Conservative Overloading in Higher-Order Logic

Steven Obua (obua@in.tum.de) *
Technische Universität München, D-85748 Garching, Boltzmannstr. 3, Germany

Abstract. Overloading in the context of higher-order logic has been used for some time now. Isabelle is the only proof-assistant that actually implements overloading *within the logic* instead of merely instrumenting the pretty-printing machinery on top of the logic.

So far there existed no satisfying theory that could explain why it is *safe* to add a mechanism of certain kinds of possibly overloaded constant definitions to ordinary higher-order logic. This is not only of theoretical interest but also of practical importance; until now it was easy to introduce inconsistencies in Isabelle by abusing overloaded definitions.

This paper addresses both the theoretical and the practical aspects of adding overloading to higher-order logic. We first define what we mean by *Higher-Order Logic with Conservative Overloading* (HOLCO). HOLCO captures how overloading is actually applied by the users of Isabelle; for example it allows to freely mix type definitions with overloaded constant definitions. We then show the consistency of HOLCO by reducing it to ordinary higher-order logic with only type definitions and no constant definitions.

Having so established our playground we show that this playground is too big for any proof-assistant implementing HOLCO; checking if definitions obey the rules of HOLCO is not even semi-decidable. We prove this by connecting this problem with the problem of deciding the termination of certain kinds of term rewriting systems (TRSs) which we call *overloading* TRSs and showing that *Post's Correspondence Problem for Prefix Morphisms* can be reduced to it.

The undecidability proof reveals strong ties between our problem and the *dependency pair method* by Arts and Giesl for proving termination of TRSs. The *dependency graph* of overloaded TRSs can be computed exactly (for general TRSs it is not computable and must be approximated). We exploit this by providing an algorithm that checks the conservativity of definitions based on the dependency pair method and a restricted form of linear polynomial interpretation; the algorithm also uses the strategy of Hirokawa and Middeldorp of recursively calculating the strongly connected components of the dependency graph. Of course the algorithm cannot successfully check all valid conservative definitions; but it is sufficiently powerful to deal with all overloaded definitions that the author has encountered so far in practice. An implementation of this algorithm is available as part of a package that adds conservative overloading to Isabelle. This package also allows to delegate the conservativity check to external tools like the *Tyrolean Termination Tool* or the *Automated Program Verification Environment*.

* Supported by the Ph.D. program "Logik in der Informatik" of the "Deutsche Forschungsgemeinschaft."

1. Introduction and Motivation

Higher-order logic (HOL) is widely used in the mechanical theorem proving community. There are several implementations of it available, among them HOL 4 [9], HOL-light [10] and Isabelle/HOL [3].

HOL has a theory extension mechanism that allows to define new constants in terms of already known ones. The name of the new, to be defined constant must be different from the names of all already defined constants. In that way it is ensured that you can do in principle without any constant definitions by just unfolding all definitions.

Isabelle/HOL inherits from its meta logic [1] two additional features, *axiomatic type classes* and *overloaded constant definitions*, both of which have been introduced in [2]. Overloading is not very useful in the absence of axiomatic type classes or a similar mechanism (and the other way around) but it is an orthogonal feature that can be studied separately. We therefore will focus on overloading alone.

1.1. Why Overloading?

So what is overloading and why is it useful? We answer these questions by giving a well-known example. Addition and the operation of forming the inverse of a given element with respect to addition are useful for different types of elements like *reals* (`real`) or *integers* (`int`). Both reals and integers are *numerical types*; in Isabelle/HOL these types are organized using axiomatic type classes and overloading [11]. We would like to refer to the addition of reals by the same constant name (`add`) that we use for talking about the addition of integers; similarly the inverse is denoted by `uminus` for both reals and integers. Actually, there may be even more types for which addition and inverse makes sense; in Isabelle we can *declare* polymorphic constants by

consts
`add` :: $\alpha \to \alpha \to \alpha$
`uminus` :: $\alpha \to \alpha$
`zero` :: α .

Addition of reals is then referred to by `add` :: `real` $\to$ `real` $\to$ `real` while addition of integers is denoted by `add` :: `int` $\to$ `int` $\to$ `int`. Often we do not need to write down the full type as we just did; type inference will often deduce the type of the constants automatically. Despite of real and integer addition sharing the same name, the *definition* of addition can of course be very different for reals and integers. The same statements also hold for the operation of forming the inverse of an element. Note that we also added a declaration for 0 (`zero`).

Subtraction is easily defined using addition and negation:

consts
 `minus` :: $\alpha \to \alpha \to \alpha$
defs
 `minus` a $b \equiv$ `add` a (`uminus` b) .

Now imagine we have defined a type of matrices α `matrix` with entries of type α [14, sect. 3 (Finite Matrices)]. Matrices are represented as functions $f ::$ `nat` $\to$ `nat` $\to \alpha$ such that $f\,j\,i \neq$ `zero` for only finitely many pairs (j, i). Such a type definition could look like this:

typedef α `matrix` =
 $\{f ::$ `nat` $\to$ `nat` $\to \alpha \mid$ `finite` $\{(j,i) \mid f\,j\,i \neq$ `zero`$\}\}$.

The functions `rep` and `abs` convert between abstract matrices and their concrete representations as functions. The zero matrix and addition and negation of matrices are defined by

defs
 `zero` :: α `matrix` $\equiv$ `abs` $(\lambda j\,i.$ `zero`$)$
 `uminus` $(a ::$ α `matrix`$)$ $\equiv$ `abs` $(\lambda j\,i.$ `uminus` (`rep` $a\,j\,i$)$)$
 `add` $(a ::$ α `matrix`$)$ $b \equiv$ `abs` $(\lambda j\,i.$ `add` (`rep` $a\,j\,i$) (`rep` $b\,j\,i$)$)$.

Note that addition etc. has been defined for the types `int matrix`, `real matrix`, α `matrix matrix` and so on as well!

Overloading makes inventing new names like `add_real`, `add_int`, `add_real_matrix` and `add_int_matrix` superfluous and helps to keep the number of *declarations* small. The example also illustrates that at the same time the number of *definitions* is reduced. This is not so much a matter of less typing; in tandem with axiomatic type classes for example we could prove that α `matrix` forms a group if α does. Thus overloading is a way to achieve modular and extensible reasoning.

Declarations, constant definitions and type definitions are *theory extensions*. It is instructive to observe the order of theory extensions in our example:

1. Declaration (of `add`, `uminus`, `zero`),
2. Definition (of `add`, `uminus`, `zero` for `real` and `int`),
3. Declaration and Definition (of `minus`),
4. Type definition (of α `matrix`),
5. Definition (of `add`, `uminus`, `zero` for α `matrix`).

Obviously one cannot assume that all definitions for a single overloaded constant are done at once; the definitions in (2.) are separated from the definitions in (5.) by (3.) and (4.), that is another constant definition and a type definition. The type definition again depends on the declaration of `zero` in (1.). The definition in (3.) depends on the definitions in (2.) and also on the definitions in (5.). The theory extensions of [2] cannot handle such a situation; in this paper we present a theory extension mechanism that can.

1.2. Is Overloading Safe?

The discussion so far should have convinced the reader that overloading is useful (see also [15], [11], [14]). But is it also safe? Not necessarily. In order to enable overloading we cannot make the assumption that when defining a new constant the name of the constant has never been used before, especially not on the right hand side of the definition. Proof-assistants like HOL 4 or HOL-light make this assumption. With it, it is easy to see that constant definitions cannot possibly endanger the consistency of a theory; all the constants can be eliminated by just unfolding the definitions of these constants [9]. Without it, we are in trouble as the following example shows:

consts
 $\mathtt{b} :: \mathtt{bool}$
defs
 $\mathtt{b} \equiv \neg\ \mathtt{b}$.

The definition of `b` leads to an inconsistent theory. Therefore we need restrictions of some kind in order to enjoy safe overloading. A first attempt might be to not allow any constant (including its type) on the right hand side of a definition to unify (without renaming) with the constant to be defined. This restriction would render the above definition illegal because $\mathtt{b} :: \mathtt{bool}$ unifies with $\mathtt{b} :: \mathtt{bool}$ while still allowing for example the definition of addition for matrices because $\mathtt{add} :: \alpha\ \mathtt{matrix}$ does not unify with $\mathtt{add} :: \alpha$.

But this restriction is not strong enough; consider

consts
 $\mathtt{A} :: \alpha \to \mathtt{bool}$
defs
 $\mathtt{A}\ (x :: \alpha\,\mathtt{list} \times \alpha) \equiv \mathtt{A}(\mathtt{snd}\,x \,\#\, \mathtt{fst}\,x)$
 $\mathtt{A}\ (x :: \alpha\,\mathtt{list}) \equiv \neg\,\mathtt{A}(\mathtt{tl}\,x, \mathtt{hd}\,x)$.

There are two definitions for `A`. The two definitions do not *overlap*, that is $\mathtt{A} :: \alpha\,\mathtt{list} \times \alpha \to \mathtt{bool}$ and $\mathtt{A} :: \beta\,\mathtt{list} \to \mathtt{bool}$ do not unify. Our restriction applies to neither the first definition nor the second

definition because $\alpha\,\texttt{list} \times \alpha$ and $\alpha\,\texttt{list}$ cannot be unified. Therefore we would call these definitions legal; but they lead to an inconsistent theory via

$$\mathtt{A}\,[x] \;=\; \mathtt{A}\,(x, []) \;=\; \neg\,\mathtt{A}\,[x].$$

Note that the latter example is accepted by Isabelle 2005!

Thus we need a different criterium for legal definitions. The one we will introduce in the next section is composed of two restrictions:

1. Two definitions may not overlap, that is the defined constants (including their respective types) do not unify (after renaming).

2. The process of unfolding definitions always terminates.

Under these two conditions overloading is safe. This will be made precise in the next section under the notion *conservative overloading*.

The rest of the paper is organized as follows: in the next section we define rigorously what we mean by conservative overloading and prove that it is safe. In the third section we discuss if and how conservative overloading can be implemented by a proof-assistant for higher-order logic, both in principle and in practice. We conclude by summarizing the results of this paper.

2. Higher-Order Logic with Conservative Overloading

Sections 2.1–2.3 introduce notation for notions in the domain of higher-order logic that are familiar to readers working in higher-order logic. Therefore the reader should feel free to skim over these sections; later on in the text the reader might want to consult these sections for reference.

From Section 2.4 on it gets interesting. After describing the different ways of extending a higher-order logic theory we explain why and in what sense these extensions can be considered safe.

2.1. Types

Let us first fix four mutually disjoint, infinite sets of names:

- $\mathcal{K}$ from which the assistant draws the names of *type constructors*,
- $\mathcal{C}$ which is used to name *constants*,
- $\mathcal{A}$ which contains the names of *type variables*,
- $\mathcal{X}$ which provides the names of *term variables*.

Furthermore we fix a function ι from $\mathcal{K}$ to $\mathbb{N}$ where $\iota(\mathtt{c})$ denotes the *arity* of the type constructor $\mathtt{c}$. Given $\mathcal{K}' \subseteq \mathcal{K}$ we can then inductively define the set $\mathcal{T}(\mathcal{K}')$ of all types with constructor names in $\mathcal{K}'$ and the type variables $tvars(\tau)$ that appear in a type τ:

$$\frac{\alpha \in \mathcal{A}}{\alpha \in \mathcal{T}(\mathcal{K}')} \qquad \frac{\tau_1, \ldots, \tau_n \in \mathcal{T}(\mathcal{K}') \qquad \mathtt{c} \in \mathcal{K}' \qquad \iota(\mathtt{c}) = n}{\mathtt{c}(\tau_1, \ldots, \tau_n) \in \mathcal{T}(\mathcal{K}')}$$

$$\frac{\alpha \in \mathcal{A}}{tvars(\alpha) = \{\alpha\}} \qquad \frac{\tau = \mathtt{c}(\tau_1, \ldots, \tau_n) \in \mathcal{T}(\mathcal{K}')}{tvars(\tau) = tvars(\tau_1) \cup \ldots \cup tvars(\tau_n)}.$$

A *type substitution* σ is a map from $\mathcal{A}$ to $\mathcal{T}(\mathcal{K})$ such that $\sigma\,\alpha \neq \alpha$ holds only for finitely many $\alpha \in \mathcal{K}$. Applying σ to a type τ means to simultaneously replace any type variable α occurring in τ by $\sigma\,\alpha$. For two types τ and τ' we write $\tau' \leq \tau$ if there is a type substitution σ such that $\tau' = \sigma\,\tau$.

2.2. Signatures and terms

A *signature* S is a triple $(\mathcal{K}_S, \mathcal{C}_S, ctype_S)$ where $\mathcal{K}_S$ and $\mathcal{C}_S$ are finite subsets of $\mathcal{K}$ and $\mathcal{C}$, respectively; the third component $ctype_S$ is a function from $\mathcal{C}_S$ to $\mathcal{T}(\mathcal{K}_S)$. Note that usually the arity ι that we have fixed once and for all is part of the signature; this is only a minor detail and does not matter. Furthermore we demand that $\mathcal{K}_S$ contains the constructors for the type of proposition `prop` and the type of functions $\rightarrow$. The arities of these constructors are $\iota(\mathtt{prop}) = 0$ and $\iota(\rightarrow) = 2$. We also demand $\{\equiv, \mathtt{rep}, \mathtt{abs}\} \subseteq \mathcal{C}_S$ with

- $ctype_S(\equiv) = \alpha \rightarrow (\alpha \rightarrow \mathtt{prop})$,
- and $ctype_S(\mathtt{rep}) = ctype_S(\mathtt{abs}) = \alpha \rightarrow \beta$

for some $\alpha, \beta \in \mathcal{A}$ and $\alpha \neq \beta$. Of course we wrote in the above and will continue to write $a \rightarrow b$ instead of $\rightarrow (a, b)$.

Table I. Functions on terms

	$x :: \tau$	$\mathtt{C} :: \tau$	$f \cdot g$	$\lambda\, x :: \tau.\, b$
$tvars\,(\cdot)$	$tvars\,(\tau)$	$tvars\,(\tau)$	$tvars\,(f) \cup tvars\,(g)$	$tvars\,(\tau) \cup tvars\,(b)$
$vars\,(\cdot)$	$\{x :: \tau\}$	$\emptyset$	$vars\,(f) \cup vars\,(g)$	$vars\,(b) \setminus \{x :: \tau\}$
$consts\,(\cdot)$	$\emptyset$	$\{\mathtt{C} :: \tau\}$	$consts\,(f) \cup consts\,(g)$	$consts\,(b)$
$\sigma\,\cdot$	$x :: \sigma\,\tau$	$\mathtt{C} :: \sigma\,\tau$	$\sigma\, f \cdot \sigma\, g$	$\lambda\, x :: \sigma\,\tau.\, \sigma\, b$

Given a signature S we define the set of its terms $Term\,(S)$ and the type $\Sigma\,(t)$ of a term t by

$$\frac{x \in \mathcal{X} \qquad \tau \in \mathcal{T}\,(\mathcal{K}_S)}{x :: \tau \in Term\,(S) \qquad \Sigma\,(x :: \tau) = \tau}$$

$$\frac{\mathtt{C} \in \mathcal{C}_S \qquad \tau \in \mathcal{T}\,(\mathcal{K}_S) \qquad \tau \leq ctype_S(\mathtt{C})}{\mathtt{C} :: \tau \in Term\,(S) \qquad \Sigma\,(\mathtt{C} :: \tau) = \tau}$$

$$\frac{f, g \in Term\,(S) \qquad \Sigma\,(f) = \Sigma\,(g) \to \tau}{f \cdot g \in Term\,(S) \qquad \Sigma\,(f \cdot g) = \tau}$$

$$\frac{x \in \mathcal{X} \qquad \tau \in \mathcal{T}\,(\mathcal{K}_S) \qquad b \in Term\,(S)}{\lambda\, x :: \tau.\, b \in Term\,(S) \qquad \Sigma\,(\lambda\, x :: \tau.\, b) = \tau \to \Sigma\,(b)}.$$

Again we prefer infix notation and write $u \equiv v$ instead of

$$((\equiv :: \Sigma\,(u) \to (\Sigma\,(u) \to \mathtt{prop})) \cdot u) \cdot v.$$

We call a term t a *proposition* iff $\Sigma\,(t) = \mathtt{prop}$; therefore every equation $u \equiv v$ is a proposition. For the set of propositions we write $Prop\,(S)$.

In the following we need several functions on terms: the type variables of a term, the free variables of a term, the constants appearing in a term and the application of a type substitution σ to a term; see their definition in Table I.

A *term substitution* f is a map from $\mathcal{X} \times \mathcal{T}\,(\mathcal{K}_S)$ to $Term\,(S)$ such that $f\,(x, \tau) \neq x :: \tau$ for only finitely many (x, τ) and $\Sigma\,(f\,(x, \tau)) = \tau$ for all (x, τ). It is extended in the obvious way to a map on terms that replaces every free variable $x :: \tau \in vars\,(t)$ in a term t by the term $f\,(x, \tau)$.

A *type definition* is a triple $(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], p)$ such that $\mathtt{c} \in \mathcal{K}_S$, $n = \iota\,\mathtt{c}$, $\alpha_i \neq \alpha_j$ for $i \neq j$, $p \in Term\,(S)$, $\Sigma\,(p) = \tau \to \mathtt{prop}$ for some type τ, $vars\,(p) = \emptyset$ and $tvars\,(p) \subseteq \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathcal{A}$.

A *constant definition* is a triple $(\mathtt{C}, \tau, u)$ such that $\mathtt{C} \in \mathcal{C}_S$, $u \in \mathit{Term}\,(S)$, $\mathit{vars}\,(u) = \emptyset$, $\tau \in \mathcal{T}\,(\mathcal{K}_S)$, $\mathit{tvars}\,(u) = \mathit{tvars}\,(\tau)$ and $\Sigma\,(u) = \tau \leq \mathit{ctype}_S\,(\mathtt{C})$.

2.3. Theories, theorems and inference rules

A *theory* $\mathfrak{T}$ is a quadruple $(S, \mathit{Axioms}_0, \mathit{Axioms}_T, \mathit{Axioms}_D)$ consisting of a signature S, a set of axioms $\mathit{Axioms}_0 \subseteq \mathit{Prop}\,(S)$, a set of type definitions Axioms_T and a set of constant definitions Axioms_D.

A constant name $\mathtt{C}$ is called *overloaded* if there are two types τ_1 and τ_2, $\tau_1 \neq \tau_2$, such that there are u_1 and u_2 with $(\mathtt{C}, \tau_1, u_1) \in \mathit{Axioms}_D$ and also $(\mathtt{C}, \tau_2, u_2) \in \mathit{Axioms}_D$.

A *theorem* is a triple $(\mathfrak{T}, \Gamma, t)$ written $\Gamma \vdash_{\mathfrak{T}} t$ such that the proposition $t \in \mathit{Prop}\,(S)$ can be deduced in the theory $\mathfrak{T}$ from the finitely many assumptions in $\Gamma \subseteq \mathit{Prop}\,(S)$ by the inference rules of higher-order logic. We will use the shorter notation $\vdash_{\mathfrak{T}} t$ for $\emptyset \vdash_{\mathfrak{T}} t$.

An *inference rule* R can be thought of as a partial function from a theory together with a finite list of parameters, among them terms and already proven theorems, to a new theorem:

$$R\,(\mathfrak{T}, p_1, \ldots, p_n) \;=\; \Gamma \vdash_{\mathfrak{T}} t.$$

Among the inference rules we have the following ones:

- $\textsc{Axiom}\,(\mathfrak{T}, p) \quad = \quad \vdash_{\mathfrak{T}} p \quad$ for $p \in \mathit{Axioms}_0$;
- $\textsc{RepAbs}\,(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, p, x :: \tau) \quad =$

  $$\{p \cdot (x :: \tau)\} \vdash_{\mathfrak{T}} (\mathtt{rep} :: \tau_r) \cdot ((\mathtt{abs} :: \tau_a) \cdot (x :: \tau)) \equiv x :: \tau$$

  and $\textsc{AbsRep}\,(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, p, x :: \tau') \quad =$

  $$\vdash_{\mathfrak{T}} (\mathtt{abs} :: \tau_a) \cdot ((\mathtt{rep} :: \tau_r) \cdot (x :: \tau')) \equiv x :: \tau',$$

  for $(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], p) \in \mathit{Axioms}_T$ and $x \in \mathcal{X}$, where

  $$\Sigma\,(p) = \tau \to \mathtt{prop},\ \tau' = \mathtt{c}\,(\alpha_1, \ldots, \alpha_n),\ \tau_r = \tau' \to \tau,\ \tau_a = \tau \to \tau';$$

- $\textsc{Def}(\mathfrak{T}, \mathtt{C}, \tau, u) \quad = \quad \vdash_{\mathfrak{T}} \mathtt{C} :: \tau \equiv u \quad$ for $(\mathtt{C}, \tau, u) \in \mathit{Axioms}_D$;
- $\textsc{Refl}\,(\mathfrak{T}, u) \quad = \quad \vdash_{\mathfrak{T}} u \equiv u \quad$ for $u \in \mathit{Term}\,(S)$;
- $\textsc{InstType}\,(\mathfrak{T}, \{h_1, \ldots, h_n\} \vdash_{\mathfrak{T}} p, \alpha_1, \tau_1, \ldots, \alpha_m, \tau_m) \quad =$

  $$\{\sigma\, h_1, \ldots, \sigma\, h_n\} \vdash_{\mathfrak{T}} \sigma\, p,$$

  where σ is the type substitution that maps $\alpha_i \in \mathcal{A}$ to $\tau_i \in \mathcal{T}\,(\mathcal{K}_S)$ provided $\alpha_i \neq \alpha_j$ for $i \neq j$;

– $\textsc{Inst}\,(\mathfrak{T}, \{h_1, \ldots, h_n\} \vdash_{\mathfrak{T}} p, x_1, \tau_1, t_1, \ldots, x_m, \tau_m, t_m) \quad =$

$$\{f\,h_1, \ldots, f\,h_n\} \vdash_{\mathfrak{T}} f\,p,$$

where f is the term substitution given bei $f(x_i, \tau_i) = t_i$ for $i = 1, \ldots, m$, provided $(x_i, \tau_i) \neq (x_j, \tau_j)$ for $i \neq j$.

There are more inference rules that deal for example with implication and quantification ([1],[9],[10]). These do not interest us very much; we only need a certain property that we will state in Section 2.6. This property holds for all inference rules we have not mentioned here.

2.4. Theory Extensions

Starting from an initial theory, new axioms, type and constant definitions are added to the theory. This is where it gets interesting for us: how do we for example ensure that the definition of possibly overloaded constants does not endanger the consistency of our theory?

A theory $\mathfrak{T}'$ is called an *extension* of a theory $\mathfrak{T}'$ if there is a series of theory extension commands E_i such that

$$\mathfrak{T}' = E_n(E_{n-1}(\ldots(E_0(\mathfrak{T}, \ldots), \ldots), \ldots), \ldots).$$

A *valid theory* is a theory that is an extension of the initial theory (see Section 2.4.1).

A *theory extension command* E can be thought of as a partial function from a theory together with a finite list of parameters, among them terms and theorems (but no theories), to the extended theory:

$$E\,(\mathfrak{T}, p_1, \ldots, p_n) \;=\; \mathfrak{T}'.$$

There are five such extension commands: Declare, Define, TypeDecl, TypeDef and AssertAxiom. In the following, the original theory is always $\mathfrak{T} = (S, \mathit{Axioms}_0, \mathit{Axioms}_T, \mathit{Axioms}_D)$ while the extended one is denoted $\mathfrak{T}' = (S', \mathit{Axioms}'_0, \mathit{Axioms}'_T, \mathit{Axioms}'_D)$.

2.4.1. *The initial theory*

The initial theory contains at least the constant $\equiv$, and at least the types `prop` and $\rightarrow$. There are more constants in the initial theory, like the constants for implication ($\Rightarrow$) and quantification ($\forall$); which there are exactly is a design choice which has been made differently for example in Isabelle or HOL-light. Furthermore we demand $\mathit{Axioms}_T = \emptyset$ and $\mathit{Axioms}_D = \emptyset$.

2.4.2. *Extending a theory by declaring a new constant*

We split the process of defining a new constant into two steps, introducing the new constant by declaring it, and defining the declared

constant. This allows the second step to be executed more than once for the same constant (name) which is crucial for overloading the constant. The extension command that corresponds to the first step is

$$\textsc{Declare}\,(\mathfrak{T}, \mathtt{C}, \tau) = \mathfrak{T}'$$

for $\tau \in \mathcal{T}\,(S)$ and $\mathtt{C} \in \mathcal{C}$ but $\mathtt{C} \notin \mathcal{C}_S$. The axioms do not change, but the signature does:

$$S' = (\mathcal{K}_S, \mathcal{C}_S \cup \{\mathtt{C}\}, \mathit{ctype}_{S'}).$$

The function $\mathit{ctype}_{S'}$ results from ctype_S by defining its value at $\mathtt{C}$:

$$\mathit{ctype}_{S'}\,(\mathtt{C}') := \begin{cases} \mathit{ctype}_S\,(\mathtt{C}') & \text{if } \mathtt{C} \neq \mathtt{C}', \\ \tau & \text{if } \mathtt{C} = \mathtt{C}'. \end{cases}$$

The resulting theory is $\mathfrak{T}' = (S', \mathit{Axioms}_0, \mathit{Axioms}_T, \mathit{Axioms}_D)$.

2.4.3. *Extending a theory by defining a constant*
Once a constant has been declared, it can be defined. We say two types τ_1 and τ_2 do *overlap* if there is a type τ such that $\tau \leq \tau_1$ and $\tau \leq \tau_2$; otherwise we call them *non-overlapping*. We extend this notion to constants: two constants $\mathtt{C}_1 :: \tau_1$ and $\mathtt{C}_2 :: \tau_2$ are called non-overlapping if $\mathtt{C}_1 \neq \mathtt{C}_2$ or τ_1 and τ_2 are non-overlapping.

The extension command for defining a constant is

$$\textsc{Define}\,(\mathfrak{T}, \mathtt{C}, \tau, u) = \mathfrak{T}'$$

where $(\mathtt{C}, \tau, u)$ is a constant definition (see Section 2.2). Let the set *Crit* be defined by

$$\begin{aligned} \mathit{Crit}_I &= \{\equiv :: \alpha \to (\alpha \to \mathtt{prop}), \ldots\}, \\ \mathit{Crit}_0 &= \bigcup \{\mathit{consts}\,(p) \,|\, p \in \mathit{Axioms}_0\}, \\ \mathit{Crit}_D &= \{\mathtt{C} :: \tau \,|\, \exists_u\,(\mathtt{C}, \tau, u) \in \mathit{Axioms}_D\}, \\ \mathit{Crit}_T &= \{\mathtt{rep} :: \alpha \to \beta, \mathtt{abs} :: \alpha \to \beta\}, \\ \mathit{Crit} &= \mathit{Crit}_I \cup \mathit{Crit}_0 \cup \mathit{Crit}_D \cup \mathit{Crit}_T. \end{aligned}$$

In the above we have $\alpha, \beta \in \mathcal{A}$ and $\alpha \neq \beta$. The set Crit_I contains all the constants that play a rôle in any of the inference rules of the logic. In addition to $\equiv$ these might be $\Rightarrow$, $\forall$, and so on.

For the Define command to complete successfully, two conditions must be met:

1. $\mathtt{C} :: \tau$ and $\mathtt{C}' :: \tau'$ are non-overlapping for all $\mathtt{C}' :: \tau' \in \mathit{Crit}$;

2. the reduction system (see Section 2.5) induced by the set of constant definitions
$$Axioms'_D = Axioms_D \cup \{(\mathtt{C}, \tau, u)\}$$
is *terminating*.

Then we have $\mathfrak{T}' = (S, Axioms_0, Axioms_T, Axioms'_D)$.

2.4.4. *Extending a theory by declaring a new type*
For the declaration of a new type the command

$$\textsc{TypeDecl}\,(\mathfrak{T}, \mathtt{c}) \;=\; \mathfrak{T}'$$

is available. The assumptions for this command to succeed are $\mathtt{c} \in \mathcal{K}$ and $\mathtt{c} \notin \mathcal{K}_S$. The axioms of the theory are left untouched, only the signature changes:

$$S' = (\mathcal{K}_S \cup \{\mathtt{c}\}, \mathcal{C}_S, ctype_S).$$

The resulting theory is therefore

$$\mathfrak{T}' = (S', Axioms_0, Axioms_T, Axioms_D).$$

2.4.5. *Extending a theory by defining a type*
The command for extending a theory by a new type is

$$\textsc{TypeDef}\,(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, \vdash_{\mathfrak{T}}\; p \cdot t) \;=\; \mathfrak{T}'.$$

The side conditions are: $\mathtt{c} \in \mathcal{K}_S$, there must be no l', p' with $(\mathtt{c}, l', p') \in Axioms_T$, and $(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], p)$ must be a type definition (see Section 2.2).

The resulting theory is

$$\mathfrak{T}' = (S, Axioms_0, Axioms_T \cup \{(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], p)\}, Axioms_D).$$

2.4.6. *Extending a theory by asserting an axiom*
Asserting an axiom can destroy all sorts of properties of a theory, for example its consistency. There is nothing we can do about this; people should be very careful when asserting axioms. So the basic advice is not to assert axioms at all but to rely on the axioms in the initial theory. But if someone really wants to assert an axiom, then we *can* make sure that this axiom does not interfere with the constant definitions so far.

The command for extending a theory by asserting an axiom is

$$\textsc{AssertAxiom}\,(\mathfrak{T}, p),$$

where $p \in \mathit{Prop}(S)$. Forming again the set Crit_D by

$$\mathit{Crit}_D = \{\mathtt{C} :: \tau \mid \exists_u (\mathtt{C}, \tau, u) \in \mathit{Axioms}_D\},$$

we demand that $\mathtt{C} :: \tau$ and $\mathtt{C}' :: \tau'$ are non-overlapping for all $\mathtt{C} :: \tau \in \mathit{consts}(p)$ and all $\mathtt{C}' :: \tau' \in \mathit{Crit}_D$.

The resulting theory is $\mathfrak{T}' = (S, \mathit{Axioms}_0 \cup \{p\}, \mathit{Axioms}_T, \mathit{Axioms}_D)$.

2.5. The Reduction System Induced by a Set of Definitions

Let us first concern ourselves with those sets that from a purely syntactic point of view could be subsets of Axioms_D for some valid theory $\mathfrak{T}$. To clarify what this actually means we inductively define $\mathit{Defs}(S)$, the *set of definitions associated with a signature* S:

- $\emptyset \in \mathit{Defs}(S)$;
- let us assume $D \in \mathit{Defs}(S)$; let us further assume that $(\mathtt{C}, \tau, u)$ is a constant definition with respect to S as defined in Section 2.2; if $\mathtt{C}' :: \tau'$ and $\mathtt{C} :: \tau$ are non-overlapping for all $(\mathtt{C}', \tau', u') \in D$ then we conclude
$$D \cup \{(\mathtt{C}, \tau, u)\} \in \mathit{Defs}(S).$$

For the set Axioms'_D and signature S from Section 2.4.3 we have by construction $\mathit{Axioms}'_D \in \mathit{Defs}(S)$. For any signature S and any $D \in \mathit{Defs}(S)$ we define the abstract reduction system [4] $RS(D) = (\mathit{Term}(S), \rightarrow_D)$ by

$$t \xrightarrow{(p,d)}_D t' \quad \text{iff} \quad \begin{array}{l}\text{there is } d = (\mathtt{C}, \tau, u) \in D \text{ and a type substitution} \\ \sigma \text{ such that } \mathtt{C} :: \sigma\,\tau \text{ occurs in } t \text{ at position } p \\ \text{and } t' \text{ can be obtained by replacing this specific} \\ \text{occurrence in } t \text{ with } \sigma\,u.\end{array}$$

We just write $t \rightarrow_D t'$ if (p, d) is of no importance to us.

The reduction system is *terminating* iff there is no infinite chain

$$t \rightarrow_D t' \rightarrow_D t'' \rightarrow_D \ldots.$$

The reduction system is *(semi-)confluent* iff for $t \rightarrow_D t'$ and $t \rightarrow^*_D t''$ we can always find s with $t' \rightarrow^*_D s$ and $t'' \rightarrow^*_D s$. As usual, $\rightarrow^*_D$ denotes the reflexive and transitive hull of $\rightarrow_D$.

THEOREM 1. *$RS(D)$ is confluent for any $D \in \mathit{Defs}(S)$.*

Proof. Assume

$$t \xrightarrow{(p,d)}_D t'$$

and

$$t \xrightarrow{(q_1,e_1)}_D t''_1 \xrightarrow{(q_2,e_2)}_D t''_2 \cdots \xrightarrow{(q_n,e_n)}_D t''_n.$$

If $p \notin \{q_1, \ldots, q_n\}$ then the reduction (p, d) is independent of the other reductions; therefore both

$$t' \xrightarrow{(q_1,e_1)}_D \cdots \xrightarrow{(q_n,e_n)}_D s \quad \text{and} \quad t''_n \xrightarrow{(p,d)}_D s$$

for some s. On the other hand, assume i is the least index such that $p = q_i$. Then $d = e_i$ follows because otherwise we would have found z (the constant in t at position p) such that $z \leq (\mathtt{C}, \tau)$ and $z \leq (\mathtt{C}', \tau')$ where $(\mathtt{C}, \tau, u) = d$ and $(\mathtt{C}', \tau', u') = e_i$ for certain u, u', which contradicts the assumption that $\mathtt{C} :: \tau$ and $\mathtt{C}' :: \tau'$ are non-overlapping for $d \neq e_i$. Thus

$$t' \xrightarrow{(q_1,e_1)}_D \cdots \xrightarrow{(q_{i-1},e_{i-1})}_D t''_i \xrightarrow{(q_{i+1},e_{i+1})}_D \cdots \xrightarrow{(q_n,e_n)}_D t''_n.$$

□

Therefore every terminating $RS(D)$ is *convergent* and hence

$$\mathcal{N}_D(t) = s \quad \text{iff } t \to_D^* s \text{ and there is no } s' \text{ such that } s \to_D s'$$

is a total and uniquely defined function on $Term(S)$.

Note that $\mathcal{N}_D$ is *type preserving*, that is

$$\Sigma(\mathcal{N}_D(t)) = \Sigma(t)$$

holds for all $t \in Term(S)$.

For a given valid theory $\mathfrak{T} = (S, Axioms_0, Axioms_T, Axioms_D)$ we know by the very definition of validness

1. $Axioms_D \in Defs(S)$,
2. $RS(Axioms_D)$ is terminating.

Therefore $\mathcal{N}_{Axioms_D}$ is well-defined; we use the shorter notation $\mathcal{N}_{\mathfrak{T}}$ for this function.

2.6. Overloading is Conservative

We are now in a position to show that adding overloading to higher-order logic as described in the previous sections is *conservative* in the sense that every valid theory

$$\mathfrak{T} = (S, Axioms_0, Axioms_T, Axioms_D)$$

can be reduced to a valid theory

$$\mathfrak{T}^- = (S, Axioms_0, Axioms_T^-, \emptyset).$$

The theory $\mathfrak{T}^-$ does not contain any constant definitions, and therefore no overloaded constant names; it can be considered a theory of ordinary higher-order logic without overloading.

We achieve this reduction by setting

$$\mathit{Axioms}_T^- = \{(\mathtt{c}, l, \mathcal{N}_{\mathfrak{T}}(p)) \mid (\mathtt{c}, l, p) \in \mathit{Axioms}_T\}.$$

More generally, let $\mathfrak{S} = (S', \ldots)$ be an extension of $\mathfrak{T}$. We define

$$\mathfrak{T}^{-\mathfrak{S}} = (S', \mathit{Axioms}_0, \mathit{Axioms}_T^{-\mathfrak{S}}, \emptyset),$$
$$\mathit{Axioms}_T^{-\mathfrak{S}} = \{(\mathtt{c}, l, \mathcal{N}_{\mathfrak{S}}(p)) \mid (\mathtt{c}, l, p) \in \mathit{Axioms}_T\}.$$

The latter construction is obviously a generalization of the first:

$$\mathfrak{T}^- = \mathfrak{T}^{-\mathfrak{T}}.$$

Corollary 1 proves that overloading is conservative. The corollary can be split in two parts that are proved separately, the first part by induction over the inference rules, the second one by induction over the theory extension commands.

THEOREM 2. *If $\mathfrak{S}$ is an extension of the valid theory $\mathfrak{T}$ then*

$$\{h_1, \ldots, h_n\} \vdash_{\mathfrak{T}} t,$$

implies

$$\{\mathcal{N}(h_1), \ldots, \mathcal{N}(h_n)\} \vdash_{\mathfrak{T}^{-\mathfrak{S}}} \mathcal{N}(t),$$

where $\mathcal{N} = \mathcal{N}_{\mathfrak{S}}$.

Proof. We set $\mathfrak{T}' = \mathfrak{T}^{-\mathfrak{S}}$. For each theorem $\Gamma \vdash_{\mathfrak{T}} t$ we show that $\mathcal{N}(\Gamma) \vdash_{\mathfrak{T}'} \mathcal{N}(t)$ is a valid theorem by rule induction:

- $\Gamma \vdash_{\mathfrak{T}} t = \vdash_{\mathfrak{T}} t = \textsc{Axiom}(\mathfrak{T}, t)$:
  Then it follows that $t \in \mathit{Axioms}_0$, and this implies $\mathcal{N}(t) = t$. Therefore we conclude

  $$\textsc{Axiom}(\mathfrak{T}', t) = \vdash_{\mathfrak{T}'} t = \vdash_{\mathfrak{T}'} \mathcal{N}(t).$$

- $\Gamma \vdash_{\mathfrak{T}} t = \textsc{RepAbs}(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, p, x :: \tau)$:
  Thus the hypotheses and the conclusion are $\Gamma = \{p \cdot (x :: \tau)\}$ and $t = (\mathtt{rep} :: \tau_r) \cdot ((\mathtt{abs} :: \tau_a) \cdot (x :: \tau)) \equiv x :: \tau$. Applying $\mathcal{N}$ to t yields $\mathcal{N}(t) = t$, applying $\mathcal{N}$ to Γ results in

  $$\mathcal{N}(\Gamma) = \mathcal{N}(\{p \cdot (x :: \tau)\}) = \{\mathcal{N}(p) \cdot (x :: \tau)\}.$$

  The obvious conclusion is

  $$\mathcal{N}(\Gamma) \vdash_{\mathfrak{T}'} \mathcal{N}(t) = \textsc{RepAbs}(\mathfrak{T}', \mathtt{c}, \alpha_1, \ldots, \alpha_n, \mathcal{N}(p), x :: \tau).$$

Because we assume $\textsc{RepAbs}\,(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, p, x :: \tau)$ to be a valid theorem, we know $(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], p) \in \mathit{Axioms}_T$ which implies $(\mathtt{c}, [\alpha_1, \ldots, \alpha_n], \mathcal{N}\,(p)) \in \mathit{Axioms}_T^-$ which implies that

$$\textsc{RepAbs}\,(\mathfrak{T}', \mathtt{c}, \alpha_1, \ldots, \alpha_n, \mathcal{N}\,(p), x :: \tau)$$

is a valid theorem, too.

– $\Gamma \vdash_{\mathfrak{T}} t = \textsc{AbsRep}\,(\mathfrak{T}, \mathtt{c}, \alpha_1, \ldots, \alpha_n, p, x :: \tau)$:
This case is handled similarly as the previous case.

– $\Gamma \vdash_{\mathfrak{T}} t \;= \vdash_{\mathfrak{T}} \mathtt{C} :: \tau \equiv u \;=\; \textsc{Def}(\mathfrak{T}, \mathtt{C}, \tau, u)$:
Applying $\mathcal{N}$ to the conclusion of the theorem yields

$$\mathcal{N}\,(\mathtt{C} :: \tau \equiv u) = (\mathcal{N}\,(\mathtt{C} :: \tau) \equiv \mathcal{N}\,(u))\,.$$

Because we assume $\textsc{Def}(\mathfrak{T}, \mathtt{C}, \tau, u)$ to be a valid theorem, we have $(\mathtt{C}, \tau, u) \in \mathit{Axioms}_D(\mathfrak{T}) \subseteq \mathit{Axioms}_D(\mathfrak{S})$. This again implies

$$\mathtt{C} :: \tau \rightarrow_{\mathit{Axioms}_D(\mathfrak{S})} u.$$

This means that applying $\mathcal{N}$ to $\mathtt{C} :: \tau$ and u yields the same result: $\mathcal{N}\,(\mathtt{C} :: \tau) = \mathcal{N}\,(u)$. We conclude:

$$\vdash_{\mathfrak{T}'} \mathcal{N}\,(\mathtt{C} :: \tau \equiv u) \;= \vdash_{\mathfrak{T}'} \mathcal{N}\,(u) \equiv \mathcal{N}\,(u) \;=\; \textsc{Refl}\,(\mathfrak{T}', \mathcal{N}\,(u)).$$

– $\Gamma \vdash_{\mathfrak{T}} t = \vdash_{\mathfrak{T}} u \equiv u = \textsc{Refl}\,(\mathfrak{T}, u)$:
This is an easy one:

$$\vdash_{\mathfrak{T}'} \mathcal{N}\,(u \equiv u) = \vdash_{\mathfrak{T}'} \mathcal{N}\,(u) \equiv \mathcal{N}\,(u) = \textsc{Refl}\,(\mathfrak{T}', \mathcal{N}\,(u)).$$

– $\Gamma \vdash_{\mathfrak{T}} t = \textsc{InstType}\,(\mathfrak{T}, \{h_1, \ldots, h_n\} \vdash_{\mathfrak{T}} p, \alpha_1, \tau_1, \ldots, \alpha_m, \tau_m)$:
Let σ be the type substitution given by the α_i's and τ_i's. We have $\Gamma = \{\sigma\, h_1, \ldots, \sigma\, h_n\}$ and $t = \sigma\, p$. By induction we know that

$$\{\sigma\,(\mathcal{N}\,(h_1)), \ldots, \sigma\,(\mathcal{N}\,(h_n))\} \vdash_{\mathfrak{T}'} \sigma\,(\mathcal{N}\,(p))$$

is a theorem in the theory $\mathfrak{T}'$. From this we must deduce that

$$\{\mathcal{N}\,(\sigma\, h_1), \ldots, \mathcal{N}\,(\sigma\, h_n)\} \vdash_{\mathfrak{T}'} \mathcal{N}\,(\sigma\, p)$$

is also a theorem in the theory $\mathfrak{T}'$. In general the inequality

$$\sigma\,(\mathcal{N}\,(t)) \neq \mathcal{N}\,(\sigma\, t)$$

holds for a term $t \in \mathit{Term}\,(S)$ but we can repair this because

$$\mathcal{N}\,(\sigma\,(\mathcal{N}\,(t))) = \mathcal{N}\,(\sigma\, t)$$

holds. Look at those constants $\mathtt{C} :: \alpha \in \mathit{consts}\,(\sigma\,(\mathcal{N}\,(t)))$ for which $\mathcal{N}\,(\mathtt{C} :: \alpha) \neq \mathtt{C} :: \alpha$ holds. Such a constant can be treated as a variable because it is guaranteed not to interfere with any of the axioms in $\mathfrak{T}'$; given any theorem ϕ of $\mathfrak{T}'$, replacing in ϕ all occurrences of $\mathtt{C} :: \alpha$ by $x :: \alpha$ for some fresh variable name $x \in \mathcal{X}$ yields again a theorem ϕ' of $\mathfrak{T}'$. Having done so, we can then instantiate this variable with $\mathcal{N}\,(\mathtt{C} :: \alpha)$ via the INST-rule.

– $\Gamma \vdash_{\mathfrak{T}} t = \textsc{Inst}\,(\mathfrak{T}, \phi, x_1, \tau_1, t_1, \ldots, x_m, \tau_m, t_m)$:

$$\mathcal{N}\,(\Gamma) \vdash_{\mathfrak{T}'} \mathcal{N}\,(t) = \textsc{Inst}\,(\mathfrak{T}', \mathcal{N}\,(\phi), x_1, \tau_1, \mathcal{N}\,(t_1), \ldots, x_m, \tau_m, \mathcal{N}\,(t_m)).$$

– $\Gamma \vdash_{\mathfrak{T}} t = \textsc{Rule}\,(\mathfrak{T}, p_1, \ldots, p_n, s_1, \ldots, s_m, \Gamma_1 \vdash_{\mathfrak{T}} t_1, \ldots, \Gamma_r \vdash_{\mathfrak{T}} t_r)$: We have already mentioned in Section 2.3 that higher-order logic has more inference rules than the five rules we handled until now. One way to complete the rule induction would be to state all these rules explicitly and treat them in separate cases. But we have handled already all the interesting cases; the cases we have not treated yet proceed all along the same line. Therefore we choose another way to complete the proof: we just give a recipe on how to proceed in the remaining cases.

We have separated the parameter list of the rule into different classes. How this separation is done is obvious for each rule; the aim of this separation is to show

$$\mathcal{N}\,(\Gamma) \vdash_{\mathfrak{T}'} \mathcal{N}\,(t) = \textsc{Rule}\,(\mathfrak{T}', p_1, \ldots, p_n, \mathcal{N}\,(s_1), \ldots, \mathcal{N}\,(s_m), \mathcal{N}\,(\Gamma_1) \vdash_{\mathfrak{T}'} \mathcal{N}\,(t_1), \ldots, \mathcal{N}\,(\Gamma_r) \vdash_{\mathfrak{T}'} \mathcal{N}\,(t_r))).$$

If this has been achieved, the left hand side must be a valid theorem, because by induction the right hand side is a valid theorem. As in the previous cases, the key is to simplify $\mathcal{N}\,(t \cdot s)$ to $t \cdot \mathcal{N}\,(s)$ if t is an operator of the logic (that is, contained in Crit_I) like $\equiv$ or $\Rightarrow$.

The proof steps for REFL and INST serve as examples of how to prove the remaining rules. The reader is invited to prove additional cases as an excercise, for example the case of *modus ponens*.

□

THEOREM 3. *Assume $\mathfrak{T}$ is a valid theory. Then $\mathfrak{T}^-$ is also valid.*

Proof. During this proof we write $A \geq B$ if

$$A = (S, \mathit{Axioms}_0, \mathit{Axioms}_T, \emptyset), \quad B = (S', \mathit{Axioms}'_0, \mathit{Axioms}_T, \emptyset),$$

S can be obtained from S' by declaring constants and types, and $\mathit{Axioms}'_0 \subseteq \mathit{Axioms}_0$. It is clear that if B is a valid theory, then so is A.

Assume $\mathfrak{T}$ is a valid theory. Then there must be a sequence of valid theories

$$\mathfrak{T}_0, \ \ldots, \ \mathfrak{T}_n = \mathfrak{T}$$

such that for all $i = 0, \ldots, n-1$ the theory $\mathfrak{T}_{i+1}$ is an extension of

$$\mathfrak{T}'_i = \textsc{TypeDef}\,(\mathfrak{T}_i, \mathsf{c}_i, \alpha_{i,1}, \ldots, \alpha_{i,n}, \vdash_{\mathfrak{T}_i} \ p_i \cdot t_i)$$

and the type definitions in $\mathfrak{T}_{i+1}$ are the same as the ones in $\mathfrak{T}'_i$:

$$\mathit{Axioms}_T(\mathfrak{T}_0) = \emptyset, \quad \mathit{Axioms}_T(\mathfrak{T}_{i+1}) = \mathit{Axioms}_T(\mathfrak{T}'_i).$$

We show by induction that $\mathfrak{T}_i^{-\mathfrak{T}}$ is a valid theory for $i = 0, \ldots, n$.

- In the base case we have to show that $\mathfrak{T}_0^{-\mathfrak{T}}$ is a valid theory. Because of
$$\mathfrak{T}_0^{-\mathfrak{T}} \geq \mathfrak{T}_0$$
this is immediate.
- For the induction step we consider $\mathfrak{T}_{i+1}^{-\mathfrak{T}}$. Because of
$$\mathfrak{T}_{i+1}^{-\mathfrak{T}} \geq (\mathfrak{T}'_i)^{-\mathfrak{T}}$$
we only need to show that $(\mathfrak{T}'_i)^{-\mathfrak{T}}$ is a valid theory provided $\mathfrak{T}_i^{-\mathfrak{T}}$ is one. Let us set $\mathcal{N} = \mathcal{N}_{\mathfrak{T}}$. We know that the theorem
$$\vdash_{\mathfrak{T}_i} \ p_i \cdot t_i$$
holds, and by Theorem 2 we deduce
$$\vdash_{\mathfrak{T}_i^{-\mathfrak{T}}} \ \mathcal{N}\,(p_i) \cdot \mathcal{N}\,(t_i).$$
This allows us to conclude
$$(\mathfrak{T}'_i)^{-\mathfrak{T}} = \textsc{TypeDef}\,(\mathfrak{T}_i^{-\mathfrak{T}}, \mathsf{c}_i, \alpha_{i,1}, \ldots, \alpha_{i,n}, \vdash_{\mathfrak{T}_i^{-\mathfrak{T}}} \ \mathcal{N}\,(p_i) \cdot \mathcal{N}\,(t_i)).$$

We have shown that

$$\mathfrak{T}_n^{-\mathfrak{T}} = \mathfrak{T}^{-\mathfrak{T}} = \mathfrak{T}^-$$

is a valid theory. □

COROLLARY 1. *Assume that* $\mathfrak{T}$ *is a valid theory. Then so is* $\mathfrak{T}^-$ *and if*

$$\{h_1, \ldots, h_n\} \vdash_{\mathfrak{T}} t,$$

then also

$$\{\mathcal{N}_{\mathfrak{T}}(h_1), \ldots, \mathcal{N}_{\mathfrak{T}}(h_n)\} \vdash_{\mathfrak{T}^-} \mathcal{N}_{\mathfrak{T}}(t).$$

Proof. The proof is immediate from Theorems 2 and 3. □

How does the above corollary relate to consistency preservation? Let us assume that our logic contains a constant $\texttt{False} :: \texttt{prop} \in \mathit{Crit}_I \cup \mathit{Crit}_0$ with the usual meaning so that inconsistency of $\mathfrak{T}$ means that

$$\vdash_{\mathfrak{T}} \texttt{False} :: \texttt{prop}$$

can be derived. Corollary 1 tells us that $\vdash_{\mathfrak{T}^-} \mathcal{N}_{\mathfrak{T}}(\texttt{False} :: \texttt{prop})$ is then derivable, too, and because of $\mathcal{N}_{\mathfrak{T}}(\texttt{False} :: \texttt{prop}) = \texttt{False} :: \texttt{prop}$ this implies the derivability of

$$\vdash_{\mathfrak{T}^-} \texttt{False} :: \texttt{prop}.$$

This means that consistency of $\mathfrak{T}^-$ implies consistency of $\mathfrak{T}$: *overloading does not endanger consistency.*

Note that conservativity in the sense of Corollary 1 is different from syntactical conservativity which has been proposed in [2] as a mandatory property of theory extensions. Our theory extensions actually violate this property which demands that if $\mathfrak{S}$ is an extension of $\mathfrak{T} = (S, \ldots)$ and $\vdash_{\mathfrak{S}} p$ and $p \in \mathit{Prop}(S)$, then also $\vdash_{\mathfrak{T}} p$. This violation is a natural consequence of the fact that declaration and definition of both constants and types are done separately (DECLARE vs. DEFINE, TYPEDECL vs. TYPEDEF).

3. Proving Termination

3.1. CONSTANT DEPENDENCIES

For any $D \in \mathit{Defs}(S)$ we have defined the reduction system $RS(D)$ so that t can be reduced to t' iff t' results from t by unfolding definitions in D. We are interested in the termination of $RS(D)$; termination does not depend on the structure of u in a definition $(\mathtt{C}, \tau, u) \in D$ but only on what constants appear in u. We therefore define the set D' of *constant dependencies* associated with a set $D \in \mathit{Defs}(S)$ by

$$D' = \{(\mathtt{C}, \tau, c) \mid \exists_u (\mathtt{C}, \tau, u) \in D \wedge c \in \mathit{consts}(u)\}$$

and collect all these sets in

$$Deps_{\mathrm{def}}(S) = \left\{ D' \mid D \in Defs(S) \right\}.$$

Although not necessarily $D' \in Defs(S)$, we can still associate the reduction system $RS(D')$ with it. The definition we gave in Section 2.5 also applies here. We restrict its domain, though; $RS(D)$ is the pair $(Term(S), \rightarrow_D)$, but $RS(D')$ is the pair $(Consts(S), \rightarrow_{D'})$ where

$$Consts(S) = \{\mathtt{C} :: \tau \mid \mathtt{C} \in \mathcal{C} \wedge \mathtt{C} :: \tau \in Term(S)\}$$

is the set of constants with respect to the signature S. This is a little bit ambiguous but should do no harm because it will be clear from the context if we are talking about $A \in Defs(S)$ or $A \in Deps_{\mathrm{def}}(S)$.

For $D' \in Deps_{\mathrm{def}}(S)$ the reduction system $RS(D')$ is not necessarily confluent but is interesting to us because of the following theorem:

THEOREM 4. *For all $D \in Defs(S)$ we have that $RS(D)$ is terminating iff $RS(D')$ is terminating, where $D' \in Deps_{def}(S)$ is the set of constant dependencies associated with D.*

Proof. We show this claim by translating infinite chains in one reduction system to infinite chains in the other.

Given an infinite chain in $RS(D')$, say

$$c_1 \rightarrow_{D'} c_2 \rightarrow_{D'} c_3 \rightarrow_{D'} \ldots,$$

we get an infinite chain in $RS(D)$

$$c_1 \rightarrow_D u_1 \rightarrow_D u_2 \rightarrow_D \ldots$$

such that $c_2 \in consts(u_1)$, $c_3 \in consts(u_2)$ and so on by performing the next reduction in the residue of the previous reduction.

On the other hand, if we are given an infinite chain

$$u_1 \rightarrow_D u_2 \rightarrow_D u_3 \rightarrow_D \ldots$$

in $RS(D)$, then the construction of an infinite chain

$$c_1 \rightarrow_{D'} c_2 \rightarrow_{D'} c_3 \rightarrow_{D'} \ldots$$

in $RS(D')$ is a repeated application of the *Pigeon-Hole Principle*: there are infinitely many reductions taking place in u_1 but there are only finitely many constants in u_1. Therefore there must be a constant c_1 in which infinitely many reductions take place. By repeating this argument we obtain c_2, and so on.

We have defined $Deps_{\mathrm{def}}(S)$ building on the definition of $Defs(S)$. We would also like to have an independent description. To this end, we define $Deps(S)$ by

- $\emptyset \in \mathit{Deps}(S)$;
- assume $d = (\mathtt{C}_1, \tau_1, \mathtt{C}_2 :: \tau_2)$ where $\mathtt{C}_1 :: \tau_1, \mathtt{C}_2 :: \tau_2 \in \mathit{Consts}(S)$ and $\mathit{tvars}(\tau_2) \subseteq \mathit{tvars}(\tau_1)$; assume furthermore $D \in \mathit{Deps}(S)$ and for all $d' = (\mathtt{C}_3, \tau_3, \mathtt{C}_4 :: \tau_4) \in D$ either $\mathtt{C}_1 :: \tau_1$ and $\mathtt{C}_3 :: \tau_3$ are non-overlapping or otherwise $\mathtt{C}_1 :: \tau_1 = \mathtt{C}_3 :: \tau_3$; then

$$D \cup \{d\} \in \mathit{Deps}(S).$$

An immediate conclusion is $\mathit{Deps}_{\mathrm{def}}(S) \subseteq \mathit{Deps}(S)$. We do not necessarily have $\mathit{Deps}(S) \subseteq \mathit{Deps}_{\mathrm{def}}(S)$, though; but for any $D \in \mathit{Deps}(S)$ we can easily construct E' and S' such that $E' \in \mathit{Deps}_{\mathrm{def}}(S')$ and $RS(D)$ terminates iff $RS(E')$ terminates. Let us define the signature S' by

$$\begin{aligned} S' &= (\mathcal{K}_S, \mathcal{C}'_S, \mathit{ctype}'_S), \\ \mathcal{C}'_S &= \mathcal{C}_S \cup \{\Phi\}, \\ \mathit{ctype}'_S(\mathtt{C}) &= \begin{cases} \mathit{ctype}_S(\mathtt{C}) & \text{if } \mathtt{C} \neq \Phi, \\ \alpha \to \beta & \text{if } \mathtt{C} = \Phi. \end{cases} \end{aligned}$$

In the above Φ is a new constant name (that is, $\Phi \in \mathcal{C} \backslash \mathcal{C}_S$), and $\alpha, \beta \in \mathcal{A}$ and $\alpha \neq \beta$. We first construct $E \in \mathit{Defs}(S)$. We can divide D into equivalence classes $D_1, \ldots, D_n$ such that

- D is the union of the D_i's,
- for $i \neq j$, any $(\mathtt{C}_1, \tau_1, c_1) \in D_i$ and any $(\mathtt{C}_2, \tau_2, c_2) \in D_j$, we have that $\mathtt{C}_1 :: \tau_1$ and $\mathtt{C}_2 :: \tau_2$ are non-overlapping,
- and such that we can write

$$D_i = \{(\mathtt{C}_i, \tau_i, \mathtt{C}_{i,1} :: \tau_{i,1}), \ldots, (\mathtt{C}_i, \tau_i, \mathtt{C}_{i,m_i} :: \tau_{i,m_i})\}.$$

Each such D_i corresponds to one definition $d_i \in E$ via

$$\begin{aligned} d_i &= (\mathtt{C}_i, \tau_i, u_i), \\ u_i &= \big(\cdots\big(((\Phi :: \tau'_i) \cdot (\mathtt{C}_{i,1} :: \tau_{i,1})) \cdot (\mathtt{C}_{i,2} :: \tau_{i,2})\big)\cdots\big) \cdot (\mathtt{C}_{i,m_i} :: \tau_{i,m_i}), \\ \tau'_i &= \tau_{i,1} \to (\tau_{i,2} \to \cdots (\tau_{i,m_i} \to \tau_i) \cdots). \end{aligned}$$

We obtain $E' \in \mathit{Deps}_{\mathrm{def}}(S')$ by setting $E = \{d_1, \ldots, d_n\} \in \mathit{Defs}(S')$.

THEOREM 5. *$RS(E')$ terminates iff $RS(D)$ terminates.*

Proof. Obviously $E' = D \cup \{d'_1, \ldots, d'_n\}$ where $d'_i = (\mathtt{C}_i, \tau_i, \Phi :: \tau'_i)$. Therefore any infinite chain in $RS(D)$ is an infinite chain in $RS(E')$. On the other hand any chain in $RS(E')$ involving a reduction step $\rightarrow_{d'_i}$ cannot be infinite but ends with this step because the constant Φ we introduced has no dependencies and cannot be reduced. Therefore any infinite chain in $RS(E')$ is also an infinite chain in $RS(D)$. □

COROLLARY 2. *We can decide termination of $RS(D)$ for any signature S and any $D \in Defs(S)$ iff we can decide termination of $RS(D)$ for any signature S and any $D \in Deps(S)$.*

Proof. This is a direct consequence of Theorems 4 and 5 and the fact that $Deps_{\mathrm{def}}(S) \subseteq Deps(S)$. □

Therefore we can focus now on how to prove termination of $RS(D)$ for $D \in Deps(S)$.

3.2. The TRS Induced by a Set of Constant Dependencies

A *first-order term rewriting system* (TRS) is a pair (F, R) where F is the signature of the term rewriting system and R is the set of *rules* (see [4], [5]). The signature of a term rewriting system is a set of function symbols with an arity associated with each symbol. A rule is a pair of first-order terms over the signature F, written

$$t \Longrightarrow t',$$

such that t is no variable and all variables of t' appear also in t.

Given a signature S (in the sense of Section 2.2) and $D \in Deps(S)$ we define the term rewriting system $TRS(D)$ induced by D:

$$\begin{aligned} F &= \mathcal{C}_S \cup \mathcal{K}_S, \\ arity(f) &= \begin{cases} \iota(f) & \text{if } f \in \mathcal{K}, \\ 1 & \text{if } f \in \mathcal{C} \end{cases} \\ R &= \left\{ \mathtt{C}(\tau) \Longrightarrow \mathtt{C}'(\tau') \mid (\mathtt{C}, \tau, \mathtt{C}' :: \tau') \in D \right\}. \end{aligned}$$

Note that the variables of $TRS(D)$ are from $\mathcal{A}$.

$TRS(D)$ is of special shape. Let us be more precise: we say that a TRS is an *overloading TRS* if all its rules r_i, $i = 1, \ldots, n$, have the form

$$r_i : f_i(a_i) \Longrightarrow g_i(b_i)$$

such that

1. neither f_i nor g_i appears in any of the terms $a_1, b_1, \ldots, a_n, b_n$,

2. if there are substitutions σ, ϑ such that $\sigma\, a_i = \vartheta\, a_j$ then $a_i = a_j$ or $f_i \neq f_j$.

Because of their origin we call the f_i's and g_i's *term symbols* and all the other symbols in F *type symbols*.

THEOREM 6. *T is an overloading TRS iff there exists a signature S and $D \in Deps(S)$ with $T = TRS(D)$.*

Proof. If $D \in Deps(S)$ then obviously $TRS(D)$ is overloading.

Assume on the other hand that we are given an overloading TRS $T = (F, \{r_1, \ldots, r_n\})$. After suitably choosing $\mathcal{K}$ etc., and fixing $\alpha \in \mathcal{A}$,

$$\begin{aligned} S &= (\mathcal{K}_S, \mathcal{C}_S, ctype_S), \\ \mathcal{C}_S &= \{f_1, \ldots, f_n, g_1, \ldots, g_n\}, \\ \mathcal{K}_S &= F \backslash \mathcal{C}_S, \\ ctype_S(\mathtt{C}) &= \alpha, \\ D &= \{(f_i, a_i, g_i :: b_i) \mid i = 1, \ldots, n\} \end{aligned}$$

delivers D such that $D \in Deps(S)$ and $T = TRS(D)$. □

Every TRS can be considered a reduction system on the first-order terms over its signature. For this reduction relation we also write $\Longrightarrow$, for its reflexive and transitive hull we write $\Longrightarrow^*$.

It is not difficult to show directly that given $D \in Deps(S)$, $RS(D)$ terminates iff $TRS(D)$ terminates. But we gain more insight by using a result that has been established in [6] about the connection between *dependency pairs* and termination of term rewriting systems. This insight is later on put to further good use when we tackle the question of how to practically prove termination of an overloading TRS.

So assume $T = (F, R)$ is an overloading TRS. We call $f \in F$ *defined* if there is a rule $f(\ldots) \Longrightarrow \ldots \in R$, otherwise we call f a *constructor*. Let us further assume that there is an injective map from the set of defined symbols to the set of constructors which preserves the arity, so that a defined $\mathtt{C} \in F$ is assigned a constructor $\tilde{\mathtt{C}}$ which appears in none of the rules. This is no restriction since we could just add new symbols to F.

We call $\langle \tilde{\mathtt{C}}_1(\tau_1), \tilde{\mathtt{C}}_2(\tau_2) \rangle$ a *dependency pair* if $\mathtt{C}_1(\tau_1) \Longrightarrow \mathtt{C}_2(\tau_2) \in R$ and $\mathtt{C}_2$ is defined. For an overloading TRS this notion of dependency pair actually coincides with the notion found in [6] or [5].

A *chain* is a (finite or infinite) sequence

$$\langle s_1, t_1 \rangle, \langle s_2, t_2 \rangle, \ldots$$

of dependency pairs such that there are substitutions (with respect to F) σ_i with

$$\sigma_i\, t_i \Longrightarrow^* \sigma_{i+1}\, s_{i+1}$$

for all $i = 1, 2, \ldots$. The following theorem has been stated and proven in [6, Theorem 6]:

THEOREM 7. *A TRS is terminating iff no infinite chain of dependency pairs exists.*

In general it is not decidable to check whether even only two dependency pairs form a chain; the next theorem teaches that in the case of an overloading TRS this check is just mere syntactic unification. In order to formulate the theorem, we need an auxiliary definition. We fix a variable β of the TRS and for any first-order term t of the TRS and also for any substitution we define the function $cap\,(t)$ by

$$\begin{aligned} cap\,(\alpha) &= \alpha, \\ cap\,(f(t_1, \ldots, t_n)) &= \begin{cases} \beta & \text{if } f \text{ is a term symbol,} \\ f(cap\,(t_1), \ldots, cap\,(t_n)) & \text{if } f \text{ is a constructor,} \end{cases} \\ cap\,(\sigma)\,\alpha &= cap\,(\sigma\,\alpha). \end{aligned}$$

THEOREM 8. *A* simple *chain is a sequence*

$$\langle s_1,\, t_1\rangle,\, \langle s_2,\, t_2\rangle,\, \ldots$$

of dependency pairs such that there are substitutions σ_i with

$$\sigma_i\, t_i = \sigma_{i+1}\, s_{i+1} \quad \textit{and} \quad cap\,(\sigma_i) = \sigma_i.$$

In an overloading TRS, the notions chain and simple chain coincide.

This is a consequence of:

THEOREM 9. *In an overloading TRS, if $\sigma\, t \Longrightarrow^* \vartheta\, s$ for two substitutions σ, ϑ and terms t, s such that $cap\,(t) = t$ and $cap\,(s) = s$, then also $cap\,(\sigma)\, t = cap\,(\vartheta)\, s$.*

Proof. Because our TRS is overloading, the only reductions that can take place in $\sigma\, t$ are the ones that take place in those regions of the term that have been replaced by β in $cap\,(\sigma\, t)$. Let us call these regions $t_1, \ldots, t_n$. Furthermore we denote the regions in $\vartheta\, s$ that have been replaced by β in $cap\,(\vartheta\, s)$ by $s_1, \ldots, s_m$. Because reductions cannot "escape" regions, and also cannot "create" or "move" regions, we have $n = m$ and t_i reduces to s_i. Replacing all regions by β makes these reductions unnecessary, and we have

$$cap\,(\sigma\, t) = cap\,(\vartheta\, s).$$

Neither t nor s contain term symbols; therefore

$$cap\,(\sigma\,t) = cap\,(\sigma)\,t, \quad cap\,(\vartheta\,s) = cap\,(\vartheta)\,s,$$

which finishes our argument. □

Proof of Theorem 8. It is clear that a simple chain is also a chain. Now assume that we are given a chain. We need to show that it actually is a simple chain. We know $\sigma_i\,t_i \Longrightarrow^* \sigma_{i+1}\,s_{i+1}$. Theorem 9 tells us that

$$cap\,(\sigma_i)\,t_i = cap\,(\sigma_{i+1})\,s_{i+1},$$

because for any dependency pair $\langle s_i,\, t_i\rangle$ we have

$$cap\,(s_i) = s_i \quad \text{and} \quad cap\,(t_i) = t_i.$$

□

COROLLARY 3. *Assume $D \in Deps\,(S)$. Then we have that $RS\,(D) = (Consts\,(S), \rightarrow_D)$ is terminating iff $TRS\,(D)$ is terminating.*

Proof. An infinite reduction sequence in $RS\,(D)$ is just a simple chain in $TRS\,(D)$ together with the substitutions which prove the simple-chain property. Theorems 8 and 7 then prove our claim.

3.3. Undecidability of Proving Termination

In the first part of this paper we have explained how overloading can be added safely to ordinary higher-order logic as it is implemented in proof-assistants like Isabelle, HOL 4 or HOL-light.

But is this enriched logic also implementable by proof-assistants? That is, if the user gives his possibly overloaded definitions D to the proof-assistant, can it decide whether to accept or reject them solely based on if the definitions play by the rules of HOLCO? The answer is no; as we will show now, one cannot decide if $RS\,(D)$ is terminating. Corollaries 2 and 3 tell us that this decision problem is equivalent to the problem of deciding if an overloading TRS is terminating.

Therefore we can show the undecidability by reducing *Post's Correspondence Problem for Prefix Morphisms* (PCPP) to the termination problem for overloading TRS's. An instance of the PCPP is the following problem: Given pairs $(a_1, b_1), \ldots, (a_n, b_n)$ of non-empty finite words over the alphabet $\{0, 1\}$ such that for $i \neq j$ neither a_i is a prefix of a_j nor b_i is a prefix of b_j, decide if there is a solution, that is a finite sequence $i_1, \ldots, i_m$ with

$$a_{i_1} a_{i_2} \ldots a_{i_m} = b_{i_1} b_{i_2} \ldots b_{i_m}.$$

The PCPP is undecidable [8].

For the above instance of the PCPP, we construct an overloading TRS. The construction is basically the one that is used in [5] for reducing *Post's Correspondence Problem*, which is the PCPP without demanding that the a_i's and b_j's cannot be prefixes of each other, to the termination problem of a general TRS.

Our TRS has five function symbols: a nullary symbol $\Box$, three unary symbols 0, 1, $\mathtt{C}$, and one ternary symbol $\mathtt{c}$. For a binary word $p = q_1 q_2 \dots q_n$ (that is $q_1, \dots, q_n \in \{0, 1\}$) we define the term

$$p\,(t) = q_1(q_2(\dots(q_n(t))\dots)).$$

For each pair (a_i, b_i) of the PCPP instance the TRS has a rule

$$\mathtt{C}\,(\mathtt{c}\,(a_i\,(\alpha), b_i\,(\beta), \gamma)) \Longrightarrow \mathtt{C}\,(\mathtt{c}\,(\alpha, \beta, \gamma)),$$

furthermore it has the rules

$$\begin{aligned} r_0: &\quad \mathtt{C}\,(\mathtt{c}\,(\Box, \Box, 0\,(\alpha))) \Longrightarrow \mathtt{C}\,(\mathtt{c}\,(0\,(\alpha), 0\,(\alpha), 0\,(\alpha))), \\ r_1: &\quad \mathtt{C}\,(\mathtt{c}\,(\Box, \Box, 1\,(\alpha))) \Longrightarrow \mathtt{C}\,(\mathtt{c}\,(1\,(\alpha), 1\,(\alpha), 1\,(\alpha))). \end{aligned}$$

THEOREM 10. *The TRS we just constructed is an overloading TRS, and it is terminating iff the corresponding PCPP instance has no solution.*

Proof. The left hand sides of no two rules unify, because for $a_i\,(\alpha)$ and $a_j(\alpha')$ to unify, a_i would need to be a prefix of a_j or the other way around. Therefore the TRS is overloading.

If the PCPP instance has a solution $i_1, \dots, i_m$, then

$$\mathtt{C}\,(\mathtt{c}\,(s\,(\Box), s\,(\Box), s(\Box)))$$

for $s = a_{i_1} \dots a_{i_m}$ starts a cyclic and therefore infinite reduction sequence.

If the TRS is not terminating, then there exists an infinite simple chain. Because all the other rules reduce the size of the term, there must be two dependency pairs in this chain that correspond to r_0 or r_1. The dependency pairs between these two pairs form a solution of the PCPP instance, after throwing away those pairs that correspond to r_0 or r_1. That the solution is not empty is ensured because directly after r_0 or r_1 have been applied, neither of them can be applied again. $\Box$

Note that we also have shown that it is undecidable if an overloading TRS admits *cyclic* reductions.

COROLLARY 4. *It is undecidable if an overloading TRS is terminating. Actually, it is not even semi-decidable.*

Proof. It is semi-decidable if any given PCPP instance has a solution. Considering that the PCPP is undecidable [8], Theorem 10 tells us that the termination of an overloading TRS cannot be semi-decidable. □

3.4. Practically Proving Termination

Corollary 4 shows that any check a proof-assistant might employ for deciding if a set of definitions is associated with a reduction system that terminates will be incomplete: there will always be definitions that *should* pass but *won't* pass.

On the other hand overloading has proven to be a very useful technique which is regularly taken advantage of by users of the proof-assistant Isabelle. Until now the use of overloading in Isabelle was an act of faith; experienced users "knew" how to use overloading only in a sound way. Therefore it is worthwhile to examine how to devise a check that can give definitive and reliable answers on almost all overloaded definitions an experienced user would issue.

We have developed an add-on to the proof-assistant Isabelle that provides functionality for checking if $RS(D)$ terminates for the set D of definitions of an Isabelle theory. The check can either be done by using external TRS termination provers, or by using a built-in termination prover [16].

3.4.1. *External Provers and the Dependency Pair Method*

There are now several tools available that can prove the termination of a wide range of term rewriting systems, among them *AProVE* [12] and the *Tyrolean Termination Tool* [13]. We have shown that checking definitions can be done by checking if an overloading TRS is terminating. Therefore we can just give this overloading TRS to one of these tools.

We have tested the viability of this approach by checking four formalizations in Isabelle/HOL, all of which make use of overloading:

Main the starting point for Isabelle/HOL users,

Bali which has been concerned with the formalization of various aspects of the programming language Java [17],

Mat a formalization of checking the bounds of real linear programs employing matrices [14],

Nom and the implementation of nominal techniques for Isabelle [15].

Our add-on extracts the overloading TRS that corresponds to the definitions of the corresponding Isabelle theories. Note that *all* definitions

of a theory are collected for this purpose; this also recursively includes the definitions of the parent theories of the theory.

Giving the four generated TRSs directly to either AProVE or TTT fails with a time-out after several minutes. Can we somehow preprocess the TRS to provide the checkers with easier input?

Both checkers use the dependency pair method invented in [6]. The basic idea there is to calculate a *dependency graph*. This graph has the dependency pairs of the TRS as its nodes and there is an edge from $\langle a, b\rangle$ to $\langle c, d\rangle$ if $\langle a, b\rangle \langle c, d\rangle$ is a chain. Therefore in general the dependency graph is not computable and it is necessary to use an approximation of the dependency graph. This approximation has an edge from $\langle a, b\rangle$ to $\langle c, d\rangle$ if there are substitutions σ, ϑ such that

$$\sigma\,(\mathit{lin}\,(\mathit{cap}\,(b))) = \vartheta\, c,$$

see for example [5, p. 252]; *lin* is a function that replaces *all* occurrences of variables by fresh variables.

We can do better for overloading TRSs. Chains are simple chains and therefore we can compute the exact dependency graph which has an edge from $\langle a, b\rangle$ to $\langle c, d\rangle$ iff there are substitutions σ, ϑ such that

$$\sigma\, b = \vartheta\, c.$$

In general the exact dependency graph has less edges than the approximated dependency graph. Take for example

$$b = \tilde{\mathtt{C}}\,(\alpha \to \alpha), \quad c = \tilde{\mathtt{C}}\,(\mathtt{real} \to \mathtt{int}).$$

Then the approximated graph has an edge from $\langle a, b\rangle$ to $\langle c, d\rangle$ because $\beta \to \gamma$ and $\mathtt{real} \to \mathtt{int}$ unify, but the exact dependency graph has no such edge.

An overloading TRS is not terminating iff it admits an infinite simple chain of dependency pairs. Mapping this infinite chain to the exact dependency graph yields an infinite path p in the graph. We call a non-empty set N of nodes such that for $n, m \in N$, there are non-empty paths from n to m and from m to n a *cyclic component* of the graph. Each cyclic component is a *strongly connected component*. If there is an infinite path p then there also must be an infinite path p' with all of its nodes belonging to the same cyclic component, because the dependency graph is finite.

Each cyclic component of the dependency graph is a set of dependency pairs, and can therefore be considered a subset of the rules of the overloading TRS. The overloading TRS is terminating iff all TRSs that correspond to a cyclic component are terminating.

Table II. Dependency Graph Statistics (on a 1.7GHz Pentium IV Mobile)

	Main	**Bali**	**Mat**	**Nom**
Number of nodes of DG	1684	5726	2311	1765
Construction time of DG	0.6 s	7.3 s	1 s	0.6 s
Number of nodes of root-cyclic DG	0	12	6	8
Construction time of root-cyclic DG	0.1 s	0.5 s	0.2 s	0.1 s
Number of cyclic components	0	5	6	1
Maximum number of nodes per component	-	4	1	8

Therefore we have found a method to break up our big overloading TRS into a couple of (much) smaller overloading TRSs whose dependency graphs consist of exactly one cyclic component. AProVE can automatically solve all of these systems; TTT time-outs for several of them in automatic mode but manages to solve all of them in semi-automatic mode with *polynomial interpretations* switched on. Both checkers only need a fraction of a second for all of the smaller term rewriting systems combined.

The dominating factor in our examples is constructing the dependency graph (DG) and calculating the cyclic components of the graph; the exact timings are listed in Table II. The construction of the dependency graph for Bali takes over seven seconds although discrimination nets ([21], [22]) are used to speed up the calculation. But we actually don't need the whole DG in order to prove termination; we only need those nodes of the DG that belong to a cyclic component. Instead we construct (without constructing the DG first) what we call the *root-cyclic* DG; this is the biggest subgraph of the DG such that all nodes are root-cyclic. The root-cyclic DG is usually much smaller than the DG. In order to determine which dependency pairs $d_i = \langle \tilde{\mathtt{C}}_i\,(\tau_i), \tilde{\mathtt{D}}_i\,(\varrho_i)\rangle$, $i = 1, \ldots, n$, are root-cyclic we construct a graph with nodes $\{\mathtt{C}_1, \mathtt{D}_1, \ldots, \mathtt{C}_n, \mathtt{D}_n\}$ and edges from $\mathtt{C}_i$ to $\mathtt{D}_i$. We call d_i root-cyclic if $\mathtt{C}_i$ and $\mathtt{D}_i$ belong to the same cyclic component of that graph.

Table II shows that this simple optimization has a big impact; we only need half a second to construct the root-cyclic DG for Bali compared to over seven seconds for the full DG.

3.4.2. *An Algorithm For Proving Termination*

For a proof-assistant it is somewhat unsatisfying and clumsy to have to rely on external TRS termination provers. Therefore we present an algorithm for proving termination of an overloading TRS that is easy to implement and powerful: it can handle all of our examples, that is Bali, Mat and Nom (Main is dealt with trivially). It is a variant of the

dependency pair method [6] and uses the recursive calculation of cyclic components from [7].

Let us look at a cyclic component of the (root-cyclic) DG and assume that the component consists of the dependency pairs $d_1, \ldots, d_N$ where $d_i = \langle s_i, t_i \rangle$. If we can find relations $\succ, \succeq \subseteq T \times T$, where T is the set of first-order terms that do not contain any term symbols, such that for all $x, y, z \in T$ and substitutions σ with $\sigma = \mathit{cap}(\sigma)$

- $\succ$ is well-founded, that is there is no infinite chain

$$u_0 \succ u_1 \succ u_2 \succ \ldots,$$

- $x \succ y$ and $y \succeq z$ implies $x \succ z$,
- $x \succeq y$ and $y \succ z$ implies $x \succ z$,
- $x \succ y$ implies $\sigma\, x \succ \sigma\, y$,
- $x \succeq y$ implies $\sigma\, x \succeq \sigma\, y$,

and such that

- $s_i \succ t_i$ for $i = 1, \ldots, M$, where $1 \leq M \leq N$,
- $s_i \succeq t_i$ for $i = M + 1, \ldots, N$,

then we can drop the nodes $d_1, \ldots, d_M$ and start all over again by calculating the cyclic components of this reduced graph.

THEOREM 11. *If we can continue this process until we have arrived at the empty graph then we have successfully shown termination.*

Proof. The whole overloading TRS terminates if each of its cyclic components corresponds to a terminating overloading TRS. The TRS that corresponds to the cyclic component we picked is not terminating iff there exists an infinite simple chain of dependency pairs

$$\langle u_1, v_1 \rangle, \langle u_2, v_2 \rangle, \ldots$$

where $\langle u_i, v_i \rangle \in \{d_1, \ldots, d_N\}$ for all i, which means that there are substitutions σ_i with $\sigma_i = \mathit{cap}(\sigma_i)$ and

$$\sigma_1\, u_1 \rhd_1 \sigma_1\, v_1 = \sigma_2\, u_2 \rhd_2 \sigma_2\, v_2 = \cdots.$$

In the above $\rhd_i$ is either $\succ$ or $\succeq$, depending on if $j \leq M$ or $j > M$, respectively, where $\langle u_i, v_i \rangle = d_j$. Here we have used the property of $\succ$ and $\succeq$ to be closed under substitutions.

If any of the d_j's for $j \leq M$ is equal to $u_i \succ v_i$ for infinitely many i then $\rhd_i$ is equal to $\succ$ for infinitely many i. This allows us to construct

an infinite descending chain with respect to $\succ$ by using the transitivity-like properties of $\succ$ and $\succeq$. But this contradicts the property of $\succ$ to be well-founded.

Therefore there exists an infinite simple chain of dependency pairs in $\{d_1, \ldots, d_N\}$ iff there exists an infinite simple chain of dependency pairs in $\{d_{M+1}, \ldots, d_N\}$. □

Which relations $\succ$ and $\succeq$ should we choose? This depends on the cyclic component we are currently examining. We use a restricted form of linear polynomial interpretation to find these relations which is a special case of the method described in [20]. Note that the full method of [20] could be applied; it could even be improved because our relations need not have the *subterm property* (which is nevertheless fulfilled for the polynomial interpretation we describe here). We do not use this more general method because finding the relations becomes much more costly in terms of runtime.

Terms are interpreted as those linear polynomials that can be viewed as functions from $\mathbb{R}_+ \times \cdots \times \mathbb{R}_+$ to $\mathbb{R}_+$. The set $\mathbb{R}_+$ denotes the set of non-negative real numbers.

To each (type) symbol f of the overloading TRS we assign a real constant $c_f \geq 0$, and to each variable α of the TRS we assign a real variable $x_{i(\alpha)}$. The function i converts variables into indices and is injective. A term t is interpreted as $\Theta(t)$:

$$\Theta(\alpha) = x_{i(\alpha)}, \qquad \Theta(f(t_1, \ldots, t_n)) = c_f + \Theta(t_1) + \ldots + \Theta(t_n) \ .$$

For $\varepsilon \geq 0$ we define the relation $>_\varepsilon$ on functions from $\mathbb{R}_+^n$ to $\mathbb{R}_+$ by

$$p >_\varepsilon q \iff p(x_1, \ldots, x_n) \geq \varepsilon + q(x_1, \ldots, x_n) \quad \text{for all } x_1, \ldots, x_n \in \mathbb{R}_+.$$

This relation is well-founded for $\varepsilon > 0$. We pull back this relation to the set of terms and define

$$u \succ_\varepsilon v \iff \Theta(u) >_\varepsilon \Theta(v).$$

We write $\succeq$ instead of $\succ_0$. Can we choose $\varepsilon > 0$ such that by setting $\succ$ to $\succ_\varepsilon$ we obtain the pair of relations we are searching for?

Obviously $\succ_\varepsilon$ and $\succeq$ would have almost all of the desired properties we need in order to apply Theorem 11. In order to get rid of the "almost" we have to show that (after possibly reordering the dependency pairs) we have $s_i \succ_\varepsilon t_i$ for $i = 1, \ldots, M$ and $M \geq 1$ and also $s_i \succeq t_i$ for $i = M+1, \ldots, N$.

For two terms u and v the relation $u \succ_\varepsilon v$ holds iff

1. $\mathit{diff}(u, v) = \Theta(u)(0, \ldots, 0) - \Theta(v)(0, \ldots, 0) \geq \varepsilon$,

2. and for all $x_1, \ldots, x_n \in \mathbb{R}_+$ and all $i = 1, \ldots, n$ the inequality $\frac{\partial}{\partial x_i}(\Theta(u)(x_1, \ldots, x_n) - \Theta(v)(x_1, \ldots, x_n)) \geq 0$ holds.

The second condition just means that for all α the inequality

$$\mathit{varcount}(u, \alpha) \geq \mathit{varcount}(v, \alpha)$$

holds; $\mathit{varcount}(t, \alpha)$ denotes the number of occurrences of the variable α in the term t. The first condition is also easy to check; the expression $\mathit{diff}(u, v)$ does not contain any variables any more.

If assignments $f \mapsto c_f$ and $\varepsilon > 0$ exist such that $\succ_\varepsilon$ and $\succeq$ fulfill all the requirements of Theorem 11 then we can actually calculate them automatically. For this we view the c_f's not any longer as fixed constants but treat them as variables. The expressions $D_i = \mathit{diff}(s_i, t_i)$ are then linear homogeneous polynomials in these variables. We restate our problem as a linear program:

$$\begin{array}{lrcl} \textbf{maximize} & D & = & D_1 + \cdots + D_N \\ \textbf{subject to} & D_1 & \geq & 0 \\ & D_2 & \geq & 0 \\ & & \vdots & \\ & D_N & \geq & 0 \ . \end{array}$$

It is understood that all variables of the linear program carry non-negativity constraints. See [18] and [19] for background information on linear programming.

THEOREM 12. *There exist assignments $f \mapsto c_f$ and $\varepsilon > 0$ such that $\succ_\varepsilon$ and $\succeq$ fulfill the conditions of Theorem 11 (modulo a possible reordering of the dependency pairs) iff*

1. *for all α and $i = 1, \ldots, N$,*

$$\mathit{varcount}(s_i, \alpha) \geq \mathit{varcount}(t_i, \alpha),$$

2. *the above real linear program is unbounded.*

In that case those dependency pairs d_i can be dropped from the dependency graph for which $\mathit{diff}(s_i, t_i) > 0$ holds.

Proof. The linear program is either feasible with D assuming a maximum $D_{\max} = 0$, or feasible with D being unbounded: it cannot be infeasible because the zero vector is a feasible solution, and it cannot be feasible and bounded with D assuming a maximum $\infty > D_{\max} > 0$

Table III. Built-In Termination Check (on a 1.7GHz Pentium IV Mobile)

	Bali	**Mat**	**Nom**
Runtime (in milli seconds)	2 ms	9 ms	12 ms
Average number of dependency pairs dropped	1.2	1	1.33

because then one could just multiply the solution vector with a constant $k > 1$ which would lead to an objective value $k\,D_{\max} > D_{\max}$.

Assume $D_{max} = 0$. Then also $D_i = 0$ for all feasible solutions and all $i = 1, \ldots, N$, so our method cannot be applied.

Assume $D_{max} = \infty$. Then there must be a feasible solution with $D_i > 0$ for at least one i. For this solution define

$$\varepsilon \;=\; \min\{D_i \,|\, D_i > 0,\, i = 1, \ldots, N\}.$$

Together with our previous explanations this concludes the proof. □

Note that the simplex method together with a non-cycling pivoting rule like Bland's rule is well-suited to deal with the above linear program. The dictionary can be obtained directly from our formulation by introducing N slack variables. The normally necessary first phase for discovering a feasible solution can be dropped because the zero vector is an obvious feasible solution. Confidence in the result can be obtained by calculating with exact fractions instead of floating point numbers.

Table III shows the runtimes of the above algorithm for the examples Bali, Mat and Nom. The average number of dropped dependency pairs can probably be improved by looking more carefully at the final dictionary; for our measurements we use Bland's rule and take the first discovered feasible ray on which the objective function is unbounded.

Important to note is that in all three examples no pivoting is necessary at all; the starting dictionary already gives away the feasible ray. That means that an even more easily implementable method could be used that works by simply counting variables and constants. For the purpose of counting constants we let $\mathit{constcount}\,(t, f)$ denote the number of occurrences of the (type) symbol f in the term t.

COROLLARY 5. *There exist assignments $f \mapsto c_f$ and $\varepsilon > 0$ such that $\succ_\varepsilon$ and $\succeq$ fulfill the conditions of Theorem 11 (modulo a possible reordering of the dependency pairs) if*

1. *for all α and $i = 1, \ldots, N$,*

$$\mathit{varcount}\,(s_i, \alpha) \;\geq\; \mathit{varcount}\,(t_i, \alpha),$$

2. *there are* $M \geq 1$ *(type) symbols* $f_1, \ldots, f_M$ *such that for all* $j = 1, \ldots, M$ *and* $i = 1, \ldots, N$,

$$\mathit{constcount}\,(s_i, f_j) \;\geq\; \mathit{constcount}\,(t_i, f_j),$$

3. *furthermore there exist indices* $1 \leq i_1, \ldots, i_M \leq N$ *such that for all* $j = 1, \ldots, M$,

$$\mathit{constcount}\,(s_{i_j}, f_j) \;>\; \mathit{constcount}\,(t_{i_j}, f_j).$$

In that case all dependency pairs d_i *can be dropped from the dependency graph for which there exists* j *with*

$$\mathit{constcount}\,(s_i, f_j) \;>\; \mathit{constcount}\,(t_i, f_j).$$

In particular, $d_{i_1}, \ldots, d_{i_M}$ *can be dropped.*

Proof. Observe that $\mathit{constcount}\,(t, f)$ is the coefficient of c_f in the linear polynomial $\Theta\,(t)$. Then it is clear that a feasible solution of the linear program with $D > 0$ is obtained by setting $c_{f_j} = 1$ for $j = 1, \ldots, M$ and $c_f = 0$ for all other f's. The statement then follows from Theorem 11. □

Basing the termination check on Corollary 5 already yields a powerful method that can deal with the examples Bali, Mat and Nom automatically. Note that the Tyrolean Termination Tool [13] cannot handle Bali in automatic mode but only in semi-automatic mode with the hint that polynomial interpretation should be used.

Nevertheless, Corollary 5 yields a strictly weaker method than Theorem 12. To understand why, look at the definitions

consts
$\mathtt{Dummy} :: \alpha \to \beta$
$\mathtt{A} :: \alpha$
defs
$\mathtt{A} :: \alpha\,\mathtt{list} \;\equiv\; \mathtt{Dummy}\,(\mathtt{A} :: \alpha\,\mathtt{vector})$
$\mathtt{A} :: \alpha\,\mathtt{vector}\,\mathtt{vector} \;\equiv\; \mathtt{Dummy}\,(\mathtt{A} :: \alpha\,\mathtt{list})$.

The dependency graph for these definitions has one cyclic component consisting of the two (root-cyclic) dependency pairs

$$\begin{aligned} d_1 &= \langle \tilde{\mathtt{A}}\,(\alpha\,\mathtt{list}),\, \tilde{\mathtt{A}}\,(\alpha\,\mathtt{vector})\rangle \;, \\ d_2 &= \langle \tilde{\mathtt{A}}\,(\alpha\,\mathtt{vector}\,\mathtt{vector}),\, \tilde{\mathtt{A}}\,(\alpha\,\mathtt{list})\rangle \;. \end{aligned}$$

The corresponding linear program reads

$$\begin{array}{lrcl} \textbf{maximize} & D &=& c_{\mathtt{vector}} \\ \textbf{subject to} & D_1 &=& c_{\mathtt{list}} - c_{\mathtt{vector}} \geq 0 \\ & D_2 &=& -c_{\mathtt{list}} + 2\,c_{\mathtt{vector}} \geq 0 \;. \end{array}$$

Corollary 5 does not help us here because both $c_{\texttt{list}}$ and $c_{\texttt{vector}}$ appear with negative coefficients in the constraints. But $c_{\texttt{list}} = 2$ and $c_{\texttt{vector}} = 1$ is a feasible solution of the linear program with $D = 1$ which means that $\{(2r, r) \mid r \geq 0\}$ is the feasible ray (on which the objective function is unbounded) we were looking for in order to apply Theorem 12! Evaluating $D_1 = 2-1 = 1 > 0$ and $D_2 = -2+2\cdot 1 = 0 \geq 0$ tells us that we can drop d_1. This leaves us with the cyclic component consisting only of d_2 and the linear program

$$\begin{array}{lrcl} \textbf{maximize} & D' & = & -c_{\texttt{list}} + 2\,c_{\texttt{vector}} \\ \textbf{subject to} & D'_1 & = & -c_{\texttt{list}} + 2\,c_{\texttt{vector}} \geq 0\ . \end{array}$$

Now Corollary 5 is applicable: $c_{\texttt{vector}}$ only appears with strictly positive coefficients in the constraints, therefore $c_{\texttt{vector}} = 1$ and $c_{\texttt{list}} = 0$ is the feasible solution we were looking for. Dropping also d_2 leaves us the empty graph, which proves termination according to Theorem 11. Note that picking $c_{\texttt{list}} = 3$ and $c_{\texttt{vector}} = 2$ in the first place would have saved us the second round; because of $D_1 = 3 - 2 = 1 > 0$ and $D_2 = -3 + 2 \cdot 2 = 1 > 0$ we could have dropped *both* dependency pairs d_1 and d_2 at once.

4. Conclusion

We have presented the first theory extension mechanism that can cope with overloading in higher-order logic as it is actually used in the proof-assistant Isabelle and shown that this mechanism is safe.

Checking if overloaded definitions play by the rules of our mechanism is not even semi-decidable. We have proven this by revealing the ties of our problem with the problem of checking the termination of a certain kind of term rewriting system.

Fortunately, most practical uses of overloaded definitions can be checked by a simple algorithm. This algorithm incorporates several ideas of recent research on termination of first-order term rewriting systems and has been implemented as an add-on [16] to the proof-assistant Isabelle. Furthermore this add-on is able to export the check in form of several TRSs to external termination provers as TTT or AProVE.

It is now possible to use overloading as just another tool when working in higher-order logic, without any doubts about its safety.

Acknowledgments

Special thanks to Markus Wenzel for pointing out to me the delicate points of overloading in connection with the HOL type system and in general, and to both Tobias Nipkow and Markus Wenzel for adding axiomatic type classes and overloading to the Isabelle proof assistant. Furthermore I thank Tjark Weber for several heated discussions at the opera and elsewhere concerning the dependency graph that enabled me to see the connection of overloading to the method of Arts and Giesl. Jürgen Giesl helped me understand several points about the dependency pair method. Stefan Berghofer referred me to [5] for background information about TRSs. Thanks to Alexander Krauss, Tjark Weber and Markus Wenzel for reading this paper and suggesting several important improvements and corrections, and also to Clemens Ballarin for reminding me not to skip too many proofs. Finally I would like to express my gratitude to Tobias Nipkow for giving me the time to work on this topic and for providing excellent working conditions.

References

1. Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, Vol. 5, No. 3, 1989, pages 363-397.
2. Markus Wenzel. Type Classes and Overloading in Higher-Order Logic. *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs '97*, LNCS 1275, Springer 1997, pages 307-322.
3. Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer 2002
4. Franz Baader, Tobias Nipkow. *Term Rewriting and All That,* Cambridge University Press 1998.
5. Terese. *Term Rewriting Systems*, Cambridge University Press 2003.
6. Thomas Arts, Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 2000, Vol. 236, pages 133-178.
7. Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Automated Deduction, 19th International Conference, CADE-19*, LNAI 2741, Springer 2003, pages 32-46.
8. Keijo Ruohonen. Reversible Machines and Post's Correspondence Problem for Biprefix Morphisms. *Journal of Information Processing and Cybernetics*, 1985, Vol. 21 (12), pages 579-595.
9. *The HOL System Description.* `http://hol.sourceforge.net/`
10. John Harrison. *The HOL Light theorem prover.* `http://www.cl.cam.ac.uk/~jrh/hol-light/`
11. Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *Journal of Automated Reasoning*, 2004, Vol. 33, No. 1, pages 29-49.
12. *AProVE - Automated Program Verification Environment.* `http://www-i2.informatik.rwth-aachen.de/AProVE/`
13. *TTT - Tyrolean Termination Tool.* `http://cl2-informatik.uibk.ac.at/ttt/`

14. Steven Obua. Proving Bounds for Real Linear Programs in Isabelle/HOL. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs '05*, LNCS 3603, Springer 2005, pages 227-244.
15. Christian Urban. Nominal Techniques in Isabelle/HOL. *Automated Deduction, 20th International Conference, CADE-20*, LNAI 3632, Springer 2005, pages 38-53.
16. Steven Obua. *How To Check Overloaded Definitions in Isabelle.* `http://www4.in.tum.de/~obua/checkdefs`
17. Project Bali. `http://isabelle.in.tum.de/Bali`
18. Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*, 2nd ed., Springer 2001.
19. Alexander Schrijver. *Theory of Linear and Integer Programming*, Wiley 1986.
20. Jürgen Giesl. Generating Polynomial Orderings for Termination Proofs. *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, Springer 1995, pages 426-431.
21. Jim Christian. Flatterms, Discrimination Nets, and Fast Term Rewriting. *Journal of Automated Reasoning*, 1993, Vol. 10, No. 1, pages 95-113.
22. Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott. *Artificial Intelligence Programming.*, Lawrence Erlbaum Associates 1980.