systems such as relational refinement types and type systems for dependency analysis and relative cost, and we verify examples that were out of reach of prior work.

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

Additional Key Words and Phrases: Relational Logic, Formal Verification, Refinement Types

ACM Reference format:
Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A Relational Logic for Higher-Order Programs. *Proc. ACM Program. Lang.* 1, ICFP, Article 21 (September 2017), 29 pages.
https://doi.org/10.1145/3110265

1 INTRODUCTION

Many important aspects of program behavior go beyond the traditional characterization of program properties as sets of traces [Alpern and Schneider 1985]. Hyperproperties [Clarkson and Schneider 2008] generalize properties and capture a larger class of program behaviors, by focusing on sets of sets of traces. As an intermediate point in this space, relational properties are sets of pairs of traces. Relational properties encompass many properties of interest, including program equivalence and refinement, as well as more specific notions such as non-interference and continuity.

2475-1421/2017/9-ART21
https://doi.org/10.1145/3110265

Relational verification is an instance of program verification that targets relational properties. Expectedly, standard verification methods such as type systems, program logics, and program analyses can be lifted to a relational setting. However, it remains a challenge to devise sufficiently powerful methods that can be used to verify a broad range of examples. In effect, most existing relational verification methods are limited in the examples that they can naturally verify, due to the fundamental tension between the syntax-directed nature of program verification, and the need to relate structurally different programs. Moreover, approaches to resolve this tension highly depend on the programming paradigm, on the class of program properties considered, and on the verification method. In the (arguably simplest) case of deductive verification of general properties of imperative programs, one approach to reduce this tension is to use self-composition [Barthe et al. 2004], which reduces relational verification to standard verification. However, reasoning about self-composed programs might be cumbersome. Alternatively, there exist expressive relational program logics that rely on an intricate set of rules to reason about a pair of programs. These logics combine two-sided rules, in which the two programs have the same top-level structure, and one-sided rules, which operate on a single program. Rules for loops are further divided into synchronous, in which both programs perform the same number of iterations, and asynchronous rules, that do not have this restriction but introduce more complexity [Barthe et al. 2017; Benton 2004].

In contrast, deductive verification of general properties of (pure) higher-order programs is less developed. One potential approach to solve the tension between the syntax-directedness, and the need to relate structurally different programs, is to reduce relational verification of pure higher-order programs to proofs in higher-order logic. There are strong similarities between this approach and self-composition: it reduces relational verification to standard verification, but this approach is very difficult to use in practice. A better alternative is to use relational refinement types such as rF* [Barthe et al. 2014], HOARe^2 [Barthe et al. 2015], DFuzz [Gaboardi et al. 2013] or RelCost [Çiçek et al. 2017]. Informally, relational refinement type systems use assertions to capture relationships between inputs and outputs of two higher-order programs. They are appealing for two reasons:

- They capture many important properties of programs in a direct and intuitive manner. For instance, the type $\{x :: \mathbb{N} \mid x_1 \leq x_2\} \to \{y :: \mathbb{N} \mid y_1 \leq y_2\}$ captures the set of pairs of functions that preserve the natural order on natural numbers, i.e. pairs of functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ such that for every $x_1, x_2 \in \mathbb{N}$, $x_1 \leq x_2$ implies $f_1(x_1) \leq f_2(x_2)$. (The subscripts 1 and 2 on a variable refer to its values in the two runs.)
- They can potentially benefit from a long and successful line of foundational [Dunfield and Pfenning 2004; Freeman and Pfenning 1991; Melliès and Zeilberger 2015; Xi and Pfenning 1999] and practical [Swamy et al. 2016; Vazou et al. 2014] research on refinement types.

Unfortunately, existing relational refinement type systems fail to support the verification of several examples. Broadly speaking, the two programs in a relational judgment may be required to have the same type and the same control flow; moreover, this requirement must be satisfied by their subprograms: if the two programs are applications, then the two sub-programs in argument position (resp. in function position) must have the same type and the same control flow; if the two programs are case expressions, they must enter the same branch, and their branches must themselves have the same control flow; if the two programs are recursive definitions, then their bodies must perform the same sequence of recursive calls; etc. This restriction, which can be found in more or less strict forms in the different relational type systems, limits the ability to carry fine-grained reasoning about terms that are structurally different. This raises the question whether the type-directed form of reasoning purported by refinement types can be reconciled with an expressive relational verification

of higher-order programs. We provide a positive answer for pure higher-order programs; extending our results to effectful programs is an important goal, but we leave it for future work.

Our starting point is the observation that relational refinement type systems are inherently restricted to reasoning about two structurally similar programs, because relational assertions are embedded into types. In order to provide broad support for one-sided rules (i.e., rules that contemplate only one of the two expressions), it is therefore necessary to consider relational assertions at the top-level, since one-sided rules have a natural formulation in this setting. Considering relational assertions at the top-level can be done in two different ways: either by supporting a rich theory of subtyping for relational refinement types, in such a way that each type admits a normal form where refinements only arise at the top-level, or simply by adapting the definitions and rules of refinement type systems so that only the top-level refinements are considered. Although both approaches are feasible, we believe that the second approach is more streamlined and leads to friendlier verification environments.

Contributions. We present a new logic, called Relational Higher Order Logic (RHOL, § 5), for reasoning about relational properties of higher-order programs written in a variant of Plotkin's PCF (§ 2). The logic manipulates judgments of the form:

$$\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$$

where Γ is a simply typed context, σ_1 and σ_2 are (possibly different) simple types, t_1 and t_2 are terms, Ψ is a set of assertions, and ϕ is an assertion. Our logic retains the type-directed nature of (relational) refinement type systems, and features typing rules for reasoning about structurally similar terms. However, disentangling types from assertions also makes it possible to define type-directed rules operating on a single term (left or right) of the judgment. This confers great expressivity to the logic, without significantly affecting its type-directed nature, and opens the possibility to alternate freely between two-sided and one-sided reasoning, as done in first-order imperative languages.

The validity of judgments is expressed relative to a set-theoretical semantics—our variant of PCF is restricted to terms which admit a set-theoretical semantics, including strongly normalizing terms. More precisely, a judgment $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$ is valid if for every valuation ρ (mapping variables in the context Γ to elements in the interpretation of their types), the interpretation of ϕ is true whenever the interpretation of (all the assertions in) Ψ is true. Soundness of the logic can be proved through a standard model-theoretic argument; however, we provide an alternative proof based on a sound and complete embedding into Higher-Order Logic (HOL, § 3). We leverage this equivalence to establish several meta-theoretical properties of the logic, notably subject reduction.

Moreover, we demonstrate that RHOL can be used as a general framework, by defining sound embedding for several relational type systems: relational refinement types (§ 6.2), the Dependency Core Calculus (DCC) for many dependency analyses, including those for information flow security (§ 6.3), and the RelCost (§ 6.4) type system for relative cost. The embedding of RelCost is particularly interesting, since it exercises the ability of our logic to alternate between synchronous and asynchronous reasoning. Finally, we verify several examples that go beyond the capabilities of previous systems (§ 7).

Related Work. While dependent type theory is the prevailing approach to reason about (pure) higher-order programs, several authors have explored another approach, which is crisply summarized by Jacobs [1999]: "A logic is always a logic over a type theory". Formalisms following this approach are defined in two stages; the first stage introduces a (dependent) type theory for writing programs, and the second stage introduces a predicate logic to reason about programs. This approach has been pursued independently in a series of works on logic-enriched type theories [Aczel

and Gambino 2000, 2006; Adams and Luo 2010; Belo 2007], and on refinement types Pfenning [2008]; Zeilberger [2016]. In the latter line of work, programs are written in an intrinsically typed λ-calculus à la Church; then, a system of sorts (a.k.a. refinements) is used to establish properties of programs typable in the first system. Our approach is similar; however, these works are developed in a unary setting, and do not consider the problem of relational verification. A further approach consists of developing a logic in an untyped setting, as is the case of LTC [Dybjer 1985].

Moreover, there is a large body of work on relational verification; we focus on type-based methods and deductive methods. Relational Hoare Logic [Benton 2004] and Relational Separation Logic [Yang 2007] are two program logics, respectively based on Hoare Logic and Separation Logic, for reasoning about relational properties of (first-order) imperative programs. These logics have been used for a broad range of examples and applications, ranging from program equivalence to compiler verification and information flow analysis. Moreover, they have been extended in several directions. For example, Probabilistic Relational Hoare Logic [Barthe et al. 2009] and approximate probabilistic Relational Hoare Logic [Barthe et al. 2012] are generalizations of Relational Hoare logic for reasoning about relational properties of (first-order) probabilistic programs. These logics have been used for a broad range of applications, including probabilistic information flow, side-channel security, proofs of cryptographic strength (reductionist security) and differential privacy. Cartesian Hoare Logic [Sousa and Dillig 2016] is also a recent generalization of Relational Hoare Logic for reasoning about bounded safety (i.e. k-safety for arbitrary but fixed k) properties of (first-order) imperative programs. This logic has been used for analyzing standard libraries. Experiments have demonstrated that such logics can be very effective in practice. Our formalism can be seen as a proposal to adapt their flexibility to pure higher-order programs.

Product programs [Barthe et al. 2011, 2004; Terauchi and Aiken 2005; Zaks and Pnueli 2008] are a general class of constructions that emulate the behavior of two programs and can be used for reducing relational verification to standard verification. While product programs naturally achieve (relative) completeness, they are often difficult to use since they require global reasoning on the obtained program—however recent works [Blatter et al. 2017] show how this approach can be automated in specific settings. Building product programs for (pure) higher-order languages is an intriguing possibility, and it might be possible to instrument RHOL using ideas from Barthe et al. [2017] to this effect; however, the product programs constructed in [Barthe et al. 2017] are a consequence, rather than a means, of relational verification.

Several type systems have been designed to support formal reasoning about relational properties for functional programs. Some of the earlier works in this direction have focused on the semantics foundations of parametricity, like the work by Abadi et al. [1993] on System R, a relational version of System F. The recent work by Ghani et al. [2016a] has further extended this approach to give better foundations to a combination of relational parametricity and impredicative polymorphism. Interestingly, similarly to RHOL, System R also supports relations between expressions at different types, although, since System R does not support refinement types, the only relations that System R can support are the parametric ones on polymorphic terms. In RHOL, we do not support parametric polymorphism à la System F currently but the relations that we support are more general. Adding parametric polymorphism will require foregoing the set-theoretical semantics, but it should still be possible to prove equivalence with a polymorphic variant of higher-order logic.

Several type systems have been proposed to reason about information flow security, a prime example of a relational property. Some examples include SLAM [Heintze and Riecke 1998], the type system underlying Flow Caml [Pottier and Simonet 2002] and DCC [Abadi et al. 1999]. Most of these type systems consider only one expression but they allow the use of information flow labels to specify relations between two different executions of the expression. As we show in this paper,

this approach can also be implemented in RHOL. We show how to translate DCC since it is one of the most general type systems; however, similar translations can also be provided for the other type systems.

Relational Hoare Type Theory (RHTT) [Nanevski et al. 2013; Stewart et al. 2013] is a formalism for relational reasoning about stateful higher-order programs. RHTT was designed to verify security properties like authorization and information flow policies but was used for the verification of hetergenous pointer data structures as well. RHTT uses a monad to separate stateful computations and relational refinements on the monadic type express relational pre- and post-conditions. RHTT supports reasoning about two different programs but the programs must have the same types at the top-level. RHTT's rules support both two- and one-sided reasoning similar to RHOL, but the focus of RHTT is on verifying properties of the program state. In particular, examples such as those in §7 or embeddings such as those in §6 were not considered in RHTT. RHTT is proved sound over a domain-theoretic model and continuity must be proven explicitly during the verification of recursive functions (rules are provided to prove continuity in many cases). In contrast, RHOL's set-theoretic model is simpler, but admits only those recursive functions that have a unique interpretation in set-theory.

Logical relations [Plotkin 1973; Statman 1985; Tait 1967] provide a fundamental tool for reasoning about programs. They have been used for a broad range of purposes, including proving unary properties (for instance strong normalization or complexity) and relational properties (for instance equivalence or information flow security). Our work can be understood as an attempt to internalize the versatility of relational logical relations in a syntactic framework. There is a large body of works on logic for logical relations, from the early works by Plotkin and Abadi [1993] to more recent works on logics for reasoning about states and concurrency by Ahmed, Birkedal, Dreyer, and collaborators among others [Dreyer et al. 2011, 2010; Jung et al. 2015; Krogh-Jespersen et al. 2017]. In particular, the IRIS logic [Jung et al. 2015] can be seen as a powerful reasoning framework for logical relations, as shown recently by Krogh-Jespersen et al. [2017] . Even if we also aim at internalize the logical relations, the goal of RHOL differ from the one of IRIS in the fact that we aim for syntax-driven relational verification.

We have already mentioned the works on relational refinement type systems for verifying cryptographic constructions [Barthe et al. 2014], for differential privacy [Barthe et al. 2015; Gaboardi et al. 2013] and for relational cost analysis [Çiçek et al. 2017]. This line of works is probably the most related to our work, however RHOL improves over all of them, as also shown by some of the embedding we give in Section 6. Another work related to this direction is the one by Asada et al. [2016]. This work proposes a technique to reduce relational refinement to standard first order refinements. Their technique is incomplete but it works well on some concrete examples. As we discussed before, we believe that some technique of this kind can be applied also to RHOL however this is orthogonal to our goal and we leave it for future investigations.

2 (A VARIANT OF) PCF

We consider a variant of PCF [Plotkin 1977] with booleans, natural numbers, lists and recursion, and recursive definitions. For the latter, we require that all recursive calls are performed on strictly smaller elements—as a consequence, the fixpoint equation derived from the definition has a unique set-theoretical solution. The precise method to enforce this requirement is orthogonal to our purposes, and could for instance be based on a syntactic guard predicate, or on sized types.

Types are defined by the grammar:

$$\tau, \sigma ::= \mathbb{B} \mid \mathbb{N} \mid \mathsf{list}_\tau \mid \tau \times \tau \mid \tau \to \tau$$

We let I range over inductive types.

Terms of the language are defined by the grammar:

$$t ::= x \mid \langle t, t\rangle \mid \pi_1\ t \mid \pi_2\ t \mid t\ t \mid \lambda x : \tau.t \mid c \mid S\ t \mid t :: t \mid \text{case } t \text{ of } 0 \mapsto t; S \mapsto t$$
$$\mid \text{case } t \text{ of } \mathsf{tt} \mapsto t; \mathsf{ff} \mapsto t \mid \text{case } t \text{ of } [] \mapsto t; \_ :: \_ \mapsto t \mid \text{letrec } f\ x = t$$

where x ranges over a set V of variables, c ranges over the set $\{\mathsf{tt}, \mathsf{ff}, 0, []\}$ of constants, and λ-abstractions are *à la* Church. The operational behavior of terms is captured by $\beta\iota\mu$-reduction $\to_{\beta\iota\mu} = \to_\beta \cup \to_\iota \cup \to_\mu$, where β-reduction, ι-reduction and μ-reduction are defined as the contextual closure of:

$$\begin{array}{rcl@{\qquad}l@{\qquad}rcl}
(\lambda x.t)\ u & \to_\beta & t[u/x] & \text{case } \mathsf{tt} \text{ of } \mathsf{tt} \mapsto u; \mathsf{ff} \mapsto v & \to_\iota & u \\
\pi_i\langle t_1, t_2\rangle & \to_\beta & t_i & \text{case } \mathsf{ff} \text{ of } \mathsf{tt} \mapsto u; \mathsf{ff} \mapsto v & \to_\iota & v \\
\text{case } 0 \text{ of } 0 \mapsto u; S \mapsto v & \to_\iota & u & \text{case } [] \text{ of } [] \mapsto u; _ :: _ \mapsto v & \to_\iota & u \\
\text{case } St \text{ of } 0 \mapsto u; S \mapsto v & \to_\iota & (v\ t) & \text{case } h :: t \text{ of } [] \mapsto u; _ :: _ \mapsto v & \to_\iota & (v\ h\ t)
\end{array}$$
$$(\text{letrec } f\ x = t)\ (C\ \vec{t}) \to_\mu t[C\ \vec{t}/x][\text{letrec } f\ x = t/f]$$

where $t[u/x]$ denotes the usual (capture-free) notion of substitution on terms (replace x by u in t). As usual, we let $=_{\beta\iota\mu}$ denote the reflexive-symmetric-transitive closure of $\to_{\beta\iota\mu}$. In particular, we only allow reduction of letrec when the argument has a constructor $C \in \{\mathsf{tt}, \mathsf{ff}, 0, S, [], ::\}$ in head position.

Judgments are of the form $\Gamma \vdash t : \tau$, where Γ is a set of typing declarations of the form $x : \sigma$, such that variables are declared at most once. The typing rules are standard, except for recursive functions. In this case, the typing rule requires that the domain of the recursive function is an inductive type (booleans, naturals, or lists here) and that the body of the recursive definition letrec $f\ x = e$ satisfies a predicate $\mathcal{D}ef(f, x, e)$ which ensures that all recursive calls are performed on smaller arguments. The typing rule for recursive definitions is thus:

$$\frac{\Gamma, f : I \to \sigma, x : I \vdash e : \sigma \qquad \mathcal{D}ef(f, x, e) \qquad I \in \{\mathbb{N}, \mathsf{list}_\tau\}}{\Gamma \vdash \text{letrec } f\ x\ = e : I \to \sigma}$$

The other rules are standard. We give set-theoretical semantics to this system. For each type τ, its interpretation $[\![\tau]\!]$ is the set of its values:

$$[\![\mathbb{B}]\!] \triangleq \mathbb{B} \quad [\![\mathbb{N}]\!] \triangleq \mathbb{N} \quad [\![\mathsf{list}_\tau]\!] \triangleq \mathsf{list}_{[\![\tau]\!]} \quad [\![\sigma \to \tau]\!] \triangleq [\![\sigma]\!] \to [\![\tau]\!]$$

where $[\![\sigma]\!] \to [\![\tau]\!]$ is the set of total functions with domain $[\![\sigma]\!]$ and codomain $[\![\tau]\!]$.

A valuation ρ for a context Γ (written $\rho \models \Gamma$) is a partial map such that $\rho(x) \in [\![\tau]\!]$ whenever $(x : \tau) \in \Gamma$. Given a valuation ρ for Γ, every term t such that $\Gamma \vdash t : \tau$ has an interpretation $(\!|t|\!)_\rho$:

$$(\!|x|\!)_\rho \triangleq \rho(x) \qquad (\!|\langle t, u\rangle|\!)_\rho \triangleq \langle (\!|t|\!)_\rho, (\!|u|\!)_\rho\rangle \qquad (\!|\pi_i\ t|\!)_\rho \triangleq \pi_i((\!|t|\!)_\rho)$$

$$(\!|\lambda x : \tau.t|\!)_\rho \triangleq \lambda v : [\![\tau]\!].(\!|t|\!)_{\rho[(\!|v|\!)_\rho/x]} \qquad (\!|c|\!)_\rho \triangleq c \qquad (\!|S\ t|\!)_\rho \triangleq S\ (\!|t|\!)_\rho \qquad (\!|t :: u|\!)_\rho \triangleq (\!|t|\!)_\rho :: (\!|u|\!)_\rho$$

$$(\!|\text{case } t \text{ of } [] \mapsto u; \_ :: \_ \mapsto v|\!)_\rho \triangleq \begin{cases} (\!|u|\!)_\rho & \text{if } (\!|t|\!)_\rho = [] \\ (\!|v|\!)_\rho\ M\ N & \text{if } (\!|t|\!)_\rho = M :: N \end{cases} \qquad (\!|\text{letrec } f\ x = t|\!)_\rho \triangleq F$$

In the case of letrec $f\ x = e$, we require that F is the unique solution of the fixpoint equation extracted from the recursive definition—existence and unicity of the solution follows from the validity of the $\mathcal{D}ef(f, x, e)$ predicate.

The interpretation of well-typed terms is sound. Moreover, the interpretation equates convertible terms. (This extends to η-conversion.)

Theorem 1 (Soundness of set-theoretic semantics)**.**

- If $\Gamma \vdash t : \tau$ and $\rho \models \Gamma$, then $(\!|t|\!)_\rho \in [\![\tau]\!]$.
- If $\Gamma \vdash t : \tau$ and $\Gamma \vdash u : \tau$ and $t =_{\beta\iota\mu} u$ and $\rho \models \Gamma$, then $(\!|t|\!)_\rho = (\!|u|\!)_\rho$.

3 HIGHER-ORDER LOGIC

Higher-Order Logic is defined as a calculus in natural deduction for a predicate logic over simply-typed terms. More specifically, its assertions are formulae over typed terms, and are defined by the following grammar:

$$\phi ::= P(t_1, \ldots, t_n) \mid \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \forall x : \tau.\phi \mid \exists x : \tau.\phi$$

where P ranges over basic predicates (as usual, we will often omit the types of bound variables, when clear from the context). We assume that predicates come equipped with an axiomatization. For instance, the predicate $\mathsf{All}(l, \lambda x.\phi)$ is defined to capture lists whose elements satisfies ϕ. This can be defined axiomatically:

$$\mathsf{All}([], \lambda x.\phi) \qquad \forall ht.\mathsf{All}(t, \lambda x.\phi) \Rightarrow \phi(h) \Rightarrow \mathsf{All}(h :: t, \lambda x.\phi)$$

We use the notation $\lambda x.\phi$ for simplicity, although we have not introduced formally a type for propositions—adding such a type is straightforward and orthogonal to our work: another alternative would be to use axiom scheme.

We define well-typed assertions using a judgment of the form $\Gamma \vdash \phi$. The typing rules are standard. A HOL judgment is then of the form $\Gamma \mid \Psi \vdash \phi$, where Γ is a simply typed context, Ψ is a set of assertions, and ϕ is an assertion, and such that $\Gamma \vdash \psi$ for every $\psi \in \Psi$, and $\Gamma \vdash \phi$. The rules of the logic are given in Figure 1, where the notation $\phi[t/x]$ denotes the (capture-free) substitution of x by t in ϕ. In addition to the usual rules for equality, implication and universal quantification, there are rules for inductive types (only the rules for lists are displayed; similar rules exist for booleans and natural numbers): the rule [LIST] models the induction principle for lists; the rules [NC] and [CONS] formalise injectivity and non overlap of constructors. A rule for strong induction [SLIST] can be considered as well, and is in fact derivable from simple induction.

$$\frac{\phi \in \Psi}{\Gamma \mid \Psi \vdash \phi}\ \mathsf{AX} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash t' : \tau \quad t =_{\beta\iota\mu} t'}{\Gamma \mid \Psi \vdash t = t'}\ \mathsf{CONV}$$

$$\frac{\Gamma \mid \Psi \vdash \phi[t/x] \quad \Gamma \mid \Psi \vdash t = u}{\Gamma \mid \Psi \vdash \phi[u/x]}\ \mathsf{SUBST} \qquad \frac{\Gamma \mid \Psi, \psi \vdash \phi}{\Gamma \mid \Psi \vdash \psi \Rightarrow \phi}\ \Rightarrow_{\mathsf{I}} \qquad \frac{\Gamma \mid \Psi \vdash \psi \Rightarrow \phi \quad \Gamma \mid \Psi \vdash \psi}{\Gamma \mid \Psi \vdash \phi}\ \Rightarrow_{\mathsf{E}}$$

$$\frac{\Gamma, x : \sigma \mid \Psi \vdash \phi}{\Gamma \mid \Psi \vdash \forall x : \sigma.\phi}\ \forall_{\mathsf{I}} \qquad \frac{\Gamma \mid \Psi \vdash \forall x : \sigma.\phi \quad \Gamma \vdash t : \sigma}{\Gamma \mid \Psi \vdash \phi[t/x]}\ \forall_{\mathsf{E}} \qquad \frac{}{\Gamma \mid \Psi \vdash \top}\ \top_{\mathsf{I}} \qquad \frac{\Gamma \mid \Psi \vdash \bot \quad \Gamma \vdash \phi}{\Gamma \mid \Psi \vdash \phi}\ \bot_{\mathsf{E}}$$

$$\frac{\Gamma \mid \Psi \vdash \phi[[]/t] \quad \Gamma, h : \sigma, t : \mathsf{list}_\sigma \mid \Psi, \phi \vdash \phi[h :: t/t]}{\Gamma \mid \Psi \vdash \forall t : \mathsf{list}_\sigma.\phi}\ \mathsf{LIST} \qquad \frac{\Gamma \vdash h :: t : \mathsf{list}_\tau}{\Gamma \mid \emptyset \vdash [] \neq h :: t}\ \mathsf{NC}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 :: t_2 = t_1' :: t_2'}{\Gamma \mid \Psi \vdash t_i = t_i'}\ \mathsf{CONS_i} \qquad \frac{\Gamma, t : \mathsf{list}_\tau \mid \Psi, \forall u : \mathsf{list}_\tau.|u| < |t| \Rightarrow \phi[u/t] \vdash \phi}{\Gamma \mid \Psi \vdash \forall t : \mathsf{list}_\tau.\phi}\ \mathsf{SLIST}$$

Fig. 1. Selected rules for HOL

Higher-Order Logic inherits a set-theoretical interpretation from its underlying simply-typed λ-calculus. We assume given for each predicate P an interpretation $[\![P]\!]$ which is compatible with

the type of P and its axioms. The interpretation of assertions is then defined in the usual way. Specifically, the interpretation $(\!|\phi|\!)_\rho$ of an assertion ϕ w.r.t. a valuation ρ includes the clauses:

$$(\!|P(t_1,\ldots,t_n)|\!)_\rho \triangleq ([\![t_1]\!]_\rho,\ldots,[\![t_n]\!]_\rho) \in [\![P]\!] \qquad (\!|\top|\!)_\rho \triangleq \tilde{\top} \qquad (\!|\bot|\!)_\rho \triangleq \tilde{\bot}$$

$$(\!|\phi_1 \wedge \phi_2|\!)_\rho \triangleq (\!|\phi_1|\!)_\rho \,\tilde{\wedge}\, (\!|\phi_2|\!)_\rho \qquad (\!|\phi_1 \Rightarrow \phi_2|\!)_\rho \triangleq (\!|\phi_1|\!)_\rho \,\tilde{\Rightarrow}\, (\!|\phi_2|\!)_\rho$$

$$(\!|\forall x : \tau.\phi|\!)_\rho \triangleq \tilde{\forall} v.v \in [\![\tau]\!] \,\tilde{\Rightarrow}\, (\!|\phi|\!)_{\rho[v/x]}$$

Higher-order logic is sound with respect to this semantics.

Theorem 2 (Soundness of set-theoretical semantics)**.** If $\Gamma \mid \Psi \vdash \phi$, then for every valuation $\rho \models \Gamma$, $\bigwedge_{\psi\in\Psi}(\!|\psi|\!)_\rho$ implies $(\!|\phi|\!)_\rho$.

In particular, higher-order logic is consistent, i.e. there is no derivation of $\Gamma \mid \emptyset \vdash \bot$ for any Γ.

4 UNARY HIGHER-ORDER LOGIC

As a stepping stone towards Relational Higher-Order Logic, we define Unary Higher-Order Logic (UHOL). UHOL retains the flavor of refinement types, but dissociates typing from assertions; judgments of UHOL are of the form:

$$\Gamma \mid \Psi \vdash t : \tau \mid \phi$$

where a distinguished variable $\mathbf{r}$, which doesn't appear in Γ, may appear (free) in ϕ as a synonym of t. A judgment is well-formed if t has type τ, Ψ is a valid set of assertions in the context Γ, and ϕ is a valid assertion in the context $\Gamma, \mathbf{r} : \tau$. Figure 2 presents selected typing rules. The [ABS] rule allows proving formulas that refer to λ-abstractions, expressing that if the argument satisfies a precondition ϕ', then the result satisfies a postcondition ϕ. The [APP] rule, dually, proves a condition ϕ on an application $t\ u$ provided that the argument u satisfies the precondition ϕ' of the function t. The [VAR] rule introduces a variable from the context with a formula proven in HOL. Rules for constants (e.g. [NIL]) work in the same way. Rule [CONS] proves a formula ϕ for a non-empty list, provided that ϕ is a consequence of some conditions ϕ', ϕ'' on its head and its tail. Rule [PAIR] allows the construction of judgments about pairs in a similar manner. The rules [PROJ$_i$] prove judgments about the projections of a pair. The rule [SUB] (subsumption) allows strengthening the assumed assertions Ψ and weakening the concluding assertion ϕ. It generates a HOL proof obligation. The rule [CASE] can be used for a case analysis over the constructor of a term. Finally, the rule [LETREC] supports inductive reasoning about recursive definitions. Recall that the domain of a recursive definition is an inductive type, for which a natural notion of size exists. If, assuming that a proposition holds for all elements smaller than the argument, we can prove that the proposition holds for the body, then the proposition must hold as well for the function. Furthermore, we require that the function we are verifying satisfies the predicate $\mathcal{D}ef(f,x,i)$, as was the case in HOL. The induction is performed over the $<$ order, which varies depending on the type of the argument.

We now discuss the main meta-theoretic results of UHOL. The following result establishes that every HOL judgment can be proven in UHOL and vice versa.

Theorem 3 (Equivalence with HOL)**.** For every context Γ, simple type σ, term t, set of assertions Ψ and assertion ϕ, the following are equivalent:

- $\Gamma \mid \Psi \vdash t : \sigma \mid \phi$
- $\Gamma \mid \Psi \vdash \phi[t/\mathbf{r}]$

$$\frac{\Gamma \vdash x : \sigma \qquad \Gamma \mid \Psi \vdash \phi[x/\mathbf{r}]}{\Gamma \mid \Psi \vdash x : \sigma \mid \phi}\ \text{VAR} \qquad \frac{\Gamma, x : \tau \mid \Psi, \phi' \vdash t : \sigma \mid \phi}{\Gamma \mid \Psi \vdash \lambda x : \tau . t : \tau \to \sigma \mid \forall x . \phi' \Rightarrow \phi[\mathbf{r}\ x/\mathbf{r}]}\ \text{ABS}$$

$$\frac{\Gamma \mid \Psi \vdash t : \tau \to \sigma \mid \forall x . \phi'[x/\mathbf{r}] \Rightarrow \phi[\mathbf{r}\ x/\mathbf{r}] \qquad \Gamma \mid \Psi \vdash u : \tau \mid \phi'}{\Gamma \mid \Psi \vdash t\ u : \sigma \mid \phi[u/x]}\ \text{APP} \qquad \frac{\Gamma \mid \Psi \vdash_{\mathsf{HOL}} \phi[[]/\mathbf{r}]}{\Gamma \mid \Psi \vdash [] : \mathsf{list}_\sigma \mid \phi}\ \text{NIL}$$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash h : \sigma \mid \phi' \qquad \Gamma \mid \Psi \vdash t : \mathsf{list}_\sigma \mid \phi'' \\ \Gamma \mid \Psi \vdash_{\mathsf{HOL}} \forall x y . \phi'[x/\mathbf{r}] \Rightarrow \phi''[y/\mathbf{r}] \Rightarrow \phi[x :: y/\mathbf{r}]\end{array}}{\Gamma \mid \Psi \vdash h :: t : \mathsf{list}_\sigma \mid \phi}\ \text{CONS} \qquad \frac{\Gamma \mid \Psi \vdash t : \sigma \times \tau \mid \phi[\pi_i(\mathbf{r})/\mathbf{r}]}{\Gamma \mid \Psi \vdash \pi_i(t) : \sigma \mid \phi}\ \text{PROJ}_\text{i}$$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash t : \sigma \mid \phi' \qquad \Gamma \mid \Psi \vdash u : \tau \mid \phi'' \\ \Gamma \mid \Psi \vdash_{\mathsf{HOL}} \forall x y . \phi'[x/\mathbf{r}] \Rightarrow \phi''[y/\mathbf{r}] \Rightarrow \phi[\langle x, y \rangle/\mathbf{r}]\end{array}}{\Gamma \mid \Psi \vdash \langle t, u \rangle : \sigma \times \tau \mid \phi}\ \text{PAIR}$$

$$\frac{\Gamma \mid \Psi \vdash t : \sigma \mid \phi' \quad \Gamma \mid \Psi \vdash_{\mathsf{HOL}} \phi'[t/\mathbf{r}] \Rightarrow \phi[t/\mathbf{r}]}{\Gamma \mid \Psi \vdash t : \sigma \mid \phi}\ \text{SUB}$$

$$\frac{\begin{array}{c}\Gamma \vdash l : \mathsf{list}_\tau \\ \Gamma \mid \Psi, l = [] \vdash u : \sigma \mid \phi \\ \Gamma \mid \Psi \vdash v : \tau \to \mathsf{list}_\tau \to \sigma \mid \forall h t . l = h :: t \Rightarrow \phi[\mathbf{r}\ h\ t/\mathbf{r}]\end{array}}{\Gamma \mid \Psi \vdash \mathsf{case}\ l\ \mathsf{of}\ [] \mapsto u; \_ :: \_ \mapsto v : \sigma \mid \phi}\ \text{LISTCASE}$$

$$\frac{\begin{array}{c}\mathcal{D}ef(f, x, e) \\ \Gamma, x : I, f : I \to \sigma \mid \Psi, \phi', \forall m . |m| < |x| \Rightarrow \phi'[m/x] \Rightarrow \phi[m/x][f\ m/\mathbf{r}] \vdash e : \sigma \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \mathsf{letrec}\ f\ x\ = e : I \to \sigma \mid \forall x . \phi' \Rightarrow \phi[\mathbf{r}\ x/\mathbf{r}]}\ \text{LETREC}$$

where $I \in \{\mathbb{N}, \mathsf{list}_\tau\}$

Fig. 2. Unary Higher-Order Logic rules

The forward implication follows by induction on the derivation of $\Gamma \mid \Psi \vdash t : \sigma \mid \phi$. The reverse implication is immediate from the rule [SUB] and the observation that $\Gamma \mid \Psi \vdash t : \sigma \mid \top$ whenever t is a term of type σ.

We lift the HOL semantics to UHOL. Terms, types and formulas are interpreted as before. Additionally, for every valuation ρ let $\rho[v/x]$ denote its unique extension ρ' such that $\rho'(y) = v$ if $x = y$ and $\rho'(y) = \rho(y)$ otherwise. The following corollary states the soundness of UHOL.

Corollary 4 (Set-theoretical soundness and consistency)**.** If $\Gamma \mid \Psi \vdash t : \sigma \mid \phi$, then for every valuation $\rho \models \Gamma$, $\bigwedge_{\psi \in \Psi} (\!|\psi|\!)_\rho$ implies $(\!|\phi|\!)_{\rho[(\!|t|\!)_\rho/\mathbf{r}]}$. In particular, there is no proof of $\Gamma \mid \emptyset \vdash t : \sigma \mid \bot$ in UHOL.

Next, we prove subject conversion for UHOL. The result follows immediately from Theorem 3 and subject conversion of HOL, which is itself a direct consequence of the [CONV] and [SUBST] rules.

Corollary 5 (Subject conversion)**.** Assume that $t =_{\beta\iota\mu} t'$ and $\Gamma \mid \Psi \vdash t : \sigma \mid \phi$. If $\Gamma \vdash t' : \sigma$ then $\Gamma \mid \Psi \vdash t' : \sigma \mid \phi$.

5 RELATIONAL HIGHER-ORDER LOGIC

Relational Higher-Order Logic (RHOL) extends UHOL's separation of assertions from types to a relational setting. Formally, RHOL is a relational type system which manipulates judgments of the form

$$\Gamma \mid \Psi \vdash t_1 : \tau_1 \sim t_2 : \tau_2 \mid \phi$$

which combine a typing judgment for a pair of PCF terms and permit reasoning about the relation between them. We therefore require that t_1 and t_2 respectively have types τ_1 and τ_2 in Γ. Well-formedness of the judgment requires Ψ to be a valid set of assertions in Γ and ϕ to be a valid assertion in $\Gamma, \mathbf{r}_1 : \tau_1, \mathbf{r}_2 : \tau_2$, where the special variables $\mathbf{r}_1$ and $\mathbf{r}_2$ are used as synonyms for t_1 and t_2 in ϕ. The informal meaning of the judgment is the expected one: If the variables in Γ are related by the assertions in Ψ, then the terms t_1 and t_2 are related by the assertion ϕ.

5.1 Proof Rules

The type system combines two-sided rules (Figure 3), which apply when the two terms have the same top-level constructors and one-sided rules (Figure 5), which analyze either one of the two terms. For instance, the [APP] rule applies when the two terms are applications, and requires that the functions t_1 and t_2 relate and the arguments u_1 and u_2 relate. Specifically, t_1 and t_2 must map values related by ϕ' to values related by ϕ, and u_1 and u_2 must be related by ϕ'. The [ABS] rule is dual. The [PAIR] rule requires that the left and right components of a pair relate independently (a stronger rule is discussed at the end of the section). The [PROJ$_i$] rules require in their premise an assertion that only refers to the the first or the second component of the pair. The rules for lists require that the two lists are either both empty, or both non-empty. The rule [CONS] requires that the two heads and the two tails relate independently. The [CASE] rule derives judgments about two case constructs when the terms over which the matching happens reduce to the same branch (i.e. have the same constructor) on both sides.

In contrast, one-sided typing rules only analyze one term; therefore, they come in two flavours: left rules (shown in Figure 5) and right rules (omitted but similar). Rule [ABS-L] considers the case where the left term is a λ-abstraction, and requires the body of the abstraction to be related to the right term u_2 whenever the argument (on the left side) satisfies a non-relational assertion ϕ'. Dually, rule [APP-L] considers the case where the left term is of the form $t_1\ u_1$, and t_1 is related to the right term u_2; specifically, t_1 should map every value satisfying ϕ' to a value satisfying ϕ. Moreover, u_1 should satisfy ϕ'. Since ϕ' is a non-relational assertion, we demand that it can be established using UHOL, not RHOL. One-sided rules for pairs and lists follow a similar pattern.

In addition, RHOL has structural rules (Figure 4). The rule [SUB] can be used for strengthening the assumed assertions and for weakening the concluding assertion; the ensuing side-conditions are discharged in HOL. Other structural rules assimilate rules of HOL. For instance, if we can prove two different assertions for the same terms we can prove the conjunction of the assertions ([$\wedge_I$]). Other logical connectives have similar rules. Finally, the rule [UHOL-L] (and a dual rule [UHOL-R]) allow falling back to UHOL in a RHOL proof.

Rules [LETREC] and [LETREC-L] introduce recursive function definitions (Figure 6). These rules allow for a style of reasoning very similar to strong induction. If, assuming that the function's specification holds for all smaller arguments, we can prove that the functions specification holds, then the specification must hold for all arguments. We require that the two functions we are relating satisfy the predicates $\mathcal{D}ef(f_i, x_i, e_i)$, as was the case in HOL and UHOL. The induction is performed over the order $(a, b) < (c, d)$, which holds whenever both $a \leq b$ and $c \leq d$, and at least one of the inequalities is strict.

$$\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2 \mid \Psi, \phi' \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}{\Gamma \mid \Psi \vdash \lambda x_1 : \tau_1.t_1 : \tau_1 \to \sigma_1 \sim \lambda x_2 : \tau_2.t_2 : \tau_2 \to \sigma_2 \mid \forall x_1, x_2.\phi' \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1][\mathbf{r}_2\ x_2/\mathbf{r}_2]}\ \text{ABS}$$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash t_1 : \tau_1 \to \sigma_1 \sim t_2 : \tau_2 \to \sigma_2 \mid \forall x_1, x_2.\phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1][\mathbf{r}_2\ x_2/\mathbf{r}_2] \\ \Gamma \mid \Psi \vdash u_1 : \tau_1 \sim u_2 : \tau_2 \mid \phi'\end{array}}{\Gamma \mid \Psi \vdash t_1 u_1 : \sigma_1 \sim t_2 u_2 : \sigma_2 \mid \phi[u_1/x_1][u_2/x_2]}\ \text{APP}$$

$$\frac{\Gamma \vdash x_1 : \sigma_1 \quad \Gamma \vdash x_2 : \sigma_2 \quad \Gamma \mid \Psi \vdash \phi[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash x_1 : \sigma_1 \sim x_2 : \sigma_2 \mid \phi}\ \text{VAR} \qquad \frac{\Gamma \mid \Psi \vdash_{\text{HOL}} \phi[\mathtt{tt}/\mathbf{r}_1][\mathtt{tt}/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash \mathtt{tt} : \mathbb{B} \sim \mathtt{tt} : \mathbb{B} \mid \phi}\ \text{TRUE}$$

$$\frac{\Gamma \mid \Psi \vdash_{\text{HOL}} \phi[[]/\mathbf{r}_1][[]/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash [] : \text{list}_{\sigma_1} \sim [] : \text{list}_{\sigma_2} \mid \phi}\ \text{NIL}$$

$$\frac{\begin{array}{cc}\Gamma \mid \Psi \vdash h_1 : \sigma_1 \sim h_2 : \sigma_2 \mid \phi' & \Gamma \mid \Psi \vdash t_1 : \text{list}_{\sigma_1} \sim t_2 : \text{list}_{\sigma_2} \mid \phi'' \\ \Gamma \mid \Psi \vdash_{\text{HOL}} \forall x_1 x_2 y_1 y_2.\phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi''[y_1/\mathbf{r}_1][y_2/\mathbf{r}_2] \Rightarrow \phi[x_1 :: y_1/\mathbf{r}_1][x_2 :: y_2/\mathbf{r}_2] & \end{array}}{\Gamma \mid \Psi \vdash h_1 :: t_1 : \text{list}_{\sigma_1} \sim h_2 :: t_2 : \text{list}_{\sigma_2} \mid \phi}\ \text{CONS}$$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash l_1 : \text{list}_{\tau_1} \sim l_2 : \text{list}_{\tau_2} \mid \mathbf{r}_1 = [] \Leftrightarrow \mathbf{r}_2 = [] \\ \Gamma \mid \Psi, l_1 = [], l_2 = [] \vdash u_1 : \sigma_1 \sim u_2 : \sigma_2 \mid \phi \\ \Gamma \mid \Psi \vdash v_1 : \tau_1 \to \text{list}_{\tau_1} \to \sigma_1 \sim v_2 : \tau_2 \to \text{list}_{\tau_2} \to \sigma_2 \mid \\ \forall h_1 h_2 t_1 t_2.l_1 = h_1 :: t_1 \Rightarrow l_2 = h_2 :: t_2 \Rightarrow \phi[\mathbf{r}_1\ h_1\ t_1/\mathbf{r}_1][\mathbf{r}_2\ h_2\ t_2/\mathbf{r}_2]\end{array}}{\Gamma \mid \Psi \vdash \text{case } l_1 \text{ of } [] \mapsto u_1; \_ :: \_ \mapsto v_1 : \sigma_1 \sim \text{case } l_2 \text{ of } [] \mapsto u_2; \_ :: \_ \mapsto v_2 : \sigma_2 \mid \phi}\ \text{LISTCASE}$$

$$\frac{\begin{array}{cc}\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi' & \Gamma \mid \Psi \vdash u_1 : \tau_1 \sim u_2 : \tau_2 \mid \phi'' \\ \Gamma \mid \Psi \vdash_{\text{HOL}} \forall x_1 x_2 y_1 y_2.\phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi''[y_1/\mathbf{r}_1][y_2/\mathbf{r}_2] \Rightarrow \phi[\langle x_1, y_1\rangle/\mathbf{r}_1][\langle x_2, y_2\rangle/\mathbf{r}_2] & \end{array}}{\Gamma \mid \Psi \vdash \langle t_1, u_1\rangle : \sigma_1 \times \tau_1 \sim \langle t_2, u_2\rangle : \sigma_2 \times \tau_2 \mid \phi}\ \text{PAIR}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \times \tau_1 \sim t_2 : \sigma_2 \times \tau_2 \mid \phi[\pi_i(\mathbf{r}_1)/\mathbf{r}_1][\pi_i(\mathbf{r}_2)/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash \pi_i(t_1) : \sigma_1 \sim \pi_i(t_2) : \sigma_2 \mid \phi}\ \text{PROJ}_\text{i}$$

Fig. 3. Two-sided rules

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi' \quad \Gamma \mid \Psi \vdash_{\text{HOL}} \phi'[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2] \Rightarrow \phi[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{SUB}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi \quad \Gamma \mid \Psi \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi'}{\Gamma \mid \Psi \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi \wedge \phi'}\ \wedge_\text{I}$$

$$\frac{\Gamma \mid \Psi, \phi'[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2] \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi}{\Gamma \mid \Psi \vdash t_1 : \sigma_2 \sim t_2 : \sigma_2 \mid \phi' \Rightarrow \phi}\ \Rightarrow_\text{I}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \mid \phi[\mathbf{r}/\mathbf{r}_1][t_2/\mathbf{r}_2] \qquad \Gamma \vdash t_2 : \sigma_2}{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{UHOL} - \text{L}$$

Fig. 4. Structural rules

$$\frac{\Gamma, x_1 : \tau_1 \mid \Psi, \phi' \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}{\Gamma \mid \Psi \vdash \lambda x_1 : \tau_1 . t_1 : \tau_1 \to \sigma_1 \sim t_2 : \sigma_2 \mid \forall x_1 . \phi' \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1]}\ \text{ABS–L}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \tau_1 \to \sigma_1 \sim u_2 : \sigma_2 \mid \forall x_1 . \phi'[x_1/\mathbf{r}_1] \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1] \quad \Gamma \mid \Psi \vdash u_1 : \tau_1 \mid \phi'}{\Gamma \mid \Psi \vdash t_1 u_1 : \sigma_1 \sim u_2 : \sigma_2 \mid \phi[u_1/x_1]}\ \text{APP–L}$$

$$\frac{\phi[x_1/\mathbf{r}_1] \in \Psi \quad \mathbf{r}_2 \notin FV(\phi) \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \mid \Psi \vdash x_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{VAR–L} \qquad \frac{\Gamma \mid \Psi \vdash \phi[[]/\mathbf{r}_1][t_2/\mathbf{r}_2] \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \mid \Psi \vdash [] : \mathsf{list}_{\sigma_1} \sim t_2 : \sigma_2 \mid \phi}\ \text{NIL–L}$$

$$\frac{\Gamma \mid \Psi \vdash h_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi' \qquad \Gamma \mid \Psi \vdash t_1 : \mathsf{list}_{\sigma_1} \sim t_2 : \sigma_2 \mid \phi'' \quad \Gamma \mid \Psi \vdash_{\mathsf{HOL}} \forall x_1 x_2 y_1 . \phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi''[y_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi[x_1 :: y_1/\mathbf{r}_1][x_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash h_1 :: t_1 : \mathsf{list}_{\sigma_1} \sim t_2 : \sigma_2 \mid \phi}\ \text{CONS–L}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{list}_\tau \quad \Gamma \mid \Psi, t_1 = [] \vdash u_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi \quad \Gamma \mid \Psi \vdash v_1 : \tau \to \mathsf{list}_\tau \to \sigma_1 \sim t_2 : \sigma_2 \mid \forall h_1 l_1 . t_1 = h_1 :: l_1 \Rightarrow \phi[\mathbf{r}_1\ h_1\ l_1/\mathbf{r}_1]}{\Gamma \mid \Psi \vdash \text{case } t_1 \text{ of } [] \mapsto u_1; \_ :: \_ \mapsto v_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{LISTCASE – L}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi' \qquad \Gamma \mid \Psi \vdash u_1 : \tau_1 \sim t_2 : \sigma_2 \mid \phi'' \quad \Gamma \mid \Psi \vdash_{\mathsf{HOL}} \forall x_1 x_2 y_1 . \phi'[x_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi''[y_1/\mathbf{r}_1][x_2/\mathbf{r}_2] \Rightarrow \phi[\langle x_1, y_1 \rangle/\mathbf{r}_1][x_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash \langle t_1, u_1 \rangle : \sigma_1 \times \tau_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{PAIR–L}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \sigma_1 \times \tau_1 \sim t_2 : \sigma_2 \mid \phi[\pi_1(\mathbf{r}_1)/\mathbf{r}_1]}{\Gamma \mid \Psi \vdash \pi_1(t_1) : \sigma_1 \sim t_2 : \sigma_2 \mid \phi}\ \text{PROJ}_1\text{–L}$$

Fig. 5. One-sided rules

$$\frac{\begin{array}{c}\mathcal{D}ef(f_1, x_1, e_1)\ \ \mathcal{D}ef(f_2, x_2, e_2) \\ \Gamma, x_1 : I_1, x_2 : I_2, f_1 : I_1 \to \sigma_1, f_2 : I_2 \to \sigma_2 \mid \\ \Psi, \phi', \forall m_1 m_2 . (|m_1|, |m_2|) < (|x_1|, |x_2|) \Rightarrow \phi'[m_1/x_1][m_2/x_2] \Rightarrow \\ \phi[m_1/x_1][m_2/x_2][f_1\ m_1/\mathbf{r}_1][f_2\ m_2/\mathbf{r}_2] \vdash e_1 : \sigma_1 \sim e_2 : \sigma_2 \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \begin{array}{l}\text{letrec } f_1\ x_1\ = e_1 : I_1 \to \sigma_1 \sim \\ \text{letrec } f_2\ x_2\ = e_2 : I_2 \to \sigma_2\end{array} \mid \forall x_1 x_2 . \phi' \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1][\mathbf{r}_2\ x_2/\mathbf{r}_2]}\ \text{LETREC}$$

$$\frac{\begin{array}{c}\mathcal{D}ef(f_1, x_1, e_1) \\ \Gamma, x_1 : I_1, f_1 : I_1 \to \sigma \mid \Psi, \phi', \forall m_1 . |m_1| < |x_1| \Rightarrow \phi'[m_1/x_1] \Rightarrow \\ \phi[m_1/x_1][f_1\ m_1/\mathbf{r}_1][t_2/\mathbf{r}_2] \vdash e_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \text{letrec } f_1\ x_1\ = e_1 : I_1 \to \sigma_2 \sim t_2 : \sigma_2 \mid \forall x_1 . \phi' \Rightarrow \phi[\mathbf{r}_1\ x_1/\mathbf{r}_1]}\ \text{LETREC – L}$$

where $I_1, I_2 \in \{\mathbb{N}, \mathsf{list}_\tau\}$

Fig. 6. Recursion rules

5.2 Discussion

Our choice of the rules is guided by the practical considerations of being able to verify examples easily, without specifically aiming for minimality or exhaustiveness. In fact, many of our rules can be derived from others, or reduced to a more elementary form. For instance:

$$\frac{\Gamma \mid \Psi \vdash t_1 : \tau_1 \to \sigma_1 \sim t_2 : \tau_2 \to \sigma_2 \mid \phi[\mathbf{r}_1\ u_1/\mathbf{r}_1][\mathbf{r}_2\ u_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash t_1\ u_1 : \sigma_1 \sim t_2\ u_2 : \sigma_2 \mid \phi}\ \text{APP}-\text{FUN}$$

$$\frac{\Gamma \mid \Psi \vdash u_1 : \tau_1 \sim u_2 : \tau_2 \mid \phi[t_1\ \mathbf{r}_1/\mathbf{r}_1][t_2\ \mathbf{r}_2/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash t_1\ u_1 : \sigma_1 \sim t_2\ u_2 : \sigma_2 \mid \phi}\ \text{APP}-\text{ARG}$$

$$\frac{\Gamma \mid \Psi \vdash t_1 : \tau_1 \sim t_2 : \tau_2 \mid \phi[\langle \mathbf{r}_1, u_1\rangle/\mathbf{r}_1][\langle \mathbf{r}_2, u_2\rangle/\mathbf{r}_2]}{\Gamma \mid \Psi \vdash \langle t_1, u_1\rangle : \tau_1 \times \sigma_1 \sim \langle t_2, u_2\rangle : \tau_2 \times \sigma_2 \mid \phi}\ \text{PAIR}-\text{FST}$$

$$\frac{\begin{array}{c}\Gamma \mid \Psi \vdash t_1 : \mathsf{list}_{\tau_1} \sim t_2 : \mathsf{list}_{\tau_2} \mid \top \\ \Gamma \mid \Psi, t_1 = [], t_2 = [] \vdash u_1 : \sigma_1 \sim u_2 : \sigma_2 \mid \phi \\ \Gamma \mid \Psi, t_2 = [] \vdash v_1 : \tau_1 \to \mathsf{list}_{\tau_1} \to \sigma_1 \sim u_2 : \sigma_2 \mid \forall h_1 l_1.t_1 = h_1 :: l_1 \Rightarrow \phi[\mathbf{r}_1\ h_1\ l_1/\mathbf{r}_1] \\ \Gamma \mid \Psi, t_1 = [] \vdash u_1 : \sigma_1 \sim v_2 : \tau_2 \to \mathsf{list}_{\tau_2} \to \sigma_2 \mid \forall h_2 l_2.t_2 = h_2 :: l_2 \Rightarrow \phi[\mathbf{r}_2\ h_2\ l_2/\mathbf{r}_2] \\ \Gamma \mid \Psi \vdash v_1 : \tau_1 \to \mathsf{list}_{\tau_1} \to \sigma_1 \sim v_2 : \tau_2 \to \mathsf{list}_{\tau_2} \to \sigma_2 \mid \\ \forall h_1 h_2 l_1 l_2.t_1 = h_1 :: l_1 \Rightarrow t_2 = h_2 :: l_2 \Rightarrow \phi[\mathbf{r}_1\ h_1\ l_1/\mathbf{r}_1][\mathbf{r}_2\ h_2\ l_2/\mathbf{r}_2]\end{array}}{\Gamma \mid \Psi \vdash \mathsf{case}\ t_1\ \mathsf{of}\ [] \mapsto u_1; \_ :: \_ \mapsto v_1 : \sigma_1 \sim \mathsf{case}\ t_2\ \mathsf{of}\ [] \mapsto u_2; \_ :: \_ \mapsto v_2 : \sigma_2 \mid \phi}\ \text{LLCASE}-\text{A}$$

Fig. 7. Some derived rules

- The structural rules to reason about logical connectives, such as $[\wedge_I]$, can be derived by induction on the length of derivations with the help of [SUB].
- The [VAR-L] (similarly, [NIL-L]) rule can be weakened, without affecting the strength of the system,

$$\frac{\phi[x_1/\mathbf{r}_1] \in \Psi \quad \mathbf{r}_2 \notin FV(\phi)}{\Gamma \mid \Psi \vdash x_1 : \sigma_1 \sim x_2 : \sigma_2 \mid \phi}\ \text{VAR}-\text{L}$$

- The premise of the [VAR] rule (and similarly, [NIL]) can be changed to $\phi[x/\mathbf{r}] \in \Psi$. We can recover the original rule by one application of [SUB].
- The rules [APP-FUN] and [APP-ARG] in Figure 7 (adapted from Ghani et al. [2016b]) can be derived from the rule [APP]. To derive [APP-FUN], instantiate ϕ' to $\mathbf{r}_1 = u_1 \wedge \mathbf{r}_2 = u_2$ in [APP]. To derive [APP-ARG], we have to prove a trivial condition $\forall x_1 x_2.\phi[t_1\ x_1/\mathbf{r}_1][t_2\ x_2/\mathbf{r}_2] \Rightarrow \phi[t_1\ x_1/\mathbf{r}_1][t_2\ x_2/\mathbf{r}_2]$ on t_1, t_2.
- The [PAIR-FST] and [PAIR-SND] rules in Figure 7 can be derived in a similar way. These rules overcome a limitation of the original [PAIR] rule, namely, that the relations for the two components of the pair must be independent. [PAIR-FST] and [PAIR-SND] allow relating, for instance, pairs of integers $\langle m_1, n_1\rangle$ and $\langle m_2, n_2\rangle$ such that $m_1 + n_1 = m_2 + n_2$.
- The [LLCASE-A] rule can be used to reason about case constructs when the terms over which we discriminate do not necessarily reduce to the same branch. It is equivalent to applying [CASE-L] followed by [CASE-R].

5.3 Meta-theory

RHOL retains the expressiveness of HOL, as formalized in the following theorem.

Theorem 6 (Equivalence with HOL)**.** For every context Γ, simple types σ_1 and σ_2, terms t_1 and t_2, set of assertions Ψ and assertion ϕ, if $\Gamma \vdash t_1 : \sigma_1$ and $\Gamma \vdash t_2 : \sigma_2$, then the following are equivalent:

- $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$
- $\Gamma \mid \Psi \vdash \phi[t_1/\mathbf{r}_1][t_2/\mathbf{r}_2]$

The proof of the forward implication proceeds by induction on the structure of derivations. The proof of the reverse implication is immediate from the rule [SUB] and the observation that $\Gamma \mid \emptyset \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \top$ whenever t_1 and t_2 are typable terms of types σ_1 and σ_2 respectively.

This immediately entails soundness of RHOL, which is expressed in the following result:

Corollary 7 (Set-theoretical soundness and consistency)**.** If $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$, then for every valuation $\rho \models \Gamma$, $\bigwedge_{\psi \in \Psi} (\!|\psi|\!)_\rho$ implies $(\!|\phi|\!)_{\rho[(\!|t_1|\!)_\rho/\mathbf{r}_1],[(\!|t_2|\!)_\rho/\mathbf{r}_2]}$. In particular, there is no proof of $\Gamma \mid \emptyset \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \bot$ for any Γ.

The equivalence also entails subject conversion (and as special cases subject reduction and subject expansion). This follows immediately from subject conversion of HOL (which, as stated earlier, is itself a direct consequence of the [CONV] and [SUBST] rules).

Corollary 8 (Subject conversion)**.** Assume that $t_1 =_{\beta\iota\mu} t_1'$ and $t_2 =_{\beta\iota\mu} t_2'$ and $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$. If $\Gamma \vdash t_1' : \sigma_1$ and $\Gamma \vdash t_2' : \sigma_2$ then $\Gamma \mid \Psi \vdash t_1' : \sigma_1 \sim t_2' : \sigma_2 \mid \phi$.

Another useful consequence of the equivalence is the admissibility of the transitivity rule.

Corollary 9 (Admissibility of transitivity rule)**.** Assume that:

- $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi$
- $\Gamma \mid \Psi \vdash t_2 : \sigma_2 \sim t_3 : \sigma_3 \mid \phi'$

Then, $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_3 : \sigma_3 \mid \phi[t_2/\mathbf{r}_2] \wedge \phi'[t_2/\mathbf{r}_1]$.

Finally, we prove an embedding lemma for UHOL. The proof can be carried by induction on the structure of derivations, or using the equivalence between UHOL and HOL (Theorem 3).

Lemma 10 (Embedding lemma)**.** Assume that:

- $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \mid \phi$
- $\Gamma \mid \Psi \vdash t_2 : \sigma_2 \mid \phi'$

Then $\Gamma \mid \Psi \vdash t_1 : \sigma_1 \sim t_2 : \sigma_2 \mid \phi[\mathbf{r}_1/\mathbf{r}] \wedge \phi'[\mathbf{r}_2/\mathbf{r}]$.

The embedding is reminiscent of the approach of Beringer and Hofmann [2007] to encode information flow properties in Hoare logic.

6 EMBEDDINGS

In this section, we establish the expressiveness of RHOL and UHOL by embedding several existing refinement type systems (3 relational and 1 non-relational) from a variety of domains. All embeddings share the common idea of *separating* the simple typing information from the more fine-grained refinement information in the translation. We use uniform notation to represent similar ideas across the different embeddings. In particular, we use vertical bars $|\cdot|$ to denote the erasure of a type into a simple type, and floor bars $\lfloor\cdot\rfloor$ to denote the embedding of the refinement of a type in a HOL formula.

For the clarity of exposition, we often present fragments or variants of systems that appear in the literature, notably excluding recursive functions that do not satisfy our well-definedness predicate. Moreover, the embeddings are given for a version of RHOL à la Curry, in which λ-abstractions do not carry the type of their bound variable.

6.1 Refinement Types

Refinement types [Freeman and Pfenning 1991; Swamy et al. 2016; Vazou et al. 2014] are a variant of simple types where for every basic type τ, there is a type $\{x : \tau \mid \phi\}$ which is inhabited by the elements t of τ that satisfy the logical assertion $\phi[t/x]$. This includes dependent refinements

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \preceq \tau} \qquad \frac{\Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \tau_2 \preceq \tau_3}{\Gamma \vdash \tau_1 \preceq \tau_3} \qquad \frac{\Gamma \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash \mathsf{list}_{\tau_1} \preceq \mathsf{list}_{\tau_2}} \qquad \frac{\Gamma \vdash \{x : \tau \mid \phi\}}{\Gamma \vdash \{x : \tau \mid \phi\} \preceq \tau}$$

$$\frac{\Gamma \vdash \tau \preceq \sigma \qquad \Gamma, \mathbf{r} : \tau \vdash \phi}{\Gamma \vdash \tau \preceq \{x : \sigma \mid \phi\}} \qquad \frac{\Gamma \vdash \sigma_2 \preceq \sigma_1 \qquad \Gamma, x : \sigma_2 \vdash \tau_1 \preceq \tau_2}{\Gamma \vdash \Pi(x : \sigma_1).\tau_1 \preceq \Pi(x : \sigma_2).\tau_2}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \Pi(x : \tau).\sigma} \qquad \frac{\Gamma \vdash t_1 : \Pi(x : \tau).\sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \sigma[t_2/x]}$$

$$\frac{\Gamma \vdash t : \mathsf{list}_\tau \qquad \Gamma \vdash t_1 : \sigma \qquad \Gamma \vdash t_2 : \Pi(h : \tau).\Pi(l : \mathsf{list}_\tau).\sigma}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ [] \mapsto t_1; \_ :: \_ \mapsto t_2 : \sigma} \qquad \frac{\Gamma \vdash \tau \preceq \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash t : \sigma}$$

$$\frac{\Gamma, x : \tau, f : \Pi(\{y : \tau \mid y < x\}).\sigma[y/x] \vdash t : \sigma \qquad \mathcal{D}ef(f, x, t)}{\Gamma \vdash \mathsf{letrec}\ f\ x = t : \Pi(x : \tau).\sigma}$$

Fig. 8. Refinement types rules (subtyping and typing)

$\Pi(x : \tau).\sigma$, in which the variable x is also bound in the refinement formulas appearing in σ. Here we present a simplified variant of these systems. (Refined) types are defined by the grammar

$$\tau \ := \ \mathbb{B} \mid \mathbb{N} \mid \mathsf{list}_\tau \mid \{x : \tau \mid \phi\} \mid \Pi(x : \tau).\tau$$

As usual, we shorten $\Pi(x : \tau).\sigma$ to $\tau \to \sigma$ if $x \notin FV(\sigma)$. We also shorten bindings of the form $x : \{x : \tau \mid \phi\}$ to $\{x : \tau \mid \phi\}$. Typing rules are presented in Figure 8; note that the [LETREC] rule requires that recursive definitions satisfy the well-definedness predicate. Judgments of the form $\Gamma \vdash \tau$ are well-formedness judgments. Judgments of the form $\Gamma \vdash \phi$ are logical judgments; we omit a formal description of the rules, but assume that the logic of assertions is consistent with HOL, i.e. $\Gamma \vdash \phi$ implies $|\Gamma| \mid \lfloor \Gamma \rfloor \vdash \phi$, where the erasure functions are defined below.

This system can be embedded into UHOL in a straightforward manner. The embedding highlights the relation between these two systems, i.e. between logical assertions embedded in the types (as in refinement types) and logical assertions at the top-level, separate from simple types (as in UHOL). The intuitive idea behind the embedding is therefore to separate refinement assertions from types. Specifically, from every refinement type we can obtain a simple type by repeatedly extracting the type τ from $\{x : \tau \mid \phi\}$. We will denote this extraction by the translation function $|\tau|$:

$$|\mathbb{B}| \triangleq \mathbb{B} \qquad |\mathbb{N}| \triangleq \mathbb{N} \qquad |\mathsf{list}_\tau| \triangleq \mathsf{list}_{|\tau|} \qquad |\{x : \tau \mid \phi\}| \triangleq |\tau| \qquad |\Pi(x : \tau).\sigma| \triangleq |\tau| \to |\sigma|$$

Since $|\tau|$ loses refinement information, we define a second translation that extracts the refinement as a logical predicate over a variable x that names the typed expression. This second translation is written $\lfloor \tau \rfloor(x)$.

$$\lfloor \mathbb{B} \rfloor(x), \lfloor \mathbb{N} \rfloor(x) \triangleq \top \qquad \lfloor \mathsf{list}_\tau \rfloor(x) \triangleq \mathrm{All}(x, \lambda y. \lfloor \tau \rfloor(y)) \qquad \lfloor \{y : \tau \mid \phi\} \rfloor(x) \triangleq \lfloor \tau \rfloor(x) \wedge \phi[x/y]$$

$$\lfloor \Pi(x : \tau).\sigma \rfloor(x) \triangleq \forall y. \lfloor \tau \rfloor(y) \Rightarrow \lfloor \sigma \rfloor(xy)$$

The refinement of simple types is trivial. If t is an expression of type $\{x : \tau \mid \phi\}$, then t must satisfy both the refinement formula ϕ and the refinement of τ. A function with a refinement type can be

interpreted in two different ways: 1) As a map whose domain is the domain type restricted to its (the type's) refinement, or 2) As a map whose domain is the entire domain type (disregarding the refinement), but whose result satisfies the co-domain's refinement only if the argument satisfies the domain's refinement. We use the second interpretation, while some prior work uses the first. Therefore, if t is an expression of type $\Pi(x : \tau).\sigma$, then it must be the case that for every x satisfying the refinement of τ, $(t\ x)$ satisfies the refinement of σ.

The refinement of a list uses the predicate All, which as defined in §3, means that all elements of a list satisfy a given formula.

The syntax of assertions and expressions is exactly the same as in HOL, and therefore there is no need for a translation. Embedding of types can be lifted to contexts in the natural way.

$$|x : \tau, \Gamma| \triangleq x : |\tau|, |\Gamma| \qquad\qquad \lfloor x : \tau, \Gamma \rfloor \triangleq \lfloor \tau \rfloor(x), \lfloor \Gamma \rfloor$$

To encode judgments, all that remains is to put the previous definitions together. The main result about embedding typing judgments is the following:

Theorem 11. If $\Gamma \vdash t : \tau$ is derivable in the refinement type system, then $|\Gamma| \mid \lfloor \Gamma \rfloor \vdash t : |\tau| \mid \lfloor \tau \rfloor(\mathbf{r})$ is derivable in UHOL.

The proof is performed by induction on the structure of derivations, using as helper result the embedding of subtyping judgments into HOL. Since it can be proven by induction that, whenever $\tau \preceq \sigma$, the type extractions $|\tau|$ and $|\sigma|$ coincide, all that needs to be checked is that $\lfloor \sigma \rfloor$ is a consequence of $\lfloor \tau \rfloor$. This is captured by the following statement.

Theorem 12. If $\Gamma \vdash \tau \preceq \sigma$ is derivable in a refinement type system, then $|\Gamma|, x : |\tau| \mid \lfloor \Gamma \rfloor, \lfloor \tau \rfloor(x) \vdash \lfloor \sigma \rfloor(x)$ is derivable in HOL.

Soundness of refinement types w.r.t. the set-theoretic semantics follows immediately from Theorem 11 and the set-theoretic soundness of UHOL (Corollary 4).

6.2 Relational Refinement Types

Relational refinement types [Barthe et al. 2014, 2015] are a variant of refinement types that can be used to express relational properties via a syntax of the form $\{\mathbf{r} :: \tau \mid \phi\}$ where ϕ is a relational assertion—i.e. it may contain a left and right copy of $\mathbf{r}$, which are denoted as $\mathbf{r}_1$ and $\mathbf{r}_2$ respectively, as well as a left and a right copy of every variable in the context. In this section, we introduce a simple relational refinement type system and establish a type-preserving translation to RHOL—we compare with existing type systems at the end of the paragraph.

The syntax of relational refinement types is given by the grammar:

$$\tau ::= \mathbb{B} \mid \mathbb{N} \mid \tau \to \tau$$
$$T, U ::= \tau \mid \mathsf{list}_T \mid \Pi(x :: T).\, U \mid \{x :: T \mid \phi\}$$

Relational refinement types are naturally ordered by a subtyping relation $\Gamma \vdash T \preceq U$, where Γ is a sequence of variables declarations of the form $x :: U$.

Typing judgments are of the form $\Gamma \vdash t_1 \sim t_2 :: T$. We present selected typing rules in Figure 9. Note that the form of judgments requires that t_1 and t_2 must have the same simple type, and the typing rules require that t_1 and t_2 have the same structure[1]. In the [CASELIST] rule, we require that both terms reduce to the same branch; the case rule for natural numbers is similar. The

[1]The typing rules displayed in the figure will in fact force t_1 and t_2 to be the same term modulo renaming. This is not the case in existing relational refinement type systems; however, rules that introduce different terms on the right and on the left are limited, since both terms still need to have the same type and most one-sided rules break this invariant. For instance, in [Barthe et al. 2014] there is a rule similar to [LLCASE-A], and a rule for reducing one of the terms of a judgment.

$$\text{VAR-RT}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x_1 \sim x_2 :: T} \qquad \text{ABS-RT}\ \frac{\Gamma, x :: T \vdash u_1 \sim u_2 :: U}{\Gamma \vdash \lambda x_1.u_1 \sim \lambda x_2.u_2 :: \Pi(x :: T).\, U}$$

$$\text{APP-RT}\ \frac{\Gamma \vdash t_1 \sim t_2 :: \Pi(x :: T).\, U \qquad \Gamma \vdash u_1 \sim u_2 :: T}{\Gamma \vdash t_1\ u_1 \sim t_2\ u_2 :: U[u_1/x_1][u_2/x_2]} \qquad \text{NIL}\ \frac{\Gamma \vdash T}{\Gamma \vdash [] \sim [] :: \mathsf{list}_T}$$

$$\text{CONS}\ \frac{\Gamma \vdash h_1 \sim h_2 :: T \qquad \Gamma \vdash t_1 \sim t_2 :: \mathsf{list}_T}{\Gamma \vdash h_1 :: t_1 \sim h_2 :: t_2 :: \mathsf{list}_T} \qquad \textsc{Sub}\ \frac{\Gamma \vdash t_1 \sim t_2 :: T \qquad \Gamma \vdash T \preceq U}{\Gamma \vdash t_1 \sim t_2 :: U}$$

$$\text{LETREC-RT}\ \frac{\begin{array}{c}\Gamma, x :: T, f :: \Pi(\{y :: T \mid (y_1, y_2) < (x_1, x_2)\}).U[y/x] \vdash t_1 \sim t_2 :: U \\ \Gamma \vdash \Pi(x :: T).\, U \qquad \mathcal{D}ef(f, x, t)\end{array}}{\Gamma \vdash \text{letrec } f_1\ x_1 = t_1 \sim \text{letrec } f_2\ x_2 = t_2 :: \Pi(x :: T).\, U}$$

$$\text{CASELIST}\ \frac{\begin{array}{c}\Gamma \vdash t_1 \sim t_2 :: \mathsf{list}_\tau \\ \Gamma \vdash t_1 = [] \Leftrightarrow t_2 = [] \qquad \Gamma \vdash u_1 \sim u_2 :: T \qquad \Gamma \vdash v_1 \sim v_2 :: \Pi(h :: \tau).\, \Pi(t :: \mathsf{list}_\tau).\, T\end{array}}{\Gamma \vdash \text{case } t_1 \text{ of } [] \mapsto u_1; \_ :: \_ \mapsto v_1 \sim \text{case } t_2 \text{ of } [] \mapsto u_2; \_ :: \_ \mapsto v_2 :: T}$$

Fig. 9. Relational Typing (Selected Rules)

[LETREC] rule uses (a straightforward adaptation of) the $\mathcal{D}ef(f, x, t)$ predicate from our simply-typed language, and requires that the two recursive definitions perform exactly the same recursive calls.

Subtyping rules are the same as in the unary case, and therefore we refer to Figure 8 for them (allowing their instantiation for relational types T, U as well as unary types σ, τ).

The embedding of refinement types into UHOL can be adapted to the relational setting. From each relational refinement type T we can extract a simple type $|T|$. On the other hand, we can erase every relational refinement type T into a relational formula $[\![T]\!]$, which is parametrized by two expressions and defined as follows:

$$[\![\mathsf{list}_\tau]\!](x_1, x_2) \triangleq \bigwedge_{i \in \{1,2\}} \mathrm{All}(x_i, \lambda y.\lfloor \tau \rfloor(y)) \qquad [\![\mathsf{list}_T]\!](x_1, x_2) \triangleq \mathrm{All2}(x_1, x_2, \lambda y_1.\lambda y_2.[\![T]\!](y_1, y_2))$$

$$[\![\{y : \tau \mid \phi\}]\!](x_1, x_2) \triangleq \bigwedge_{i \in \{1,2\}} \lfloor \tau \rfloor(x_i) \wedge \phi[x_i/y]$$

$$[\![\{y :: T \mid \phi\}]\!](x_1, x_2) \triangleq [\![T]\!](x_1, x_2) \wedge \phi[x_1/y_1][x_2/y_2]$$

$$[\![\Pi(y : \tau).\sigma]\!](x) \triangleq \bigwedge_{i \in \{1,2\}} \forall y.\lfloor \tau \rfloor(y) \Rightarrow \lfloor \sigma \rfloor(xy)$$

$$[\![\Pi(y :: T).\, U]\!](x_1, x_2) \triangleq \forall y_1 y_2.[\![T]\!](y_1, y_2) \Rightarrow [\![\sigma]\!](x_1 y_1,\ x_2 y_2)$$

The predicate All2 relates two lists elementwise and is defined axiomatically:

$$\mathrm{All2}([],[],\lambda x_1.\lambda x_2.\phi)$$

$$\forall h_1 h_2 t_1 t_2.\mathrm{All}(t_1,t_2,\lambda x_1.\lambda x_2.\phi) \Rightarrow \phi(h_1,h_2) \Rightarrow \mathrm{All}(h_1 :: t_1, h_2 :: t_2, \lambda x_1.\lambda x_2.\phi)$$

To extend the embedding to contexts, we need to duplicate every variable in them:

$$|x :: T, \Gamma| \triangleq x_1, x_2 : |T|, |\Gamma| \qquad \llbracket x :: T, \Gamma \rrbracket \triangleq \llbracket T \rrbracket(x_1, x_2), \llbracket \Gamma \rrbracket$$

Now we state the main result:

Theorem 13 (Soundness of embedding)**.** If $\Gamma \vdash t_1 \sim t_2 :: T$, then $|\Gamma| \mid \llbracket \Gamma \rrbracket \vdash t_1 : |T| \sim t_2 : |T| \mid \llbracket T \rrbracket(\mathbf{r}_1, \mathbf{r}_2)$ Also, if $\Gamma \vdash T \preceq U$ then $|\Gamma|, x_1, x_2 : |T| \mid \llbracket \Gamma \rrbracket, \llbracket T \rrbracket(x_1, x_2) \vdash \llbracket U \rrbracket(x_1.x_2)$.

Proof. The proof proceeds by induction on the structure of derivations. □

Soundness of relational refinement types w.r.t. set-theoretical semantics follows immediately from Theorem 13 and the set-theoretical soundness of RHOL (Corollary 7).

Corollary 14 (Soundness of relational refinement types)**.** If $\Gamma \vdash t_1 \sim t_2 :: T$, then for every valuation $\theta \models \Gamma$ we have $(\langle\!| t_1 |\!\rangle_\theta, \langle\!| t_2 |\!\rangle_\theta) \in \langle\!| T |\!\rangle_\theta$.

6.3 Dependency Core Calculus

The Dependency Core Calculus (DCC) [Abadi et al. 1999] is a higher-order calculus with a type system that tracks data dependencies. DCC was designed as a unifying framework for dependency analysis and it was shown that many other calculi for information flow analysis [Heintze and Riecke 1998; Volpano et al. 1996], binding-time analysis [Hatcliff and Danvy 1997], and program slicing, all of which track dependencies, can be embedded in DCC. Here, we show how a fragment of DCC can be embedded into RHOL. Transitively, the corresponding fragments of all the aforementioned calculi can also be embedded in RHOL. (The fragment of DCC we consider excludes recursive functions. DCC admits general recursive functions, while our definition of RHOL only admits a subset of these. Extending the embedding to recursive functions admitted by RHOL is not difficult.)

DCC is an extension of the simply typed lambda-calculus with a monadic type family $\mathbb{T}_\ell(\tau)$, indexed by *labels* ℓ, which are elements of a lattice. Unlike other uses of monads, DCC's monad does not encapsulate any effects. Instead, its only goal is to track dependence. The type system forces that the result of an expression of type $\mathbb{T}_\ell(\tau)$ can *depend* on an expression of type $\mathbb{T}_{\ell'}(\tau')$ only if $\ell' \sqsubseteq \ell$ in the lattice. Dually, if $\ell' \not\sqsubseteq \ell$, then even if an expression e of type $\mathbb{T}_\ell(\tau)$ mentions a variable x of type $\mathbb{T}_{\ell'}(\tau')$, then e's result must be *independent* of the substitution provided for x during evaluation.

For simplicity and without any loss of generality, we consider here only a two point lattice $\{L, H\}$ with $L \sqsubset H$. The syntax of DCC's types and expressions is shown below. We use e to denote DCC expressions, to avoid confusion with HOL's expressions.

$$\begin{array}{lcl} \tau & ::= & \mathbb{B} \mid \tau \to \tau \mid \tau \times \tau \mid \mathbb{T}_\ell(\tau) \\ e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{tt} \mid \mathsf{ff} \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{tt} \mapsto e_t; \mathsf{ff} \mapsto e_f \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \mid \eta_\ell(e) \\ & & \mid \mathsf{bind}(e_1, x.e_2) \end{array}$$

Here, $\eta_\ell(e)$ and $\mathsf{bind}(e_1, x.e_2)$ are respectively the return and bind constructs for the monad $\mathbb{T}_\ell(\tau)$. Typing rules for these two constructs are shown below. Typing rules for the remaining constructs are the standard ones.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \eta_\ell(e) : \mathbb{T}_\ell(\tau)} \qquad \frac{\Gamma \vdash e_1 : \mathbb{T}_\ell(\tau_1) \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \tau_2 \searrow \ell}{\Gamma \vdash \mathsf{bind}(e_1, x.e_2) : \tau_2}$$

The crux of the dependency tracking is the relation $\tau_2 \searrow \ell$ in the premise of the rule for bind. The relation, read "τ_2 protected at level ℓ" and defined below, informally means that all primitive (boolean) values extractable from e_2 are protected by a monadic construct of the form $\mathbb{T}_{\ell'}(\tau)$, with $\ell \sqsubseteq \ell'$. Hence, the rule forces that the result obtained by eliminating the type $\mathbb{T}_\ell(\tau_1)$ flow only into types protected at ℓ in this sense.

$$\frac{\ell \sqsubseteq \ell'}{\mathbb{T}_{\ell'}(\tau) \searrow \ell} \qquad \frac{\tau \searrow \ell}{\mathbb{T}_{\ell'}(\tau) \searrow \ell} \qquad \frac{\tau_1 \searrow \ell \qquad \tau_2 \searrow \ell}{\tau_1 \times \tau_2 \searrow \ell} \qquad \frac{\tau_2 \searrow \ell}{\tau_1 \to \tau_2 \searrow \ell}$$

This fragment of DCC has a relational set-theoretic interpretation. For every type τ, we define a carrier set $|\tau|$:

$$|\mathbb{B}| \triangleq \mathbb{B} \qquad |\tau_1 \to \tau_2| \triangleq |\tau_1| \to |\tau_2| \qquad |\tau_1 \times \tau_2| \triangleq |\tau_1| \times |\tau_2| \qquad |\mathbb{T}_\ell(\tau)| \triangleq |\tau|$$

Next, every type τ is interpreted as a lattice-indexed family of relations $\lfloor \tau \rfloor_a \subseteq |\tau| \times |\tau|$. The role of the lattice element a is that it defines what can be *observed* in the system. Specifically, an expression of type $\mathbb{T}_\ell(\tau)$ can be observed only if $\ell \sqsubseteq a$. When $\ell \not\sqsubseteq a$, expressions of type $\mathbb{T}_\ell(\tau)$ look like "black-boxes". Technically, we force $\lfloor \mathbb{T}_\ell(\tau) \rfloor_a = |\tau| \times |\tau|$ when $\ell \not\sqsubseteq a$. DCC's typing rules are sound with respect to this model. The soundness implies that if $\ell \not\sqsubseteq \ell'$ and $x : \mathbb{T}_\ell(\mathbb{B}) \vdash e : \mathbb{T}_{\ell'}(\mathbb{B})$, then for $e_1, e_2 : \mathbb{T}_\ell(\mathbb{B})$, $e[e_1/x]$ and $e[e_2/x]$ are equal booleans in the set-theoretic model. This result, called noninterference, formalizes that DCC's dependency tracking is correct.

To translate DCC to RHOL, we actually embed this set-theoretic model in RHOL. We start by defining an erasing translation, $|\tau|$, from DCC's types into RHOL's simple types. This translation is exactly the same as the definition of carrier sets shown above, except that we treat $\times$ and $\to$ as RHOL's syntactic type constructs instead of set-theoretic constructs. Next, we define an erasure of terms:

$$|\mathsf{tt}| \triangleq \mathsf{tt} \qquad |\mathsf{ff}| \triangleq \mathsf{ff} \qquad |\mathsf{case}\ e\ \mathsf{of}\ \mathsf{tt} \mapsto e_t; \mathsf{ff} \mapsto e_f| \triangleq \mathsf{case}\ |e|\ \mathsf{of}\ \mathsf{tt} \mapsto |e_t|; \mathsf{ff} \mapsto |e_f| \qquad |x| \triangleq x$$

$$|\lambda x.e| \triangleq \lambda x.|e| \qquad |e_1\ e_2| \triangleq |e_1|\ |e_2| \qquad |\langle e_1, e_2\rangle| \triangleq \langle |e_1|, |e_2|\rangle \qquad |\pi_1(e)| \triangleq \pi_1(|e|)$$

$$|\pi_2(e)| \triangleq \pi_2(|e|) \qquad |\eta_\ell(e)| \triangleq |e| \qquad |\mathsf{bind}(e_1, x.e_2)| \triangleq (\lambda x.|e_2|)\ |e_1|$$

It is fairly easy to see that if $\vdash e : \tau$ in DCC, then $\vdash |e| : |\tau|$. Next, we define the lattice-indexed family of relations $\lfloor \tau \rfloor_a$ in HOL. For technical convenience, we write the relations as logical assertions, indexed by variables x, y representing the two terms to be related.

$$\lfloor \mathbb{B} \rfloor_a(x, y) \triangleq (x = \mathsf{tt} \wedge y = \mathsf{tt}) \vee (x = \mathsf{ff} \wedge y = \mathsf{ff})$$

$$\lfloor \tau_1 \to \tau_2 \rfloor_a(x, y) \triangleq \forall v, w.\, \lfloor \tau_1 \rfloor_a(v, w) \Rightarrow \lfloor \tau_2 \rfloor_a(x\ v, y\ w)$$

$$\lfloor \tau_1 \times \tau_2 \rfloor_a(x, y) \triangleq \lfloor \tau_1 \rfloor_a(\pi_1(x), \pi_1(y)) \wedge \lfloor \tau_2 \rfloor_a(\pi_2(x), \pi_2(y))$$

$$\lfloor \mathbb{T}_\ell(\tau) \rfloor_a(x, y) \triangleq \begin{cases} \lfloor \tau \rfloor_a(x, y) & \ell \sqsubseteq a \\ \top & \ell \not\sqsubseteq a \end{cases}$$

The most important clause is the last one: When $\ell \not\sqsubseteq a$, any two x, y are in the relation $\lfloor \mathbb{T}_\ell(\tau) \rfloor_a$. This generalizes to all protected types in the following sense.

Lemma 15. If $\ell \not\sqsubseteq a$ and $\tau \searrow \ell$, then $\vdash \forall x, y.(\lfloor \tau \rfloor_a(x, y) \equiv \top)$ in HOL.

The translations extend to contexts as follows:

$$|x^1 : \tau_1, \ldots, x^n : \tau_n| \triangleq x_1^1 : |\tau_1|, x_2^1 : |\tau_1|, \ldots, x_1^n : |\tau_n|, x_2^n : |\tau_n|$$
$$\lfloor x^1 : \tau_1, \ldots, x^n : \tau_n \rfloor_a \triangleq \lfloor \tau_1 \rfloor_a(x_1^1, x_2^1), \ldots, \lfloor \tau_n \rfloor_a(x_1^n, x_2^n)$$

The following theorem states that the whole translation is sound: It preserves well-typedness. In the statement of the theorem, $|e|_1$ and $|e|_2$ replace each variable x in $|e|$ with x_1 and x_2, respectively.

Theorem 16 (Soundness of embedding). If $\Gamma \vdash e : \tau$ in DCC, then for all $a \in \{L, H\}$: $|\Gamma| \mid \lfloor \Gamma \rfloor_a \vdash |e|_1 : |\tau| \sim |e|_2 : |\tau| \mid \lfloor \tau \rfloor_a(\mathbf{r}_1, \mathbf{r}_2)$.

DCC's noninterference theorem is a corollary of this theorem and the soundness of RHOL in set theory.

6.4 Relational Cost

RelCost [Çiçek et al. 2017] is a relational refinement type-and-effect system designed to reason about relative cost—the difference in the execution costs of two similar programs or of two runs of the same program on two different inputs. RelCost combines reasoning about the maximum and minimum costs of a single program with relational reasoning about the relative cost of two programs. RelCost is based on the observation that relational reasoning about structurally related expressions can improve precision in reasoning about the relative cost, but if this approach fails one can always fall back to establishing an upper bound on the relative cost the difference of the maximum cost of one program and the minimum cost of the other. Here, we show how a fragment of RelCost can be embedded into RHOL. Similar to what we did for DCC, to just convey the main intuition, we consider a fragment of RelCost excluding recursive functions. The syntax of RelCost is based on two sorts of types:

$$\begin{array}{rcll} A & ::= & \mathbb{N} \mid \mathsf{list}_A[n] \mid A \xrightarrow{\mathrm{exec}(k,l)} A \mid \forall i \overset{\mathrm{exec}(k,l)}{::} S.\, A & \text{(unary types)} \\ \tau & ::= & \mathbb{N}_r \mid \mathsf{list}_\tau[n]^\alpha \mid \tau \xrightarrow{\mathrm{diff}(k)} \tau \mid \forall i \overset{\mathrm{diff}(k)}{::} S.\, \tau \mid UA \mid \square\, \tau & \text{(relational types)} \end{array}$$

Unary types are used to type one program and they are mostly standard except for the effect annotation $\mathrm{exec}(k, l)$ on arrow types and universally quantified types representing the min and max cost k and l of the body of the closure, respectively. Relational types ascribe two programs, so they are interpreted as pairs of expressions. In relational types, arrow types and universally quantified types have an effect annotation $\mathrm{diff}(k)$ representing the relative cost k of the two closures. Besides, the superscript α refines list types with the number of elements that can differ in two lists. The type UA is the weakest relation over elements of the unary type A, i.e. it can be used to type two arbitrary terms, while the type $\square\, \tau$ is the diagonal subrelation of τ, i.e. it can be used to type only two terms that are equal. There are two kinds of typing judgments, unary and relational:

$$\Delta; \Phi; \Omega \vdash_k^l t : A \qquad \Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim l : \tau$$

The unary judgment states that the execution cost of t is lower bounded by k and upper bounded by l, and the expression t has the unary type A. The relational judgment states that the relative cost of t_1 with respect to t_2 is upper bounded by l and the two expressions have the relational type τ. Here Ω is a unary type environment, Γ is a relational type environment, Δ is an environment for index variables and Φ for assumed constraints over the index terms. Figure 10 shows selected rules.

To embed RelCost in RHOL, we define a monadic-style cost-instrumented translation of RelCost types. The translation is given in two-steps: First, we define an erasure of cost and size information into simple types and then we define a cost-passing style translation of simple types with a value-translation and an expression-translation. The erasure function is defined as follows:

$$|\mathbb{N}| \triangleq |\mathbb{N}_r| \triangleq \mathbb{N} \qquad |\mathsf{list}_A[n]| \triangleq |\mathsf{list}_A[n]^\alpha| \triangleq \mathsf{list}_{|A|} \qquad |UA| \triangleq |\square\, A| \triangleq |A|$$

$$|\forall i \overset{\mathrm{exec}(k,l)}{::} S.\, A| \triangleq |\forall i \overset{\mathrm{diff}(k)}{::} S.\, A| \triangleq \mathbb{N} \to |A| \qquad |A \xrightarrow{\mathrm{diff}(k)} B| \triangleq |A \xrightarrow{\mathrm{exec}(k,l)} B| \triangleq |A| \to |B|$$

$$\textsc{var}\ \frac{\Omega(x) = A}{\Delta; \Phi_a; \Omega \vdash^0_0 x : A} \qquad \textsc{const}\ \frac{}{\Delta; \Phi_a; \Omega \vdash^0_0 \mathsf{n} : \mathsf{int}} \qquad \textsc{lam}\ \frac{\Delta; \Phi_a; x : A_1, \Omega \vdash^l_k t : A_2}{\Delta; \Phi_a; \Omega \vdash^0_0 \lambda x.t : A_1 \xrightarrow{\mathrm{exec}(k,l)} A_2}$$

$$\textsc{app}\ \frac{\Delta; \Phi_a; \Omega \vdash^{l_1}_{k_1} t_1 : A_1 \xrightarrow{\mathrm{exec}(k,l)} A_2 \qquad \Delta; \Phi_a; \Omega \vdash^{l_2}_{k_2} t_2 : A_1}{\Delta; \Phi_a; \Omega \vdash^{l_1+l_2+l+c_{app}}_{k_1+k_2+k+c_{app}} t_1\ t_2 : A_2}$$

$$\textsc{r-var}\ \frac{\Gamma(x) = \tau}{\Delta; \Phi_a; \Gamma \vdash x \ominus x \lesssim 0 : \tau} \qquad \textsc{r-const}\ \frac{}{\Delta; \Phi_a; \Gamma \vdash \mathsf{n} \ominus \mathsf{n} \lesssim 0 : \mathrm{int}_r}$$

$$\textsc{r-cons1}\ \frac{\Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \lesssim l_1 : \tau \qquad \Delta; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \lesssim l_2 : \mathrm{list}_\tau[n]^\alpha}{\Delta; \Phi_a; \Gamma \vdash \mathrm{cons}(t_1, t_2) \ominus \mathrm{cons}(t_1', t_2') \lesssim l_1 + l_2 : \mathrm{list}_\tau[n+1]^{\alpha+1}}$$

$$\textsc{r-cons2}\ \frac{\Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \lesssim l_1 : \Box\tau \qquad \Delta; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \lesssim l_2 : \mathrm{list}_\tau[n]^\alpha}{\Delta; \Phi_a; \Gamma \vdash \mathrm{cons}(t_1, t_2) \ominus \mathrm{cons}(t_1', t_2') \lesssim l_1 + l_2 : \mathrm{list}_\tau[n+1]^\alpha}$$

$$\textsc{r-caseL}\ \frac{\begin{array}{c}\Delta; \Phi_a; \Gamma \vdash t \ominus t' \lesssim l : \mathrm{list}_\tau[n]^\alpha \qquad \Delta; \Phi_a \wedge n = 0; \Gamma \vdash t_1 \ominus t_1' \lesssim l' : \tau' \\ i, \Delta; \Phi_a \wedge n = i+1; h : \Box\tau, tl : \mathrm{list}_\tau[i]^\alpha, \Gamma \vdash t_2 \ominus t_2' \lesssim l' : \tau' \\ i, \beta, \Delta; \Phi_a \wedge n = i+1 \wedge \alpha = \beta+1; h : \tau, tl : \mathrm{list}_\tau[i]^\beta, \Gamma \vdash t_2 \ominus t_2' \lesssim l' : \tau'\end{array}}{\Delta; \Phi_a; \Gamma \vdash \begin{array}{rcl}\mathrm{case}\ t\ \mathrm{of\ nil} & \to & t_1 \\ \mid h :: tl & \to & t_2\end{array} \ominus \begin{array}{rcl}\mathrm{case}\ t'\ \mathrm{of\ nil} & \to & t_1' \\ \mid h :: tl & \to & t_2'\end{array} \lesssim l + l' : \tau'}$$

$$\textsc{r-lam}\ \frac{\Delta; \Phi_a; x : \tau_1, \Gamma \vdash t_1 \ominus t_2 \lesssim l : \tau_2}{\Delta; \Phi_a; \Gamma \vdash \lambda x.t_1 \ominus \lambda x.t_2 \lesssim 0 : \tau_1 \xrightarrow{\mathrm{diff}(l)} \tau_2} \qquad \textsc{r-app}\ \frac{\Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_1' \lesssim l_1 : \tau_1 \xrightarrow{\mathrm{diff}(l)} \tau_2 \qquad \Delta; \Phi_a; \Gamma \vdash t_2 \ominus t_2' \lesssim l_2 : \tau_1}{\Delta; \Phi_a; \Gamma \vdash t_1\ t_2 \ominus t_1'\ t_2' \lesssim l_1 + l_2 + l : \tau_2}$$

$$\textsc{r-iLam}\ \frac{\begin{array}{c}i :: S, \Delta; \Phi_a; \Gamma \vdash t \ominus t' \lesssim l : \tau \\ i \notin \mathrm{FIV}(\Phi_a; \Gamma)\end{array}}{\Delta; \Phi_a; \Gamma \vdash \Lambda t \ominus \Lambda t' \lesssim 0 : \forall i \overset{\mathrm{diff}(l)}{::} S.\ \tau}$$

$$\textsc{r-iApp}\ \frac{\begin{array}{c}\Delta; \Phi_a; \Gamma \vdash t \ominus t' \lesssim l : \forall i \overset{\mathrm{diff}(l')}{::} S.\ \tau \\ \Delta \vdash J : S\end{array}}{\Delta; \Phi_a; \Gamma \vdash t[\,] \ominus t'[\,] \lesssim l + l'[J/i] : \tau\{J/i\}}$$

$$\textsc{nochange}\ \frac{\begin{array}{c}\Delta; \Phi_a; \Gamma \vdash t \ominus t \lesssim l : \tau \\ \forall x \in dom(\Gamma).\ \Delta; \Phi_a \models \Gamma(x) \sqsubseteq \Box\Gamma(x)\end{array}}{\Delta; \Phi_a; \Gamma, \Gamma'; \Omega \vdash t \ominus t \lesssim 0 : \Box\tau}$$

$$\textsc{switch}\ \frac{\Delta; \Phi_a; \overline{\Gamma} \vdash^{l_1}_{k_1} t_1 : A \qquad \Delta; \Phi_a; \overline{\Gamma} \vdash^{l_2}_{k_2} t_2 : A}{\Delta; \Phi_a; \Gamma \vdash t_1 \ominus t_2 \lesssim l_1 - k_2 : U\,A}$$

Fig. 10. RelCost Unary and Relational Typing (Selected Rules)

$$(\!|x|\!) \triangleq (x, 0) \qquad (\!|\lambda x.t|\!) \triangleq (\lambda x.(\!|t|\!), 0) \qquad (\!|\Lambda.t|\!) \triangleq (\lambda\_.(\!|t|\!), 0)$$

$$(\!|tu|\!) \triangleq \mathsf{let}\ x = (\!|t|\!)\ \mathsf{in}\ \mathsf{let}\ y = (\!|u|\!)\ \mathsf{in}\ \mathsf{let}\ z = \pi_1(x)\ \pi_1(y)\ \mathsf{in}\ (\pi_1(z), \pi_2(x) + \pi_2(y) + \pi_2(z) + c_{app})$$

$$(\!|t[]|\!) \triangleq \mathsf{let}\ x = (\!|t|\!)\ \mathsf{in}\ \mathsf{let}\ y = \pi_1(x)\ 0\ \mathsf{in}\ (\pi_1(y), \pi_2(x) + \pi_2(y) + c_{iapp}) \qquad (\!|\mathsf{nil}|\!) \triangleq (\mathsf{nil}, 0)$$

$$(\!|\mathsf{cons}(t_1, t_2)|\!) \triangleq \mathsf{let}\ x = (\!|t_1|\!)\ \mathsf{in}\ \mathsf{let}\ y = (\!|t_2|\!)\ \mathsf{in}\ (\pi_1(x) :: \pi_1(y), \pi_2(x) + \pi_2(y))$$

$$(\!|\ \mathsf{case}\ t'\ \mathsf{of}\ \mathsf{nil}\ \to t_1' \mid h :: tl \to t_2'|\!) \triangleq \begin{cases} \mathsf{let}\ x = (\!|t'|\!)\ \mathsf{in}\ \ \mathsf{case}\ \pi_1(x)\ \mathsf{of} \\ \mathsf{nil}\ \ \to \mathsf{let}\ y = (\!|t_1'|\!)\ \mathsf{in}\ (\pi_1(y), \pi_2(x) + \pi_2(y) + c_{case}) \\ \ \mid h :: tl \to \mathsf{let}\ y = (\!|t_2'|\!)\ \mathsf{in}\ (\pi_1(y), \pi_2(x) + \pi_2(y) + c_{case}) \end{cases}$$

Fig. 11. Cost-instrumented translation of expressions.

The cost-passing style translation of *simple* types is

$$(\!|\mathbb{N}|\!)_v \triangleq \mathbb{N} \qquad (\!|\mathsf{list}_A|\!)_v \triangleq \mathsf{list}_{(\!|A|\!)_v} \qquad (\!|A \to B|\!)_v \triangleq (\!|A|\!)_v \to (\!|B|\!)_e \qquad (\!|A|\!)_e \triangleq (\!|A|\!)_v \times \mathbb{N}$$

Guided by the translation of types above we can provide a cost-instrumented translation of simply-typed λ-expressions (Figure 11). This translation maps an expression of the simple type τ to an expression of type $\tau \times \mathbb{N}$, where the second component is the number of reduction steps under an eager, call-by-value reduction strategy (which is the semantics of RelCost). It is fairly easy to see that this translation preserves typability and that it counts steps accurately.

However, this translation forgets the cost and size information in types. To recover these, we define a HOL formula for every unary type. But, first, we define axiomatically a predicate $\mathsf{listU}(n, l, P)$ that captures size information about lists:

$$\forall l, P.\mathsf{listU}(0, l, P) \equiv l = []$$

$$\forall n, l, P.\mathsf{listU}(n+1, l, P) \equiv \exists w_1, w_2.l = w_1 :: w_2 \wedge P(w_1) \wedge \mathsf{listU}(n, w_2, P)$$

We can now define a HOL formula inductively on unary types.

$$\lfloor \mathbb{N} \rfloor_v(x) \triangleq \top \qquad \lfloor \mathsf{list}_A[n] \rfloor_v(x) \triangleq \mathsf{listU}(n, x, \lfloor A \rfloor_v)$$

$$\lfloor A \xrightarrow{\mathsf{exec}(k,l)} B \rfloor_v(x) \triangleq \forall y.\lfloor A \rfloor_v(y) \Rightarrow \lfloor B \rfloor_e^{k,l}(xy)$$

$$\lfloor \forall i \overset{\mathsf{exec}(k,l)}{::} S.\, A \rfloor_v(x) \triangleq \forall y.\top \Rightarrow \forall i.\lfloor A \rfloor_e^{k,l}(xy) \qquad \lfloor A \rfloor_e^{k,l}(x) \triangleq \lfloor A \rfloor_v(\pi_1 x) \wedge k \leq \pi_2 x \leq l$$

The type translation can also be extended to type environments: $(\!|x_1 : A_1, \ldots, x_n : A_n|\!) = x_1 : (\!|A_1|\!)_v, \ldots, x_n : (\!|A_n|\!)_v$ Similarly, we can associate to a type environment an HOL context that we can use to recover the cost and size information: $\lfloor x_1 : A_1, \ldots, x_n : A_n \rfloor = \lfloor A_1 \rfloor_v(x_1), \ldots, \lfloor A_n \rfloor_v(x_n)$. Now we can provide a cost-instrumented translation of unary judgments.

Theorem 17. If $\Delta; \Phi; \Omega \vdash_k^l t : A$, then: $(\!|\Omega|\!), \Delta \mid \Phi, \lfloor \Omega \rfloor \vdash (\!|t|\!) : (\!|A|\!)_e \mid \lfloor A \rfloor_e^{k,l}(\mathbf{r})$

For the embedding of cost and size information in the relational case we first define a predicate $\mathsf{listR}(n, l_1, l_2, a, P)$ in HOL axiomatically:

$$\forall l_1, l_2, a, P.\mathsf{listR}(0, l_1, l_2, a, P) \equiv l_1 = l_2 = []$$

$$\forall n, l_1, l_2, a, P.\mathsf{listR}(n+1, l_1, l_2, a, P) \equiv \begin{array}{c} \exists w_1, z_1, w_2, z_2. l_1 = w_1 :: w_2 \wedge l_2 = z_1 :: z_2 \wedge P(w_1, z_1) \wedge \\ (((w_1 = z_1) \wedge \mathsf{listR}(n, w_2, z_2, a, P)) \vee \\ (a > 0 \wedge \exists b.\, a = b + 1 \wedge \mathsf{listR}(n, w_2, z_2, b, P))) \end{array}$$

Let $\overline{\tau}$ denote RelCost's erasure of the binary type τ to a unary type.[2] This erasure maps $\mathsf{list}_\tau[n]^\alpha$ to $\mathsf{list}_{\overline{\tau}}[n]$, $\tau \xrightarrow{\mathrm{diff}(l)} \sigma$ to $\overline{\tau} \xrightarrow{\mathrm{exec}(0,\infty)} \overline{\sigma}$, etc. Next, we define HOL formulas for the binary types.

$$\llbracket \mathbb{N} \rrbracket_v(x, y) \triangleq x = y \qquad \llbracket UA \rrbracket_v(x, y) \triangleq \lfloor A \rfloor_v(x) \wedge \lfloor A \rfloor_v(y)$$

$$\llbracket \Box \tau \rrbracket_v(x, y) \triangleq (x = y) \wedge (\llbracket \tau \rrbracket_v(x, y)) \qquad \llbracket \tau \xrightarrow{\mathrm{diff}(l)} \sigma \rrbracket_v(x, y) \triangleq$$

$$\lfloor \overline{\tau} \xrightarrow{\mathrm{exec}(0,\infty)} \overline{\sigma} \rfloor_v(x) \wedge \lfloor \overline{\tau} \xrightarrow{\mathrm{exec}(0,\infty)} \overline{\sigma} \rfloor_v(y) \wedge (\forall z_1, z_2. \llbracket \tau \rrbracket_v(z_1, z_2) \Rightarrow \llbracket \sigma \rrbracket_e^l(xz_1, yz_2))$$

$$\llbracket \forall i \overset{\mathrm{diff}(l)}{::} S.\, \tau \rrbracket_v(x, y) \triangleq$$

$$\lfloor \forall i \overset{\mathrm{exec}(0,\infty)}{::} S.\, \overline{\tau} \rfloor_v(x) \wedge \lfloor \forall i \overset{\mathrm{exec}(0,\infty)}{::} S.\, \overline{\tau} \rfloor_v(y) \wedge (\forall z_1 z_2. \top \Rightarrow \forall i. \llbracket \tau \rrbracket_e^l(xz_1, yz_2))$$

$$\llbracket \mathsf{list}_\tau[n]^\alpha \rrbracket_v(x, y) \triangleq \mathsf{listR}(n, x, y, \alpha, \llbracket \tau \rrbracket_v) \qquad \llbracket \tau \rrbracket_e^l(x, y) \triangleq \llbracket \tau \rrbracket_v(\pi_1 x, \pi_1 y) \wedge (\pi_2 x - \pi_2 y \leq l)$$

The type translation can also be extended to relational type environments pointwise: $\|x^1 : \tau_1, \ldots, x^n : \tau_n\| \triangleq x_1^1 : (\!|\tau_1|\!)_v, x_2^1 : (\!|\tau_1|\!)_v, \ldots, x_1^n : (\!|\tau_n|\!)_v, x_2^n : (\!|\tau_n|\!)_v$ We also need to derive from a type relational environment an HOL context that remembers the cost and size information: $\llbracket x^1 : \tau_1, \ldots, x^n : \tau_n \rrbracket \triangleq \llbracket \tau_1 \rrbracket_v(x_1^1, x_2^1), \ldots, \llbracket \tau_n \rrbracket_v(x_1^n, x_2^n)$. Now we can provide the translation of relational judgments.

Theorem 18. If $\Delta; \Phi; \Gamma \vdash t_1 \ominus t_2 \lesssim l : \tau$, then: $\|\Gamma\|, \Delta \mid \Phi, \llbracket \Gamma \rrbracket \vdash (\!|t_1|\!)_1 : (\!|\tau|\!)_e \sim (\!|t_2|\!)_2 : (\!|\tau|\!)_e \mid \llbracket \tau \rrbracket_e^l(\mathbf{r}_1, \mathbf{r}_2)$, where $(\!|t_i|\!)_j$ is a copy of t_i where each variable x is replaced by a variable x_j for $j \in \{1, 2\}$.

RelCost's type-soundness theorem can be derived from Theorem 18 and the soundness of RHOL in set theory.

7 EXAMPLES

We present some illustrative examples to show how RHOL's rules work in practice. Our first example shows the functional equivalence of two recursive functions that are synchronous—they perform the same number of recursive calls. The second example shows the equivalence of two asynchronous recursive functions. Our third example illustrates reasoning about the relative cost of two programs, using an encoding similar to that of RelCost, but the example cannot be verified in RelCost itself.

[2] In RelCost, this erasure is written $|\tau|$. We use a different notation to avoid confusion with our own erasure function from RelCost's types to simple types.

7.1 First Example: Factorial

We show that the two following standard implementations of factorial, with and without an accumulator, are functionally equivalent:

$$\text{fact}_1 \triangleq \text{letrec } f_1\ n_1 = \text{case } n_1 \text{ of } 0 \mapsto 1; S \mapsto \lambda x_1.(S\ x_1) * (f_1\ x_1)$$

$$\text{fact}_2 \triangleq \text{letrec } f_2\ n_2 = \lambda acc.\text{case } n_2 \text{ of } 0 \mapsto acc; S \mapsto \lambda x_2.f_2\ x_2\ ((S\ x_2) * acc)$$

Our goal is to prove that:

$$\emptyset \mid \emptyset \vdash \text{fact}_1 : \mathbb{N} \to \mathbb{N} \sim \text{fact}_2 : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid \forall n_1 n_2.n_1 = n_2 \Rightarrow \forall acc.(\mathbf{r}_1\ n_1) * acc = \mathbf{r}_2\ n_2\ acc$$

The proof starts by applying [LETREC] rule, which has its main premise:

$$\Psi \vdash \begin{array}{l}\text{case } n_1 \text{ of}\\ \quad 0 \mapsto 1;\\ \quad S \mapsto \lambda x_1.(S\ x_1) * (f_1\ x_1)\end{array} \sim \begin{array}{l}\lambda acc.\ \text{case } n_2 \text{ of}\\ \quad 0 \mapsto acc;\\ \quad S \mapsto \lambda x_2.f_2\ x_2\ ((S\ x_2) * acc)\end{array} \mid \forall acc.\mathbf{r}_1 * acc = \mathbf{r}_2\ acc$$

where $\Psi \triangleq n_1 = n_2, \forall y_1 y_2.(y_1, y_2) < (n_1, n_2) \Rightarrow y_1 = y_2 \Rightarrow \forall a.(f_1\ y_1) * a = f_2\ y_2\ a$.

To prove this, we start by applying the one-sided [ABS-R] rule, with a trivial condition on *acc*. Then we can apply a two-sided [CASE] rule, which has 3 premises:

- $\Psi \vdash n_1 = 0 \Leftrightarrow n_2 = 0$
- $\Psi, n_1 = 0, n_2 = 0 \vdash 1 \sim acc \mid \mathbf{r}_1 * acc = \mathbf{r}_2$
- $\Psi \vdash \lambda x_1.(S\ x_1) * (f_1\ x_1) \sim \lambda x_2.f_2\ x_2\ ((S\ x_2) * acc) \mid \forall x_1 x_2.n_1 = S\ x_1 \Rightarrow n_2 = S\ x_2 \Rightarrow (\mathbf{r}_1\ x_1) * acc = \mathbf{r}_2\ x_2$

Premise 1 is a direct consequence of the assertion $n_1 = n_2$ in Ψ. Premise 2 is a trivial arithmetic identity which can be proven in HOL (using rule SUB or by invoking Theorem 6). To prove premise 3, we first apply the (two-sided) [ABS] rule, which leaves the following proof obligation:

$$\Psi, n_1 = S\ x_1, n_2 = S\ x_2 \vdash (S\ x_1) * (f_1\ x_1) \sim f_2\ x_2\ ((S\ x_2) * acc) \mid \mathbf{r}_1 * acc = \mathbf{r}_2$$

This is proven in HOL by instantiating the inductive hypothesis in Ψ with $y_1 \mapsto x_1, y_2 \mapsto x_2, a \mapsto (S\ x_1) * acc$.

7.2 Second Example: Take and Map

This example establishes the equivalence of two programs that compute the same result, but using different number of recursive calls. Consider the following function *take* that takes a list l and a natural number n and returns the first n elements of the list (or the whole list if its length is less than n).

$$\begin{array}{rl} take \triangleq & \text{letrec } f_1\ l_1 = \lambda n_1.\ \text{case } l_1 \text{ of } [] \mapsto []\\ & \qquad\qquad \_ :: \_ \mapsto \lambda h_1 t_1.\ \text{case } n_1 \text{ of } 0 \mapsto []\\ & \qquad\qquad\qquad\qquad S \mapsto \lambda y_1.h_1 :: (f_1\ t_1\ y_1)\end{array}$$

Next, consider the standard function *map* that applies a function g to every element of a list l pointwise.

$$\begin{array}{rl} map \triangleq & \text{letrec } f_2\ l_2 = \lambda g_2.\ \text{case } l_2 \text{ of } [] \mapsto\ []\\ & \qquad\qquad ; \_ :: \_ \mapsto\ \lambda h_2 t_2.(g_2\ h_2) :: (f_2\ t_2\ g_2)\end{array}$$

Intuitively, it should be clear that for all g, n, l, $map\ (take\ l\ n)\ g = take\ (map\ l\ g)\ n$ (mapping g over the first n elements of the list is the same as mapping over the whole list and then taking the first n elements). However, the computations on the two sides of the equality are very different: For a list l of length more than n, $map\ (take\ l\ n)\ g$ only examines the first n elements, whereas

take (*map* l g) n traverses the whole list. In the following we formalize this property in RHOL (Theorem 19) and outline the high-level idea of the proof. The full proof is in the appendix.

Theorem 19. $l_1, l_2 : \mathsf{list}_{\mathbb{N}}, n_1, n_2 : \mathbb{N}, g_1, g_2 : \mathbb{N} \to \mathbb{N} \mid l_1 = l_2, n_1 = n_2, g_1 = g_2 \vdash$
$map\ (take\ l_1\ n_1)\ g_1 : \mathsf{list}_{\mathbb{N}} \sim take\ (map\ l_2\ g_2)\ n_2 : \mathsf{list}_{\mathbb{N}} \mid \mathbf{r}_1 = \mathbf{r}_2$

Proof idea. Since the two sides make an unequal number of recursive calls, we need to reason asynchronously on the two sides (specifically, we use the rule [LLCASE-A]). However, equality cannot be established inductively with asynchronous reasoning: If two function applications are to be shown equal, and a recursion step is taken in only one of them, then the induction hypothesis cannot be applied. So, we strengthen the induction hypothesis, replacing the assertion $\mathbf{r}_1 = \mathbf{r}_2$ in the theorem statement with $\mathbf{r}_1 \sqsubseteq \mathbf{r}_2 \wedge |\mathbf{r}_1| = \mathsf{min}(n_1, |l_1|) \wedge |\mathbf{r}_2| = \mathsf{min}(n_2, |l_2|)$ where $\sqsubseteq$ denotes the prefix ordering on lists and $|\cdot|$ is the list length function. This assertion implies $\mathbf{r}_1 = \mathbf{r}_2$ and can be established inductively. The full proof is in the appendix, but at a high-level, the proof requires proving two judgments, one for the inner map-take pair and another for the outer one:

- $\Psi \vdash take\ l_1\ n_1 \sim map\ l_2\ g_2 \mid \mathbf{r}_1 \sqsubseteq_{g_2} \mathbf{r}_2$
- $\Psi \vdash map \sim take \mid \forall m_1 m_2 . m_1 \sqsubseteq_{g_2} m_2 \Rightarrow (\forall g_1 . g_1 = g_2 \Rightarrow \forall x_2 . x_2 \geq |m_1| \Rightarrow (\mathbf{r}_1\ m_1\ g_1) \sqsubseteq (\mathbf{r}_2\ m_2\ x_2))$

where $m_1 \sqsubseteq_g m_2$ is an axiomatically defined predicate equivalent to $(\mathsf{map}\ m_1\ g) \sqsubseteq m_2$ and Ψ are the assumptions in the statement of the theorem (in particular, $l_1 = l_2$). The proof of the first premise proceeds by an analysis of *map* using synchronous rules. For the second premise, after applying [LETREC] we apply the asynchronous [LLCASE-A] rule, and then prove the following premises:

(1) $\Psi, \Phi, x_2 \geq |m_1|, g_1 = g_2, m_1 = [], m_2 = [] \vdash [] \sim [] \mid \mathbf{r}_1 \sqsubseteq \mathbf{r}_2$
(2) $\Psi, \Phi, x_2 \geq |m_1|, g_1 = g_2, m_1 = [] \vdash [] \sim \lambda h_2 t_2 . \mathsf{case}\ x_2\ \mathsf{of}\ 0 \mapsto []; S \mapsto \lambda y_2 . h_2 :: f_2\ t_2\ y_2 \mid \forall h_2 t_2 . m_2 = h_2 :: t_2 \Rightarrow \mathbf{r}_1 \sqsubseteq (\mathbf{r}_2\ h_2\ t_2)$
(3) $\Psi, \Phi, x_2 \geq |m_1|, g_1 = g_2, m_2 = [] \vdash \lambda h_1 t_1 . (g_1\ h_1) :: (f_1\ t_1\ g_1) \sim [] \mid \forall h_1 t_1 . m_1 = h_1 :: t_1 \Rightarrow (\mathbf{r}_1\ h_1\ t_1) \sqsubseteq \mathbf{r}_2$
(4) $\Psi, \Phi, x_2 \geq |m_1|, g_1 = g_2 \vdash \lambda h_1 t_1 . (g_1\ h_1) :: (f_1\ t_1\ g_1) \sim \lambda h_2 t_2 . \mathsf{case}\ x_2\ \mathsf{of}\ 0 \mapsto []; S \mapsto \lambda y_2 . h_2 :: f_2\ t_2\ y_2 \mid \forall h_1 t_1 h_2 t_2 . m_1 = h_1 :: t_1 \Rightarrow m_2 = h_1 :: t_1 \Rightarrow (\mathbf{r}_1\ h_1\ t_1) \sqsubseteq (\mathbf{r}_2\ h_2\ t_2)$

where Φ is the inductive hypothesis obtained from the [LETREC] application. The first two premises follow directly from the definition of $\sqsubseteq$, while the third one follows from the contradictory assumptions $m_1 \sqsubseteq_g m_2$, $m_1 = h_1 :: t_1$ and $m_2 = []$. The last premise is proved by first applying the [NATCASE-R] rule and then applying the induction hypothesis. □

7.3 Third Example: Insertion Sort

Insertion sort is a standard sorting algorithm that sorts a list $h :: t$ by sorting the tail t recursively and then inserting h at the appropriate position in the sorted tail. Consider the following implementations of the insertion function, insert, and the insertion sort function, isort, each returning a pair, whose first element is the usual output list (inserted list for insert and sorted list for isort) and whose second element is the *number of comparisons* made during the execution (assuming an eager, call-by-value evaluation strategy).

$$\begin{aligned}
\mathsf{insert} \triangleq \lambda x.\ &\mathsf{letrec}\ \mathit{insert}\ l = \mathsf{case}\ l\ \mathsf{of}\ [] \mapsto ([x], 0);\\
&\quad _ :: _ \mapsto \lambda h\, t.\ \mathsf{case}\ x \leq h\ \mathsf{of}\\
&\qquad\qquad \mathsf{tt} \mapsto\ (x :: l, 1);\\
&\qquad\qquad \mathsf{ff} \mapsto\ \mathsf{let}\ s = \mathit{insert}\ t\ \mathsf{in}\\
&\qquad\qquad\quad (h :: (\pi_1\ s), 1 + (\pi_2\ s))
\end{aligned}$$

$$\begin{aligned}\mathsf{isort} \triangleq \mathsf{letrec}\ \mathit{isort}\ l = \mathsf{case}\ l\ \mathsf{of}\ [] \mapsto ([], 0);\\ \_ :: \_ \mapsto \lambda h\, t.\ \ \mathsf{let}\ s = \mathit{isort}\ t\\ \mathsf{let}\ s' = \mathsf{insert}\ h\ (\pi_1\ s)\ \mathsf{in}\\ (\pi_1\ s', (\pi_2\ s) + (\pi_2\ s'))\end{aligned}$$

Using this implementation, we prove the following interesting fact about insertion sort: Among all lists of the same length, insertion sort computes the fastest (with fewest comparisons) on lists that are already sorted. This is a property about the relational cost of insertion sort (on two different inputs), which cannot be established in RelCost. To state the property in RHOL, we define a list predicate $\mathsf{sorted}(l)$ in HOL axiomatically:

$$\mathsf{sorted}([]) \equiv \top \qquad \forall h\, t.\, \mathsf{sorted}(h :: t) \equiv (\mathsf{sorted}(t) \wedge h \leq \mathsf{lmin}(t))$$

where the function $\mathsf{lmin}(l)$ returns the minimum element of l:

$$\mathsf{lmin} \triangleq \mathsf{letrec}\ f\ l = \mathsf{case}\ l\ \mathsf{of}\ [] \mapsto \infty; \_ :: \_ \mapsto \lambda h\, t.\, \mathsf{min}(h, f\ t)$$

As in the previous example, let $|\cdot|$ be the standard list length function. The property of insertion sort mentioned above is formalized in the following theorem. In words, the theorem says that if isort is executed on lists x_1 and x_2 of the same length and x_1 is sorted, then the number of comparisons made during the sorting of x_1 is no more than the number of comparisons made during the sorting of x_2.

Theorem 20. Let $\tau \triangleq \mathsf{list}_{\mathbb{N}} \to \mathsf{list}_{\mathbb{N}}$. Then, $\bullet \mid \bullet \vdash \mathsf{isort} : \tau \sim \mathsf{isort} : \tau \mid \forall x_1\, x_2.\, (\mathsf{sorted}(x_1) \wedge |x_1| = |x_2|) \Rightarrow \pi_2(\mathbf{r}_1\ x_1) \leq \pi_2(\mathbf{r}_2\ x_2)$.

A full proof is shown in the appendix. The proof proceeds mostly synchronously in the two sides. Following the structure of isort, we apply the rules [LETREC], [LISTCASE] and [APP] + [ABS] (for the let binding, which, as usual, is defined as a function application), followed by an application of the inductive hypothesis for the recursive call to *isort*. Eventually, we expose the call to insert on both sides. At this point, the observation is that since x_1 is already sorted, its head element must be no greater than all elements in its tail, so insert must return immediately with at most 1 comparison on the x_1 side. Formally, this last proof step can be completed by switching to either UHOL or HOL and using subject conversion; we switch to HOL in the appendix.

8 CONCLUSION

We have developed Relational Higher-Order Logic, a new formalism to reason about relational properties of (pure) higher-order programs written in a simply typed λ-calculus with inductive types and recursive definitions. The system is expressive, has solid foundations via an equivalence with Higher-Order Logic, and yet retains the (nice) "feel" of relational refinement type systems. An important direction for future work is to extend Relational Higher-Order Logic to effectful programs. Natural directions include integrating the state monad, and the Giry monad for probability sub-distributions. One particularly exciting perspective is to broaden the scope of relational cost analysis to probabilistic programs, and to prove relational costs for different data structures. There are also many potential applications to security, differential privacy, machine learning, and probabilistic programming.

For practical purposes, it will also be interesting to build prototype implementations of Relational Higher-Order Logic. We believe that much of the technology developed for (relational) refinement types, and in particular the automated generation of verification conditions (maybe with user hints to switch between unary and binary modes of reasoning) and the connection to SMT-solvers can be lifted without significant hurdle to Relational Higher-Order Logic.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful and thoughtful comments. This article is based on research that has been supported, in part, by NSF under grant TWC-1565365.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 147–160. DOI: http://dx.doi.org/10.1145/292540.292555

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal Parametric Polymorphism. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. 157–170. DOI: http://dx.doi.org/10.1145/158511.158622

Peter Aczel and Nicola Gambino. 2000. Collection Principles in Dependent Type Theory. In *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers (Lecture Notes in Computer Science)*, Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack (Eds.), Vol. 2277. Springer, 1–23. DOI: http://dx.doi.org/10.1007/3-540-45842-5_1

Peter Aczel and Nicola Gambino. 2006. The generalised type-theoretic interpretation of constructive set theory. *J. Symb. Log.* 71, 1 (2006), 67–103. DOI: http://dx.doi.org/10.2178/jsl/1140641163

Robin Adams and Zhaohui Luo. 2010. Classical predicative logic-enriched type theories. *Ann. Pure Appl. Logic* 161, 11 (2010), 1315–1345. DOI: http://dx.doi.org/10.1016/j.apal.2010.04.005

Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185. DOI: http://dx.doi.org/10.1016/0020-0190(85)90056-0

Kazuyuki Asada, Ryosuke Sato, and Naoki Kobayashi. 2016. Verifying relational properties of functional programs by first-order refinement. *Science of Computer Programming* (2016).

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. 200–214. DOI: http://dx.doi.org/10.1007/978-3-642-21437-0_17

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. 100–114. DOI: http://dx.doi.org/10.1109/CSFW.2004.17

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

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

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 90–101. DOI: http://dx.doi.org/10.1145/1480881.1480894

Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 161–174. http://dl.acm.org/citation.cfm?id=3009896

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 97–110. DOI: http://dx.doi.org/10.1145/2103656.2103670

João Filipe Belo. 2007. Dependently Sorted Logic. In *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale del Friuli, Italy, May 2-5, 2007, Revised Selected Papers (Lecture Notes in Computer Science)*, Marino Miculan, Ivan Scagnetto, and Furio Honsell (Eds.), Vol. 4941. Springer, 33–50. DOI: http://dx.doi.org/10.1007/978-3-540-68103-8_3

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations.. In *Proceedings of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'04*, Neil D. Jones and Xavier Leroy (Eds.). 14–25.

Lennart Beringer and Martin Hofmann. 2007. Secure information flow and program logics. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. IEEE Computer Society, 233–248. DOI: http://dx.doi.org/10.1109/CSF.2007.30

Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. 2017. Deductive Verification with Relational Properties. In *In Proc. of the 23th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017), Uppsala, Sweden.* To Appear.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 316–329. http://dl.acm.org/citation.cfm?id=3009858

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *Proceedings of CSF'08*. 51–65.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Logical Methods in Computer Science* 7, 2 (2011). DOI: http://dx.doi.org/10.2168/LMCS-7(2:16)2011

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010. A relational modal logic for higher-order stateful ADTs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 185–198. DOI: http://dx.doi.org/10.1145/1706299.1706323

Joshua Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 281–292. DOI: http://dx.doi.org/10.1145/964001.964025

Peter Dybjer. 1985. Program Verification in a Logical Theory of Constructions. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science)*, Jean-Pierre Jouannaud (Ed.), Vol. 201. Springer, 334–349. DOI: http://dx.doi.org/10.1007/3-540-15975-4_46

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. DOI: http://dx.doi.org/10.1145/113445.113468

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 357–370. DOI: http://dx.doi.org/10.1145/2429069.2429113

Neil Ghani, Fredrik Nordvall Forsberg, and Alex Simpson. 2016a. Comprehensive Parametric Polymorphism: Categorical Models and Type Theory. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 3–19. DOI: http://dx.doi.org/10.1007/978-3-662-49630-5_1

Neil Ghani, Fredrik Nordvall Forsberg, and Alex Simpson. 2016b. Comprehensive Parametric Polymorphism: Categorical Models and Type Theory. In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Bart Jacobs and Christof Löding (Eds.), Vol. 9634. Springer, 3–19. DOI: http://dx.doi.org/10.1007/978-3-662-49630-5_1

John Hatcliff and Olivier Danvy. 1997. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science* 7 (1997), 507–541.

Nevin Heintze and Jon G. Riecke. 1998. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 365–377. DOI: http://dx.doi.org/10.1145/268946.268976

B. Jacobs. 1999. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. DOI: http://dx.doi.org/10.1145/2676726.2676980

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 218–231. http://dl.acm.org/citation.cfm?id=3009877

Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 3–16. DOI: http://dx.doi.org/10.1145/2676726.2676970

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.* 35, 2 (2013), 6:1–6:41. DOI: http://dx.doi.org/10.1145/2491522.2491523

Frank Pfenning. 2008. Church and Curry: Combining Intrinsic and Extrinsic Typing. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, C.Benzmüller, C.Brown, J.Siekmann, and R.Statman (Eds.).

College Publications, 303–338.

Gordon Plotkin. 1973. Lambda-definability and logical relations. (1973).

Gordon Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223 – 255. DOI: http://dx.doi.org/10.1016/0304-3975(77)90044-5

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*. 361–375. DOI: http://dx.doi.org/10.1007/BFb0037118

François Pottier and Vincent Simonet. 2002. Information flow inference for ML. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. 319–330. DOI: http://dx.doi.org/10.1145/503272.503302

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 57–69. DOI: http://dx.doi.org/10.1145/2908080.2908092

R. Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2-3 (May 1985), 85–97. http://dx.doi.org/10.1016/s0019-9958(85)80001-2

Gordon Stewart, Anindya Banerjee, and Aleksandar Nanevski. 2013. Dependent types for enforcement of information flow and erasure policies in heterogeneous data structures. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*. 145–156. DOI: http://dx.doi.org/10.1145/2505879.2505895

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. DOI: http://dx.doi.org/10.1145/2837614.2837655

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. DOI: http://dx.doi.org/10.2307/2271658

Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis Symposium*, Chris Hankin and Igor Siveroni (Eds.), Vol. 3672. 352–367.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. DOI: http://dx.doi.org/10.1145/2628136.2628161

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. 1996. A sound type system for secure flow analysis. *Journal of Computer Security* 4, 3 (1996), 1–21.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. DOI: http://dx.doi.org/10.1145/292540.292560

Hongseok Yang. 2007. Relational separation logic. 375, 1-3 (2007), 308–334.

Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Formal Methods (Lecture Notes in Computer Science)*, Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere (Eds.), Vol. 5014. 35–51.

Noam Zeilberger. 2016. Principles of Type Refinement. (2016). http://noamz.org/oplss16/refinements-notes.pdf Notes for OPLSS 2016 school.