

# Conditional Transfer Rule: Reference Manual

Mihails Milehins

February 6, 2026

## Abstract

The document presents a reference manual for the framework *Conditional Transfer Rule*: a collection of experimental utilities for *unoverloading* [10] of definitions and synthesis of *conditional transfer rules* [6] for the object logic *Isabelle/HOL* (e.g., see [13]) of the formal proof assistant *Isabelle* [19] written in *Isabelle/ML* [17, 24].

## Acknowledgements

The author would like to acknowledge the assistance that he received from the users of the mailing list of Isabelle [1] in the form of answers given to his general queries. Special thanks go to Fabian Immler for the development and implementation of the original algorithm for unoverloading of definitions [7], for suggesting the original idea for the implementation of a framework for the relativization of definitions (the idea evolved from [8]) and for providing an outline of the first feasible algorithm for this task (implemented as *CTR II*), to Andrei Popescu for trying the software and providing feedback, to Kevin Kappelmann for providing an explanation of certain aspects of [9], to Alexander Krauss for providing an explanation of certain aspects of [10], to Andreas Lochbihler for providing an outline of an improved algorithm for the relativization of definitions (not currently implemented), to Andreas Lochbihler and Dmitriy Traytel for providing an explanation of the existing functionality of the framework Conditional Parametricity [4]. Furthermore, the author would like to acknowledge the positive impact of [21] and [24] on his ability to code in Isabelle/ML. Moreover, the author would like to acknowledge the positive role that numerous Q&A posted on the Stack Exchange network [2] played in the development of this work. The author would also like to express gratitude to all members of his family and friends for their continuous support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Background . . . . .	5
1.2	Structure and organization . . . . .	5
<b>2</b>	<b>UD</b>	<b>6</b>
2.1	Introduction . . . . .	6
2.1.1	Background . . . . .	6
2.1.2	Purpose and scope . . . . .	6
2.1.3	Related and previous work . . . . .	6
2.2	Theory . . . . .	6
2.3	Syntax . . . . .	7
2.4	Examples . . . . .	8
2.4.1	Type classes . . . . .	8
2.4.2	Low-level overloading . . . . .	8
<b>3</b>	<b>CTR</b>	<b>10</b>
3.1	Introduction . . . . .	10
3.1.1	Background . . . . .	10
3.1.2	Purpose and scope . . . . .	10
3.1.3	Related and previous work . . . . .	10
3.2	Theory . . . . .	11
3.3	Syntax . . . . .	11
3.3.1	Background . . . . .	11
3.3.2	<i>ctr-relator</i> . . . . .	11
3.3.3	<i>ctr</i> . . . . .	12
3.4	Examples . . . . .	13
3.4.1	CTR I . . . . .	13
3.4.2	CTR II . . . . .	14
	<b>References</b>	<b>16</b>

# 1 Introduction

## 1.1 Background

The framework Conditional Transfer Rule (CTR) provides several experimental *Isabelle/Isar* [26, 25, 23] commands that are aimed at the automation of unoverloading of definitions and synthesis of conditional transfer rules in the object logic Isabelle/HOL of the formal proof assistant Isabelle.

## 1.2 Structure and organization

The remainder of the reference manual is organized into two explicit sections, one for each sub-framework of the CTR:

- *Unoverload Definition* (UD): automated elimination of sort constraints and unoverloading of definitions
- *Conditional Transfer Rule* (CTR): automated synthesis of conditional transfer rules from definitions

It should be noted that the abbreviation CTR will be used to refer both to the general framework and the sub-framework.

## 2 UD

### 2.1 Introduction

#### 2.1.1 Background

This section presents a reference manual for the sub-framework UD. The UD can be used for the elimination of *sort constraints* (e.g., see [5]) and unoverloading of definitions in the object logic Isabelle/HOL of the formal proof assistant Isabelle. The UD evolved from the author's work on an extension of the framework *Types-To-Sets* (see [12, 14, 8, 7], for a description of the framework *Types-To-Sets* and [16] for a description of the author's extension) and builds upon certain ideas expressed in [10].

#### 2.1.2 Purpose and scope

The primary functionality of the framework is available via the Isabelle/Isar command **ud**. This command automates the processes of the elimination of sort constraints and unoverloading of definitions. Thus, the command **ud** allows for the synthesis of the convenience constants and theorems that are usually needed for the application of the derivation step 2 of the original relativization algorithm of *Types-To-Sets* (see subsection 5.4 in [12]). However, it is expected that the command can be useful for other purposes.

#### 2.1.3 Related and previous work

The functionality provided by the command **ud** shares similarities with the functionality provided by the algorithms for the elimination of sort constraints and elimination of overloading that were presented in [10] and with the algorithm associated with the command **unoverload\_definition** that was proposed in [7]. Nonetheless, technically, unlike **unoverload\_definition**, the command **ud** does not require the additional axiom UO associated with *Types-To-Sets* for its operation (see [12], [7]), it uses the *definitional axioms* (e.g., see [10]) instead of arbitrary theorems supplied by the user and it is independent of the infrastructure associated with the *axiomatic type classes* [18, 22, 5].

It should also be mentioned that the Isabelle/ML code from the main distribution of Isabelle was frequently reused during the development of the UD. Lastly, it should be mentioned that the framework SpecCheck [9] was used for unit testing the framework UD.

### 2.2 Theory

The general references for this subsection are [10] and [13]. The command **ud** relies on a restricted (non-recursive) variant of the *classical overloading elimination algorithm* that was originally proposed in [10]. It is assumed that there exists a variable  $ud_{\text{with}}$  that stores theorems of the form  $c_\tau = c_{\text{with}} \bar{x}$ , where  $c_\tau$  and  $c_{\text{with}}$  are distinct *constant-instances* and  $\bar{x}$  is a finite sequence of *uninterpreted constant-instances*, such that, if  $c_\tau$  depends on a type variable  $\alpha_\Upsilon$ , with  $\Upsilon$  being a *type class* [18, 22, 5] that depends on the overloaded constants  $\bar{x}'$ , then  $\bar{x}$  contains  $\bar{x}'$  as a subsequence. Lastly, the binary operation  $\cup$  is defined in a manner such that for any sequences  $\bar{x}$  and  $\bar{x}'$ ,  $\bar{x} \cup \bar{x}'$  is a sequence that consists of all elements of the union of the elements of  $\bar{x}$  and  $\bar{x}'$  without duplication. Assuming an underlying *well-formed definitional theory*  $D$ , the input to the algorithm is a constant-instance  $c_\sigma$ . Given the constant-instance  $c_\sigma$ , there exists at most one definitional axiom  $c_\tau = \phi_\tau[\bar{x}]$  in  $D$  such that  $c_\sigma \leq c_\tau$ : otherwise the *orthogonality* of  $D$  and, therefore, the *well-formedness* of  $D$  are violated ( $\phi$  is assumed to be parameterized by the types that it can have with respect to the type substitution operation, and  $\bar{x}$  in  $c_\tau = \phi_\tau[\bar{x}]$  is a list of all uninterpreted constant-instances that occur in  $\phi_\tau[\bar{x}]$ ).

If a definitional axiom  $c_\tau = \phi_\tau[\bar{x}]$  such that  $c_\sigma \leq c_\tau$  exists for the constant-instance  $c_\sigma$ , then the following derivation is applied to it by the algorithm

$$\frac{}{\vdash c_\tau = \phi_\tau[\bar{x}]} \quad (1)$$

$$\frac{}{\vdash c_\sigma = \phi_\sigma[\bar{x}]} \quad (2)$$

$$\frac{}{\vdash c_\sigma = \phi_{\text{with}}[\bar{x} \cup \bar{x}']} \quad (3)$$

$$\frac{}{\vdash c_{\text{with}} = (\lambda \bar{f}. \phi_{\text{with}}[\bar{f}])} \quad (4)$$

$$\frac{}{\vdash c_{\text{with}} \text{ ?}\bar{f} = \phi_{\text{with}}[\text{?}\bar{f}]} \quad (5)$$

$$\frac{}{\vdash c_{\text{with}}(\bar{x} \cup \bar{x}') = \phi_{\text{with}}[\bar{x} \cup \bar{x}']} \quad (6)$$

$$\vdash c_\sigma = c_{\text{with}}(\bar{x} \cup \bar{x}')$$

In step 1, the previously established property  $c_\sigma \leq c_\tau$  is used to create the (extended variant of the) type substitution map  $\rho$  such that  $\sigma = \rho(\tau)$  (see [11]) and perform the type substitution in  $c_\tau = \phi_\tau[\bar{x}]$  to obtain  $c_\sigma = \phi_\sigma[\bar{x}]$ ; in step 2, the collection of theorems  $ud_{\text{with}}$  is unfolded, using it as a term rewriting system, possibly introducing further uninterpreted constants  $\bar{x}'$ ; in step 3, the term on the right-hand side of the theorem is processed by removing the sort constraints from all type variables that occur in it, replacing every uninterpreted constant-instance (this excludes all built-in constants of Isabelle/HOL) that occurs in it by a fresh term variable, and applying the abstraction until the resulting term is closed: this term forms the right-hand side of a new definitional axiom of a fresh constant  $c_{\text{with}}$  (if the conditions associated with the definitional principles of Isabelle/HOL [13] are satisfied); step 4 is justified by the beta-contraction; step 5 is a substitution of the uninterpreted constants  $\bar{x} \cup \bar{x}'$ ; step 6 follows trivially from the results of the application of steps 2 and 5.

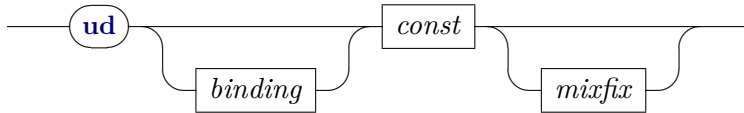
The implementation of the command **ud** closely follows the steps of the algorithm outlined above. Thus, at the end of the successful execution, the command declares the constant  $c_{\text{with}}$  and stores the constant-instance definition that is obtained at the end of step 3 of the algorithm UD; furthermore, the command adds the theorem that is obtained after the execution of step 6 of the algorithm to  $ud_{\text{with}}$ .

Unlike the classical overloading elimination algorithm, the algorithm employed in the implementation of the command **ud** is not recursive. Thus, the users are responsible for maintaining an adequate collection of theorems  $ud_{\text{with}}$ . Nonetheless, in this case, the users can provide their own unoverloaded constants  $c_{\text{with}}$  and the associated theorems  $c_\sigma = c_{\text{with}} \bar{x}$  for any constant-instance  $c_\sigma$ . From the perspective of the relativization algorithm associated with Types-To-Sets this can be useful because there is no guarantee that the automatically synthesized constants  $c_{\text{with}}$  will possess desirable parametricity characteristics (e.g., see [11] and [8]). Unfortunately, the implemented algorithm still suffers from the fundamental limitation that was already outlined in [10], [12] and [14]: it does not offer a solution for handling the constants whose types contain occurrences of the type constructors whose type definitions contain occurrences of unresolvable overloading.

### 2.3 Syntax

This subsection presents the syntactic categories that are associated with the command **ud**. It is important to note that the presentation is only approximate.

$$\mathbf{ud} : \text{theory} \rightarrow \text{theory}$$



`ud` (*b*) `const` (*mixfix*) provides access to the algorithm for the elimination of sort constraints and unoverloading of definitions that was described in subsection 2.2. The optional binding *b* is used for the specification of the names of the entities added by the command to the theory and the optional argument *mixfix* is used for the specification of the concrete inner syntax for the constant in the usual manner (e.g., see [25]). If either *b* or *mixfix* are not specified by the user, then the command introduces sensible defaults. Following the specification of the definition of the constant, an additional theorem that establishes the relationship between the newly introduced constant and the constant provided by the user as an input is established and added to the dynamic fact *ud-with*.

## 2.4 Examples

In this subsection, some of the capabilities of the UD are demonstrated by example. The examples that are presented in this subsection are expected to be sufficient for beginning an independent exploration of the framework, but do not cover the entire spectrum of the functionality and the problems that one may encounter while using it.

### 2.4.1 Type classes

**definition** *mono where*

$$\text{mono } f \leftrightarrow (\forall x y. x \leq y \longrightarrow f x \leq f y)$$

We begin the exploration of the capabilities of the framework by considering the constant *UD-Reference.mono*. It is defined as

$$\text{mono } f = (\forall x y. x \leq y \longrightarrow f x \leq f y)$$

for any  $f :: 'a :: \text{order} \Rightarrow 'b :: \text{order}$ . The constant is unoverloaded using the command `ud`:

`ud <mono>`

The invocation of the command above declares the constant *mono.with* that is defined as

$$\text{mono.with } \text{less-eq } \text{less-eqa} \equiv \lambda f. \forall x y. \text{less-eqa } x y \longrightarrow \text{less-eq } (f x) (f y)$$

and provides the theorem *mono.with* given by

$$\text{UD-Reference.mono} \equiv \text{mono.with } (\leq) (\leq).$$

The theorems establish the relationship between the unoverloaded constant *mono.with* and the overloaded constant *UD-Reference.mono*: both theorems are automatically added to the dynamic fact *ud-with*.

### 2.4.2 Low-level overloading

The following example closely follows Example 5 in section 5.2. in [10].

`consts pls :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a`

**overloading**

`pls-nat  $\equiv$  pls::nat  $\Rightarrow$  nat  $\Rightarrow$  nat`

```

pls-times ≡ pls::'a × 'b ⇒ 'a × 'b ⇒ 'a × 'b
begin
definition pls-nat :: nat ⇒ nat ⇒ nat where pls-nat a b = a + b
definition pls-times :: 'a × 'b ⇒ 'a × 'b ⇒ 'a × 'b
  where pls-times ≡ λx y. (pls (fst x) (fst y), pls (snd x) (snd y))
end

ud pls-nat ⟨pls::nat ⇒ nat ⇒ nat⟩
ud pls-times ⟨pls::'a × 'b ⇒ 'a × 'b ⇒ 'a × 'b⟩

```

As expected, two new unoverloaded constants are produced via the invocations of the command **ud** above. The first constant, *pls-nat.with*, corresponds to *pls*::*nat* ⇒ *nat* ⇒ *nat* and is given by

$$pls-nat.with \equiv (+),$$

the second constant, *pls-times.with*, corresponds to

$$pls::'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$$

and is given by

$$pls-times.with \text{ pls } pls \equiv \lambda x y. (pls \text{ (fst } x) \text{ (fst } y), pls \text{ (snd } x) \text{ (snd } y)).$$

The theorems that establish the relationship between the overloaded and the unoverloaded constants are given by

$$pls \equiv pls-nat.with$$

and

$$pls \equiv pls-times.with \text{ pls } pls.$$

The definitions of the constants *pls-nat.with* and *pls-times.with* are consistent with the ones suggested in [10]. Nonetheless, of course, it is important to keep in mind that the command **ud** has a more restricted scope of applicability than the algorithm suggested in [10].

## 3 CTR

### 3.1 Introduction

#### 3.1.1 Background

This section presents a reference manual for the sub-framework CTR that can be used for the automated synthesis of conditional transfer rules. The CTR evolved from the frameworks *Conditional Parametricity* (CP) [4] and Types-To-Sets that are available as part of the main distribution of Isabelle. However, it does not require either the axiom LT or the axiom UO associated with the Types-To-Sets for its operation. The CTR introduces the following Isabelle/Isar commands:

$$\begin{aligned} \mathbf{ctr} & : \text{local-theory} \rightarrow \text{local-theory} \\ \mathbf{ctr-relator} & : \text{local-theory} \rightarrow \text{local-theory} \end{aligned}$$

#### 3.1.2 Purpose and scope

The primary functionality of the CTR is available via the command **ctr**. The command **ctr**, given a definition of a constant, attempts to provide a conditional transfer rule for this constant, possibly under arbitrary user-defined side conditions. The process of the synthesis of a transfer rule for a constant may or may not involve the declaration and synthesis of a definition of a new (*relativized*) constant. The command provides an interface for the application of two plausible algorithms for the synthesis of the transfer rules that have a restricted and overlapping scope of applicability. The first algorithm (*CTR I*) was developed and implemented in [4]. An outline of the second algorithm (*CTR II*) was proposed in [15] and [8]: CTR II relies on the functionality of the *transfer-prover* (see subsection 4.6 in [11]). More details about CTR II are given in the next subsection.

The command **ctr-relator** can be used for the registration of the, so-called, *ctr-relators* that need to be provided for every non-nullary type constructor that occurs in the type of the constant-instance whose definition is passed as an argument to CTR II. However, as a fallback solution, the command **ctr** may use the *relators* that are associated with the standard *BNF* infrastructure of Isabelle/HOL (e.g., see [20]). The only necessary condition for the registration of the *ctr-relators* is that they must satisfy the type-constraint associated with the action of a BNF on relations (e.g., see [20] or [3]).

#### 3.1.3 Related and previous work

As already mentioned, the CTR evolved from the framework CP that is available as part of the main distribution of Isabelle. Furthermore, CTR provides an interface to the main functionality associated with the framework CP and builds upon many ideas that could be associated with it. The primary reason for the development of the command **ctr** instead of extending the implementation of the existing command **parametric-constant** provided as part of the CP was the philosophy of non-intervention with the development version of Isabelle that was adopted at the onset of the design of the CTR. Perhaps, in the future, the functionality of the aforementioned commands can be integrated.

It should also be mentioned that the Isabelle/ML code from the main distribution of Isabelle was frequently reused during the development of CTR. Lastly, it should be mentioned that the framework SpecCheck [9] was used for unit testing the framework CTR.

## 3.2 Theory

Assume the existence of an underlying well-formed definitional theory  $D$  and a context  $\Gamma$ ; assume the existence of a map  $\mathbf{ctr}_{\text{Rel}}$  from a finite set of non-nullary type constructors to relators, mapping each non-nullary type constructor in its domain to a valid relator for this type constructor. The inputs to CTR II are a constant-instance definition  $\vdash c_{? \bar{\gamma}} K = \phi[? \bar{\gamma}]$  of the constant-instance  $c_{? \bar{\gamma}} K$  and the map  $\mathbf{trp}$  from the set of all schematic type variables in  $? \bar{\gamma}$  to the set of (fixed) binary relations  $x_{\alpha \rightarrow \beta \rightarrow \mathbb{B}}$  in  $\Gamma$  with non-overlapping type variables ( $? \bar{\gamma}$  corresponds to a sequence of all distinct type variables that occur in the type  $? \bar{\gamma} K$ , while  $K$ , applied using a postfix notation, contains all information about the type constructors of the type  $? \bar{\gamma} K$ , such that the non-nullary type constructors associated with  $K$  form a subset of the domain of  $\mathbf{ctr}_{\text{Rel}}$ ). CTR II consists of three parts: *synthesis of a parametricity relation*, *synthesis of a transfer rule* and *post-processing*.

**Synthesis of a parametricity relation.** An outline of an algorithm for the conversion of a type to a *parametricity relation* is given in subsection 4.1.1 in [11]. Thus, every nullary type constructor in  $? \bar{\gamma} K$  is replaced by the identity relation  $=$ , every non-nullary type constructor  $\kappa$  in  $? \bar{\gamma} K$  is replaced by its corresponding relator  $\mathbf{ctr}_{\text{Rel}}(\kappa)$  and every type variable  $? \gamma$  in  $? \bar{\gamma} K$  is replaced by  $\mathbf{trp}(? \gamma)$ , obtaining the parametricity relation  $R_{\bar{\alpha} K \rightarrow \bar{\beta} K \rightarrow \mathbb{B}}$ .

**Synthesis of a transfer rule.** First, the goal  $R \phi[\bar{\alpha}] \phi[\bar{\beta}]$  is created in  $\Gamma$  and an attempt to prove it is made using the algorithm associated with the method *transfer-prover*. If the proof is successful, nothing else needs to be done in this part. Otherwise, an attempt to find a solution for  $? a$  in  $R (? a_{\bar{\alpha} K}) \phi[\bar{\beta}]$  is made, once again, using the *transfer-prover*. The result of the successful completion of the synthesis is a transfer rule  $\Gamma \vdash R \psi[\bar{\alpha}, \bar{x}] \phi[\bar{\beta}]$ , where  $\bar{x}$  is used to denote a sequence of typed variables, each of which occurs free in the context  $\Gamma$  (the success of this part is not guaranteed).

**Post-processing.** If  $\psi[\bar{\alpha}, \bar{x}] = \phi[\bar{\alpha}]$  after the successful completion of part 2 of the algorithm, then the definitions of the constant-instances  $c_{\bar{\alpha} K}$  and  $c_{\bar{\beta} K}$  are folded, resulting in the deduction  $\Gamma \vdash R c_{\bar{\alpha} K} c_{\bar{\beta} K}$ . Otherwise, if  $\psi[\bar{\alpha}, \bar{x}] \neq \phi[\bar{\alpha}]$ , then a new constant  $d$  is declared such that  $\vdash d_{\sigma[? \bar{\alpha}]} = (\lambda \bar{x}. \psi[? \bar{\alpha}, \bar{x}])$ , where  $\sigma$  is determined uniquely by  $\bar{x}$  and  $? \bar{\alpha} K$ . In this case,  $\Gamma \vdash R \psi[\bar{\alpha}, \bar{x}] \phi[\bar{\beta}]$  can be restated as  $\Gamma \vdash R (d_{\sigma[? \bar{\alpha}]} \bar{x}) c_{\bar{\beta} K}$ . This result can be exported to the global theory context and forms a conditional transfer rule for  $c_{? \bar{\gamma} K}$ .

CTR II can perform the synthesis of the transfer rules for constants under arbitrary user-defined side conditions automatically. However, the algorithm guarantees neither that it can identify whether a transfer rule exists for a given constant under a given set of side conditions, nor that it will be found if it does exist.

## 3.3 Syntax

### 3.3.1 Background

This subsection presents the syntactic categories that are associated with the command **ctr** and closely related auxiliary commands and attributes. It is important to note that the presentation is only approximate.

### 3.3.2 ctr-relator

**ctr-relator** : *local-theory*  $\rightarrow$  *local-theory*



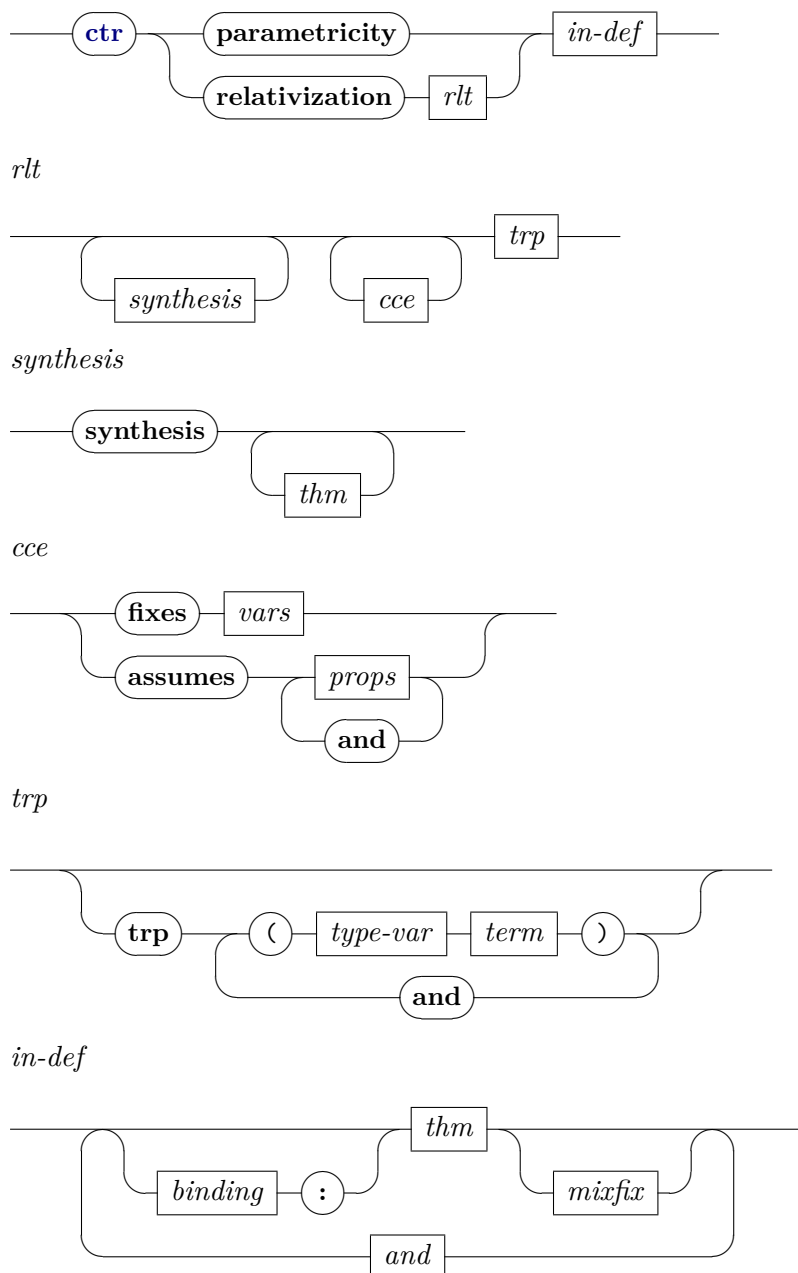
**ctr-relator**  $c$  saves the ctr-relator  $c$  to a database of ctr-relators for future reference. A ctr-relator is defined as any constant that has the type of the form

$$(\alpha_1 \Rightarrow \beta_1 \Rightarrow \mathbb{B}) \Rightarrow \dots \Rightarrow (\alpha_n \Rightarrow \beta_n \Rightarrow \mathbb{B}) \Rightarrow (\alpha_1 \dots \alpha_n) \kappa \Rightarrow (\beta_1 \dots \beta_n) \kappa \Rightarrow \mathbb{B},$$

where  $\alpha_1 \dots \alpha_n$  and  $\beta_1 \dots \beta_n$  are distinct type variables,  $n$  is a positive integer and  $\kappa$  is a type constructor.

### 3.3.3 *ctr*

**ctr** : *local-theory*  $\rightarrow$  *local-theory*



**ctr** provides access to two algorithms for the automated synthesis of the transfer rules and the relativization of constants based on their definitions.

**parametricity** indicates that CTR I needs to be invoked by the command.

**relativization** indicates that CTR II needs to be invoked by the command.

**synthesis** *thm* indicates that the relativization of the inputs needs to be performed and post-processed using the simplifier with the collection of rewrite rules stated as the fact *thm*. If the optional argument *thm* is not provided, then the default *simpset* is used for post-processing. If the keyword **synthesis** is omitted, then no attempt to perform the relativization is made. The keyword **synthesis** is relevant only for CTR II.

**trp** ( $?a_1 A_1$ ) **and** ... **and** ( $?a_n A_n$ ) indicates that the type variable that has the indexname  $?a_i$  ( $1 \leq i \leq n$ ,  $n$  is a non-negative integer) is meant to correspond to the binary relation  $A_i$  in the parametricity relation constructed by the command prior to the statement of the transfer rule (for further information see subsection 4.1 in [11]). This is relevant only for CTR II.

**in** (*b*) *def-thm* (*mixfix*) is used for the specification of the input to the algorithms that are associated with the command **ctr**. *def-thm* is meant to be a fact that consists of exactly one theorem of the form  $A ?a_1 \dots ?a_n \simeq f ?a_1 \dots ?a_n$ , where  $A$  is a constant,  $\simeq$  is either meta- or object-equality,  $n$  is a non-negative integer,  $?a_1 \dots ?a_n$  are schematic variables and  $f$  is a suitable formula in  $n$  arguments (however, there are further implicit restrictions). If a new constant is introduced by the command, then the optional argument *mixfix* is used for the specification of the concrete inner syntax for the constant in the usual manner (e.g. see [25]). The optional binding *b* is used for the specification of the names of the entities that are provided after the successful execution of the command. It is hoped that the algorithm chosen for the specification of the names is sufficiently intuitive and does not require further explanation. If either *b* or *mixfix* are not specified by the user, then the command introduces sensible defaults. Multiple theorems may be provided after the keyword **in**, employing the keyword **and** as a separator. In this case, the parameterized algorithm associated with the command is applied repeatedly to each input theorem in the order of their specification from left to right and the local context is augmented incrementally.

### 3.4 Examples

In this subsection, some of the capabilities of the CTR are demonstrated by example. The examples that are presented in this subsection are expected to be sufficient to begin an independent exploration of the framework, but do not cover the entire spectrum of the functionality and the problems that one may encounter while using it.

The examples that are presented in this subsection continue the example of the application of the command **ud** to the definition of the constant *UD-Reference.mono* that was presented in subsection 2.4.

#### 3.4.1 CTR I

As already explained, the command **ctr** can be used to invoke the algorithm associated with the command **parametric-constant** for the synthesis of a transfer rule for a given constant. For example, the invocation of

```
ctr parametricity
  in mono: mono.with-def
```

generates the transfer rule *mono-transfer*:

*If bi-total A and (A ==> A ==> (=)) (≤) (≤) and (B ==> B ==> (=)) (≤) (≤) then*

$((A \implies B) \implies (=))$  *order-class.mono order-class.mono.*

A detailed explanation of the underlying algorithm can be found in [4].

### 3.4.2 CTR II

The first example in this subsection demonstrates how CTR II can be used to produce a parametricity property identical to the one produced by CTR I above:

#### ctr relativization

```
fixes A1 :: 'a ⇒ 'b ⇒ bool and A2 :: 'c ⇒ 'd ⇒ bool
assumes [transfer-rule]: bi-total A1
trp (?'a A1) and (?'b A2)
in mono': mono.with-def
```

This produces the theorem *mono'.transfer*:

```
If bi-total A1 then
((A2 ⟹ A2 ⟹ (=)) ⟹ ⟹
(A1 ⟹ A1 ⟹ (=)) ⟹ ⟹ (A1 ⟹ A2) ⟹ ⟹ (=))
mono.with mono.with.
```

which is identical to the theorem *mono-transfer* generated by CTR I.

Of course, there is very little value in trying to establish a parametricity property using CTR II due to the availability of CTR I. However, it is often the case that the constant is not parametric under a given set of side conditions. Nonetheless, in this case, it is often possible to define a new *relativized constant* that is related to the original constant under the parametricity relation associated with the original constant. It is expected that the most common application of CTR II will be the synthesis of the relativized constants.

For example, suppose one desires to find a conditional transfer rule for the constant *mono.with* such that (using the conventions from the previous example) the relation *A1* is right total, but not, necessarily, left total. While, under such restriction on the involved relations, the constant *mono.with* is no longer conditionally parametric, its relativization exists and can be found using the transfer prover. To automate the relativization process, the user merely needs to add the keyword **synthesis** immediately after the keyword **relativization**:

#### ctr relativization

```
synthesis ctr-simps
fixes A1 :: 'a ⇒ 'b ⇒ bool and A2 :: 'c ⇒ 'd ⇒ bool
assumes [transfer-domain-rule]: Domainp A1 = (λx. x ∈ U1)
and [transfer-rule]: right-total A1
trp (?'a A1) and (?'b A2)
in mono-ow: mono.with-def
```

This produces the definition *mono-ow-def*:

```
mono-ow U1 less-eq less-eqa f ≡
∀ x∈U1. ∀ y∈U1. less-eqa x y ⟶ less-eq (f x) (f y)
```

and the theorem *mono-ow.transfer*:

```
If Domainp A1 = (λx. x ∈ U1) and right-total A1 then
((A2 ⟹ A2 ⟹ (=)) ⟹ ⟹
(A1 ⟹ A1 ⟹ (=)) ⟹ ⟹ (A1 ⟹ A2) ⟹ ⟹ (=))
(mono-ow U1) mono.with.
```

It should be noted that, given that the keyword **synthesis** was followed by the name of the named collection of theorems *ctr-simps*, this collection was used in post-processing of the result of the relativization. The users can omit simplification entirely by specifying the collection *ctr-blank* instead of *ctr-simps*.

## References

- [1] Isabelle mailing-list, . URL <https://lists.cam.ac.uk/pipermail/cl-isabelle-users/>.
- [2] Stack Exchange, . URL <https://stackexchange.com/>.
- [3] J. C. Blanchette, L. Gheri, A. Popescu, and D. Traytel. Bindings as Bounded Natural Functors. *Proceedings of the ACM on Programming Languages*, 3(POPL):22:1–22:34, 2019.
- [4] J. Gilcher, A. Lochbihler, and D. Traytel. Conditional Parametricity in Isabelle/HOL. In *TABLEAUX/FroCoS/ITP*, Brasília, Brazil, 2017.
- [5] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502, pages 160–174. Springer, Heidelberg, 2007. ISBN 978-3-540-74464-1.
- [6] B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs*, volume 8307, pages 131–146. Springer, Heidelberg, 2013. ISBN 978-3-319-03545-1.
- [7] F. Immler. Automation for unloading definitions, 2019. URL <http://isabelle.in.tum.de/repos/isabelle/rev/ab5a8a2519b0>.
- [8] F. Immler and B. Zhan. Smooth Manifolds and Types to Sets for Linear Algebra in Isabelle/HOL. In A. Mahboubi and M. O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal*, CPP 2019, pages 65–77. ACM, New York, 2019. ISBN 978-1-4503-6222-1.
- [9] K. Kappelmann, L. Bulwahn, and S. Willenbrink. SpecCheck - Specification-Based Testing for Isabelle/ML. *Archive of Formal Proofs*, 2021.
- [10] A. Krauss and A. Schropp. A Mechanized Translation from Higher-Order Logic to Set Theory. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, volume 6172, pages 323–338. Springer, Heidelberg, 2010. ISBN 978-3-642-14051-8.
- [11] O. Kunčar. *Types, Abstraction and Parametric Polymorphism in Higher-Order Logic*. PhD thesis, Technische Universität München, Munich, 2015.
- [12] O. Kunčar and A. Popescu. From Types to Sets by Local Type Definitions in Higher-Order Logic. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, volume 9807, pages 200–218. Springer, Heidelberg, 2016. ISBN 978-3-319-43144-4.
- [13] O. Kunčar and A. Popescu. Comprehending Isabelle/HOL’s Consistency. In H. Yang, editor, *Programming Languages and Systems*, volume 10201, pages 724–749. Springer, Heidelberg, 2017. ISBN 978-3-662-54433-4.
- [14] O. Kunčar and A. Popescu. From Types to Sets by Local Type Definition in Higher-Order Logic. *Journal of Automated Reasoning*, 62(2):237–260, 2019.
- [15] P. Lammich. Automatic Data Refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 84–99. Springer, Heidelberg, 2013. ISBN 978-3-642-39634-2.
- [16] M. Milehins. Extension of Types-To-Sets. *Archive of Formal Proofs*, 2021.
- [17] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, Massachusetts, 1997. ISBN 978-0-262-63181-5.

- [18] T. Nipkow and G. Snelting. Type Classes and Overloading Resolution via Order-Sorted Unification. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 1–14. Springer, Heidelberg, 1991. ISBN 978-3-540-47599-6.
- [19] L. C. Paulson. Natural Deduction as Higher-Order Resolution. *The Journal of Logic Programming*, 3(3):237–258, 1986.
- [20] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *27th Annual IEEE Symposium on Logic in Computer Science*, pages 596–605, Dubrovnik, 2012. IEEE.
- [21] C. Urban. *The Isabelle Cookbook: A Gentle Tutorial for Programming Isabelle/ML*. 2019.
- [22] M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 307–322. Springer, Heidelberg, 1997. ISBN 978-3-540-69526-4.
- [23] M. Wenzel. Isar — A Generic Interpretative Approach to Readable Formal Proof Documents. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin-Mohring, editors, *Theorem Proving in Higher Order Logics*, volume 1690, pages 167–183. Springer, Heidelberg, 1999. ISBN 978-3-540-66463-5.
- [24] M. Wenzel. The Isabelle/Isar Implementation. 2019.
- [25] M. Wenzel. The Isabelle/Isar Reference Manual. 2019.
- [26] M. M. Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Technische Universität München, Munich, 2001.