A Calculus for Type Predicates and Type Coercion

Martin Giese

Johann Radon Institute for Computational and Applied Mathematics,
Altenbergerstr. 69, A-4040 Linz, Austria
martin.giese@oeaw.ac.at

Abstract. We extend classical first-order logic with subtyping by type predicates and type coercion. Type predicates assert that the value of a term belongs to a more special type than the signature guarantees, while type coercion allows using terms of a more general type where the signature calls for a more special one. These operations are important e.g. in the specification and verification of object-oriented programs. We present a tableau calculus for this logic and prove its completeness.

1 Introduction

When mathematicians have shown that the ratio $\frac{p}{q}$ is in fact a natural number, they have no qualms about writing $(\frac{p}{q})!$, even though the factorial is not defined for a rational argument which $\frac{p}{q}$, syntactically speaking, is. This is very different from the common practice in strongly typed programming languages, like e.g. Java: here, an explicit *type coercion* or *cast* is needed in most cases to adjust the type of the fraction to the the required argument type of the factorial function, maybe $(\mathit{rat\text{-}to\text{-}nat}(\frac{p}{q}))!$. The compiled program might check at runtime that the value of the fraction is indeed a natural number, and signal an error otherwise. Typically, such a strongly typed programming language will also provide *type predicates*, which allow checking whether a value is indeed of a more special type than its syntax implies.

Previous work published about tableaux for logics with subtyping ('order-sorted logic'), e.g. [3,2] does not consider type coercions. There are no type predicates in the work of Schmitt and Wernecke, whereas Weidenbach does not consider static typing. We will amend these shortcomings in the present paper, by first defining in Sect. 2 a logic with static typing, subtypes, type coercions, type predicates, and equality, which is not too complicated. In Sect. 3, we give a tableau calculus for this logic, which is somewhat more complicated than might be expected. And in Sect. 4, we give the completeness proof for that calculus, which is actually the main contribution of this paper.

2 A Logic with Type Predicates and Type Coercions

2.1 Types

Let us first clarify an important distinction. In the definition of our logic, there are going to be two kinds of entities that will carry a type: terms and domain elements.

B. Beckert (Ed): TABLEAUX 2005, LNAI 3702, pp. 123–137, 2005.

- The type of a term is given by the signature, more precisely, a term's type is the type declared as return type of the term's outermost function symbol. The syntax is defined in such a way that a term of a certain type can be used wherever the syntax calls for a term of a supertype. To make it clear that we mean this notion of type, we will talk about the *static type of a term*.
- When we *evaluate* a term using some interpretation, we get an element of the domain. Every element of the domain has exactly one type. Our semantics is defined in such a way that the type of the value of a term will be a subtype of, or equal to the static type of the term. When we mean this notion of type, we will talk about the *dynamic type of a domain element*.

For example let us assume two types Z, signifying the integers, and Q, signifying the rationals. We want Z to be a subtype of Q. $zero, one, two : Z$ are constants of static type Z. The operation $div : Q, Q \to Q$ produces a term of static type Q from two terms of static type Q. Since Z is a subtype of Q, we can use two as argument for div. The static type of the term $div(two, two)$ is Q, since that is the return type of div. The domain of the standard model consists of the set of rational numbers, where the integers have dynamic type Z, and all other elements have dynamic type Q. In the standard model, the *value* of $div(two, two)$ is the domain element 1. The dynamic type of the value of $div(two, two)$ is therefore Z, which is all right since Z is a subtype of Q.

We assume as given a set of types $\mathcal{T}$ with a partial ordering $\sqsubseteq$. We require $\mathcal{T}$ to be closed under greatest lower bounds, i.e. for any two $A, B \in \mathcal{T}$, there is a $C \in \mathcal{T}$ with $C \sqsubseteq A$ and $C \sqsubseteq B$, such that for any other $D \in \mathcal{T}$ with $D \sqsubseteq A$ and $D \sqsubseteq B$, $D \sqsubseteq C$. It is well known that C is then uniquely defined, and we write $A \sqcap B$ for C.

In the approach of Weidenbach [2], types are just unary predicates and the relationship between types is given by axioms of a certain form. These unary predicates correspond to our type predicates, which limit the dynamic type of a domain element. There is no concept of static typing of terms and variables. Indeed, there is a straightforward modelling of type casts in the Weidenbach setting, but it requires working with ill-typed terms. In our calculus, only well-typed formulae are ever constructed.

2.2 Syntax

A signature for our logic consists of a set of function symbols and predicate symbols. Each function and predicate symbol has a (possibly empty) list of argument types and function symbols also have a return type. We write

$$f : A_1, \ldots, A_n \to A$$

resp.

$$p : A_1, \ldots, A_n$$

to express that the function f, resp. predicate p has argument types $A_1, \ldots, A_n$ and return type A. Constants are included in this definition as functions without arguments.

To simplify things, we do not allow overloading: any function or predicate symbol can have only one list of argument types and return type.

We also provide ourselves with a set of variables, each of which has a type. We write $v : A$ to say that v has type A.

The syntax of our logic is like that of usual classical first order logic, except that we have a series of function symbols and a series of predicate symbols with a predefined meaning, and for which we will use a special syntax.

Definition 1. *We inductively define the system of sets* $\{T_A\}_{A \in \mathcal{T}}$ *of* terms of static type A *to be the least system of sets such that*

- $x \in T_A$ *for any variable* $x : A$,
- $f(t_1, \ldots, t_n) \in T_A$ *for any function symbol* $f : A_1, \ldots, A_n \to A$, *and terms* $t_i \in T_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *for* $i = 1, \ldots, n$,
- $(A)t \in T_A$ *for any term* $t \in T_{A'}$ *where* A' *is an arbitrary type.*

We write the static type of t *as* $\sigma(t) := A$ *for any term* $t \in T_A$.

A term $(A)t$ is a *type coercion*, also called a *type cast* or *cast* for short. Its intended semantics is that whatever the static type of t, *if* the dynamic type of the value of t is $\sqsubseteq A$, then the value of $(A)t$ is the same as that of t, but the term has the static type A, permitting it to be used in a context where a term of that type is required.

Definition 2. *We inductively define the set of formulae* F *to be the least set such that*

- $p(t_1, \ldots, t_n) \in F$ *for any predicate symbol* $p : A_1, \ldots, A_n$ *and terms* $t_i \in T_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ *for* $i = 1, \ldots, n$,
- $t_1 \doteq t_2 \in F$ *for any terms* $t_1 \in T_{A_1}$ *and* $t_2 \in T_{A_2}$,
- $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi, \ldots \in F$ *for any* $\phi, \psi \in F$.
- $\forall x.\phi$, $\exists x.\phi \in F$ *for any* $\phi \in F$ *and any variable* x.
- $t \sqsubseteq A \in F$ *for any term* t *and type* A.

The formulae $t_1 \doteq t_2$ *and* $t_2 \doteq t_1$ *are considered syntactically identical. An* atom *is a formula of the shape* $p(t_1, \ldots, t_n)$, *or* $t_1 \doteq t_2$, *or* $t \sqsubseteq A$. *A* literal *is an atom or a negated atom. A closed formula is defined as usual to be a formula without free variables.*

The formula $t \sqsubseteq A$ is intended to express that the dynamic type of the value of t is a subtype of or equal to A.

2.3 Semantics

The terms and formulae of our logic will be evaluated with respect to a *structure* $\mathcal{S} = (\mathcal{D}, \mathcal{I})$, consisting of a *domain* and an *interpretation*.

The semantics of a term is going to be a value in the domain $\mathcal{D}$. Each domain element $x \in \mathcal{D}$ has a dynamic type $\delta(x) \in \mathcal{T}$.

While each domain element has a unique dynamic type, it may still occur as the value of terms of different static types. Our semantic definitions will be such, that the dynamic type of the value of a term is guaranteed to be a subtype of the static type of the term. We denote the set of valid domain elements for a certain static type by

$$\mathcal{D}_A := \{x \in \mathcal{D} \mid \delta(x) \sqsubseteq A\}$$

We will require each of these sets to be non-empty:

$$\mathcal{D}_A \neq \emptyset \quad \text{for all types } A \in \mathcal{T}.$$

This restriction spares us the trouble of handling quantification over empty domains.

An interpretation $\mathcal{I}$ assigns a meaning to every function and predicate symbol given by the signature. More precisely, for a function symbol

$$f : A_1, \ldots, A_n \to A \quad ,$$

the interpretation yields a function

$$\mathcal{I}(f) : \mathcal{D}_{A_1} \times \ldots \times \mathcal{D}_{A_n} \to \mathcal{D}_A$$

and for a predicate symbol

$$p : A_1, \ldots, A_n$$

some set of tuples of domain elements.

$$\mathcal{I}(p) \subseteq \mathcal{D}_{A_1} \times \ldots \times \mathcal{D}_{A_n} \quad .$$

For type coercions, we require $\mathcal{I}$ to exhibit a particular behaviour, namely

$$\mathcal{I}((A))(x) = x \quad \text{if } \delta(x) \sqsubseteq A.$$

Otherwise, $\mathcal{I}((A))(x)$ may be an arbitrary element of $\mathcal{D}_A$. Finally, for type predicates, we require that

$$\mathcal{I}(\sqsubseteq A) = \mathcal{D}_A \quad .$$

A variable assignment β is simply a function that assigns some domain value $\beta(x) \in \mathcal{D}_A$ to every variable $x : A$.

β^d_x denotes the modification of the variable assignment β such that $\beta^d_x(x) = d$ and $\beta^d_x(y) = \beta(y)$ for all variables $y \neq x$.

For a structure $\mathcal{S} = (\mathcal{D}, \mathcal{I})$, we can now define the valuation function $\mathrm{val}_\mathcal{S}$ that takes a variable assignment and a term and yields the domain element that is the value of that term. Our definitions will ensure that $\mathrm{val}_\mathcal{S}(\beta, t) \in \mathcal{D}_A$ for any $t \in T_A$.

Definition 3. *We inductively define the valuation function* $\mathrm{val}_\mathcal{S}$ *by*

- $\mathrm{val}_\mathcal{S}(\beta, x) = \beta(x)$ *for any variable* x.
- $\mathrm{val}_\mathcal{S}(\beta, f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_\mathcal{S}(\beta, t_1), \ldots, \mathrm{val}_\mathcal{S}(\beta, t_n))$.
- $\mathrm{val}_\mathcal{S}(\beta, (A)t) = \mathcal{I}((A))(\mathrm{val}_\mathcal{S}(\beta, t))$.

For a ground term t, we simply write $\mathrm{val}_{\mathcal{S}}(t)$.

Next, we define the validity relation $\mathcal{S}, \beta \models \phi$ that says whether a formula ϕ is valid in $\mathcal{S} = (\mathcal{D}, \mathcal{I})$ under some variable assignment β.

Definition 4. *We inductively define the validity relation* $\models$ *by*

- $\mathcal{S}, \beta \models p(t_1, \ldots, t_n)$ *iff* $(\mathrm{val}_{\mathcal{S}}(\beta, t_1), \ldots, \mathrm{val}_{\mathcal{S}}(\beta, t_n)) \in \mathcal{I}(p)$.
- $\mathcal{S}, \beta \models t_1 \doteq t_2$ *iff* $\mathrm{val}_{\mathcal{S}}(\beta, t_1) = \mathrm{val}_{\mathcal{S}}(\beta, t_2)$.
- $\mathcal{S}, \beta \models \neg\phi$ *iff not* $\mathcal{S}, \beta \models \phi$, *etc. for* $\phi \vee \psi$, $\phi \wedge \psi, \ldots$
- $\mathcal{S}, \beta \models \forall x.\phi$ *iff* $\mathcal{S}, \beta^d_x \models \phi$ *for every* $d \in \mathcal{D}_A$ *where* $x : A$, *i.e. A is the static type of the variable x. Similarly for* $\mathcal{S}, \beta \models \exists x.\phi$.
- $\mathcal{S}, \beta \models t \sqsubseteq A$ *iff* $\delta(\mathrm{val}_{\mathcal{S}}(\beta, t)) \sqsubseteq A$.

We write $\mathcal{S} \models \phi$, *for a closed formula ϕ, since β is then irrelevant. A closed formula ϕ is* unsatisfiable *iff* $\mathcal{S} \models \phi$ *for no structure ϕ.*

3 A Tableau Calculus

The main difficulty in finding a complete calculus for this logic is that it must be possible to prove e.g. a formula $s \doteq t$ containing casts in the presence of various type predicate literals $u \sqsubseteq A$, $v \sqsubseteq B$, if the latter imply the equality of s and t. The first idea would be to have a rule

$$\frac{\begin{array}{c}\phi[z/(A)t] \\ t \sqsubseteq A\end{array}}{\phi[z/t]}\ \mathit{wrong}$$

that allows us to *remove* a cast if there is a type predicate that guarantees that the cast is successful. The problem here is that the formula $\phi[z/t]$ might no longer be well-typed, since the static type of t might be too general for the position where $(A)t$ stood – in fact, that is probably why the cast is there in the first place. One needs to go the opposite way: there must be a rule that allows to *add* casts wherever they are guaranteed not to change the semantics of a term. But this leads to difficulties in the completeness proof, since such a rule makes terms grow, defying the usual induction arguments employed in a Hintikka style completeness proof. We show how to solve this problem in Sect. 4.

In order to avoid the duplication of rules due to duality, we give a calculus for formulae in negation normal form, i.e. negation appears only before atoms $p(\ldots)$, or $t_1 \doteq t_2$, or $t \sqsubseteq A$.

The calculus is a 'ground', i.e. Smullyan style tableau calculus, without free variables. We expect it to be possible without difficulties to design a free variable version of the calculus, using order-sorted unification, in the style of [3,2].

See Fig. 1 for the rules of our calculus. Starting from an initial tableau containing only one branch with one formula, namely the one that we want to prove unsatisfiable, we build a tableau by applying these rules, adding the formulae beneath the line whenever the ones above the line are present. The β

$$\frac{\phi \wedge \psi}{\begin{array}{c}\phi\\ \psi\end{array}}\ \alpha \qquad \frac{\phi \vee \psi}{\phi \quad \psi}\ \beta$$

$$\frac{\forall x.\phi}{\phi[x/t]}\ \gamma \qquad \frac{\exists x.\phi}{\phi[x/c]}\ \delta$$

with $t \in T_A$ ground, if $x : A$. with $c : A$ a new constant, if $x : A$.

$$\frac{\begin{array}{l}\phi[z/t_1]\\ t_1 \doteq t_2\end{array}}{\phi[z/t_2]}\ \text{apply-}\doteq \quad \text{if } \sigma(t_2) \sqsubseteq \sigma(t_1). \qquad \frac{\begin{array}{l}\phi[z/t_1]\\ t_1 \doteq t_2\end{array}}{\phi[z/(A)t_2]}\ \text{apply-}\doteq' \quad \text{with } A := \sigma(t_1).$$

$$\frac{t_1 \doteq t_2}{\begin{array}{l}t_2 \sqsubseteq \sigma(t_1)\\ t_1 \sqsubseteq \sigma(t_2)\end{array}}\ \text{type-}\doteq \qquad \frac{\begin{array}{c}t \sqsubseteq A\\ t \sqsubseteq B\end{array}}{t \sqsubseteq A \sqcap B}\ \text{type-}\sqcap$$

$$\frac{}{t \sqsubseteq \sigma(t)}\ \text{type-static}$$

$$\frac{\begin{array}{l}\phi[z/t]\\ t \sqsubseteq A\end{array}}{\phi[z/(A)t]}\ \text{cast-add} \quad \text{where } A \sqsubseteq \sigma(t). \qquad \frac{(\neg)(A)t \sqsubseteq B}{\begin{array}{l}t \sqsubseteq A\\ (\neg)t \sqsubseteq B\end{array}}\ \text{cast-type}$$

$$\frac{\phi[z/(A)t]}{\phi[z/t]}\ \text{cast-del} \quad \text{where } \sigma(t) \sqsubseteq A.$$

$$\frac{\begin{array}{l}t \sqsubseteq A\\ \neg t \sqsubseteq B\end{array}}{\bot}\ \text{close-}\sqsubseteq \quad \text{with } A \sqsubseteq B. \qquad \frac{\neg t \doteq t}{\bot}\ \text{close-}\doteq$$

$$\frac{\phi, \neg\phi}{\bot}\ \text{close}$$

Fig. 1. The rules of our calculus

rule splits the current branch into two new branches, all other rules don't split. A tableau is closed, if $\bot$ is present on all branches.

Note in particular the cast-add rule, which can add a cast around a term, whenever there is a type predicate to guarantee that the value of the term remains the same with the cast.

The type-$\doteq$ rule might be a little unexpected at first. Remember that if $t_1 \doteq t_2$ holds, the terms have the same value $v \in \mathcal{D}$, so their values have the same dynamic type $\delta(v)$. On the other hand, the dynamic types of the values of terms are subtypes of their static types, so $\delta(v) \sqsubseteq \sigma(t_1)$ and $\delta(v) \sqsubseteq \sigma(t_2)$, which

motivates the type-$\doteq$ rule. It is needed to deal with equalities between terms of unequal static types, see the proof of Lemma 4.

The rule cast-type allows us to remove the cast from $(A)t$, if $t \sqsubseteq A$ is present, but only in the special case of the top-level of a type predicate literal. Such literals are a special case in our completeness proof, see Lemma 6.

Theorem 1. *A closed tableau can be constructed from a closed formula ϕ, iff ϕ is unsatisfiable.*

The proof of soundness, i.e. that a formula for which a closed tableau exists is indeed unsatisfiable, is rather standard, so we will not go into it here. The interesting part is the completeness proof, which is the main result of this paper.

4 Completeness

The standard approach for proving completeness fails because the cast 'normalization' rule cast-add, which is to be used to make semantically equal terms syntactically equal, makes terms grow. We can however work around this by using a modified signature and calculus.

Modified Signature. We show completeness in a modified calculus that uses an extended signature. We then show how a proof in the modified calculus can be transformed into one in the normal calculus. We modify the signature by adding a function symbol $f^A : \ldots \to A$ for any function symbol $f : \ldots \to B$ with $A \sqsubseteq B$. The semantics of $f^A(\ldots)$ will be defined to be the same as that of $f(\ldots)$, only with the static type A. Similarly, we add modified casts $(B)^A t$ with static type A for $A \sqsubseteq B$. We will say that $f^A(\ldots)$ and $(B)^A t$ carry a *superscript cast to* A. The idea of these superscript casts is to normalize terms such that each term carries a cast to the most specific type known for it. The modified calculus is just like the one of Fig. 1, except that rules cast-add and cast-type are replaced by those shown in Fig. 2, and the cast-strengthen rule is added.

$$\frac{\phi[z/t] \qquad t \sqsubseteq A}{\phi[z/t^A]}\ \text{cast-add} \quad \text{where } A \sqsubseteq \sigma(t). \qquad \frac{(\neg)t^A \sqsubseteq B \qquad t \sqsubseteq A}{(\neg)t \sqsubseteq B}\ \text{cast-type} \qquad \frac{\phi[z/t^A] \qquad t \sqsubseteq B}{\phi[z/t^B]}\ \text{cast-strengthen} \quad \text{where } B \sqsubseteq A.$$

Fig. 2. Alternative rules in the modified calculus

In these rules, t^A stands for either $f^A(\ldots)$ or $(B)^A t'$, depending on what kind of term t is. However, t itself may not have a superscript cast in the rules cast-add and cast-strengthen, since a term can have only one superscript cast. The rule cast-del of the original calculus can now be applied for casts with or without superscript casts.

To show completeness of this modified calculus, we need to show a *model lemma* that states that a (possibly infinite) open branch on which a certain set of relevant rule applications has been performed has a model. Showing from this that there is a closed tableau for every unsatisfiable formula ϕ is then standard.

The initial formula ϕ contains only a finite set of types as static types of terms or in type predicates $t \sqsubseteq A$. Some of the rules of our calculus can introduce finite intersections of types, but there will still always be only finitely many types in a tableau. In particular, $\sqsubseteq$ is a Noetherian ordering on $\mathcal{T}$.

Therefore, we can state the following definition:

Definition 5. *We call a tableau branch H* type-saturated*, if every possible application of type-$\sqcap$ and type-static for all ground terms t has been performed on H.*

Let H be a type-saturated branch. For any term t, we define the most specific known type *$\kappa_H(t)$ w.r.t. H to be the unique least type A under the subtype ordering with $t \sqsubseteq A \in H$.*

Existence and uniqueness of $\kappa_H(t)$ are guaranteed by the saturation w.r.t. the type-static and type-$\sqcap$ rules.

The purpose of our applications of cast-add, cast-strengthen, and cast-del is to bring terms to a normalized form, where the top operator of every subterm is superscripted with the most specific known type for the subterm, and there are no casts $(A)t$ which are redundant in the sense that $\sigma(t) \sqsubseteq A$.

Definition 6. *Let H be type-saturated. A term t is* normalized w.r.t. *H iff*

- *$t = f^A(t_1, \ldots, t_n)$ where all t_i are normalized, and $A = \kappa_H(f(t_1, \ldots, t_n))$, or*
- *$t = (B)^A t'$ where t' is normalized, $A = \kappa_H((B)t')$, and $\sigma(t') \not\sqsubseteq B$.*

We designate the set of ground terms normalized w.r.t. H by Norm_H*. An atom ϕ is* normalized w.r.t. *H, iff for some terms $t_i \in \mathrm{Norm}_H$,*

- *$\phi = t_1 \doteq t_2$,*
- *$\phi = p(t_1, \ldots, t_n)$, or*
- *$\phi = f(t_1, \ldots, t_n) \sqsubseteq A$, or*
- *$\phi = (B)t_1 \sqsubseteq A$, where $\sigma(t_1) \not\sqsubseteq B$.*

A normalized literal is a normalized atom or the negation of one. Note: normalized type predicate atoms do not *have superscript casts on the top level operator.*

Definition 7. *We call a tableau branch H* saturated*, if H is type-saturated, and for each of the following rules, if all premisses are in H, then so is one of the conclusions:*

- *the α, β, and δ rules,*
- *the type-$\doteq$, cast-type, cast-strengthen, cast-del, and close-... rules on literals,*
- *the γ rule for every formula $\forall x.\phi \in H$ and every normalized ground term $t \in \mathrm{Norm}_H \cap T_A$ for $A \sqsubseteq B$ if $x : B$.*

- *the apply-$\doteq$ rule for equations $t_1 \doteq t_2$ with equal static types $\sigma(t_1) = \sigma(t_2)$,*
- *the cast-add rule, except on the top level of a type predicate literal, i.e. $\phi[z/t] = t \sqsubseteq A$ for some A.*

We do not apply cast-add on the top level of type literals $t \sqsubseteq A$: normalizing $t \sqsubseteq A$ to $t^A \sqsubseteq A$ would of course destroy the information of the literal. And rather than normalizing $t \sqsubseteq B$ to $t^A \sqsubseteq B$, we join the literals to $t \sqsubseteq A \sqcap B$.

Saturation w.r.t. cast-add and cast-strengthen implies that terms will be equipped with superscript casts for their most specific known types. Conversely, the following lemma guarantees that a superscript cast cannot appear without the corresponding type predicate literal, if we forbid spurious applications of the γ-rule for non-normalized terms.

Lemma 1. *Let H be a saturated branch obtained by a derivation in which the γ rule is only applied for terms t^A with $t \sqsubseteq A \in H$. Let $\phi \in H$ contain a subterm t^A, i.e. either $f^A(\ldots)$ or $(B)^A t'$. Then H also contains the formula $t \sqsubseteq A$.*

Proof. We show this by induction on the length of the derivation of ϕ from formulae without superscript casts. As induction hypothesis, we can assume that the statement is true for the original formulae in the rule application which added ϕ to the branch. If ϕ results from a rule application where one of the original formulae already contained the term t^A, we are finished. Otherwise, ϕ might be the result of an application of the following rules:

- γ: The assumption of this lemma is that a quantifier can get instantiated with a term t^A only if $t \sqsubseteq A \in H$.
- apply-$\doteq$: If the superscript cast t^A does not occur in the term t_2, there must have been some $s^A[t_1]$ in ϕ which contains an occurrence of t_1 and $t^A = s^A[t_2]$. By the induction hypothesis, $s[t_1] \sqsubseteq A \in H$. But if H is saturated, then we must also have $s[t_2] \sqsubseteq A \in H$.
- cast-add: If ϕ is the result of adding ^{A} to t, then $t \sqsubseteq A$ is of course present. But t^A might also be $s^A[r^C]$ where where $r \sqsubseteq C \in H$ and ^{C} was added to r. Like for apply-$\doteq$, the induction hypothesis guarantees that $s[r] \sqsubseteq A \in H$. Due to saturation, we then also have $s[r^C] \sqsubseteq A \in H$
- cast-strengthen: The same argument as for cast-add applies.
- cast-del: The argument is again the same.

None of the other rules can introduce new terms t^A. □

Domain for Saturated Branch. Given a saturated branch H, we will construct a model that makes all formulae in H true. We start by defining the domain as

$$\mathcal{D} := \mathrm{Norm}_H / \sim_H \quad ,$$

where $\sim_H$ is the congruence induced by those (normalized) ground equations in H that have equal static type on both sides, i.e.

$$s \sim_H t \quad :\Leftrightarrow \quad s = t \text{ or } [s \doteq t \in H \text{ and } \sigma(s) = \sigma(t)]$$

We will denote the typical element of $\mathcal{D}$ by $[t]$ where t is a normalized term, and the [] denote taking the congruence class.

$\sim_H$ is a congruence relation on normalized terms, meaning that

- $\sim_H$ is an equivalence relation on Norm_H.
- If $t \sim_H t'$, then $f^A(\ldots, t, \ldots) \sim_H f^A(\ldots, t', \ldots)$ for all argument positions of all function symbols f^A.
- If $t \sim_H t'$, then $(B)^A t \sim_H (B)^A t'$ for all casts $(B)^A$.

These properties follow from the saturation w.r.t. apply-$\doteq$.

We need to assign a dynamic type to every element $[t] \in \mathcal{D}$. We will simply take the static type of t. As t is normalized, this is the superscript on the outermost operator. The dynamic type is well-defined: the static type of all elements of the congruence class is the same, as we look only at equations of identical static types on both sides.

Interpretation for Saturated Branch. We now define the interpretation $\mathcal{I}$ of function and predicate symbols. We will give the definitions first, and then show that all cases are actually well-defined.

We define

$$\mathcal{I}(f)([t_1], \ldots, [t_n]) := [f^C(t_1, \ldots, t_n)] \qquad [A]$$

where $C := \kappa_H(f(t_1, \ldots, t_n))$. Similarly,

$$\mathcal{I}((B))([t']) := \begin{cases} [t'] & \text{if } \sigma(t') \sqsubseteq B, \\ [(B)^C t'] & \text{otherwise,} \end{cases} \qquad [B]$$

where $C := \kappa_H((B)t')$.

We want the function symbols with superscript casts to be semantically equivalent to the versions without, but we need to make sure that the static types are right. We use the interpretation of casts to ensure this.

$$\mathcal{I}(f^A)([t_1], \ldots, [t_n]) := \mathcal{I}((A))(\mathcal{I}(f)([t_1], \ldots, [t_n])) \quad . \qquad [C]$$

Similarly, for casts with superscript casts,

$$\mathcal{I}((B)^A)([t']) := \mathcal{I}((A))(\mathcal{I}((B))([t'])) \quad . \qquad [D]$$

For predicates, we define

$$([t_1], \ldots, [t_n]) \in \mathcal{I}(p) \quad \Leftrightarrow \quad p(t_1, \ldots, t_n) \in H \quad , \qquad [E]$$

as usual in Hintikka set constructions.

To show that these are valid definitions, we will need the following lemma:

Lemma 2. *Let H be a saturated branch and $t \sim_H t'$. Then $\kappa_H(t) = \kappa_H(t')$.*

Proof. To see that this is the case, note that due to the definition of $\sim_H$, we have either $t = t'$ or $t \doteq t' \in H$. If $t = t'$, the result follows immediately. Otherwise, due to the saturation w.r.t. apply-$\doteq$, it follows that $t \sqsubseteq A \in H$ iff $t' \sqsubseteq A \in H$ for all types A. Thus, the most specific known types for t and t' must be equal. □

We will now show the validity of definitions $[A]$ to $[E]$.

Proof. For $[A]$, we need to show that

- $f^C(t_1, \ldots, t_n) \in \mathrm{Norm}_H$, if all $t_i \in \mathrm{Norm}_H$.
- The static type of $f^C(t_1, \ldots, t_n)$ is a subtype of the result type of f.
- $[f^C(t_1, \ldots, t_n)]$ is independent of the choice of particular $t_1, \ldots, t_n$.

Now, $f^C(t_1, \ldots, t_n)$ is indeed normalized w.r.t. H, if all t_i are normalized, since C is defined as $\kappa_H(f(t_1, \ldots, t_n))$, as required by the definition of normalized terms. As for the second point, the result type of f is of course the static type of $f(t_1, \ldots, t_n)$, which is a supertype of $C = \kappa_H(f(t_1, \ldots, t_n))$, which in turn is the static type of $f^C(t_1, \ldots, t_n)$. Finally, if $t_i \sim_H t'_i$ for $i = 1, \ldots, n$, then $\kappa(f(t_1, \ldots, t_n)) = \kappa(f(t'_1, \ldots, t'_n))$, due to Lemma 2, so C is independent of the choice of representatives. Also, $f^C(t_1, \ldots, t_n) \sim_H f^C(t'_1, \ldots, t'_n)$, since $\sim_H$ is a congruence relation.

For $[B]$, we need to make sure that the cast doesn't change its argument if its dynamic type is $\sqsubseteq B$. Since the dynamic type of $[t']$ is the static type of t', this is ensured by the first case. In the other case, one sees that $(B)^C t'$ is indeed a normalized term of static type $C \sqsubseteq B$. As for the choice of representatives, if $t' \sim_H t''$, then they have the same static type, so the same case applies for both. In the first case, $[t'] = [t'']$ trivially holds. In the second case C is well-defined due to Lemma 2, and $(B)^C t' \sim_H (B)^C t''$, again because $\sim_H$ is a congruence.

For $[C]$, and $[D]$, note that $\mathcal{I}((A))$ will deliver a value with dynamic type $\sqsubseteq A$ in both cases, as we required for superscript casts. These definitions do not require any choice of representatives.

Finally, for $[E]$, n-fold application of the saturation w.r.t. apply-$\doteq$ guarantees that if $t_i \sim_H t'_i$ for $i = 1, \ldots, n$, then $p(t_1, \ldots, t_n) \in H$ iff $p(t'_1, \ldots, t'_n) \in H$. Therefore, the definition of $\mathcal{I}(p)$ is independent of the choice of the t_i.

□

A central property of our interpretation is that normalized ground terms get evaluated to their equivalence classes:

Lemma 3. *Let H be a saturated branch and let $\mathcal{S} = (\mathcal{D}, \mathcal{I})$ be the structure previously defined. For any normalized ground term $t \in \mathrm{Norm}_H$, $\mathrm{val}_{\mathcal{S}}(t) = [t]$.*

Proof. One shows this by induction on the term structure of t. Let $t = s^A$ be a normalized ground term, i.e. all its subterms are normalized, the top level operator carries a superscript cast to $A = \kappa_H(s)$, and the top level operator is not a redundant cast. By the induction hypothesis, $\mathrm{val}_{\mathcal{S}}(t_i) = [t_i]$ for all subterms

t_i of t. If the top level operator is a function term, $t = f^A(t_1, \ldots, t_n)$, $\mathrm{val}_{\mathcal{S}}(t) = \mathcal{I}((A))(\mathcal{I}(f)([t_1], \ldots, [t_n])) = \mathcal{I}((A))([f^A(t_1, \ldots, t_n)]) = [f^A(t_1, \ldots, t_n)] = [t]$. If $t = (B)^A t'$ is a cast, then $\mathrm{val}_{\mathcal{S}}(t) = \mathcal{I}((A))(\mathcal{I}((B))([t']))$. Since the cast is not redundant, this is equal to $\mathcal{I}((A))([(B)^A t']) = [(B)^A t'] = [t]$. □

Model Lemma. We now need to show that the constructed model satisfies all formulae in a saturated tableau branch. As usual, this is done by induction on some notion of formula complexity. However, for different kinds of formulae, the required notion will be a different one. We therefore split the whole proof into a series of lemmas, each of which requires a different proof idea.

First, we will show that all normalized literals are satisfied in $\mathcal{S} = (\mathcal{D}, \mathcal{I})$.

Lemma 4. *Let H be a saturated open branch and let $\mathcal{S} = (\mathcal{D}, \mathcal{I})$ be the structure defined previously. $\mathcal{S} \models \phi$ for any literal $\phi \in H$ that is normalized w.r.t. H.*

Proof. Lemma 3 tells us that all normalized subterms t_i of ϕ get evaluated to $[t_i]$. We will start with positive literals, i.e. non-negated atoms. For $\phi = p(t_1, \ldots, t_n)$, the result follows directly from our semantics and point $[E]$ in the definition of the interpretation $\mathcal{I}$. For a type predicate $f(t_1, \ldots, t_n) \sqsubseteq A$,

$$\mathrm{val}_{\mathcal{S}}(\beta, f(t_1, \ldots, t_n)) = [f^B(t_1, \ldots, t_n)]$$

for $B = \kappa_H(f(t_1, \ldots, t_n)) \sqsubseteq A$. Hence, the dynamic type of the value will be $\sqsubseteq A$, and therefore $\mathcal{S} \models f(t_1, \ldots, t_n) \sqsubseteq A$. Similarly, for $(B)t_1 \sqsubseteq A$,

$$\mathrm{val}_{\mathcal{S}}(\beta, (B)t_1) = \begin{cases} [t_1] & \text{if } \sigma(t_1) \sqsubseteq B, \\ [(B)^C t_1] & \text{otherwise,} \end{cases}$$

where $C = \kappa_H((B)t_1) \sqsubseteq A$. The first case is excluded, because $(B)t_1 \sqsubseteq A$ is normalized. In the second case, the dynamic type is clearly $C \sqsubseteq A$, so we are done.

For an equality atom $t_1 \doteq t_2 \in H$ with t_1 and t_2 of equal static types, $\mathrm{val}_{\mathcal{S}}(\beta, t_1) = \mathrm{val}_{\mathcal{S}}(\beta, t_2)$, since $t_1 \sim_H t_2$, so $\mathcal{S} \models t_1 \doteq t_2$. If t_1 and t_2 are of unequal static types A and B, respectively, since t_1 and t_2 are normalized, $t_1 = s_1^A$ and $t_2 = s_2^B$ for some terms s_1 and s_2. Also due to normalization, there must be literals $s_1 \sqsubseteq A \in H$ and $s_2 \sqsubseteq B \in H$. Due to saturation w.r.t. the type-$\doteq$ rule, we must also have literals $s_1 \sqsubseteq B \in H$ and $s_2 \sqsubseteq A \in H$. Now since H is type-saturated, H also contains $s_1 \sqsubseteq A \sqcap B$ and $s_2 \sqsubseteq A \sqcap B$ which, given that $A \neq B$, means that s_1^A and s_2^B are in fact *not* normalized which contradicts our assumptions.[1]

If ϕ is a negative literal, we again have to look at the different possible forms. If $\phi = \neg p(t_1, \ldots, t_n)$, assume that $\mathcal{S} \not\models \phi$. Then $\mathcal{S} \models p(t_1, \ldots, t_n)$, which implies that also $p(t_1, \ldots, t_n) \in H$, in which case H would be closed, which is a contradiction. For a negated type predicate $\phi = \neg t \sqsubseteq A$, t will be evaluated to an

[1] One sees here why the apply-$\doteq'$ rule that inserts a cast to make static types fit is not necessary, just convenient. It is not even necessary (but probably *very* convenient) to allow application of equations where the static type gets more special. Application for identical types is enough if we have the type-$\doteq$ and cast normalization rules.

object of dynamic type $C := \kappa_H(t)$. Assume $\mathcal{S} \not\models \phi$. Then $C \sqsubseteq A$. There must be a literal $t \sqsubseteq C \in H$, and therefore H is closed by the close-$\sqsubseteq$ rule. Finally, for a negated equation $\phi = \neg t_1 \doteq t_2$, assume $\mathcal{S} \not\models \phi$. Then $\mathcal{S} \models t_1 \doteq t_2$, and since t_1 and t_2 are normalized, $t_1 \sim_H t_2$. Then, either $t_1 = t_2$, implying that H is closed using close-$\doteq$, or there is an equation $t_1 \doteq t_2 \in H$. Saturation w.r.t. apply-$\doteq$ tells us that $\neg t_2 \doteq t_2$ must then also be in H, so H would again be closed, contradicting our assumptions. □

Based on the previous lemma, we can show that almost all atoms are satisfied, with the exception of one special case which requires separate treatment.

Lemma 5. *Let H be a saturated open branch and $\mathcal{S}$ the same structure as before. $\mathcal{S} \models \phi$ for any literal $\phi \in H$, except if ϕ is a type predicate literal $\phi = (\neg)t^B \sqsubseteq A$ with a superscript cast on the top-level operator.*

Proof. We show this by induction on ϕ with an order $\succ$ defined as follows: $\phi \succ \psi$ iff $sct(\phi) \gg sct(\psi)$, where $sct(\phi)$ is the multiset of the superscript cast types of terms occurring in ϕ, except at the top-level of type predicate atoms, terms without superscript cast being mapped to a pseudo-type ∞, which is larger than all other types. $\gg$ is the multiset ordering induced by the supertype ordering $\sqsupset$, with the mentioned extension for ∞.

So let us assume as induction hypothesis that $\mathcal{S} \models \psi$ for all $\psi \in H$ with $\phi \succ \psi$, except for the case that ψ is a type-predicate literal with a superscript cast on the top-level.

The case that ϕ is normalized is covered in Lemma 4. Otherwise, ϕ contains

(a) a redundant cast $(B)u$, or
(b) a subterm u that is a function or cast application without superscript cast, and that is not at the top-level of a type predicate atom, or
(c) a subterm u that is a function or cast application u^B with a superscript cast to B where a more specific type $C \sqsubseteq B$ is known, i.e. $u \sqsubseteq C \in H$.

In case (a), saturation w.r.t. cast-del guarantees the existence of a formula $\phi' \in H$ that results from deleting the superfluous cast (B) from u. Since ϕ' has one operator less than ϕ, $sct(\phi')$ results from $sct(\phi)$ by deleting one element.[2] Therefore $\phi \succ \phi'$, and the induction hypothesis tells us that $\mathcal{S} \models \phi'$. We also know that $\mathrm{val}_{\mathcal{S}}((B)u) = \mathrm{val}_{\mathcal{S}}(u)$. Let ψ be such that $\phi = \psi[z/(B)u]$ and $\phi' = \psi[z/u]$. The substitution lemma gives us $\mathcal{S} \models \phi$ iff $\mathcal{S} \models \psi[z/(B)u]$ iff $\mathcal{S}, \{z \leftarrow \mathrm{val}_{\mathcal{S}}((B)u)\} \models \psi$ iff $\mathcal{S}, \{z \leftarrow \mathrm{val}_{\mathcal{S}}(u)\} \models \psi$ iff $\mathcal{S} \models \psi[z/u]$ iff $\mathcal{S} \models \phi'$.

In case (b), due to the type completeness of H, there is some literal $u \sqsubseteq C \in H$. This literal must be smaller than ϕ: if ϕ is a type predicate literal, u must be below the top level, so $sct(u \sqsubseteq C)$ is only a subset of that for ϕ, otherwise, $sct(u \sqsubseteq C)$ lacks at least the top level superscript cast of u. Thus, we can apply the induction hypothesis, to get $\mathcal{S} \models u \sqsubseteq C$, and therefore $\mathrm{val}_{\mathcal{S}}((C)u) = \mathrm{val}_{\mathcal{S}}(u)$. Saturation w.r.t. cast-add tells us that there is also $\phi' \in H$ in where ϕ' results from replacing u by u^C in t. Since the term u without superscript cast is considered to be

[2] In the case of deleting a redundant cast $(B)u \sqsubseteq A$ at the top level of a type predicate atom, the removed element is actually the superscript cast of u and not that of $(B)u$.

larger than u^C by our ordering, $\phi \succ \phi'$ and the ind. hyp. gives us $\mathcal{S} \models \phi'$. Like for (a), the result follows by an application of the substitution lemma.

Finally, in case (c), it is saturation w.r.t. cast-strengthen which tells us that if $\phi \in H$, then also $\phi' \in H$, where ϕ is the result of replacing u^B by u^C in ϕ. Now, $u \sqsubseteq C \in H$, and this literal is smaller than ϕ under $\succ$, as in case (b). Hence, $\mathcal{S}$ satisfies $u \sqsubseteq C$, and since $B \sqsubseteq C$, also $u \sqsubseteq B$. Thus, $\mathrm{val}_{\mathcal{S}}((B)u) = \mathrm{val}_{\mathcal{S}}(u) = \mathrm{val}_{\mathcal{S}}((C)u)$. Like for (b), we can apply the ind. hyp. to get $\mathcal{S} \models \phi'$, and the substitution lemma gives us $\mathcal{S} \models \phi$. □

This is easily extended to also allow type predicate literals *with* top-level superscript casts:

Lemma 6. *Let H be a saturated open branch and $\mathcal{S}$ the same structure as before. $\mathcal{S} \models \phi$ for any literal $\phi \in H$.*

Proof. Most literals ϕ are covered by Lemma 5. For the special case of type predicate literals $(\neg)t^A \sqsubseteq B$ for terms with superscript casts, Lemma 1 guarantees that $t \sqsubseteq A$ will also be present, so $\kappa_H(t) \sqsubseteq A$, and therefore $\mathrm{val}_{\mathcal{S}}(t^A) = \mathrm{val}_{\mathcal{S}}(t)$. Due to saturation w.r.t. cast-type, we must have $(\neg)t \sqsubseteq B \in H$, and by Lemma 5, $\mathcal{S} \models (\neg)t \sqsubseteq B$. Since t and t^A are evaluated the same we also get, using the substitution lemma, $\mathcal{S} \models (\neg)t^A \sqsubseteq B$. □

Now, we can finally show that $\mathcal{S}$ is actually a model for all of H.

Lemma 7 (Model Lemma). *Let H be a saturated open branch and let $\mathcal{S} = (\mathcal{D}, \mathcal{I})$ be the structure defined previously. $\mathcal{S}$ is a model for H, i.e. $\mathcal{S} \models \phi$ for all $\phi \in H$.*

Proof. We prove this as usual by an induction on the number of $\wedge, \vee, \forall, \exists$ occurring in ϕ.

Let $\phi \in H$, and assume that $\mathcal{S} \models \psi$ for all ψ smaller than ϕ.

If ϕ is a literal, the result is provided by Lemma 6.

The cases for conjunction, disjunction, and existential quantifiers are completely standard. For universal quantifiers, we convince ourselves that instantiation with normalized ground terms is enough:

Let $\phi = \forall x.\phi_1$ with $x : A$. Then $\phi_1[x/t] \in H$ for all normalized ground terms $t \in T_B \cap \mathrm{Norm}_H$ of static type $B \sqsubseteq A$. For all such t, we may apply the ind. hyp. , since ϕ_1 lacks the occurrence of $\forall$, so $\mathcal{S} \models \phi[x/t]$. Due to the substitution lemma, $\mathcal{S}, \{x \leftarrow \mathrm{val}_{\mathcal{S}}(t)\} \models \phi$, and because of Lemma 3, $\mathcal{S}, \{x \leftarrow [t]\} \models \phi$. Now $\{[t] \mid t \in \mathrm{Norm}_H, \sigma(t) \sqsubseteq A\} = \mathcal{D}_A$ in our structure, so $\mathcal{S} \models \phi$. □

Proof Transformation to Original Calculus. Now that we have shown completeness of the calculus with superscript casts, it only remains to show how to emulate a proof in that calculus using the original rule set of Fig. 1.

The idea is quite simple: every occurrence of $f^A(\ldots)$ can be replaced by $(A)f(\ldots)$, and every $(B)^A t$ by $(A)(B)t$. The modified cast-add, cast-strengthen, and cast-type rules can then be emulated by the original ones as follows:

$$\frac{\phi[z/t], t \sqsubseteq A}{\phi[z/t^A]}\ \text{cast-add} \quad \rightsquigarrow \quad \frac{\phi[z/t], t \sqsubseteq A}{\phi[z/(A)t]}\ \text{cast-add} \quad ,$$

$$\frac{(\neg)t^A \sqsubseteq B, t \sqsubseteq A}{(\neg)t \sqsubseteq B}\ \text{cast-type} \quad \rightsquigarrow \quad \frac{(\neg)(A)t \sqsubseteq B, t \sqsubseteq A}{(\neg)t \sqsubseteq B}\ \text{cast-type} \quad ,$$

and for $C \sqsubseteq A$:

$$\frac{\phi[z/t^A], t \sqsubseteq C}{\phi[z/t^C]}\ \text{cast-strengthen} \quad \rightsquigarrow \quad \frac{\dfrac{\phi[z/(A)t], t \sqsubseteq C}{\phi[z/(A)(C)t]}\ \text{cast-add}}{\phi[z/(C)t]}\ \text{cast-del} \quad .$$

5 Conclusion and Future Work

We have defined a logic with static subtyping, type coercions and type predicates, given a tableau calculus for this logic, and shown its completeness.

Future work includes the accommodation of free variables. This should be possible without difficulties along the lines of [3,2], using order-sorted unification for branch closure. More interestingly, we believe that our cast normalization rules cast-add, cast-del, etc. can be formulated as *destructive* simplification rules. In that case the completeness proof would probably have to be based on saturation modulo redundancy in the style of [1].

The presented calculus is not very efficient as an automated theorem proving calculus, even with destructive cast normalization rules: One should at least get rid of the type-static rule. One can also think of a criterion that limits the insertion of casts to those cases where they can actually be of help in closing some branch. Maybe cast normalization can be restricted to cases where two literals are already unifiable except for some casts.

Acknowledgments

The author would like to thank Bernhard Beckert and Richard Bubel for the interesting discussions that led to these results, and the anonymous referees for their useful comments.

References

1. Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.
2. Christoph Weidenbach. First-order tableaux with sorts. *Journal of the Interest Group in Pure and Applied Logics, IGPL*, 3(6):887–906, 1995.
3. Wolfgang Wernecke and Peter H. Schmitt. Tableau calculus for order-sorted logic. In U. Hedstück, C.-R. Rollinger, and K. H. Bläsius, editors, *Sorts and Types in Artificial Intelligence, Proc. of the Workshop, Ehringerfeld*, volume 418 of *Lecture Notes in AI*, pages 49–60. Springer Verlag, 1990.