A Language with Type-Dependent Equality

Florian Rabe[https://orcid.org/0000−0003−3040−3655]

Computer Science, FAU Erlangen-Nuremberg

Abstract. In soft type systems terms and types exist independently and typing is a binary relation between them. That allows the same term to have multiple types, which is in particular the case in the presence of subtyping. Thus, a soft type system may define equality in such a way that two terms can be equal at one type but unequal at another type.
We explore the design of soft type systems with such a type-dependent equality. The most promising application is that it yields a more natural treatment of quotient types: if two terms can be different at the base type but equal at the quotient type, we can use the same representation in both types without incurring the cost of using equivalence classes. That can help formalize mathematics, where the official definition of quotients uses equivalence classes but practical notations usually do not. The main drawback of such a system is that the substitution of equals by equals becomes more complex as it now depends on the type with which the equal terms are used.
We analyze the general problem, show examples from major soft-typed proof assistants, and then present a simple language that allows studying type-dependent equality in a simple rigorous setting.

1 Introduction

To work with mathematical content in computer systems, it is necessary to represent it in formal languages. So far combining the flexibility of informal mathematical language with strong tool support has proved challenging. All major proof assistants are at least partially motivated by this but must make different trade-offs to obtain tool support. And despite massive progress, no current system is even close to mimicking the flexibility of mathematical language [Wie07,KR20]. Arguably that is part of the reason why adoption of proof assistants by mathematicians is much slower than hoped.

A fundamental issue is the decidability of typing and equality, which is not a priority (arguable even a negative priority) in mathematics but often essential in proof assistants. Therefore, features like subtyping via predicate types $\{x : A|p(x)\}$ and equating objects via quotient types A/r are very difficult to design in formal systems.

In **hard-typed** systems, every term has a unique type (possibly up to some equality on types). Examples used in proof assistants are dependent type theories such as the one in Coq [Coq15] and the higher-order logic family [Gor88]. Hard typing often yields better computational properties (in particular type inference,

decidable equality) but requires formalization artifacts when mathematical operations on sets cannot be mapped direct to corresponding operations on types. For examples, quotients must usually be modeled via Setoids $\{A : \texttt{type}, r : A \to A \to \texttt{bool}\}$ and subtypes via dependent products $\Sigma x : A.P(x)$. Lean [dKA+15] uses an interesting compromise: it adds kernel-level proof automation for those artifacts even though they are library-level definitions and not part of the underlying logic.

Our focus here is on **soft-typed** systems, where terms exist independently of types and typing is giving by a binary predicate between terms and types. In particular, every term can have many different types, and types can be seen as unary predicates on terms. Proof assistants using soft typing include Nuprl [CAB+86,ABC+06] and Mizar [TB85]. These can accommodate predicate types (both) and quotient types (Nuprl) naturally at the cost of making typing and equality undecidable. A compromise solution is employed in PVS [ORS92]: it uses a hard type system with a lattice of soft predicate subtypes for each hard type.

In this paper we look at one particular feature in the design of soft type systems: type-dependent equality (TDE). With TDE, the equality operator is defined in such a way that the derivability of the formula $s =_A t$ depends on the type A — the same two terms might be equal at one type but unequal at a different type. Note that TDE is not a design option in hard type systems at all because there terms do not have multiple different types to begin with.

This paper is motivated by the observation that TDE, while used in some systems, has received disproportionately little systematic attention relative to its potential benefits and fundamental nature. In fact, the author coined the name "type-dependent equality" for the occasion of this paper. Even in proof assistants featuring (some variant of) TDE, it is not prominently advertised. This is all the more important as some of the TDE-related language design trade-offs in these systems are very subtle and little-known even to experts in the area of formalized mathematics.

We take a broad approach focusing on general ideas and presenting only a simple formal language with TDE. This is warranted because typing and equality are cross-cutting issues that affect virtually every other language feature and thus cannot be studied in isolation. Sect. 2 extensively discusses the benefits and dangers of TDE and how it interrelates with other language features. It also describes some of the subtleties of how TDE is or is not realized in current proof assistants. Then Sect. 3 introduces the syntax and semantics of a formal language with TDE. It serves as an example to better understand TDE and as a starting point to develop more advanced languages with TDE. Sect. 4 concludes.

2 Motivating Considerations and Related Work

2.1 Type-Dependent Equality

Benefits Equality is usually seen as a binary relation on objects. Even in typed languages, which often use a ternary equality relation $s =_A t$, equality is often

absolute in the sense that the derivability of $s =_A t$ does not depend on A. The only effect of A in $=_A$ is to restrict s and t to be of type A — if s and t have both type A and B, then either both $s =_A t$ and $s =_B t$ hold or neither.

A type-*dependent* equality (TDE) relation is rare, and it is instructive to ponder why. In hard-typed languages such as HOL [Gor88] or the calculus of constructions underlying Coq [Coq15], all types are disjoint. Equality is ternary because it is a polymorphic family of binary relations, one for each type. The question of TDE does not come up because terms s and t never have two different types A and B.

Set theoretic languages are usually based on an axiomatization of set theory in first-order logic as in Mizar [TB85] or higher-order logic as in Isabelle/ZF [PC93]. In both cases, it is common to use a fixed base type U for the universe of sets, and binary equality on U is a primitive notion. In these languages, soft typing is an emergent feature: we can add the concept of types as unary predicates on U. But the binary equality on U both historically and foundationally precedes the introduction of soft types. Thus, TDE is usually not considered.

The only major system with systematic TDE that the author could find is Nuprl [CAB+86]. It uses TDE to handle quotients naturally by changing the equality at the quotient type. For example, it allows having both $0 \neq_{\mathbb{Z}} 2$ and $0 =_{\mathbb{Z}/\mathtt{mod}_2} 2$ without inconsistency. The key idea is to use a different equality relation at the quotient type than at the base type. This trick has the benefit that no fiddling with equivalence classes is needed: the term $2 : \mathbb{Z}$ can also be used as an element of $2 : \mathbb{Z}/\mathtt{mod}_2$. Thus, the canonical projection into the quotient is a no-op, which is far superior computationally and notationally to the standard approach of using equivalence classes.

Mizar [TB85] uses a light variant of TDE for record types, which we discuss in more detail in Sect. 2.4.

Arguably, TDE is how quotients are handled in practical mathematics as well. Even though mathematics officially defines the quotient as the set of equivalence classes, mathematical notation almost always reuses the terms of the original types as elements of the quotient — with the implicit understanding that related terms are equal when seen as elements of the quotient. However, this is considered a notational simplification, and the foundations of mathematics do not include a rigorous treatment of TDE.

Analogy to Typing There is an elegant way to fit TDE into a general framework of formal systems if we think of typing as type-dependent definedness (TDD). Instead of thinking of $a : A$ as a binary predicate, we can think of it as a type-dependent unary predicate $:_A$ applied to a.

That yields a language with a unary and a binary type-dependent predicate: TDD $:_A$ and TDE $=_A$. TDD regulates which values are acceptable values at type A, and TDE regulates which of those are equal. The semantics of types can then be given by partial equivalence relations on objects [All87]: the denotational semantics of A is the universe U restricted to objects satisfying $:_A$ quotiented by $=_A$.

It is well-known that partial equivalence relations enjoy nice closure properties. In particular, TDE makes it easy to introduce

- predicate types $A|p$ for a unary predicate p on A: a copy of A with definedness restricted to p
- quotient types A/r for an equivalence relation r on A: a copy of A with equality broadened to r.

Dangers Tweaking the equality relation at a type corresponds to changing the introduction rule of equality to make more things equal. Consequently, the elimination rule is applicable more often so that a naive version of TDE can easily be inconsistent. The elimination rule of equality varies between languages but is usually a substitution rule like

$$\frac{x : A \vdash F(x) : \texttt{bool} \qquad \vdash s =_A s' \qquad \vdash F(s)}{\vdash F(s')}\text{Sub}$$

This shows us how TDE can cause trouble: Assume in addition to the premises of Sub, we also have $x : B \vdash F(x)$ and $\vdash s : B$ and $\vdash s' : B$ — a common situation in languages with subtyping. Then TDE+Sub is inconsistent if the plausible situation arises where $\vdash \neg s =_B s'$ and $\vdash F(s)$ and $\vdash \neg F(s')$. Intuitively, whenever terms can have multiple types, the equality relations at those types must be consistent with each other. At least if A is a subtype of B, then $s =_A s'$ should imply $s =_B s'$, i.e., the rule

$$\frac{\vdash s =_A s' \qquad \vdash A <: B}{\vdash s =_B s'}$$

should be present. Otherwise, we would no longer have a canonical embedding of A into B, which would be an unreasonably high price to pay.

But the details subtly depend on the specific language. To understand how Nuprl, which uses Sub, avoids inconsistency, the author had to resort to direct communication with the developers after exhausting the documentation and failing at reverse engineering: Nuprl prevents the above situation because $x : A \vdash F(x) : \texttt{bool}$ would not hold, thus making Sub not applicable. In fact and maybe surprisingly, it is possible in Nuprl to have $\vdash F(g) : \texttt{bool}$ for every closed term $g : A$ but still $x : A \nvdash F(x) : \texttt{bool}$. That is because the Nuprl proof system (of which well-formedness of propositions is a special case) privileges closed terms — there is even a type *Base* of all closed terms. This unusual behavior is motivated also by other design considerations but is critical for the soundness of TDE+Sub in Nuprl.

2.2 Abstract Definitions and Quotient Types

Standard mathematics defines quotients via equivalence classes. It is instructive to ask whether there are alternative definitions.

Abstract vs. Concrete Definitions We speak of an **abstract** definition of a language feature if the operations that form sets and their elements are axiomatized through their characteristic properties. We speak of a **concrete** definition if the operations are given as abbreviations of existing expressions.

A paradigmatic abstract definition is the Cartesian product, which is usually specified to have formation operator $\_ \times \_$, introduction form $(\_,\_)$, elimination forms $\__1$ and $\__2$, computation properties $(a,b)_1 = a$ and $(a,b)_2 = b$ (intuitively: elimination of introduction is identity), representation property $p = (p_1, p_2)$ (intuitively: introduction of elimination is identity; or introduction is surjective), and extensionality property $p = q$ iff $p_i = q_i$ for $i = 1, 2$ (intuitively: elimination is injective). To show the consistency, at least one concrete definition must be given, e.g., via Wiener pairs or Kuratowski pairs.

A paradigmatic concrete definition is the set of functions, which is usually defined by formation operator $A \to B = \{f \subseteq A \times B | \forall x : A.\exists^1 y : B.(x,y) \in f\}$, introduction form $\lambda x : A.t(x) = \{(a, t(a)) : a \in A\}$, elimination form $f\,a = \texttt{the}\, b : B.(a,b) \in f$. For concrete constructions, the characteristic properties are the corresponding ones but must be proved instead of being axioms: computation $(\lambda x : A.t(x))\,a = t(a)$, representation $f = \lambda x : A.f\,x$, extensionality $f = g$ iff $f\,a = g\,a$ for $a : A$.

Abstract definitions allow for more scalable reasoning, can usually be done in simpler languages, allow defining language features orthogonally, and are more portable across systems. Thus, mathematics and computer science often prefer them, e.g., λ calculus is the abstract definition of function types obtained by abstracting from the concrete one used in mathematics.

Abstract Quotient Types It is maybe surprising that quotient types are usually defined concretely by using equivalence classes:

$$A/r = \{[a]_r : a \in A\}, \qquad [a]_r = \{a' : A | r\,a\,a'\} \qquad (*)$$

Maybe this is because there is no natural competing concrete definition. Contrary to function sets, there is no commonly used abstract specification of quotient sets either. But we can systematically obtain an abstract specification by analogy to the ones above:

- formation operator A/r for a binary equivalence relation r on A
- introduction form $[a]_r$ for $a : A$
- elimination form $t(\rho q) : B$ for $q : A/r$ and $t(x) : A \xrightarrow{r} B$, where we write ρq for picking an arbitrary representative of class q and $t(x) : A \xrightarrow{r} B$ as a shorthand for $x : A \vdash t(x) : B$ and $r\,a\,a'$ implies $t(a) = t(a')$,
- computation property $t(\rho[a]_r) = t(a)$
- representation property $[\_]_r : A \xrightarrow{r} B$ and $q = [\rho q]_r$ whenever $q : A/r$
- extensionality property $p = q$ iff $t(\rho p) = t(\rho q)$ whenever $t(x) : A \xrightarrow{r} B$

Nuprl uses this abstract definition of quotient types that is a primitive part of the logic. An alternative definition of quotients equivalent to the one in Nuprl is given in [Nog02]. Introduction form $[\_]_r$ and elimination form ρ are no-ops, which is critical for performance and convenience. Mizar defines quotients concretely

using $(*)$. Coq and Lean define quotients concretely via setoids, but that does not satisfy the above specification as equality is not redefined. However, Lean provides kernel-level support to eliminate that artifact.

2.3 Predicate and Quotient Types

Predicate Types If p is a unary predicate on A, we write $A|p$ for the predicate subtype of A given by p. An abstract specification can be given by

- formation operator $A|p$
- introduction form $a|p : A|p$ if $a : A$ and $p\,a$
- elimination form $\iota s : A$ for $s : A|p$, and $p\,(\iota s)$
- computation property $\iota\,(a|p) = a$
- representation property $s = (\iota s)|p$ whenever $s : A|p$
- extensionality property $s = t$ iff $\iota s = \iota t$ for $s, t : A|p$

Mizar uses a concrete definition to introduce predicate types. Nuprl and PVS use abstract ones that are primitive parts of the logics. In all three systems, introduction form $a|p$ and elimination form ι are no-ops (i.e., the introduction form incurs a proof obligation that the system must try to discharge automatically). Hard-typed languages such as HOL or dependent type theory usually define $\mathrm{set}\,A = A \to \texttt{bool}$ as the power type of A. Then p itself can be used instead of $A|p$. But this does not satisfy the above specification as p is a value and not a type. The HOL proof assistant family employ the conservative extension principle to turn the value p into a fresh type [Gor88]: essentially given p, it adds a fresh type S, a partial function $A \to^? S$ and the function $\iota : S \to A$ such that S becomes bijective to $A|p$. PVS uses HOL plus primitive operations corresponding to the above specification. Because it supports both $p : \mathrm{set}\,A$ and the type $A|p$, conversions between the two are commonly used.

Duality The abstract definitions of predicate and quotient types are dual in the following sense:

property	predicate type $A\|p$	quotient type A/r
formation uses	unary predicate p	binary predicate r
introduction	requires satisfaction of p	canonical projection $[\_]_r$
elimination	canonical injection ι	requires preservation of r
minimal predicate	$A\|\lambda x.\texttt{false} \cong \texttt{void}$ (initial)	$A/\lambda xy.\texttt{false} \cong A$
total predicate	$A\|\lambda x.\texttt{true} \cong A$	$A/\lambda xy.\texttt{true} \cong \texttt{unit}$ (terminal)

Here, to enhance the duality, we do not require r to be an equivalence. Instead, we allow any binary relation and assume the semantics uses the generated equivalence relation. Nuprl and the setoid encoding require an equivalence relation.

The duality is weaker when using standard concrete definitions. Here concrete predicate types make it easy to use no-ops for introduction. But for concrete quotient types, the projection $[\_]_r$ is expensive. A guiding motivation for our work is to retain the duality and use no-ops in both cases.

This becomes particularly practical when chaining: Using no-ops, in $A|p|q$, we can use a predicate q on A as opposed to $A|p$. And we simply have $A|p|q =$

$A|q|p = A|(\lambda x.p\,x \wedge q\,x)$. For quotient types, we desire the corresponding chain rule $A/r/s = A/s/r = A/(\lambda xy.r\,x\,y \vee s\,x\,y$ as opposed to awkwardly defining s on equivalence classes.

2.4 Records and Predicate/Quotient Types

Lax vs. Strict Records For simplicity, we do not give formal rules for record types and instead consider them by example. We use record types like $R := \{x : A, y : B\}$ with introduction form $r := [x = a, y = b] : R$ and elimination forms $r.x : A$ and $r.y : B$. Specifically, we discuss the relation between R and $S := \{x : A\}$ as well as the forgetful functor $F : R \to S$. Using this example, we introduce a distinction between lax and strict records that is critical in the presence of soft typing and TDE.

If $\vdash r : S$, we speak of **lax records**. Intuitively, the elements of a lax record type are all records that have *at least* the fields prescribed by the type. Thus, larger record types have fewer record values. To check $r : S$, we only check $r.x : A$ and ignore the additional field $r.y$. Therefore, R is a subtype of S, and F is the subtype embedding and a no-op, and the empty record type $\{\}$ is the type of all records.

This is the case in most languages with primitive record types. In this situation, TDE (possibly only for record types) is a natural feature: for $s, t : R$, we can put $s =_R t$ if $s.x =_A t.x$ and $s.y =_B t.y$, but $s =_S t$ already if $s.x =_A t.x$. Thus, the subtype R has the stronger equality, and more terms may be equal at the supertype S. In particular, we might have $s =_S t$ but $s \neq_R t$.

This is how PVS records work. Mizar structures work almost the same way: However, the type argument of the equality relation is not explicit in Mizar. Instead, it is inferred to be the most specific type of the argument records, i.e. if $s, t : R$, then $s = t$ denotes $s =_R t$. In addition to R being a subtype of S, Mizar introduces explicit syntax for F to drop the additional fields, and then the equality $s =_S t$ can be stated as $F(s) = F(t)$. Nuprl does not have primitive records but derives lax record types as functions from some index set (representing the field labels) to types.

If $\nvdash r : S$, we speak of **strict records**. Intuitively, the elements of a strict record type are all records that have *exactly* the fields prescribed by the type. $F(r)$ explicitly removes the field y from r, and the empty record $\{\}$ is a unit type. This is common in languages that use derived record types (e.g., by generating an axiomatic specification, via product types $A \times B$, or via single-constructor inductive types $R = \texttt{inductive}\, R(A, B)$). Examples are Coq and Isabelle. Here F can be an expensive operation, especially if many record fields must be copied or if the value of the field y must be re-inferred later on. TDE is not an option because no two records have both type R and type S.

There is no subtyping between strict record types R and S, and the relation between them can be better understood as a quotient. $\lambda rr' : R.\ r.x =_A r'.x$ is an equivalence relation on R, and F is the canonical projection. Contrary to

general quotients, the equivalence relation always has canonical representatives: we can use the elements of S because S is isomorphic to the quotient.

Record Subtypes vs. Record Quotients We can apply the quotient intuition also to lax records. As for strict records, we have the same equivalence relation on R, and F is also the canonical projection. However, the quotient type cannot be expressed: F is not surjective, and S is not isomorphic to the quotient — S is much bigger than the quotient (a supertype of R even). For that reason, Mizar introduces an additional operator that maps every record type S (which are lax by default) to the corresponding strict record type S^{strict} and provides syntax for the functor F that now maps $R \to S^{\text{strict}}$.

Conversely, we can apply the subtype intuition to strict records. But F is not a no-op and thus not a proper subtype embedding. But languages that support implicit coercions can insert F automatically, thus creating the look and feel of subtyping for the user.

Thus, languages have substantial freedom in how to combine record types with subtyping and quotient typing as well as TDE.

Mathematical Records In standard mathematics, records are not used explicitly. But effectively, they occur frequently, e.g., when using algebraic structures such as when S is Magma (tuple of a set and an operation) and R is Monoid (Magma with an additional unit). We can think of them as strict records derived via Cartesian products.

The cost of the explicit forgetful functor F is harmless here because the representation change from r to $F(r)$ is not done on paper and left to an implicit coercion in the reader's mind, i.e., we simply write r instead of $F(r)$. Therefore, mathematical language can flexibly switch between the subtype intuition (every monoid is a magma) and the quotient intuition (monoids can be projected to their magma).

3 Formal Language Definition

We develop a minimal formal language with systematic TDE.

3.1 Syntax and Inference System

Grammar The grammar is given below. Types A are user-declared base types a, type variables k, built-in base type `bool`, predicate types $A|p$, quotient types A/r, and function types $A \to B$. Simple functions are needed to give the language practical expressivity and to form the unary/binary predicates p and r. Dependent functions types or polymorphic type operators can be added easily but are not essential in the sequel. We add record types in Sect. 3.3.

Terms are user-declared constants c, bound variables x, the usual logical connectives and quantifiers on the type `bool`, and the usual λ-abstraction and application forms for $A \to B$. $A|p$ and A/r do not have associated term formation

rules because they are populated by the same terms as A with introduction and elimination being no-ops. The distinction between A on the one hand and $A|p$ and A/r on the other hand is relegated to the judgments $t : A$ (TDD) and $s =_A t$ (TDE). In keeping with logical practice, the equality judgment is part of the term syntax as a `bool`-valued predicate. But the typing judgment must be a meta-level judgment because it defines which syntax is well-formed to begin with and because of subtle soundness issues discussed below.

$$\begin{array}{lll} A, B & ::= & a \mid k \mid \texttt{bool} \mid A|p \mid A/r \mid A \to B \\ s, t, p, r & ::= & c \mid x \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}(s)t\,\texttt{else}\,t' \mid \text{(logical operators)} \\ & \mid & \lambda x : A.t \mid t\,t \mid s =_A t \\ \Gamma & ::= & (x : \texttt{type} \mid x : A \mid t)^* \end{array}$$

User-declared theories Θ (omitted in the grammar) introduce global knowledge: base types $a : \texttt{type}$, constants $c : A$ for $A : \texttt{type}$, and axioms t for $t : \texttt{bool}$. Everything below should be understood as relative to a fixed theory, which we do not make explicit in the notation. Correspondingly, contexts Γ collect local (α-renamable) knowledge: type variables $k : \texttt{type}$, typed variables $x : A$ for $A : \texttt{type}$, and local assumptions t for $t : \texttt{bool}$.

Judgments The judgments are given below. As usual for soft type theories, the rules for the typing and provability judgments are mutually recursive. A subtyping judgment is defined below as an abbreviation. Because types can contain terms, we also need a judgment for equality of types and a rule for substitution of equal types for each other, but we omit those here.

$\Gamma \vdash$	Γ is a well-formed context
$\Gamma \vdash A : \texttt{type}$	A is a well-formed type
$\Gamma \vdash t : A$	t is well-formed at type A
$\Gamma \vdash t$	boolean t is provable

$$\frac{\Gamma, x : A \vdash t(x) : B}{\Gamma \vdash \lambda x : A.t(x) : A \to B} \qquad \frac{\Gamma \vdash s : \texttt{bool} \quad \Gamma, s \vdash t : A \quad \Gamma, \neg s \vdash t' : A}{\Gamma \vdash \texttt{if}(s)t\,\texttt{else}\,t' : A}$$

$$\frac{\Gamma \vdash s : A \quad \Gamma \vdash t : A}{\Gamma \vdash s =_A t : \texttt{bool}} \qquad \frac{\Gamma \vdash s : \texttt{bool} \quad \Gamma, s \vdash t : \texttt{bool}}{\Gamma \vdash s \wedge t : \texttt{bool}} \qquad \frac{\Gamma \vdash s \quad \Gamma, s \vdash t}{\Gamma \vdash s \wedge t}$$

$$\frac{\Gamma, x : A \vdash t(x) : \texttt{bool}}{\Gamma \vdash (\forall x : A.t(x)) : \texttt{bool}} \qquad \frac{\Gamma \vdash \forall x : A.t(x) \quad \Gamma \vdash s : A}{\Gamma \vdash t(s)}$$

Fig. 1. Selected Rules for Standard Operators

Rules for Standard Operators We omit most of the rules for context formation, logical operators, and functions and only give some selected rules in Fig. 1. We write $t(x)$ for a term with a distinguished free variable x and accordingly $t(s)$ for the substitution of s for x.

The rule for λ-abstraction is as usual. Because no constraints are put on $t(x)$, we later on incur the proof obligation that every term $t(x)$ respects $=_A$. The formation rule for equality shows that $s =_A t$ is only well-formed if s and t already have type A. The rule for if-then-else is interesting because a local assumption for the truth/falsity of the condition s is available to show the well-formedness of the expression t and t' in the then/else branch. Similarly, the rules for the quantifiers (here shown: $\forall$) and binary logical operators (here shown: $\wedge$) may use the truth of their first part to show the well-formedness of the second part; for the binary operators, that corresponds to lazy evaluation. That is important because the typing rules for predicate and quotient types are able to use those assumptions.

The proof rules for the binary operators (here shown: the introduction rule for $\wedge$) are similarly sequential. Contrary to both standard hard-typed higher-order logic and soft-typed Mizar, we allow empty types, which is the natural choice when working with predicate types. Therefore, some quantifier rules (here shown: the elimination rule of the universal) only allow terms whose free variables are from the current context; consequently, e.g., $\forall x : A.t(x) \Rightarrow \exists x : A.t(x)$ is only a theorem if A is known to be non-empty.

Type formation:

$$\frac{\Gamma \vdash A : \mathtt{type} \quad \Gamma \vdash p : A \to \mathtt{bool}}{\Gamma \vdash A|p : \mathtt{type}} \qquad \frac{\Gamma \vdash A : \mathtt{type} \quad \Gamma \vdash r : A \to A \to \mathtt{bool}}{\Gamma \vdash A/r : \mathtt{type}}$$

Introduction (TDD):

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash p\,t}{\Gamma \vdash t : A|p} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : A/r}$$

Elimination:

$$\frac{\Gamma \vdash t : A|p}{\Gamma \vdash t : A} \qquad \frac{\Gamma \vdash t : A|p}{\Gamma \vdash p\,t}$$

$$\frac{\Gamma \vdash s : A/r \quad \Gamma, x : A \vdash t(x) : B \quad \Gamma, x : A, y : A, r\,x\,y \vdash t(x) =_B t(y)}{\Gamma \vdash t(s) : B}$$

Equality (TDE):

$$\frac{\Gamma \vdash s : A|p \quad \Gamma \vdash t : A|p \quad \Gamma \vdash s =_A t}{\Gamma \vdash s =_{A|p} t} \qquad \frac{\Gamma \vdash s : A \quad \Gamma \vdash t : A \quad \Gamma \vdash r\,s\,t}{\Gamma \vdash s =_{A/r} t}$$

Fig. 2. Rules for Predicate and Quotient Types

Rules for Predicate and Quotient Types Fig. 2 gives all rules for predicate and quotient types. Together with the proof rules below, it is straightforward to show that these satisfy the abstract specifications described in Sect. 2: introduction and elimination forms are no-ops, and computation, representation, and extensionality are trivial.

Note that the equality rule for quotient types does not have to construct the equivalence closure of r. We only make all terms equal that satisfy $r\,s\,t$, at which point the usual rules for equality already induce the equivalence closure.

Equality introduction and elimination:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t =_A t} \qquad \frac{\Gamma \vdash s =_A s' \qquad \Gamma, x : A \vdash t(x) : \texttt{bool} \qquad \Gamma \vdash t(s)}{\Gamma \vdash t(s')}\,\text{Sub}$$

Fig. 3. Rules for Equality

Rules for Equality Fig. 3 gives the rules for equality. Reflexivity is the introduction rule. As usual, symmetry and transitivity can be derived.

The elimination rule Sub is the usual substitution rule. It is deceptively simple: its soundness is very sensitive to subtle variations in the language. The problem is that the type A may affect the derivability of the premise $s =_A s'$ but does not explicitly occur in the conclusion of the rule. Thus, whenever $t(s)$ and $\neg t(s')$, we can try to force an inconsistency by choosing some A at which s and s' are equal, e.g., a suitable quotient of a sufficiently large type. The type system must prevent that by allowing $t(x)$ to use the variable $x : A$ only in ways that cannot distinguish A-equal terms.

For example, it may be unexpected that our syntax does not include `bool`-values terms $t \in A$. We could include that, say with a formation rule

$$\frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : \texttt{type}}{\Gamma \vdash t \in A : \texttt{bool}}$$

Here the first premise is necessary to ensure that only well-typed terms t may be used. But there is a deep problem: it would allow constructing terms that do not preserve equality. For example, assume a theory that declares the usual type $\mathbb{N}$ and constants for the natural numbers, and let $\texttt{mod}_m$ be equivalence modulo m and `Prime` be the prime number property. Then for a variable $x : \mathbb{N}/\texttt{mod}_2$, the boolean $x \in \mathbb{N}|\texttt{Prime}$ would be well-formed and equivalent to $\texttt{Prime}(x)$. But that boolean would break the soundness of Sub as we have $2 =_{\mathbb{N}/\texttt{mod}_2} 4$ but $2 \in \mathbb{N}|\texttt{Prime}$ and $4 \notin \mathbb{N}|\texttt{Prime}$.

Remark 1. The author had originally included $t \in A : \texttt{bool}$ in the syntax but failed to obtain soundness even after trying multiple variants of the syntax and formation rule of $t \in A$.

However, without a `bool`-valued predicate $\in$, not much expressivity is lost: we can still express $x \in \mathbb{N}|p$ by simply putting $p\,x$. And we can easily add syntactic sugar to recover $\in$-based notations for that. Despite the similar expressivity, the inconsistency problem does not arise in that case: if $x : \mathbb{N}/\mathtt{mod}_2$, the boolean $\mathtt{Prime}\,x$ is ill-formed because $\mathtt{Prime} : \mathbb{N} \to \mathtt{bool}$ cannot be applied to $x : \mathbb{N}/\mathtt{mod}_2$.

As a general language design principle, we must be careful what the syntax lets us do with terms of quotient types. In particular, given a variable $x : A/r$, it must not be allowed to inspect x via arbitrary predicates on A. The elimination rule of the quotient type already guarantees that only equality-preserving operations can be applied. But we must also check the interaction of quotient types with every other primitive operation in the language. This is formally established in the soundness theorem below.

Subtyping In soft-typed systems, subtyping $A <: B$ is usually defined via

$$x : A \text{ implies } x : B \text{ for all } x \qquad (a)$$

In the presence of TDE, the following stronger condition is more practical:

$$x : A, y : A, x =_A y \text{ implies } x =_B y \text{ for all } x, y \qquad (b)$$

Note that (b) implies (a). (b) is only relevant in the presence of TDE: without TDE, it trivially follows from (a). Therefore, we use (b) to define the subtyping judgment $\vdash A <: B$.

We can now derive the usual contra/co-variance rules for $A \to B$ as well as $A|p <: A$ and $A <: A/r$. Maybe surprisingly, the quotient type becomes a supertype of the base type even though its semantics is a set with lower cardinality. For every type A, we have the following subtype hierarchy from the void type to the unit type:

$$A|\lambda x.\mathtt{false} <: A|p <: A|\lambda x.\mathtt{true} = A = A/\lambda xy.\mathtt{false} <: A/r \subseteq A/\lambda xy.\mathtt{true}$$

On the left hand side, these capture the no-op canonical embeddings of predicate types via increasingly inclusive predicates, on the right hand side the no-op canonical projections of the quotient types via increasingly inclusive predicates.

3.2 Semantics

Partial Equivalence Relations We recall the basic properties of partial equivalence relations (PERs). A PER S on set U is a symmetric and transitive binary relation on U. We write $\operatorname{dom} S = \{u \in U | (u,u) \in S\} = \{u \in U | \exists v \in U.(u,v) \in S\}$ for the set of elements touched by S, and $S|_V = S \cap V^2$ for the restriction of S to $V \subseteq U$, and $\mathrm{PER}(R)$ for the PER generated by the binary relation R on U. Then $S|_{\operatorname{dom} S}$ is an equivalence relation on $\operatorname{dom} S$, and we write $S^{/}$ for the corresponding quotient (in the usual set-theoretical sense).

Overview We assume a set-theoretical universe U and interpret syntax according to the table below:

Syntax	Semantics	Intended meaning
type A	$[\![A]\!] \subseteq U \times U$	$\mathrm{PER}([\![A]\!])^{/}$
terms t	elements $[\![t]\!] \in U$	
typing $\vdash t : A$	$[\![t]\!] \in \mathrm{dom}\,[\![A]\!]$	equivalence class of $[\![t]\!]$ in $\mathrm{PER}([\![A]\!])^{/}$
equality $\vdash s =_A t$	$([\![s]\!], [\![t]\!]) \in \mathrm{PER}([\![A]\!])$	
subtyping $\vdash A <: B$	$[\![A]\!] \subseteq [\![B]\!]$	

Note that even though equality depends on the type, the interpretation of terms does not. Every term has an absolute meaning defined by induction on the language of terms, not by induction on typing derivations. The intended meaning of a type is a quotient of a subset of U. Terms can have multiple types, and the intended meaning of term t seen as an element of type A is the equivalence class of $[\![t]\!]$ in the intended meaning of A.

Interpretation of Identifiers A *model* maps every part of the theory to its interpretation:

- a base type a to a PER $[\![a]\!] \subseteq U \times U$
- a constant $c : A$ to an element $[\![c]\!] \in \mathrm{dom}\,[\![A]\!]$
- an axiom t to a proof that t holds.

An *assignment* maps every part of a context to its interpretation accordingly. We write $\alpha, x \mapsto u$ for the extension of α with a case for x. The function $[\![-]\!]$ is actually relative to a model of the theory (which we omit from the notation) and an assignment α for the context. Note that the interpretation of types depends on the assignment even in the absence of type variables because terms can occur in types.

Interpretation Function Given a fixed model (which we omit from the notation), we define the *interpretation function* $[\![-]\!]^\alpha$ for all types and terms in context Γ under an assignment α to Γ.

Constants a and c are interpreted according to the model, variables k and x according to the assignment. The remaining cases are:

$$[\![\mathtt{bool}]\!]^\alpha = \{(0,0),(1,1)\} \quad \text{i.e.,} \quad \mathrm{dom}\,[\![\mathtt{bool}]\!]^\alpha = \{0,1\}$$

$$[\![\mathtt{true}]\!]^\alpha = 1 \qquad [\![\mathtt{false}]\!]^\alpha = 0 \qquad [\![\mathtt{if}(s)t\,\mathtt{else}\,t']\!]^\alpha = \begin{cases} [\![t]\!]^\alpha & \text{if } [\![s]\!]^\alpha = 1 \\ [\![t']\!]^\alpha & \text{otherwise} \end{cases}$$

$$[\![s =_A t]\!]^\alpha = \begin{cases} 1 & \text{if } ([\![s]\!]^\alpha, [\![t]\!]^\alpha) \in \mathrm{PER}([\![A]\!]) \\ 0 & \text{otherwise} \end{cases}$$

$$[\![A \to B]\!]^\alpha = \\ \left\{(f,g) \in (\mathrm{dom}\,[\![A]\!]^\alpha \to \mathrm{dom}\,[\![B]\!]^\alpha)^2 \mid (u,v) \in [\![A]\!]^\alpha \text{ implies } (f(u), g(v)) \in [\![B]\!]^\alpha\right\}$$

$$[\![\lambda x : A.t(x)]\!]^\alpha = \left\{(u, [\![t(x)]\!]^{\alpha, x \mapsto u}) \,:\, u \in \mathrm{dom}\,[\![A]\!]^\alpha\right\}$$

$$[\![f\ a]\!]^\alpha = [\![f]\!]^\alpha([\![a]\!]^\alpha)$$
$$[\![A|p]\!]^\alpha = [\![A]\!]^\alpha \ \cap\ \left\{u \in \operatorname{dom}[\![A]\!]^\alpha \mid [\![p]\!]^\alpha(u) = 1\right\}^2$$
$$[\![A/r]\!]^\alpha = [\![A]\!]^\alpha \ \cup\ \left\{(u,v) \in \operatorname{dom}[\![A]\!]^{\alpha\,2} \mid [\![r]\!]^\alpha(u)(v) = 1\right\}$$

where we abbreviate $V^2 = V \times V$ as usual.

Soundness The soundness theorem consists of multiple statements:
- the main theorem that provable booleans hold (3)
- the well-definedness of the interpretation function (1+2)
- the usual substitution lemma that interpretation commutes with substitution (4a)
- a lemma specific to the PER semantics that ensures that every term with free variables preserves equality (4b)

Theorem 1. *For any theory and model, we have*
1. *if* $\Gamma \vdash A : \texttt{type}$*, then* $[\![A]\!]^\alpha \subseteq U \times U$
2. *if* $\Gamma \vdash t : A$*, then* $[\![t]\!]^\alpha \in \operatorname{dom}[\![A]\!]^\alpha$ *for all* α
3. *if* $\Gamma \vdash b$*, then* $[\![b]\!]^\alpha = 1$ *for all* α
4. *for any* $\Gamma, x : A \vdash t(x) : B$
   (a) *if* $\Gamma \vdash s : A$*, then* $[\![t(s)]\!]^\alpha = [\![t(x)]\!]^{\alpha, x \mapsto [\![s]\!]}$ *for all* α
   (b) *if* $(u,v) \in [\![A]\!]^\alpha$*, then* $([\![t(x)]\!]^{\alpha,x\mapsto u}, [\![t(x)]\!]^{\alpha,x\mapsto v}) \in [\![B]\!]^\alpha$ *for all* α

Proof. All statements are proved in a joint induction on derivations. We only mention a few critical cases. (4b) is needed to prove (2) for the case of λ-abstraction. (4a) and (4b) are needed to prove (3) for the case of Sub.

We have so far not investigated any completeness properties. It is reasonable to expect those are related to the completeness of HOL for Henkin models. But it is non-obvious how to combine Henkin and PER semantics.

3.3 Lax Record Types

There are several ways to extend the language with record types. Lax records are particularly attractive with TDE, and we sketch one way to add them.

Syntax and Semantics We use contexts as record types and substitutions γ as record values. That yields relatively powerful record types, which may contain type fields, value fields, and axioms:

$$\begin{aligned} A &::= \{\Gamma\} \mid t.k \\ t &::= [\gamma] \mid t.x \\ \gamma &::= (k = A \mid x = t \mid P)^* \\ P &::= \text{(proof terms omitted)} \end{aligned}$$

A substitution γ for context Γ maps every type/term/assumption declaration in Γ to an appropriate type/term/proof. The last case of that requires extending the syntax with a term language for proofs, which we omit.

We omit the typing and equality rules, which are complex but routine. For example, the equality rule for an example record type is

$$\frac{\Gamma \vdash r.k = s.k : \texttt{type} \qquad \Gamma \vdash r.x =_{r.k} s.x}{\Gamma \vdash r =_{\{k:\texttt{type},x:k\}} s}$$

where $r.k = s.k : \texttt{type}$ is an instance of the type equality judgment we omitted above. The semantics is straightforward except for the usual problem of needing some kind of universe hierarchy because record types containing type fields are too big to be interpreted as a set in the universe. We gloss over that issue.

Mizar's strictness operator is not needed because lax records with TDE allows expressing equalities at different record types. Like with predicate and quotient typing, applying a forgetful functor is a no-op.

Subtyping and TDE With the addition of lax records, we can form subtypes via record subtyping in addition to predicate subtyping. The main effect this has on the language design is that the argument of Rem. 1 becomes less compelling: While a membership test $t \in A|p$ can be replaced with $p\,t$, a similar workaround does not exist for record subtyping.

Assume we added a `bool`-valued predicate $r \in S : \texttt{bool}$ for $r : R$ and record types $S <: R$. For example, this would allow inspecting an input $x : R$ to see if it provides more fields than guaranteed by R. A typical application would be to employ a more efficient semigroup algorithm if the input is a monoid. This can also be used if the additional fields are not uniquely determined, e.g., to check if a vector space comes with a distinguished base. Membership tests like this are routine in soft-typed computer algebra systems such as Gap [Lin07] or SageMath [S+13].

But attempts to add such tests to the syntax run into soundness issues. For example, assume record types $Monoid <: Semigroup$, a record $M : Semigroup$ that happens to satisfy the monoid axioms, and $M' : Monoid$ arising from M by adding the additional fields. Then $M =_{Semigroup} M'$ and $M \notin Monoid$ and $M' \in Monoid$, which makes Sub unsound if a naive membership test is added.

The author currently does not have a satisfactory solution for soundly testing record membership in the presence of TDE.

4 Conclusion

We coined the term "type-dependent equality" (TDE) for an existing but not widely known feature in soft-typed languages. We provided an overview of the advantages and pitfalls of designing formal systems with TDE and described their realizations in major proof assistants. We have used that to design a simple language with TDE that allows for an elegant treatment of predicate and quotient types. Importantly, many critical operations are no-ops, which is advantageous notationally and computationally.

Many aspects of the work are folklore such as the PER semantics for soft type systems or have been implemented before such as Nuprl's TDE and quotient

types. The main contribution is to collect and analyze all these aspects in a simple formal language that exhibits the main characteristics while allowing a rigorous and clear presentation. Critically, the soundness theorem clarifies the consistency issues that must be taken care of when designing TDE-languages. And the syntax and semantics are simple enough to make the formal verification of the soundness theorem feasible.

Thus, the work provides an ideal starting point for designing more advanced TDE-languages that could allow for better formalizations of mathematical practices than supported by current proof assistants.

References

ABC+06. S. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

All87. S. Allen. *A Non-type-theoretic Semantics for Type-theoretic Language*. PhD thesis, Cornell University, 1987.

CAB+86. R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

Coq15. Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

dKA+15. L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In A. Felty and A. Middeldorp, editors, *Automated Deduction*, pages 378–388. Springer, 2015.

Gor88. M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.

KR20. C. Kaliszyk and F. Rabe. A Survey of Languages for Formalizing Mathematics. In C. Benzmüller and B. Miller, editors, *Intelligent Computer Mathematics*, pages 138–156. Springer, 2020.

Lin07. S. Linton. GAP: groups, algorithms, programming. *ACM Communications in Computer Algebra*, 41(3):108–109, 2007.

Nog02. A. Nogin. Quotient Types: A Modular Approach. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 263–280. Springer, 2002.

ORS92. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.

PC93. L. Paulson and M. Coen. Zermelo-Fraenkel Set Theory, 1993. Isabelle distribution, ZF/ZF.thy.

S+13. W. Stein et al. *Sage Mathematics Software*. The Sage Development Team, 2013. http://www.sagemath.org.

TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.

Wie07. F. Wiedijk. The QED Manifesto Revisited. In *From Insight to Proof, Festschrift in Honour of Andrzej Trybulec*, pages 121–133, 2007.