Termination of Isabelle Functions via Termination of Rewriting*

Alexander Krauss,[1] Christian Sternagel,[2] René Thiemann,[2] Carsten Fuhs,[3] and Jürgen Giesl[3]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Institute of Computer Science, University of Innsbruck, Austria
[3] LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. We show how to automate termination proofs for recursive functions in (a first-order subset of) Isabelle/HOL by encoding them as term rewrite systems and invoking an external termination prover. Our link to the external prover includes full proof reconstruction, where all necessary properties are derived inside Isabelle/HOL without oracles. Apart from the certification of the imported proof, the main challenge is the formal reduction of the proof obligation produced by Isabelle/HOL to the termination of the corresponding term rewrite system. We automate this reduction via suitable tactics which we added to the IsaFoR library.

1 Introduction

In a proof assistant based on higher-order logic (HOL), such as Isabelle/HOL [15], recursive function definitions typically require a termination proof. To release the user from finding suitable termination arguments manually, it is desirable to automate these termination proofs as much as possible.

There have already been successful approaches to port and adapt existing termination techniques from term rewriting and other areas to Isabelle [5,12]. They indeed increase the degree of automation for termination proofs of HOL functions. However, these approaches do not cover all powerful techniques that have been developed in term rewriting, e.g., [7,20]. These techniques are implemented in a number of termination tools (e.g., AProVE [9], T$_{\text{T}}$T$_2$ [11] and many others) that can show termination of (first-order) term rewrite systems (TRSs) automatically. (In the remainder we use 'termination tool' exclusively to refer to such fully automatic and external provers.) Instead of porting further proof techniques to Isabelle, we prefer to use the existing termination tools, giving direct access to an abundance of methods and their efficient implementations.

Using termination tools inside proof assistants has been an open problem for some time and is often mentioned as future work when discussing certification of termination proofs [3,6]. However, this requires more than a communication interface between two programs. In LCF-style proof assistants [10] such as Isabelle, all proofs must be checked by a small trusted kernel. Thus, integrating external tools as unverified oracles is unsatisfactory: any error in the external tool or in

* Supported by the DFG grant GI 274/5-3 and the FWF project P22767-N13.

the integration code would compromise the overall soundness. Instead, the external tool must provide a certificate that can be checked by the proof assistant.

Our approach involves the following steps.

1. Generate the definition of a TRS $\mathcal{R}^f$ which corresponds to the function f.
2. Prove that termination of $\mathcal{R}^f$ indeed implies the termination goal for f.
3. Run the termination tool on $\mathcal{R}^f$ and obtain a certificate.
4. Replay the certificate using a formally verified checker.

While steps 1 and 3 are not hard, and the ground work for step 4 is already available in the IsaFoR library [17,19], which formalizes term rewriting and several termination techniques,[1] this paper is concerned with the missing piece, the reduction of termination proof obligations for HOL functions to the termination of a TRS. This is non-trivial, as the languages differ considerably. Termination of a TRS expresses the well-foundedness of a relation over terms, i.e., of type $(term \times term)\ set$, where *term*s are first-order terms. In contrast, the termination proof obligation for a HOL function states the well-foundedness of its call relation, which has the type $(\alpha \times \alpha)\ set$, where α is the argument type of the function. In essence, we must move from a shallow embedding (the functional programming fragment of Isabelle/HOL) to a deep embedding (the formalization of term rewriting in IsaFoR).

The goal of this paper is to provide this formal relationship between termination of first-order HOL functions and termination of TRSs. More precisely, we develop a tactic that automatically reduces the termination proof obligation of a HOL function to the termination problem of a TRS. This allows us to use arbitrary termination tools for fully automated termination proofs inside Isabelle. Thus, powerful termination tools become available to the Isabelle user, while retaining the strong soundness guarantees of an LCF-style proof assistant. Since our approach is generic, it automatically benefits from future improvements to termination tools and the termination techniques within IsaFoR. Our implementation is available as part of IsaFoR.

Outline of this paper. We give a short introduction on term rewriting, HOL and HOL functions in §2. Then we show our main result in §3 on how to systematically discharge the termination proof obligation of a HOL function via proving termination of a TRS. In §4 we present some examples which show the strengths and limitations of our technique. How to extend our approach to support more HOL functions is discussed in §5. We conclude in §6.

2 Preliminaries

2.1 Higher-Order Logic

We consider classical HOL, which is based on simply-typed lambda-calculus, enriched with a simple form of ML-like polymorphism. Among its basic types are a type *bool* of truth values and a function space type constructor $\Rightarrow$ (where $\alpha \Rightarrow \beta$ denotes the type of total functions mapping values of type α to values of

[1] See http://cl-informatik.uibk.ac.at/software/ceta for a list of supported techniques.

type β). Sets are modeled by a type α *set*, which just abbreviates $\alpha \Rightarrow$ *bool*.

By an add-on tool, HOL supports algebraic datatypes, which includes the types *nat* (with constructors 0 and *Suc*) and *list* (with constructors [] and #).

Another add-on tool, the *function package* [13], completes the functional programming layer by allowing recursive function definitions, which are not covered by the primitives of the logic. Since it internally employs a well-founded recursion principle, it requires the user to prove well-foundedness of a certain relation, extracted automatically from the function definition (cf. §2.3). This proof obligation, by its construction, directly corresponds to the termination of the function being defined. It is the proof of this goal that we want to automate.

As opposed to functional programming languages, there is no operational semantics for HOL; the meaning of its expressions is instead given by a set-theoretic denotational semantics. As a consequence, there is no direct notion of evaluation or termination of an expression. Thus, when we informally say that we prove "termination of a HOL function," this simply means that we discharge the proof obligation produced by the function package.

2.2 Supported Fragment

Isabelle supports a wide spectrum of specifications, using various forms of inductive, coinductive and recursive definitions, as well as quantifiers and Hilbert's choice operator. Clearly, not all of them can be easily expressed using TRSs. Thus, we must limit ourselves to a subset which is sufficiently close to rewriting, and consider only algebraic datatypes, given by a set of constructors together with their types, and recursive functions, given by their defining equations with pattern matching. Additionally, we impose the following restrictions:

1. Functions and constructors must be first-order (no functions as arguments).
2. Patterns are constructor terms and must be linear and non-overlapping.
3. Patterns must be complete.
4. Expressions consist of variables, function applications, and case-expressions only. In particular, partial applications and λ-abstractions are excluded.

Linearity is always satisfied by function definitions that are accepted by Isabelle's function package, and pattern overlaps are eliminated automatically. For ease of presentation, we assume that there is no mutual recursion (f calls g and g calls f) and no nested recursion (arguments of a recursive call contain other recursive calls; they may of course contain calls to other defined functions).

Most of the above restrictions are not fundamental, and we discuss in §5 how some of them can be removed. Our chosen fragment of HOL rather represents a compromise between expressive power and a reasonably simple presentation and implementation of our reduction technique. Note that case-expressions encompass the simpler if-expressions, which can be seen as case-expressions on type *bool*. Isabelle's (non-recursive and monomorphic) let-expressions can simply be inlined or replaced by case-expressions if patterns are involved.

The functions *half* and *log* below (*log* computes the logarithm) illustrate our supported fragment and will be used as running examples throughout this paper.

$half\ 0 = 0$
$half\ (Suc\ 0) = 0$
$half\ (Suc\ (Suc\ n)) = Suc\ (half\ n)$
$log\ n = (case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m)))$

2.3 Function Definitions by Well-Founded Recursion

When the user writes a recursive definition, the function package analyzes the equations and extracts the recursive calls. This information is then used to synthesize the termination proof obligation.

Formally, we define the operation CALLS_f that computes the set of calls to f inside an expression, each together with a condition under which it occurs.

- $\text{CALLS}_f(g\ e_1\ \ldots\ e_k) \equiv \text{CALLS}_f(e_1) \cup \ldots \cup \text{CALLS}_f(e_k)$ if g is a constructor or a defined function other than f,
- $\text{CALLS}_f(f\ e_1\ \ldots\ e_n) \equiv \text{CALLS}_f(e_1) \cup \ldots \cup \text{CALLS}_f(e_n) \cup \{(e_1, \ldots, e_n, \mathit{True})\}$,
- $\text{CALLS}_f(x) \equiv \varnothing$ for all variables x, and
- $\text{CALLS}_f(case\ e\ of\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \equiv \text{CALLS}_f(e) \cup (\text{CALLS}_f(e_1) \wedge e = p_1) \cup \ldots \cup (\text{CALLS}_f(e_k) \wedge e = p_k)$ where $\text{CALLS}_f(e_i) \wedge e = p_i$ is like $\text{CALLS}_f(e_i)$, but every $(t_1, \ldots, t_m, \varphi) \in \text{CALLS}_f(e_i)$ is replaced by $(t_1, \ldots, t_m, \varphi \wedge e = p_i)$.

The termination proof obligation requires us to exhibit a strongly normalizing relation $\succ$ such that for each defining equation $f\ p_1\ \ldots\ p_n = e$ and each $(r_1, \ldots, r_n, \phi) \in \text{CALLS}_f(e)$ we can prove $\phi \implies (p_1, \ldots, p_n) \succ (r_1, \ldots, r_n)$.

Consider for example the definition of *half*, where we have $\text{CALLS}_{half}(0) \equiv \varnothing$ and $\text{CALLS}_{half}(Suc\ (half\ n)) \equiv \{(n, \mathit{True})\}$. We obtain the following obligation.

1. $SN\ ?R$
2. $\forall n.\ (Suc\ (Suc\ n),\ n) \in ?R$

The variable $?R :: (nat \times nat)\ set$ is a *schematic variable*, which can be instantiated during the proof, i.e., it can be seen as existentially quantified.

For *log*, we have $\text{CALLS}_{log}(case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m))) \equiv \{(Suc\ m,\ half\ n = Suc\ m)\}$, and the following proof obligation is produced.

1. $SN\ ?R$
2. $\forall n\ m.\ half\ n = Suc\ m \implies (n,\ Suc\ m) \in ?R$

Two things should be noted here. First, the fact that the recursive call is guarded by a case-expression is reflected by a condition in the corresponding subgoal. Without this condition, which models the usual evaluation behavior of *case*, the goal would be unprovable. Second, the goal may refer to previously defined functions. To prove it, we must refer to properties of these functions, either through their definitions, or through other lemmas about them.

When the proof obligation is discharged, the function package can use the result to derive the recursive equations as theorems (previously, they were just conjectures—consider the recursive equation $f\ x = Suc\ (f\ x)$, which is inconsistent). Additionally, an induction rule is provided, which expresses "induction along the computation." The induction rules for *half* and *log* are shown below.

$$P\ 0 \Longrightarrow P\ (Suc\ 0) \Longrightarrow (\forall n.\ P\ n \Longrightarrow P\ (Suc\ (Suc\ n))) \Longrightarrow \forall n.\ P\ n$$
$$(\forall n.\ (\forall m.\ half\ n = Suc\ m \Longrightarrow P\ (Suc\ m)) \Longrightarrow P\ n) \Longrightarrow \forall n.\ P\ n$$

2.4 IsaFoR - Term Rewriting Formalized in Isabelle/HOL

In the following, we assume that the reader is familiar with the basics of term rewriting [1]. Many notions and facts from rewriting have been formalized in the Isabelle library IsaFoR [19]. Before we can give the reduction from termination of HOL functions to termination of corresponding TRSs in §3, we need some more details on IsaFoR. Terms are represented straightforwardly by the datatype:

datatype $(\alpha, \beta)\ term = Var\ \beta \mid Fun\ \alpha\ ((\alpha, \beta)\ term\ list)$

The type variables α and β, which represent function and variable symbols, respectively, are always instantiated with the type *string* in our setting. Hence, we abbreviate $(string, string)\ term$ by $term$ in the following. For example, the term $\mathsf{f}(\mathsf{x},\mathsf{y})$ is represented by Fun "f" [Var "x", Var "y"]. A TRS is represented by a value of type $(term \times term)\ set$.

The semantics of a TRS is given by its rewrite relation $\to_{\mathcal{R}}$, defined by closing $\mathcal{R}$ under contexts and substitutions. Termination of $\mathcal{R}$ is formalized as $SN\ (\to_{\mathcal{R}})$.

IsaFoR formalizes many criteria commonly used in automated termination proofs. Ultimately, it contains an executable and terminating function

$$check\text{-}proof :: (term \times term)\ list \Rightarrow proof \Rightarrow bool$$

and a proof of the following soundness theorem:

Theorem 1 (Soundness of Check). $check\text{-}proof\ \mathcal{R}\ prf \Longrightarrow SN\ (\to_{\mathcal{R}})$

Here, *prf* is a certificate (i.e., a termination proof of $\mathcal{R}$) from some external source, encoded as a value of a suitable datatype, and $\mathcal{R}$ is the TRS under consideration.[2] Whenever *check-proof* returns *True* for some given TRS $\mathcal{R}$ and certificate *prf*, we have established (inside Isabelle) that *prf* is a valid termination proof for $\mathcal{R}$. Thus, we can prove termination of concrete TRSs inside Isabelle.

The technical details on the supported termination techniques and the structure of the certificate (i.e., the type *proof*) are orthogonal to our use of the check function, which only relies on Theorem 1.

2.5 Terminology and Notation

The layered nature of our setting requires that we carefully distinguish three levels of discourse. Primarily, there is higher-order logic (implemented in Isabelle/HOL), in which all mechanized reasoning takes place. The termination goals we ultimately want to solve are formulated on this level. Of course, the syntax of HOL consists of terms, but to distinguish them from the embedded term language of term rewriting, we refer to them as *expressions*. They are uniformly written in *italics* and follow the conventions of the lambda-calculus (in particular, function application is denoted by juxtaposition). HOL equality is

[2] To be executable, *check-proof* demands that $\mathcal{R}$ is given as a list of rules and not as a set. We ignore this difference, since it is irrelevant for this paper.

denoted by $=$. For example, the definition of *half* above is a HOL expression.

The second level is the "sub-language" of first-order terms, which is deeply embedded into HOL by the datatype *term*. When we speak of a *term*, we always refer to a value of that type, not an arbitrary HOL expression. While this embedding is simple and adequate, the concrete syntax with the *Fun* and *Var* constructors and string literals is rather unwieldy. Hence, for readability, we use sans-serif font to abbreviate the constructors and the quotes: Instead of *Var* "v" we write v, and instead of *Fun* "f" $[\ldots]$ we write $\mathsf{f}(\ldots)$, omitting the parentheses $()$ for nullary functions. This recovers the well-known concrete syntax of term rewriting, but we must keep in mind that the constructors and strings are still present, although they are not written as such.

Finally, we must relate the two languages with each other, and describe the proof procedures that derive the relevant properties. While the properties themselves can be stated in HOL for each concrete instance, the general schema cannot, as it must talk about "all HOL expressions." Thus, we use a meta-language as another level above HOL, in which we express the transformations and tactics. This level corresponds to our implementation (in ML). Functions of the meta-language are written in SMALL CAPITALS (e.g., $\textsc{calls}_f$), and variables of the meta-language, which typically range over arbitrary HOL expressions or patterns, are written e or p, possibly with annotations. For HOL expressions that are arguments of recursive calls we also use r. Equality of the meta-language is written $\equiv$ and denotes syntactic equality of HOL expressions. In particular, $e \equiv e'$ implies $e = e'$, since HOL's equality is reflexive.

Both embeddings are deep, that is, each level can talk about the syntax of the lower levels. As a simple example, the concept of a ground term can be defined as a recursive HOL function $\mathit{ground} :: \mathit{term} \Rightarrow \mathit{bool}$:

$$\mathit{ground}\ (\mathit{Var}\ x) = \mathit{False}$$
$$\mathit{ground}\ (\mathit{Fun}\ f\ ts) = (\forall t{\in}\mathit{set}(ts).\ \mathit{ground}\ t)$$

Then we can immediately deduce that $\mathit{ground}\,(\mathsf{f}(\mathsf{x})) = \mathit{False}$, due to the presence of x. Note however that the similar-looking statement $\mathit{ground}\,(\mathsf{f}(x)) = \mathit{False}$ is not uniformly true. More precisely, its universal closure $\forall x.\ \mathit{ground}\,(\mathsf{f}(x)) = \mathit{False}$ does not hold, since we could instantiate x with the term c (i.e., *Fun* "c" $[]$). Thus, we must not confuse variables of the different levels. Obviously, we cannot quantify over a variable x, which is just the *Var* constructor applied to a string.

Similarly, the meta-language can talk about the syntax of HOL, as in the definition of $\textsc{calls}_f$, which is recursive over the structure of HOL expressions.

3 The Reduction to Rewriting

3.1 Encoding Expressions and Defining Equations

We define a straightforward encoding of HOL expressions as terms, denoted by the meta-level operation ENC. For case-free expressions, we turn variables into term variables and (curried) applications into applications on the term level:

$$\begin{aligned}\textsc{enc}(x) &\equiv \mathsf{x}\\ \textsc{enc}(f\ e_1\ \ldots\ e_n) &\equiv \mathsf{f}(\textsc{enc}(e_1),\ldots,\textsc{enc}(e_n))\end{aligned}$$

Each case-expression is replaced by a new function symbol, for which we will include additional rules below. To simplify bookkeeping, we assume that each occurrence of a case-expression is annotated with a unique integer j.

$$\begin{aligned}&\textsc{enc}(\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k)\\ &\quad\equiv \mathsf{case_j}(\textsc{enc}(e),\textsc{enc}(y_1),\ldots,\textsc{enc}(y_m))\end{aligned}$$

where $y_1,\ldots,y_m$ are all variables that occur free in some e_i but not in p_i.

The operation RULES yields the rewrite rules for a function or case-expression. For a function f with defining equations $\ell_1 = r_1,\ \ldots,\ \ell_k = r_k$, they are

$$\textsc{rules}(f) \equiv \{\ \textsc{enc}(\ell_1) \to \textsc{enc}(r_1),\ \ldots,\ \textsc{enc}(\ell_k) \to \textsc{enc}(r_k)\ \}\,.$$

For the case-expression $\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k$ we have

$$\begin{aligned}\textsc{rules}(\mathit{case}_j) \equiv \{\ &\mathsf{case_j}(\textsc{enc}(p_1),\textsc{enc}(y_1),\ldots,\textsc{enc}(y_m)) \to \textsc{enc}(e_1),\\ &\ldots,\\ &\mathsf{case_j}(\textsc{enc}(p_k)),\textsc{enc}(y_1),\ldots,\textsc{enc}(y_m)) \to \textsc{enc}(e_k)\ \}\,.\end{aligned}$$

We define the TRS for f as $\mathcal{R}^f = \textsc{rules}(f) \cup \bigcup_{g\in\mathcal{S}_f} \textsc{rules}(g)$ where $\mathcal{S}_f$ is the set of all functions that are used (directly or indirectly) by f. Our encoding is similar to the well known technique of unraveling which transforms conditional into unconditional TRSs [14,16].[3]

For example, $\mathcal{R}^{log}$ is defined as follows and completely contains $\mathcal{R}^{half}$.

$$\begin{aligned}\mathsf{half}(\mathsf{0}) &\to \mathsf{0} & \mathsf{log}(\mathsf{n}) &\to \mathsf{case_0}(\mathsf{half}(\mathsf{n}))\\ \mathsf{half}(\mathsf{Suc}(\mathsf{0})) &\to \mathsf{0} & \mathsf{case_0}(\mathsf{0}) &\to \mathsf{0}\\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(\mathsf{n}))) &\to \mathsf{Suc}(\mathsf{half}(\mathsf{n})) & \mathsf{case_0}(\mathsf{Suc}(\mathsf{m})) &\to \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(\mathsf{m})))\end{aligned}$$

3.2 Embedding Functions

At this point, we have defined a translation, but we cannot reason about it in Isabelle, since ENC is only an extra-logical concept, defined on the meta-level. In fact, it is easy to see that it cannot be defined in HOL: If we had a HOL function *enc* satisfying $\mathit{enc}\ 0 = \mathsf{0}$ and $\mathit{enc}\ (\mathit{half}\ 0) = \mathsf{half}(\mathsf{0})$, we would immediately have a contradiction, since $\mathit{half}\ 0 = 0$, and $\mathsf{half}(\mathsf{0}) \neq \mathsf{0}$, but a function must always yield the same result on the same input.

In a typical reflection scenario, we would proceed by defining an interpretation for *term*. For example, if we were modeling the syntax of integer arithmetic expressions, then we could define a function $\mathit{eval} :: \mathit{term} \Rightarrow \mathit{int}$ (possibly also depending on a variable assignment) which interprets terms as integers. However,

[3] It would be possible to directly generate dependency pair problems. However, techniques like [18] and several termination tools rely on the notion of "minimal chains," which is not ensured by our approach.

in our setting, the result type of such a function is not fixed, as our terms represent HOL expressions of arbitrary types. Thus, the result type of *eval* would depend on the actual term it is applied to. This cannot be expressed in a logic without dependent types, which means we cannot use this approach here.

Instead, we take the opposite route: For all relevant types σ, we define a function $emb_\sigma :: \sigma \Rightarrow term$, mapping values of type σ to their canonical term representation.

Using Isabelle's type classes, we use a single overloaded function *emb*, which belongs to a type class *embeddable*. Concrete datatypes can be declared to be instances of this class by defining *emb*, usually by structural recursion w.r.t. the datatype. For example, here are the definitions for the types *nat* and *list*:

$$\begin{array}{ll} emb\ 0 = \mathsf{0} & emb\ [\,] = \mathsf{Nil} \\ emb\ (Suc\ n) = \mathsf{Suc}(emb\ n) & emb\ (x\ \#\ xs) = \mathsf{Cons}(emb\ x,\ emb\ xs) \end{array}$$

This form of definition is canonical for all algebraic datatypes, and suitable definitions of *emb* can be automatically generated for all user-defined datatypes, turning them into instances of the class *embeddable*. This is analogous to the instances generated automatically by Haskell's "deriving" statement. It is also possible to manually provide the definition of *emb* for other types if they behave like datatypes like the predefined type *bool* for the Booleans.

Note that by construction, the result of *emb* is always a constructor ground term. For a HOL expression e that consists only of datatype constructors, (e.g., $Suc\ (Suc\ 0)$), we have $emb\ e = \textsc{enc}(e)$. For other expressions this is not the case, e.g., $emb\ (half\ 0) = emb\ 0 = \mathsf{0}$, but $\textsc{enc}(half\ 0) \equiv \mathsf{half}(\mathsf{0})$.

To formulate our proofs, we need another encoding of expressions as terms: The operation GENC is a slight variant of ENC, which treats variables differently, mapping them to their embeddings instead of term variables.

$$\begin{aligned} \textsc{genc}(x) &\equiv emb\ x \\ \textsc{genc}(f\ e_1\ \ldots\ e_n) &\equiv \mathsf{f}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_n)) \\ \textsc{genc}(case_j\ e\ of\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k) & \\ \equiv \mathsf{case_j}(\textsc{genc}(e), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \end{aligned}$$

where $y_1, \ldots, y_m$ are all variables that occur free in some e_i but not in p_i.

Hence, $\textsc{genc}(e)$ never contains term variables. However, it contains the same HOL variables as e. For example, $\textsc{genc}(half\ (Suc\ n)) \equiv \mathsf{half}(\mathsf{Suc}(emb\ n))$.

3.3 Rewrite Lemmas

The definitions of $\mathcal{R}^{half}$ and $\mathcal{R}^{log}$ above are straightforward, but reasoning with them is clumsy and low-level: To establish a single rewrite step, we must extract the correct rule (that is, prove that it is in the set $\mathcal{R}^{half}$ or $\mathcal{R}^{log}$), invoke closure under substitution, and construct the correct substitution explicitly as a function of type $string \Rightarrow term$.

To avoid such repetitive reasoning, we automatically derive an individual lemma for each rewrite rule. From the definition of $\mathcal{R}^{half}$, we obtain the following rules, which we call *rewrite lemmas*:

$\mathsf{half}(\mathsf{0}) \to_{\mathcal{R}^{half}} \mathsf{0}$ $\qquad$ $\mathsf{half}(\mathsf{Suc}(\mathsf{0})) \to_{\mathcal{R}^{half}} \mathsf{0}$
$\forall t.\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(t))) \to_{\mathcal{R}^{half}} \mathsf{Suc}(\mathsf{half}(t))$

Note that the term variable n in the last rule has been turned into a universally-quantified HOL variable by applying the "generic substitution" $\{\mathsf{n} \mapsto t\}$. The advantage of this format is that applying a rewrite rule merely involves instantiating a universal quantifier, for which we can use the matching facilities of Isabelle. In particular, we can instantiate t with *emb* n, which in general results in a rewrite lemma of the form $\textsc{genc}(f\ p_1\ \ldots\ p_n) \to_{\mathcal{R}} \textsc{genc}(e)$ for a defining equation $f\ p_1\ \ldots\ p_n = e$.

3.4 The Simulation Property

The following property connects our generated TRSs with HOL expressions.

Definition 2 (Simulation Property). *For every expression e and $\mathcal{R} = \bigcup\{\mathcal{R}^f \mid f$ occurs in $e\}$, the* simulation property for e *is the statement*

$$\textsc{genc}(e) \to^*_{\mathcal{R}} emb\ e.$$

As we cannot quantify over all HOL expressions within HOL itself, we cannot formalize that the simulation property holds for all e.

However, we will devise a tactic that derives this property for any given concrete expression. The basic building blocks of such proofs are lemmas of the following form, which are derived for each function symbol and can be composed to show the simulation property for a given expression.

Definition 3 (Simulation Lemma). *The* simulation lemma *for a function f of arity n is the statement*

$$\forall x_1 \ldots x_n.\ \mathsf{f}(emb\ x_1, \ldots, emb\ x_n) \to^*_{\mathcal{R}^f} emb\ (f\ x_1\ \ldots\ x_n)\ .$$

E.g., the simulation lemma for *half* is $\forall n.\ \mathsf{half}(emb\ n) \to^*_{\mathcal{R}^{half}} emb\ (half\ n)$.

The lemma claims that the rules that we produced for f can indeed be used to reduce a function application to the (embedding of) the value of the function. Of course, this way of saying "$\mathcal{R}^f$ computes f" admits the possibility that there are other $\mathcal{R}^f$-reductions that lead to different normal forms or that do not terminate, since we are not requiring confluence or strong normalization. But this form of simulation lemma is sufficient for our purpose.

We show in §3.6 how simulation lemmas are proved automatically.

3.5 Reduction of Termination Goals

After having proved termination of $\mathcal{R}^f$ using a termination tool in combination with IsaFoR and Theorem 1, we now show how to use this result to solve the termination goal for the HOL function f. Recall from §2.3 that we must exhibit a strongly normalizing relation $\succ$ such that $\phi \implies (p_1, \ldots, p_n) \succ (r_1, \ldots, r_n)$ for all $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e)$ for each defining equation $f\ p_1\ \ldots\ p_n = e$.

To this end, we first define $\rightsquigarrow$ as $\rightarrow_{\mathcal{R}^f} \cup \rhd$ where $\rhd$ is the strict subterm relation. The addition of $\rhd$ is required to strip off constructors and non-recursive function applications that are wrapped around recursive calls in right-hand sides of $\mathcal{R}^f$. Since $\rightarrow_{\mathcal{R}^f}$ is strongly normalizing and closed under contexts, also $\rightsquigarrow$ is strongly normalizing. This allows us to finally choose $\succ$ as the following relation.

$$(x_1, \ldots, x_n) \succ (y_1, \ldots, y_n) \text{ iff } \mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\ y_1, \ldots, \mathit{emb}\ y_n)$$

It remains to show that the arguments of recursive calls decrease w.r.t. $\succ$. That is, for each recursive call we have a goal of the form

$$\phi \implies \mathsf{f}(\mathit{emb}\ p_1, \ldots, \mathit{emb}\ p_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n)$$

where $f\ p_1\ \ldots\ p_n = e$ is a defining equation of f and $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e)$. In the following, we illustrate the automated proof of this goal.

Note that since the p_i's are patterns, we have $\mathit{emb}\ p_i = \textsc{genc}(p_i)$, and hence

$$\begin{aligned}
& \mathsf{f}(\mathit{emb}\ p_1, \ldots, \mathit{emb}\ p_n) && \\
=\ & \mathsf{f}(\textsc{genc}(p_1), \ldots, \textsc{genc}(p_n)) && (p_i \text{ are patterns}) \\
\equiv\ & \textsc{genc}(f\ p_1\ \ldots\ p_n) && (\text{definition of } \textsc{genc}) \\
\rightarrow_{\mathcal{R}^f}\ & \textsc{genc}(e) && (\text{rewrite lemma})
\end{aligned}$$

Thus, it remains to construct a sequence $\textsc{genc}(e) \rightsquigarrow^* \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n)$, which reduces the right-hand side of the definition to a particular recursive call, eliminating any surrounding context. We proceed recursively over e.

- If $e \equiv g\ e_1\ \ldots\ e_m$ for a constructor g or a defined function symbol $g \not\equiv f$, then $(r_1, \ldots, r_n, \phi) \in \textsc{calls}(e_i)$ for some particular i. Hence, we have

$$\begin{aligned}
& \textsc{genc}(e) && \\
\equiv\ & \mathsf{g}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_m)) && (\text{definition of } \textsc{genc}) \\
\rhd\ & \textsc{genc}(e_i) && (\text{definition of } \rhd) \\
\rightsquigarrow^*\ & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) && (\text{apply tactic recursively})
\end{aligned}$$

- If $e \equiv f\ e_1\ \ldots\ e_n$ then (since we excluded nested recursion) we have $e_i = r_i$ for all i. Hence, we have

$$\begin{aligned}
& \textsc{genc}(e) && \\
\equiv\ & \mathsf{f}(\textsc{genc}(r_1), \ldots, \textsc{genc}(r_n)) && (\text{definition of } \textsc{genc}) \\
\rightarrow^*_{\mathcal{R}^f}\ & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) && (\text{simulation property})
\end{aligned}$$

- If $e \equiv\ \mathit{case}_j\ e_0\ \mathit{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ then we distinguish where the recursive call is located. If $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e_0)$, then we have

$$\begin{aligned}
& \textsc{genc}(e) && \\
\equiv\ & \mathsf{case}_\mathsf{j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) && (\text{definition of } \textsc{genc}) \\
\rhd\ & \textsc{genc}(e_0) && (\text{definition of } \rhd) \\
\rightsquigarrow^*\ & \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n) && (\text{apply tactic recursively})
\end{aligned}$$

  Otherwise, $\phi \equiv (\chi \wedge e_0 = p_i)$ for some χ and $1 \leqslant i \leqslant k$, and $(r_1, \ldots, r_n, \chi) \in \textsc{calls}(e_i)$. We may therefore use the assumption $e_0 = p_i$ and proceed with

$$
\begin{array}{rll}
 & \textsc{genc}(e) & \\
\equiv & \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(definition of } \textsc{genc}) \\
\to^*_{\mathcal{R}^f} & \mathsf{case_j}(emb\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(simulation property)} \\
= & \mathsf{case_j}(emb\ p_i, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(assumption } e_0 = p_i) \\
= & \mathsf{case_j}(\textsc{genc}(p_i), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(since } p_i \text{ is a pattern)} \\
\to_{\mathcal{R}^f} & \textsc{genc}(e_i) & \text{(rewrite lemma)} \\
\rightsquigarrow^* & \mathsf{f}(emb\ r_1, \ldots, emb\ r_n) & \text{(apply tactic recursively)}
\end{array}
$$

3.6 Proof of the Simulation Property

We have seen that for the reduction of termination goals it is essential to use the simulation property $\textsc{genc}(e) \to^*_{\mathcal{R}^f} emb\ e$ for expressions e that occur below recursive calls or within conditions that guard a recursive call. Below, we show how this property is derived for an individual expression, assuming that we already have simulation lemmas for all functions that occur in it. We again proceed recursively over e.

- If e is a HOL variable x then $\textsc{genc}(e) \equiv \textsc{genc}(x) \equiv emb\ x \equiv emb\ e$ and thus, the result follows by reflexivity of $\to^*_{\mathcal{R}^f}$.
- If $e \equiv g\ e_1\ \ldots\ e_k$ for a function symbol g then

$$
\begin{array}{rll}
 & \textsc{genc}(e) & \\
\equiv & \mathsf{g}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_k)) & \text{(definition of } \textsc{genc}) \\
\to^*_{\mathcal{R}^f} & \mathsf{g}(emb\ e_1, \ldots, emb\ e_k) & \text{(apply tactic recursively)} \\
\to^*_{\mathcal{R}^f} & emb\ (g\ e_1\ \ldots\ e_k) & \text{(simulation lemma for } g) \\
\equiv & emb\ e &
\end{array}
$$

- If $e \equiv case_j\ e_0\ of\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ then we construct the following rewrite sequence:

$$
\begin{array}{rll}
 & \textsc{genc}(e) & \\
\equiv & \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(definition of } \textsc{genc}) \\
\to^*_{\mathcal{R}^f} & \mathsf{case_j}(emb\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(apply tactic recursively)}
\end{array}
$$

  Now we apply a case analysis on e_0, which must be equal (in HOL, not syntactically) to one of the patterns. In each particular case we may assume $e_0 = p_i$. Then we continue:

$$
\begin{array}{rll}
 & \mathsf{case_j}(emb\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \\
= & \mathsf{case_j}(emb\ p_i, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(assumption } e_0 = p_i) \\
= & \mathsf{case_j}(\textsc{genc}(p_i), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \text{(since } p_i \text{ is a pattern)} \\
\to_{\mathcal{R}^f} & \textsc{genc}(e_i) & \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^f} & emb\ e_i & \text{(apply tactic recursively)} \\
= & emb\ e & \text{(assumption } e_0 = p_i)
\end{array}
$$

The tactic above assumes that simulation lemmas for all functions in e are already present. Note the simulation lemma is trivial to prove if f is a constructor, since $\mathsf{f}(emb\ x_1, \ldots, emb\ x_n) = emb\ (f\ x_1\ \ldots\ x_n)$ by definition of *emb*.

For defined symbols of non-recursive functions the simulation lemmas are derived by unfolding the definition of the function and applying the tactic above. Thus, simulation lemmas are proved bottom-up in the order of function dependencies. When a function is recursive, the proof of its simulation lemma proceeds by induction using the induction principle from the function definition.

Example 4. We show how the simulation lemma for *log* is proved, assuming that the simulation lemmas for 0, *Suc*, and *half* are already available.

So our goal is to show $\mathsf{log}(emb\ n) \to^*_{\mathcal{R}^{log}} emb\ (log\ n)$ for any $n :: nat$. We apply the induction rule of *log* and obtain the following induction hypothesis.

$$\forall m.\ half\ n = Suc\ m \implies \mathsf{log}(emb\ (Suc\ m)) \to^*_{\mathcal{R}^{log}} emb\ (log\ (Suc\ m))$$

Let c abbreviate *case half n of* $0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m))$. Then

$$\begin{array}{rll}
 & \mathsf{log}(emb\ n) & \\
\to_{\mathcal{R}^{log}} & \mathsf{case_0}(\mathsf{half}(emb\ n)) & \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{case_0}(emb\ (half\ n)) & \text{(simulation lemma of } half\text{)}
\end{array}$$

We continue by case analysis on *half n*. We only present the more interesting case *half* $n = Suc\ m$ (the other case *half* $n = 0$ is similar):

$$\begin{array}{rll}
 & \mathsf{case_0}(emb\ (half\ n)) & \\
= & \mathsf{case_0}(emb\ (Suc\ m)) & \text{(assumption } half\ n = Suc\ m\text{)} \\
= & \mathsf{case_0}(\mathsf{Suc}(emb\ m)) & \text{(def. of } emb\text{)} \\
\to_{\mathcal{R}^{log}} & \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(emb\ m))) & \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{Suc}(\mathsf{log}(emb\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{Suc}(emb\ (log\ (Suc\ m))) & \text{(induction hypothesis)} \\
\to^*_{\mathcal{R}^{log}} & emb\ (Suc\ (log\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\
= & emb\ c & \text{(assumption } half\ n = Suc\ m\text{)} \\
= & emb\ (log\ n) & \text{(def. of } log\text{)}
\end{array}$$

4 Examples

We show some characteristic examples that illustrate the strengths and weaknesses of our approach. Each example is representative for several similar ones that occur in the Isabelle distribution.

Example 5. Consider binary trees defined by the type

datatype *tree* $= E \mid N\ tree\ nat\ tree$

and a function *remdups* that removes duplicates from a tree. The function is defined by the following equations (the auxiliary function *del* removes all occurrences of an element from a tree; we omit its straightforward definition here):

$$\begin{aligned}&\mathit{remdups}\ E = E\\&\mathit{remdups}\ (N\ l\ x\ r) = N\ (\mathit{remdups}\ (\mathit{del}\ x\ l))\ x\ (\mathit{remdups}\ (\mathit{del}\ x\ r))\end{aligned}$$

The termination argument for *remdups* relies on a property of *del*: the result of *del* is smaller than its argument. In Isabelle, the user must manually state and prove (by induction) the lemma $\mathit{size}\ (\mathit{del}\ x\ t) \leq \mathit{size}\ t$, before termination can be shown. Here, *size* is an overloaded function generated automatically for every algebraic datatype.

For a termination tool, termination of the related TRS is easily proved using standard techniques, eliminating the need for finding and proving the lemma.

Example 6. The following function (originally due to Boyer and Moore [4]) normalizes conditional expressions consisting of atoms (AT) and if-expressions (IF).

$$\begin{aligned}&\mathit{norm}\ (AT\ a) = AT\ a\\&\mathit{norm}\ (IF\ (AT\ a)\ y\ z) = IF\ (AT\ a)\ (\mathit{norm}\ y)\ (\mathit{norm}\ z)\\&\mathit{norm}\ (IF\ (IF\ u\ v\ w)\ y\ z) = \mathit{norm}\ (IF\ u\ (IF\ v\ y\ z)\ (IF\ w\ y\ z))\end{aligned}$$

Isabelle's standard size measure is not sufficient to prove termination of *norm*, and a custom measure function must be specified by the user. Using a termination tool, the proof is fully automatic and no measure function is required.

Example 7. The Isabelle distribution contains the following implementation of the merge sort algorithm (transformed into non-overlapping rules internally):

$$\begin{aligned}&\mathit{msort}\ [\,] = [\,]\\&\mathit{msort}\ [x] = [x]\\&\mathit{msort}\ xs = \mathit{merge}\ (\mathit{msort}\ (\mathit{take}\ (\mathit{length}\ xs\ \mathit{div}\ 2)\ xs))\ (\mathit{msort}\ (\mathit{drop}\\&(\mathit{length}\ xs\ \mathit{div}\ 2)\ xs))\end{aligned}$$

The situation is similar to Example 5, as we must know how *take* and *drop* affect the length of the list. However, in this case, Isabelle's list theory already provides the necessary lemmas, e.g., $\mathit{length}\ (\mathit{take}\ n\ xs) = \mathit{min}\ n\ (\mathit{length}\ xs)$. Together with the built-in arithmetic decision procedures (which know about *div* and *min*), the termination proof works fully automatically.

For termination tools, the proof is a bit more challenging and requires techniques that are not yet formalized in IsaFoR (in particular, the technique of *rewriting dependency pairs* [8]). Thus, our connection to termination tools cannot handle *msort* yet. However, when this technique is added to IsaFoR in the future, no change will be required in our implementation to benefit from it.

These examples show the main strength of our reduction to rewriting: absolutely no user input in the form of lemmas or measure functions is required. On the other hand, Isabelle's ability to pick up previously established results can make the built-in termination prover surprisingly strong in the presence of

a good library, as the *msort* example shows. Even though that example can be solved by termination tools (and only the formalization lags behind), it shows an intrinsic weakness of the approach, since existing facts are not used and must be rediscovered by the termination tool if necessary.

5 Extensions

In this section, we reconsider the restrictions imposed in §2.2.

Nested Recursion. So far, we excluded nested recursion like $f\ (Suc\ n) = f\ (f\ n)$. The problem is that to prove termination of f we need its simulation lemma to reduce the inner call in the proof of the outer call, cf. §3.5. But proving the simulation lemma uses the induction rule of f, which in turn requires termination.

To solve this problem, we can use the partial induction rule that is generated by the function package even before a termination proof [13]. This rule, which is similar to the one used previously, contains extra domain conditions of the form $dom_f\ x$. It allows us to derive the restricted simulation lemma $dom_f\ n \implies \mathsf{f}(emb\ n) \to^*_{\mathcal{R}^f} emb\ (f\ n)$. In the termination proof obligation for the outer recursive call, we may assume this domain condition for the inner call (a convenience provided by the function package), so that this restricted form of simulation lemma suffices. Hence, dealing with nested recursion simply requires a certain amount of additional bookkeeping.

Underspecification. So far, we require functions to be completely defined, i.e., no cases are missing in left-hand sides or case-expressions. However, $head\ (x \# xs) = x$ is a common definition. It is internally completed by $head\ [\,] = undefined$ in Isabelle, where $undefined :: \alpha$ is an arbitrary but unknown value of type α.

For such functions, we cannot derive the simulation lemma, since this would require $\mathsf{head}(\mathsf{Nil})$ to be equal to *emb undefined*, which is an unknown term of the form $\mathsf{Suc}^k(\mathsf{0})$. The obvious idea of adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{undefined}$ to the TRS does not work, since $\mathsf{undefined}$ cannot be equal to *emb undefined*.

We can solve the problem by using fresh variables for unspecified cases, e.g., by adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{x}$. Then, the simulation lemma holds. However, the resulting TRS is no longer terminating. This new problem can be solved by using a variant of innermost rewriting, which would require support by IsaFoR as well as the termination tool.

Non-Representable Types and Polymorphism. Clearly, our embedding is limited to types that admit a term representation. This excludes uncountable types such as real numbers and most function types. However, even if such types occur in HOL functions, they may not be relevant for termination. Then, we can simply map all such values to a fixed constant by defining, e.g., $emb\ (r :: real) = \mathsf{real}$. Hence, the simulation lemmas for functions returning real numbers are trivial to prove. Furthermore, a termination proof that does not rely on these values works without problems. Like for underspecified functions, the generated TRS

no longer models the original function completely, but is only an abstraction that is sufficient to prove termination.

A similar issue arises with polymorphic functions: To handle a function of type α *list* $\Rightarrow$ α *list* we need a definition of *emb* on type α. Mapping values of type α to a constant is unproblematic, since the definition is irrelevant for the proof. However, a class instance α :: *embeddable* would violate the type class discipline. This can be solved by either replacing the use of type classes by explicit dictionary constructions (where emb_{list} would take the embedding function to use for the list elements as an argument), or by restricting α to class *embeddable*. Since the type class does not carry any axioms, the system allows us to remove the class constraint from the final theorem, so no generality is lost.

Higher-Order Functions. Higher-order functions pose new difficulties. First, we cannot hope to define *emb* on function types. In particular, this means that we cannot even state the simulation lemma for a function like *map*. Second, the termination conditions for functions with higher-order recursion depend on user-provided congruence rules of a certain format [13]. These congruence rules then influence the form of the premise ϕ in the termination condition.

A partial solution could be to create a first-order function map_f for each invocation of *map* on a concrete function f. Commonly used combinators like *map*, *filter* and *fold* could be supported in this way.

6 Conclusion

We have presented a generic approach to discharge termination goals of HOL functions by proving termination of a corresponding generated TRS. Hence, where before a manual termination proof might have been required, now external termination tools can be used. Since our approach is not tied to any particular termination proof technique, its power scales up as the capabilities of termination tools increase and more techniques are formalized in IsaFoR.

A complete prototype of our implementation is available in the IsaFoR/CeTA distribution (version 1.18, http://cl-informatik.uibk.ac.at/software/ceta), which also includes usage examples. It remains as future work to extend our approach to a larger class of HOL functions. Moreover, the implementation has to be more smoothly embedded into the Isabelle system such that a user can easily access the provided functionality. The general approach is not limited to Isabelle, and could be ported to other theorem provers like Coq, which has similar recursive definition facilities (e.g., [2]) and rewriting libraries similar to IsaFoR [3,6].

Acknowledgment. Jasmin Blanchette gave helpful feedback on a draft of this paper.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

2. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Proc. FLOPS'06*, volume 3945 of *LNCS*, pages 114–129. Springer, 2006.
3. F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comp. Science*, 2011. To appear.
4. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979.
5. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In *Proc. TPHOLs'07*, volume 4732 of *LNCS*, pages 38–53. Springer, 2007.
6. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. FroCoS'07*, volume 4720 of *LNCS*, pages 148–162. Springer, 2007.
7. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.
8. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Appl. Algebr. Eng. Comm.*, 12(1,2):39–72, 2001.
9. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR'06*, volume 4130 of *LNCS*, pages 281–286. Springer, 2006.
10. M. Gordon. From LCF to HOL: A short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.
11. M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA'09*, volume 5595 of *LNCS*, pages 295–304. Springer, 2009.
12. A. Krauss. Certified size-change termination. In *Proc. CADE'07*, volume 4603 of *LNCS*, pages 460–475. Springer, 2007.
13. A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010.
14. M. Marchiori. Logic programs as term rewriting systems. In *ALP '94*, volume 850 of *LNCS*, pages 223–241. Springer, 1994.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
16. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Algebr. Eng. Comm.*, 12(1-2):73–116, 2001.
17. C. Sternagel. *Automatic Certification of Termination Proofs*. PhD thesis, Institut für Informatik, Universität Innsbruck, Austria, 2010.
18. C. Sternagel and R. Thiemann. Certified subterm criterion and certified usable rules. In *Proc. RTA'10*, volume 6 of *LIPIcs*, pages 325–340, 2010.
19. R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.
20. H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.