

Term Rewriting in Logics of Partial Functions^{*}

Matthias Schmalz

ETH Zurich, Switzerland ^{**}

Abstract. We devise a theoretical foundation of *directed rewriting*, a term rewriting strategy for logics of partial functions, inspired by term rewriting in the Rodin platform. We prove that directed rewriting is sound and show how to supply new rewrite rules in a soundness preserving fashion. In the context of Rodin, we show that directed rewriting makes a significant number of conditional rewrite rules unconditional. Our work not only allows us to point out a number of concrete ways of improving directed rewriting in Rodin, but also has applications in other logics of partial functions. Additionally, we give a semantics for the logic of Event-B.

1 Introduction

Partiality is a common phenomenon in computer science: programs may not terminate, throw exceptions, or have undesired side-effects such as memory corruption. Reasoning about partial functions often involves proving a term to be *well-defined*; informally, a term is *well-defined* if and only if the involved functions are always applied to arguments within their domains. Well-definedness proofs are often perceived as an annoying distraction. Moreover, a theorem prover for a logic of partial functions can be overwhelmed by the number of well-definedness subgoals arising during proofs.

This paper tackles the problem of term rewriting in logics of partial functions. Many unconditional rewrite rules that are sound in logics of total functions become conditional when transferred to logics of partial functions; e.g., rewriting $\$x - \x to 0 is typically sound only under the condition that $\$x$ is well-defined. Note that the '\$' in $\$x$ emphasizes that $\$x$ matches arbitrary, possibly ill-defined, terms. Proving such well-definedness conditions even when applying the simplest rewrite rules constitutes a significant overhead.

In order to make concrete statements about the practical impact of our results, we focus on a particular logic of partial functions, although our results are of a more general nature. *Event-B* [1] is a formal method for modeling discrete state transition systems. Event-B is based on a logic of partial functions. *Rodin* [2] is the corresponding development environment. Interestingly, Rodin often avoids proving well-definedness: Rodin rewrites $\$x - \x to 0 without checking well-definedness of $\$x$, and $\$x \div \x to 1 without proving that $\$x$ differs from

^{*} This is the extended version of an article that appeared on ICFEM in 2011.

^{**} This research is supported by the EU funded FP7 project Deploy (Grant N° 214158).

0. This raises several questions, in particular, why term rewriting in Rodin is sound and to what extent Rodin avoids solving well-definedness conditions. The question of soundness has been addressed in [11], but only for a fragment of the logic (see Sect. 6).

In this paper, we provide a theoretical foundation for Rodin’s rewriting strategy: *directed rewriting*. Usually, the term t may be rewritten to u if and only if $t \equiv u$, i.e., t and u are semantically equal. In directed rewriting, t may be rewritten to u if and only if $t \sqsubseteq u$, where \sqsubseteq designates the *flat domain order* \sqsubseteq (see e.g. [18, p. 61]). We explain how conditional directed rewrite rules can be applied within proofs and why their application is sound. Directed rewriting is *unsafe* in general, i.e., it may transform a provable statement into an unprovable one and thus lead a proof attempt into a dead end. However, we show that it is straightforward to avoid this unsafety. Moreover, we provide evidence that directed rewriting significantly reduces the number of well-definedness checks required during proofs.

It is desirable that the set of rewrite rules used in proofs can be extended in a soundness preserving fashion. Event-B requires operators and binders to be monotonic w.r.t. the flat domain order; this is one of the reasons why directed rewriting is successful. However, if all operators are required to be monotonic, it is difficult to express soundness proof obligations for user-supplied rewrite rules. We point out a solution to this problem for a class of rewrite rules that often occur in practice.

Soundness proofs obviously presuppose a well-understood notion of validity. Other work introducing Event-B’s logic disregards the details about partial functions [1] or is confined to an untyped fragment of Event-B’s logic [12], in particular excluding sets. We therefore start our exposition with a novel presentation of syntax and semantics of Event-B’s logic.

The main contribution of this paper is the insight how a simple change in the rewriting strategy makes a significant number of conditional rewrite rules unconditional. Although we present our results in the context of Event-B, they have applications in other logics of partial functions such as LPF [10] and PVS [17], as we will explain in Sect. 6. We believe that our novel presentation of Event-B’s logic not only helps to understand directed rewriting, but is also useful for further developments such as mathematical (viz. conservative) extensions [5,20] and generic instantiation [22]. As a direct application, our results suggest several ways of improving Rodin’s current implementation of directed rewriting.

2 Abstract Syntax

Sequences. By t_1, \dots, t_n and $t_1 \dots t_n$ we denote the sequence of length n with t_i at position i , for $1 \leq i \leq n$. Variables denoting sequences are written in bold and underlined. We write t_i for the i th element of \underline{t} when it exists, $|\underline{t}|$ for \underline{t} ’s length, and $\underline{t}, \underline{t}'$ or $(\underline{t}, \underline{t}')$ for the concatenation of \underline{t} and \underline{t}' .

2.1 Types and Terms

Types. Roughly speaking, Event-B has a Hindley-Milner style type system, just like HOL [8] and ML. A *signature* Σ consists of three pairwise disjoint sets of *type operators*, (*ordinary operators*), and *binders*. We assume we are given infinitely many *type variables* that are distinct from the type operators given by signatures. A signature Σ assigns a non-negative integer, called *arity*, to each type operator. The signatures we consider in this paper all include the *boolean* type operator \mathcal{B} of arity zero. The set of *types (over Σ)* is the smallest set such that every type variable is a type, and $\tau(\underline{\nu})$ is a type if $\underline{\nu}$ is a sequence of types and τ a type operator of arity $|\underline{\nu}|$.

Examples of type operators in Rodin include the *integer type* \mathcal{Z} of arity 0 and the *powerset* type operator \mathcal{P} of arity 1. Informally, \mathcal{Z} denotes the set of all integers, $\mathcal{P}(\mathcal{Z})$ the set of all subsets of \mathcal{Z} , and $\mathcal{P}(\alpha)$ the set of all subsets of α .

A (*type*) *substitution* σ (*over Σ*) consists of a sequence $\underline{\alpha}$ of pairwise distinct type variables and a sequence $\underline{\mu}$ of types, where $|\underline{\alpha}| = |\underline{\mu}|$, and is written $[\underline{\alpha} := \underline{\mu}]$. The substitution σ maps the type ν to the type $\nu\sigma$ obtained by simultaneously replacing every occurrence of α_i by μ_i , $1 \leq i \leq |\underline{\alpha}|$, in ν . The type sequence $\underline{\nu}'$ is an *instance* of $\underline{\nu}$ iff $|\underline{\nu}'| = |\underline{\nu}|$ and there is a substitution σ such that $\nu'_i = \nu_i\sigma$, for $1 \leq i \leq |\underline{\nu}|$.

Terms. The signature Σ associates with each ordinary operator f a sequence of types $\underline{\nu}$ (the *argument type*) and a type μ (the *result type*), written as $f \circledast \underline{\nu} \rightsquigarrow \mu$ or, if $\underline{\nu}$ is empty, also as $f \circledast \mu$. With each binder Q the signature Σ associates a non-empty sequence $\underline{\nu}$ (the *bound variable type*), a non-empty sequence $\underline{\mu}$ (the *argument type*), and a type ξ (the *result type*), written as $Q \circledast (\underline{\nu} \rightsquigarrow \underline{\mu}) \rightsquigarrow \xi$.

Examples of operators in Rodin include $0 \circledast \mathcal{Z}$, *conjunction* $\wedge \circledast (\mathcal{B}, \mathcal{B}) \rightsquigarrow \mathcal{B}$, and *membership* $\in \circledast (\alpha, \mathcal{P}(\alpha)) \rightsquigarrow \mathcal{B}$. An example of a binder is *universal quantification* $\forall \circledast (\alpha \rightsquigarrow \mathcal{B}) \rightsquigarrow \mathcal{B}$, which informally takes a function mapping elements of α to booleans and yields a boolean.

We assume we are given infinitely many *variable names* that always differ from the operators and binders given by signatures. An (*ordinary*) *variable* $x \circledast \nu$ consists of a variable name x and a type ν . An *operator variable* $\$f \circledast \underline{\nu} \rightsquigarrow \mu$ consists of a variable name f , *argument types* $\underline{\nu}$ and a *result type* μ ; if $\underline{\nu}$ is empty, we usually write $\$f \circledast \mu$ instead of $\$f \circledast \rightsquigarrow \mu$.

Conditions T1-T4 below inductively define *terms (over Σ)*. By \underline{t} *has type* $\underline{\nu}$ (abbreviated as $\underline{t} \circledast \underline{\nu}$) we mean $|\underline{t}| = |\underline{\nu}|$ and t_i is of type ν_i , $1 \leq i \leq |\underline{t}|$.

- T1: Every ordinary variable of type ν is a term of type ν .
- T2: If $f \circledast \underline{\nu} \rightsquigarrow \mu$ is an operator and $\underline{t} \circledast \underline{\nu}'$ a sequence of terms, then $f(\underline{t} \circledast \underline{\nu}') \circledast \mu'$ is a term of type μ' , provided $(\underline{\nu}', \mu')$ is an instance of $(\underline{\nu}, \mu)$.
- T3: If $\$f \circledast \underline{\nu} \rightsquigarrow \mu$ is an operator variable and $\underline{t} \circledast \underline{\nu}$ a sequence of terms, then $\$f(\underline{t} \circledast \underline{\nu}) \circledast \mu$ is a term of type μ .
- T4: If $Q \circledast (\underline{\nu} \rightsquigarrow \underline{\mu}) \rightsquigarrow \xi$ is a binder, $\underline{x} \circledast \underline{\nu}'$ a sequence of pairwise distinct variables, and $\underline{t} \circledast \underline{\mu}'$ a sequence of terms, then $(Q \underline{x} \circledast \underline{\nu}' \cdot \underline{t} \circledast \underline{\mu}') \circledast \xi'$ is a term of type ξ' , provided $(\underline{\nu}', \underline{\mu}', \xi')$ is an instance of $(\underline{\nu}, \underline{\mu}, \xi)$.

A term of type \mathcal{B} is called *formula*. Rodin imposes further restrictions on terms; in particular, terms containing variables with the same name but different types are in some cases rejected. We ignore these restrictions, because they have no logical significance and would merely complicate our presentation without adding clarity.

Remark 1. Traditionally [1,13], the logic of Event-B is presented like a first-order logic with separate syntactic categories of “expressions” and “predicates”. Intuitively, expressions stand for terms and predicates for formulae. Expressions may however have boolean type and there are conversions between boolean expressions and predicates, i.e., $\text{bool}(\varphi)$ is the boolean expression corresponding to the predicate φ , and $t = \text{bool}(\top)$ is the predicate corresponding to the boolean expression t . We therefore abandon the strict separation between expressions and predicates, and define only the syntactic category of terms. To translate from our representation to Rodin’s, one merely needs to insert appropriate conversions, e.g., $\forall x \varepsilon \mathcal{B} \cdot x \in \{\top\} \wedge x$ corresponds to $\forall x \varepsilon \mathcal{B} \cdot x \in \{\text{bool}(\top)\} \wedge x = \text{bool}(\top)$.

Operators and operator variables have much in common, but are used for different purposes. Intuitively, an operator has a fixed meaning, while the meaning of an operator variable is unspecified. A term of the form $\$f(\underline{t})$ serves as placeholder, e.g., when specifying rewrite rules (Sect. 4.1). Rodin does not explicitly support operator variables, which is convenient for term rewriting (Sect. 4.3), but introduces challenges when reasoning about the soundness of rewrite rules (Sect. 5).

We adopt the usual definitions of *bound* and *free* (ordinary) variables. Unless mentioned otherwise, we consider *alpha-congruent* terms, i.e., terms that informally speaking differ only in the names of bound variables (see e.g. [9]), as identical.

2.2 An Example Signature

For the sake of illustration, we define the signature Σ_1 introducing

- the type operators \mathcal{B} and \mathcal{Z} of arity 0 and \mathcal{P} of arity 1,
- the operators $\text{D} \varepsilon \alpha \rightsquigarrow \mathcal{B}$, $\bullet \varepsilon \alpha$, and $= \varepsilon (\alpha, \alpha) \rightsquigarrow \mathcal{B}$,
- the operators $\top \varepsilon \mathcal{B}$, $\perp \varepsilon \mathcal{B}$, and $\neg \varepsilon \mathcal{B} \rightsquigarrow \mathcal{B}$,
- the operators $\wedge \varepsilon (\mathcal{B}, \mathcal{B}) \rightsquigarrow \mathcal{B}$, $\vee \varepsilon (\mathcal{B}, \mathcal{B}) \rightsquigarrow \mathcal{B}$, and $\Rightarrow \varepsilon (\mathcal{B}, \mathcal{B}) \rightsquigarrow \mathcal{B}$,
- the operators $\in \varepsilon (\alpha, \mathcal{P}(\alpha)) \rightsquigarrow \mathcal{B}$, $\emptyset \varepsilon \mathcal{P}(\alpha)$, and $\cap \varepsilon (\mathcal{P}(\alpha), \mathcal{P}(\alpha)) \rightsquigarrow \mathcal{P}(\alpha)$,
- the operators $0 \varepsilon \mathcal{Z}$ and $1 \varepsilon \mathcal{Z}$, and $\text{mod} \varepsilon (\mathcal{Z}, \mathcal{Z}) \rightsquigarrow \mathcal{Z}$, and
- the binders $\forall \varepsilon (\alpha \rightsquigarrow \mathcal{B}) \rightsquigarrow \mathcal{B}$ and $\text{collect} \varepsilon (\alpha \rightsquigarrow \mathcal{B}) \rightsquigarrow \mathcal{P}(\alpha)$.

Rodin provides more symbols than those in Σ_1 (see [20]). The operators D and \bullet are not available in Rodin, but simplify our presentation. Intuitively, the term $\text{D}(t)$ is true if t is well-defined and otherwise false. The term \bullet is always ill-defined. Our results do not depend on the availability of \bullet . Whenever required, we will discuss the relevance of our results for signatures without D .

To improve readability, we use infix notation and leave out parentheses when the precedence is clear. We leave out type constraints “ $\text{§ } \nu$ ” when the types are clear or irrelevant. Terms of the form `collect $x \cdot \varphi$` are written $\{x \mid \varphi\}$. The formula $\forall x_1 \cdot \dots \cdot \forall x_{|\underline{x}|} \cdot \varphi$ is abbreviated as $\forall \underline{x} \cdot \varphi$. By default, a term is of the most general type, and we assign the same types to different occurrences of a variable name. See [13] for Rodin’s concrete syntax conventions.

3 Semantics

An important decision is in which logic to formalize the denotations of Event-B’s types and terms. The best option can certainly not be uniquely determined. We have decided to define the semantics of Event-B’s logic by an embedding in higher-order logic (HOL) for the following reasons. First, HOL has a well-understood set theoretic semantics [3,8]. Second, Event-B’s logic closely resembles HOL; the main difference lies in the treatment of partial functions. This similarity allows us to keep our presentation of semantics concise. Third, there are powerful theorem provers for HOL such as Isabelle/HOL [15]. Hence, an embedding of Event-B’s logic into HOL enables us to use Isabelle/HOL as a theorem prover for Rodin. We regard it as promising to integrate Isabelle/HOL into Rodin, because Isabelle/HOL provides powerful proof tactics and Isabelle/HOL’s proofs are more trustworthy than Rodin’s thanks to Isabelle’s LCF architecture. Isabelle/HOL can also be used to prove meta-theorems about Event-B that are hard to formalize in Event-B itself; for example, in [20] we use Isabelle/HOL to prove soundness of Event-B proof rules.

Our main sources of information are [1,12,13]; these sources give an intuition about the intended semantics, but also leave questions open. We have resolved these questions through discussions with other Rodin developers.

3.1 Isabelle/HOL

To make the paper more accessible to readers not familiar with Isabelle/HOL, we summarize the most relevant features. Isabelle [19] is a generic theorem prover supporting various logics. The term HOL from now on refers to the instantiation of Isabelle to HOL.

HOL’s type system (see [8,23]) essentially coincides with ML’s. The notation $t :: \nu$ indicates that the term t has type ν . A term of type $\nu \Rightarrow \mu$ denotes a function taking one argument of type ν and yielding a result of type μ . Functions taking n arguments, $n > 0$, are represented by terms of type $\nu_1 \Rightarrow \dots \Rightarrow \nu_{n+1}$. The type operator \Rightarrow associates to the right. The application of the function f to the n arguments x_1, \dots, x_n is written $f \ x_1 \ \dots \ x_n$ and should be read as $(\dots (f \ x_1) \ \dots) \ x_n$. Function application has higher precedence than infix operators: $f \ x + 1$ is to be read as $(f \ x) + 1$.

Terms of type $\nu \Rightarrow \mu$ represent total functions. Partial functions are therefore sometimes approximated by total functions: HOL’s integer division `div` is a total function of type `int \Rightarrow int \Rightarrow int`. The developers of Isabelle/HOL have decided

that $x \text{ div } (0 :: \text{int})$ equals 0. The way how a partial function is approximated by a total function varies from case to case: in particular, the “least integer” $\text{Least } \{x :: \text{int}. \text{True}\}$ is left unspecified.

3.2 Option Types

We develop a theory EB_0 providing auxiliary definitions. The standard theory of HOL introduces *option types*, defined by

datatype $\alpha \text{ option} = \text{Some } \alpha \mid \text{None}.$

Intuitively, the type $\alpha \text{ option}$ contains copies of all elements of α and a constant None . If x is of type α , then $\text{Some } x$ is the copy of x in $\alpha \text{ option}$. The theory EB_0 introduces the notations $\alpha\uparrow$ for $\alpha \text{ option}$, $x\uparrow$ for $\text{Some } x$, and \bullet for None .

Moreover, EB_0 defines the functions WD , T , F , WT by

$$\begin{aligned} \text{WD } x &= (x \neq \bullet), \\ \text{T } \varphi &= (\varphi = \text{True}\uparrow), & \text{F } \varphi &= (\varphi = \text{False}\uparrow), & \text{WT } \varphi &= (\varphi \neq \text{False}\uparrow). \end{aligned}$$

The function WD takes a term t of type $\nu\uparrow$ and indicates whether t differs from \bullet . The term t is *well-defined* iff $\text{WD } t$ is valid and *ill-defined* iff $\neg(\text{WD } t)$ is valid. The functions T and F take a term φ of type $\text{bool}\uparrow$ and indicate whether φ equals $\text{True}\uparrow$ or $\text{False}\uparrow$, respectively. Moreover, $\text{WT } \varphi$ indicates whether φ is *weakly true*, i.e., equal to $\text{True}\uparrow$ or equal to \bullet . Finally, we say that a function f whose arguments and result have option types is *strict* iff $\text{WD}(f \underline{x}) \longrightarrow \text{WD } x_1 \wedge \dots \wedge \text{WD } x_{|\underline{x}|}$ is valid.

3.3 Denotations of Types and Terms

Given a signature Σ , a *structure* (over Σ) specifies the *denotations* of Event-B types and terms over Σ . Technically, a structure $(M, \llbracket \cdot \rrbracket)$ consists of a HOL theory M extending EB_0 and a *denotation function* $\llbracket \cdot \rrbracket$ mapping Event-B symbols, types, and terms to HOL symbols, types, and terms, respectively. For a sequence \underline{t} of types or terms, $\llbracket \underline{t} \rrbracket$ abbreviates $(\llbracket t_1 \rrbracket, \dots, \llbracket t_{|\underline{t}|} \rrbracket)$.

For every Event-B type variable α , we define $\llbracket \alpha \rrbracket = \alpha$; here we assume, without loss of generality, that α is a HOL type variable. For every Event-B type operator τ of arity n , $\llbracket \tau \rrbracket$ is a HOL type operator taking n arguments¹. The boolean type \mathcal{B} denotes bool , and a type $\tau(\underline{\nu})$ denotes the type $(\llbracket \underline{\nu} \rrbracket) \llbracket \tau \rrbracket$, i.e., the result of applying the type operator $\llbracket \tau \rrbracket$ to the types $\llbracket \underline{\nu} \rrbracket$.

While the denotation of a type is obtained by renaming type operators, the situation is more involved for terms. The denotation function $\llbracket \cdot \rrbracket$ maps operators $f \circ \underline{\nu} \rightsquigarrow \mu$ to HOL constants of type $\llbracket \nu_1 \rrbracket \uparrow \Rightarrow \dots \Rightarrow \llbracket \nu_{|\underline{\nu}|} \rrbracket \uparrow \Rightarrow \llbracket \mu \rrbracket \uparrow$ and binders $Q \circ (\underline{\nu} \rightsquigarrow \underline{\mu}) \rightsquigarrow \xi$ to HOL constants of type

$$(\llbracket \nu_1 \rrbracket \Rightarrow \dots \Rightarrow \llbracket \nu_{|\underline{\nu}|} \rrbracket \Rightarrow \llbracket \mu_1 \rrbracket \uparrow) \Rightarrow \dots \Rightarrow (\llbracket \nu_1 \rrbracket \Rightarrow \dots \Rightarrow \llbracket \nu_{|\underline{\nu}|} \rrbracket \Rightarrow \llbracket \mu_{|\underline{\mu}|} \rrbracket \uparrow) \Rightarrow \llbracket \xi \rrbracket \uparrow.$$

¹ In this regard, we also view HOL’s *type synonyms* as type operators.

An operator is *strict* iff its denotation is strict.

An Event-B term of type ν denotes a HOL term of type $\llbracket \nu \rrbracket \uparrow$ as follows:

1. $\llbracket x \circledast \nu \rrbracket = (x :: \llbracket \nu \rrbracket) \uparrow$.
2. $\llbracket f(\mathbf{t}) \circledast \mu' \rrbracket = ((\llbracket f \rrbracket \llbracket \mathbf{t} \rrbracket) :: \llbracket \mu' \rrbracket) \uparrow$.
3. $\llbracket \$f(\mathbf{t}) \circledast \mu \rrbracket = ((\$f \llbracket \mathbf{t} \rrbracket) :: \llbracket \mu \rrbracket) \uparrow$.
4. $\llbracket (Q\mathbf{x} \circledast \underline{\nu} \cdot \mathbf{t}) \circledast \xi' \rrbracket = ((\llbracket Q \rrbracket (\lambda \mathbf{x} :: \llbracket \underline{\nu} \rrbracket. \llbracket \mathbf{t} \rrbracket)) \dots (\lambda \mathbf{x} :: \llbracket \underline{\nu} \rrbracket. \llbracket \mathbf{t} \rrbracket)) :: \llbracket \xi' \rrbracket) \uparrow$.

For convenience, we assume that, for each Event-B variable name x , both x and $\$x$ are available as variable names in M. In 4, the notation $\underline{\mathbf{x}} :: \underline{\nu}$ abbreviates $(x_1 :: \nu_1) \dots (x_{|\underline{\mathbf{x}}|} :: \nu_{|\underline{\nu}|})$.

An Event-B term is *well-defined* (*ill-defined*) iff its denotation is well-defined (ill-defined). The Event-B terms t and u are *equivalent* iff $\llbracket t \rrbracket = \llbracket u \rrbracket$ is valid. An Event-B formula φ is *valid* iff $\top \llbracket \varphi \rrbracket$ is valid.

Readers familiar with HOL may wonder why Event-B has the syntactic categories of operators and binders. HOL provides only one operator, namely function application, and one binder, namely lambda-abstraction, and views the remaining operators and binders as constants of suitable function types. It would however be difficult to organize the logic of Event-B in a similar way, because Event-B's *constants* (i.e., operators with empty argument types) denote terms of type $\nu \uparrow$; hence constants cannot be used to represent operators or binders, simply because they have inappropriate types.

3.4 An Example Structure

We define a structure $(\text{EB}_1, \llbracket \cdot \rrbracket)$ over Σ_1 , the signature introduced in Sect. 2.2. It reflects the intended semantics of the symbols introduced by Σ_1 , i.e., the semantics that is used to validate the inference rules implemented in Rodin. The denotations of Rodin's remaining symbols are defined in [20].

We start with the denotations of type operators, D, and \bullet :

$$\begin{aligned} \llbracket \mathcal{B} \rrbracket &= \text{bool}, & \llbracket \mathcal{Z} \rrbracket &= \text{int}, & \llbracket \mathcal{P} \rrbracket &= \text{set}, \\ \llbracket \text{D} \rrbracket x &= (\text{WD } x) \uparrow, & \llbracket \bullet \rrbracket &= \bullet. \end{aligned}$$

The *strict extension* F of a function f taking n arguments, $n \geq 0$, is given by

$$F x_1 \dots x_n = \begin{cases} \bullet & (\neg(\text{WD } x_1) \vee \dots \vee \neg(\text{WD } x_n)) \\ (f y_1 \dots y_n) \uparrow & (x_1 = y_1 \uparrow \wedge \dots \wedge x_n = y_n \uparrow). \end{cases}$$

Intuitively, F behaves as f for well-defined arguments and is ill-defined if an argument is ill-defined. The denotations of most operators are strict extensions:

the denotation of	\top	\perp	\neg	$=$	\emptyset	\in	\cap	0	1
is the strict extension of	True	False	\neg	$=$	{}	\in	\cap	$0 :: \text{int}$	$1 :: \text{int}$

Thus, if x is an ordinary variable, then $x = x$ is valid, because x is well-defined. If however $\$x$ is an operator variable, then $\$x = \x is not valid.

The operator `mod` is strict; its result is ill-defined if an argument is negative:

$$\llbracket \text{mod} \rrbracket x \uparrow y \uparrow = \begin{cases} (x \text{ mod } y) \uparrow & (x \geq 0 \wedge y > 0) \\ \bullet & (\text{otherwise}). \end{cases}$$

The denotation of conjunction is given by

$$\llbracket \wedge \rrbracket \varphi \psi = \begin{cases} \text{True} \uparrow & (\text{T } \varphi \wedge \text{T } \psi) \\ \text{False} \uparrow & (\text{F } \varphi \vee \text{F } \psi) \\ \bullet & (\text{otherwise}). \end{cases}$$

The denotations of disjunction and implication are defined such that $\$ \varphi \vee \$ \psi$ is equivalent to $\neg(\neg \$ \varphi \wedge \neg \$ \psi)$ and $\$ \varphi \Rightarrow \$ \psi$ is equivalent to $\neg \$ \varphi \vee \$ \psi$.

Note that conjunction, disjunction, and implication are not strict. In particular, both $\perp \wedge \$ \varphi$ and $\$ \varphi \wedge \perp$ are equivalent to \perp . Contrast this to intersection: neither $\emptyset \cap \$R$ nor $\$R \cap \emptyset$ is equivalent to \emptyset , because $\$R$ is not necessarily well-defined and intersection is strict.

Finally, $\llbracket \forall \rrbracket$ and $\llbracket \text{collect} \rrbracket$ are defined such that

$$\begin{aligned} \llbracket \forall x \cdot \varphi \rrbracket &= \begin{cases} \text{True} \uparrow & (\forall x. \text{T } \llbracket \varphi \rrbracket) \\ \text{False} \uparrow & (\exists x. \text{F } \llbracket \varphi \rrbracket) \\ \bullet & (\text{otherwise}), \end{cases} \\ \llbracket \{x \mid \varphi\} \rrbracket &= \begin{cases} \{x. \text{T } \llbracket \varphi \rrbracket\} \uparrow & (\forall x. \text{WD } \llbracket \varphi \rrbracket) \\ \bullet & (\text{otherwise}). \end{cases} \end{aligned}$$

The universal quantifier may be viewed as generalized conjunction: for instance, $\forall x \varepsilon \mathcal{B} \cdot \$ \varphi(x)$ is equivalent to $\$ \varphi(\top) \wedge \$ \varphi(\perp)$. The variables bound by a binder range over well-defined values: $\forall x \cdot x = x$ is valid, and $\{x \mid x \neq x\}$ is equivalent to the empty set \emptyset .

3.5 Substitutions

Intuitively, a type substitution $[\underline{\alpha} := \underline{\nu}]$ is applied to a term by simultaneously replacing α_i by ν_i , $1 \leq i \leq |\underline{\alpha}|$. The details are similar as in HOL (cf. [8, CH. 15]) and can be found in Sect. A. We define the *denotation* $\llbracket [\underline{\alpha} := \underline{\nu}] \rrbracket$ of a type substitution by $[\underline{\alpha} := \underline{\nu}]$. From this, we prove the following duality property:

Lemma 2. *If t is an Event-B term and σ a type substitution, then $\llbracket t\sigma \rrbracket = \llbracket t \rrbracket \llbracket \sigma \rrbracket$.*

Type variables intuitively serve as place holders for types. By defining the effect of type substitutions on terms, we make precise how exactly type variables can be instantiated. Lemma 2 provides a semantic characterization of type substitutions σ ; in particular, if $\text{T } \llbracket \varphi \rrbracket$ is valid, then so is $\text{T } \llbracket \varphi\sigma \rrbracket$.

An *operator substitution* σ is written

$$[\$f_1(\underline{\$x}^1) := u_1, \dots, \$f_n(\underline{\$x}^n) := u_n],$$

where $\$f_1, \dots, \f_n are pairwise distinct, the elements of $\underline{\$x}^i$ are pairwise distinct, the type of $\underline{\$x}^i$ is the argument type of $\$f_i$, and the type of u_i the result type of $\$f_i$, for $1 \leq i \leq n$. Let us consider three examples. First, we will see that $(\$f \wedge \neg \$f)[\$f := \top]$ equals $\top \wedge \neg \top$. Second, consider $\forall x \cdot \$f(x)$; by writing $\$f(x)$ instead of $\$f$, we indicate that the term substituted for $\$f(x)$ may have free occurrences of x . We will see that $(\forall x \cdot \$f(x))[\$f(\$y) := (\$y = 0)]$ equals $\forall x \cdot x = 0$. Finally, $(\forall x \cdot \$f)[\$f := (x = 0)]$ equals $\forall y \cdot x = 0$.

The *denotation* $\llbracket \sigma \rrbracket$ of σ is

$$[\$f_1 := \lambda \underline{\$x}^1. \llbracket u_1 \rrbracket, \dots, \$f_n := \lambda \underline{\$x}^n. \llbracket u_n \rrbracket].$$

To obtain a duality property analog to Lemma 2, we define the *result of applying* σ to a term as follows:

1. $x\sigma = x$,
2. $f(\underline{t})\sigma = f(\underline{t}\sigma)$,
3. $\$g(\underline{t})\sigma = \$g(\underline{t}\sigma)$, provided $\$g$ differs from $\$f_i$, for $1 \leq i \leq n$,
4. $\$f_i(\underline{t})\sigma = u_i[\$x_1^i := t_1\sigma, \dots, \$x_{|\underline{\$x}^i|}^i := t_{|\underline{\$x}^i|}\sigma]$, for $1 \leq i \leq n$,
5. $(Q\underline{y} \cdot \underline{t})\sigma = Q\underline{y} \cdot \underline{t}\sigma$, provided no element of \underline{y} occurs free in u_1, \dots, u_n .

In 2, 3, and 5, we adopt the convention that $\underline{t}\sigma$ abbreviates $t_1\sigma, \dots, t_{|\underline{t}|}\sigma$. The proviso of 5 is achieved by suitable renaming of bound variables. Operator substitutions fulfill a duality property analog to Lemma 2:

Lemma 3. *If t is a term and σ an operator substitution, then $\llbracket t\sigma \rrbracket = \llbracket t \rrbracket \llbracket \sigma \rrbracket$.*

If we defined substitutions for ordinary variables, they would fail to satisfy a property analog to Lemma 2 and 3. Suppose we define an ordinary substitution σ such that $x\sigma$ is \bullet . Then $\llbracket x\sigma \rrbracket$ is \bullet , but there is no way of defining $\llbracket \sigma \rrbracket$ such that $\llbracket x \rrbracket \llbracket \sigma \rrbracket$ equals \bullet , because $\llbracket x \rrbracket$ is defined as $x\uparrow$.

4 Term Rewriting

In the following, the available types and terms, and their denotations are specified by a signature Σ and a structure $(M, \llbracket \cdot \rrbracket)$ over Σ . If Σ provides symbols of Σ_1 (cf. Sect. 2.2), we assume that their denotations are given according to $(EB_1, \llbracket \cdot \rrbracket)$ (cf. Sect. 3.4).

In Event-B, proofs are organized in terms of hypothetical statements, called *sequents*. A *sequent* $\underline{\psi} \vdash \varphi$ consists of a finite set $\{\underline{\psi}\}$ of formulae, called *hypotheses*, and a single formula φ , called *goal*. When a sequent is used to express a desired property of an Event-B model, we refer to it as *proof obligation*.

We consider several ways of defining denotations of sequents, all of the form

$$\llbracket \psi_1, \dots, \psi_n \vdash \varphi \rrbracket = (\mathcal{H}[\psi_1] \wedge \dots \wedge \mathcal{H}[\psi_n] \longrightarrow \mathcal{G}[\varphi]).$$

The functions \mathcal{H} and \mathcal{G} range over \mathbb{T} and WT . We distinguish *WW*-, *WS*-, *SW*-, and *SS*-semantics, where the first letter indicates the choice of \mathcal{H} and the second

the choice of \mathcal{G} ; the letter S (“strong”) represents the choice T, and the letter W (“weak”) represents the choice WT. A sequent is *WW*-, *WS*-, *SW*-, or *SS*-*valid* iff its denotation is valid according to *WW*-, *WS*-, *SW*-, or *SS*-semantics, respectively. As Event-B is based on *SW*-validity, we refer to *SW*-validity also as *validity*.

An *inference rule* is written $\frac{\underline{\Gamma}}{\Gamma_0}$ and consists of a possibly empty sequence $\underline{\Gamma}$ of sequents, called *antecedents*, and a single sequent Γ_0 , called *consequent*. It is *sound* iff validity of all antecedents implies validity of the consequent. We are interested in *backwards proofs*: in a backwards proof a sequent Δ is proved by first choosing a rule with Δ as consequent and then proving the antecedents of the rule. This is repeated up to the point where no sequents remain to be proved.

Note that rules are schematic in the sense that they may contain operator variables, which can be instantiated by substitution. Because of Lemmas 2 and 3, soundness is closed under substitution, i.e., if an inference rule is sound, then so is the result of applying a substitution to its antecedents and consequent. However, side-conditions that require variables not to occur free in formulae must be stated informally. Defining a formal representation covering all inference rules of Event-B is not in the scope of this paper.

The choice between *WW*-, *WS*-, *SW*-, and *SS*-semantics influences which inference rules are sound. Unfortunately it is not an easy choice; for each of the four semantics there exist an unsound rules whose unsoundness may be hard to accept for some reader:

semantics	unsound rule	semantics	unsound rule
<i>WW</i>	$\frac{\underline{\mathcal{X}} \vdash \psi}{\underline{\mathcal{X}}, \neg\psi \vdash \perp}$ not _E	<i>WS</i>	$\frac{}{\underline{\mathcal{X}}, \varphi \vdash \varphi}$ hyp
<i>SS</i>	$\frac{\underline{\mathcal{X}}, \varphi \vdash \perp}{\underline{\mathcal{X}} \vdash \neg\varphi}$ not _I	<i>SW</i>	$\frac{\underline{\mathcal{X}} \vdash \psi \quad \underline{\mathcal{X}}, \psi \vdash \varphi}{\underline{\mathcal{X}} \vdash \varphi}$ cut

It is therefore not surprising that different logics are based on different semantics: Event-B [12] and PVS [17] choose *SW*-semantics; LPF [4], the logic underlying VDM, is based on *SS*-semantics; Owe [16] favors *WS*-semantics. To our best knowledge, only Owe explains his choice by a comparison between the four semantics. In Section 4.2, we will show why *SW*-semantics, and *SW*-semantics only, is well-suited for term rewriting. We thus provide a novel argument in favor of *SW*-semantics.

4.1 Rewriting Terms to Equivalent Terms

When rewriting terms to equivalent terms in Event-B’s logic, it is often necessary to check well-definedness of the term to be rewritten. Consider for example

$$(\$x = \$x) \equiv \top. \tag{1}$$

A rule $t \equiv u$ is *sound* iff t and u are equivalent; hence, (1) is unsound. The unsoundness becomes evident when rewriting the goal $D(\bullet = \bullet)$ to $D(\top)$. However, the rule is sound under the precondition $D(\$x)$, i.e., whenever a well-defined term is substituted for $\$x$. Similarly, the following rules are sound only under appropriate well-definedness preconditions:

$$\$x \in \$R \cap \$S \equiv \$x \in \$R \wedge \$x \in \$S, \quad (2)$$

$$\$x \in \emptyset \equiv \perp, \quad (3)$$

$$0 \in \{x \mid \$\varphi(x)\} \equiv \$\varphi(0). \quad (4)$$

The need for well-definedness preconditions stems from the special status of \bullet , i.e., (1) is unsound because equality is strict and therefore not reflexive, (2) is unsound because intersection is strict and conjunction is not, (3) is unsound because membership is strict, and (4) is unsound because there are substitutions σ such that $\$\varphi(0)\sigma$ is well-defined whereas $\$\varphi(1)\sigma$ is not. Well-definedness preconditions can be avoided to some extent by choosing appropriate semantics. However, this would result in difficulties when applying Event-B to the problems it has initially been designed for.

During conditional rewriting as in, e.g., Isabelle [14], it is often hard to predict and control which conditions can be solved and whether a conditional rewrite rule is applied. The choice of tactic that solves conditions is tricky: if it is too weak, many conditional rewrite rules become useless; if it is too powerful, term rewriting becomes slow. Because solving conditions is undecidable, these problems are inevitable in general; but we will see that we can sometimes do better.

In the following sections, we present *directed rewriting*. *Directed rewrite rules* have well-definedness preconditions, but these preconditions can always be solved, and therefore do not need to be checked. We thus show how the before mentioned problems can be avoided for a non-trivial class of rewrite rules. We will also demonstrate the practical relevance of directed rewriting.

4.2 Directed Rewriting

A *directed rewrite rule* consists of a (pre)condition $\varphi \circ \mathcal{B}$, a *left-hand side* $t \circ \nu$, and a *right-hand side* $u \circ \nu$, and is written

$$\frac{\varphi}{t \circ \nu \sqsubseteq u \circ \nu}. \quad (5)$$

An *unconditional* rule has the condition \top and is written $t \circ \nu \sqsubseteq u \circ \nu$. For the semantics of directed rewrite rules, we recall the *flat domain order* \sqsubseteq defined by

$$\forall x y. x \sqsubseteq y \iff (\text{WD } x \longrightarrow x = y).$$

The rewrite rule (5) denotes $\text{WT}[\varphi] \longrightarrow \llbracket t \rrbracket \sqsubseteq \llbracket u \rrbracket$ and is sound iff its denotation is valid. Directed rewrite rules are not symmetric: soundness of $t \sqsubseteq u$ does not imply soundness of $u \sqsubseteq t$. The reader may want to check that the rules (1–4) can be recast as unconditional sound directed rewrite rules.

Alternatively, we could have defined the denotation of (5) by $\mathbb{T}[\varphi] \longrightarrow \llbracket t \rrbracket \sqsubseteq \llbracket u \rrbracket$. Our choice of semantics is motivated by the observation (cf. Sect. B.1) that the majority of the conditional rewrite rules available in Rodin are already sound w.r.t. our semantics, which interprets the condition φ as $\text{WT}[\varphi]$. The advantage of our semantics over the alternative semantics, which interprets φ as $\mathbb{T}[\varphi]$, is that $\text{D}(\varphi)$ does not need to be proved when applying the rule; this makes conditional rules more generally applicable. If a rule is sound only w.r.t. the alternative semantics we may replace the condition φ by $\text{D}(\varphi) \wedge \varphi$.

The following lemma shows how to use directed rewrite rules to rewrite formulae:

Lemma 4. *If the rule*

$$\frac{\psi}{\varphi_1 \sqsubseteq \varphi_2} \quad (6)$$

is sound, then so are

$$\frac{\underline{\mathbf{x}} \vdash \psi \quad \underline{\mathbf{x}} \vdash \varphi_2}{\underline{\mathbf{x}} \vdash \varphi_1} \quad \text{and} \quad \frac{\underline{\mathbf{x}} \vdash \psi \quad \underline{\mathbf{x}}, \varphi_2 \vdash \varphi}{\underline{\mathbf{x}}, \varphi_1 \vdash \varphi}. \quad (7)$$

Intuitively, if the rule in (6) is sound, it may be used to rewrite a hypothesis or goal φ_1 to φ_2 in a backwards proof. Moreover, when applying a conditional rule, one has to prove the condition ψ from the other hypotheses $\underline{\mathbf{x}}$ of the sequent at hand. Note that, although, in general, φ_1 and φ_2 are equivalent only if φ_1 is well-defined, there is no need to prove well-definedness of φ_1 when applying the rewrite rule. This is how directed rewriting avoids solving well-definedness conditions.

Lemma 4 is an immediate consequence of the definitions of soundness and the fact that sequents have SW-semantics. The assertion does not hold if sequents have WW-, WS-, or SS-semantics: take the empty sequence for $\underline{\mathbf{x}}$, \top for ψ , \bullet for φ_1 , and \top or \perp for φ_2 . Thus, SW-semantics is the only of the four semantics, under which directed rewriting is sound.

So far we are assuming that we want to apply directed rewrite rules from left to right, i.e., to rewrite φ_1 to φ_2 . If we wanted to apply them in reverse direction, i.e., to rewrite φ_2 to φ_1 , this would be sound only under WS-semantics. We believe that rewriting in reverse direction is not useful in practice, because it would make rules like $\$x = \$x \sqsubseteq \top$, $\$\varphi \wedge \neg\$\varphi \sqsubseteq \perp$, $\$R \cap \emptyset \sqsubseteq \emptyset$, and many others inapplicable.

4.3 Rewriting Subterms

Term rewriting would not be very useful without the possibility of rewriting subterms. We now explain under which assumptions directed rewriting of subterms is sound.

A HOL function f is *monotonic* iff

$$\forall \underline{\mathbf{x}} \underline{\mathbf{y}}. x_1 \sqsubseteq y_1 \wedge \cdots \wedge x_{|\underline{\mathbf{x}}|} \sqsubseteq y_{|\underline{\mathbf{y}}|} \longrightarrow f \underline{\mathbf{x}} \sqsubseteq f \underline{\mathbf{y}}.$$

For uniformity, we also consider terms of type $\nu\uparrow$ as *monotonic*. The order \sqsubseteq is lifted to functions in the usual point-wise fashion: $f \sqsubseteq g \iff (\forall x. f\ x \sqsubseteq g\ x)$. The denotation Q of a binder is *monotonic* iff

$$\forall \underline{f}\ \underline{g}. f_1 \sqsubseteq g_1 \wedge \dots \wedge f_{|\underline{f}|} \sqsubseteq g_{|\underline{g}|} \longrightarrow Q\ \underline{f} \sqsubseteq Q\ \underline{g}.$$

Similarly, an operator or binder is monotonic iff its denotation is.

The following lemma shows that directed rewrite rules may be applied to subterms of the hypothesis or the goal at hand. The restriction is that only arguments of *monotonic* operators and binders may be rewritten.

Lemma 5. *If the directed rewrite rule $\frac{\varphi}{t \sqsubseteq t'}$ is sound, then so are*

$$\frac{\varphi}{f(\underline{u}, t, \underline{u}') \sqsubseteq f(\underline{u}, t', \underline{u}')} \quad \text{and} \quad \frac{\forall \underline{x}. \varphi}{Q\underline{x} \cdot \underline{v}, t, \underline{v}' \sqsubseteq Q\underline{x} \cdot \underline{v}, t', \underline{v}'},$$

where f is a monotonic operator and Q a monotonic binder.

Proof. Suppose the rule

$$\frac{\varphi}{t \sqsubseteq t'} \tag{8}$$

is sound. Soundness of

$$\frac{\varphi}{f(\underline{u}, t, \underline{u}') \sqsubseteq f(\underline{u}, t', \underline{u}')}$$

immediately follows from the monotonicity of f .

For the remaining assertion, note that soundness of (8) entails validity of

$$\text{WT } \llbracket \varphi \rrbracket \longrightarrow \llbracket t \rrbracket \sqsubseteq \llbracket t' \rrbracket.$$

By universal generalization,

$$\forall \underline{x}. \text{WT } \llbracket \varphi \rrbracket \longrightarrow \llbracket t \rrbracket \sqsubseteq \llbracket t' \rrbracket$$

is valid. Note that, $\forall x. \psi_1 \longrightarrow \psi_2$ implies $(\forall x. \psi_1) \longrightarrow (\forall x. \psi_2)$. Hence,

$$(\forall \underline{x}. \text{WT } \llbracket \varphi \rrbracket) \longrightarrow (\forall \underline{x}. \llbracket t \rrbracket \sqsubseteq \llbracket t' \rrbracket)$$

is valid. By the definition of \sqsubseteq on functions,

$$(\forall \underline{x}. \text{WT } \llbracket \varphi \rrbracket) \longrightarrow (\lambda \underline{x}. \llbracket t \rrbracket) \sqsubseteq (\lambda \underline{x}. \llbracket t' \rrbracket)$$

is valid. Soundness of

$$\frac{\forall \underline{x}. \varphi}{Q\underline{x} \cdot \underline{v}, t, \underline{v}' \sqsubseteq Q\underline{x} \cdot \underline{v}, t', \underline{v}'}$$

follows from monotonicity of Q and the fact that $\text{WT } \llbracket \forall \underline{x}. \varphi \rrbracket$ is equivalent to $\forall \underline{x}. \text{WT } \llbracket \varphi \rrbracket$. \square

Note that every strict operator is monotonic. Moreover, every operator and binder of Σ_1 is monotonic; for most operators this is obvious, because they are strict. In fact, all operators and binders available in Rodin are monotonic [20]. However, neither the well-definedness operator D is monotonic, nor the denotations of operator variables that take at least one argument. Hence, rewriting subterms is sound for the logic implemented by Rodin, but rewriting arguments of D or of operator variables is in general unsound.

4.4 Safety

In this section we address the question under which conditions directed rewriting is *safe*. Informally, an *unsafe* backwards step transforms a valid sequent into an invalid one and thus drives the proof into a dead end. We restrict the discussion to unconditional rules, assuming that conditional rules are applied only if their conditions can be solved.

Formally, an inference rule is *safe* iff validity of its consequent implies validity of all antecedents. The inference rules resulting from sound directed rewrite rules are in general unsafe: take the rule rewriting $\vdash \bullet$ to $\vdash \perp$ as an example.

An inference rule is *WS-safe* iff WS-validity of the consequent implies WS-validity of all antecedents. Note that the inference rules resulting from sound directed rewrite rules are WS-safe. Hence, if (i) the sequent at hand is WS-valid, and (ii) only WS-safe rules are applied, then only WS-valid (and therefore valid) sequents will arise during the proof attempt. We will examine under which circumstances Conditions (i) and (ii) are true.

Note that $\underline{\psi} \vdash \varphi$ is valid iff $D(\underline{\psi}), D(\varphi), \underline{\psi} \vdash \varphi$ is WS-valid (and therefore also valid). So it is straightforward to establish Condition (i) by a mild (and validity preserving) modification of the proof obligation at hand.

Of course, inference rules may be only safe but not WS-safe: consider $\frac{\vdash \bullet}{\vdash \top}$.

Such rules need to be avoided to fulfill Condition (ii). We can always transform a safe inference rule into a WS-safe one (in a soundness preserving manner) by adding well-definedness conditions to the antecedents. That is actually unnecessary for the inference rules available in Rodin: by inspecting the list of available rules, we observe that every inference rule of Rodin is safe iff it is WS-safe. So Condition (ii) is fulfilled in Rodin if the user avoids applying unsafe rules.

In summary, directed rewriting is unsafe in general; but after slight modifications of the proof obligation and the proof calculus, directed rewriting is safe whenever only safe inference rules are applied within proofs. For Rodin, such modifications of the proof calculus are unnecessary².

4.5 Practical Relevance

We have seen that directed rewriting is sound if (i) sequents are interpreted in SW-semantics and (ii) the operators and binders “surrounding” the term to be

² We do not make a statement about required modifications of Rodin’s proof obligations, because we are not aware of a document specifying them.

rewritten are monotonic. There is no agreement in the literature on which of the four sequent semantics is best; we view directed rewriting as a novel argument in favor of SW-semantics. Clearly, non-monotonic operators are sometimes useful; e.g., we will see how to apply them for reasoning about the soundness of rewrite rules (see Sect. 5). We have however experienced that monotonicity is an acceptable restriction for many applications. Other researchers seem to make similar experiences: Jones et al. [10] point out that non-monotonic operators are “not needed for specifying software systems” or “seldom employed in proofs”. PVS [17] supports only monotonic operators and binders.

To understand how often directed rewrite rules arise in practice, we have analyzed the rewrite rules available in the Rodin platform. New rules for Rodin’s term rewriter are chosen and implemented based on the requests of Rodin users. The set of available rules therefore reflects which rules are important in practice. The details of our analysis can be found in Sect. B.2.

We say that a sound directed rewrite rule is *truly* directed iff its condition does *not* imply equivalence of its left- and right-hand side. In total, Rodin implements 453 unconditional directed rewrite rules; 165 of them (about 36%) are truly directed. Without directed rewriting, Rodin would have to prove well-definedness of the left-hand side whenever applying one of these rules. Moreover, Rodin implements 53 conditional directed rewrite rules of which 42 (about 79%) are truly directed. Thus, in a significant number of cases directed rewriting makes conditional rewrite rules unconditional or weakens the condition of a rewrite rule. We therefore conclude that directed rewriting constitutes an important optimization of Rodin’s term rewriter.

The reader may have the impression that directed rewriting mainly compensates for problems introduced by the fact that Event-B’s logic explicitly supports partial functions. But this is not entirely true. In logics of total functions it is quite common to approximate partial functions by underspecified total functions. In such a logic, $x \bmod 0$ denotes an unspecified integer. Therefore $x \bmod x$ is equivalent to 0 only if $\neg(x = 0)$. In Event-B and with directed rewriting, we can avoid the condition $\neg(x = 0)$ by restating the rule as $\$x \bmod \$x \sqsubseteq 0$. Thus, directed rewriting not only compensates for problems introduced by explicit partiality, but also makes rules unconditional that are commonly conditional in logics of total functions. In the case of Rodin, there are 35 such rules.

5 Proving User Supplied Rules Sound

Rodin provides a generic term rewriter to which the user can supply new rewrite rules [11]. The term rewriter accepts a new rule only if the user formally proves its soundness. Rules with conjunctions, disjunctions, implications, universal or existential quantifiers on the left-hand side are however rejected, because it has been unclear how to generate the required soundness proof obligations. Operator variables with non-empty argument types are not supported either. Below, we show how to overcome these limitations for a practically relevant class of rewrite rules.

In a logic with the well-definedness operator D and operator variables, it is straightforward to express soundness proof obligations:

Lemma 6. *The directed rewrite rule $\frac{\varphi}{t \sqsubseteq u}$ is sound iff $D(\varphi) \Rightarrow \varphi, D(t) \vdash D(u) \wedge t = u$ is valid.*

However, Rodin supports neither the operator D nor operator variables, which makes Lemma 6 inapplicable. A way out would be to make the operator D and operator variables available. Unfortunately, such a change would not come cheaply, because monotonicity assumptions are hard-wired in several places, in particular in Rodin's term rewriter. Therefore, we develop a method for expressing soundness proof obligations with only monotonic operators and binders and without operator variables.

Before going into the gory details, let us consider an example. For convenience, we introduce the operator `restrict` with argument type (\mathcal{B}, α) and result type α whose denotation is given by $\llbracket \text{restrict} \rrbracket \varphi x = (\text{if } \top \varphi \text{ then } x \text{ else } \bullet)$. Consider the rule $(\exists x \cdot x = \$y) \sqsubseteq \top$; Lemma 6 suggests, after slight simplifications, the following soundness proof obligation:

$$D(\exists x \cdot x = \$y) \vdash \exists x \cdot x = \$y. \quad (9)$$

To eliminate the operator variable $\$y$, we extend the underlying signature by the operators `dy` and `sy`; `dy` is of type \mathcal{B} and `sy` has the same type as $\$y$. The denotations of `dy` and `sy` are left unspecified, except that we require $\text{WD } \llbracket \text{dy} \rrbracket$ and $\text{WD } \llbracket \text{sy} \rrbracket$ to be valid. Intuitively, $\text{restrict}(\text{dy}, \text{sy})$ is equivalent to $\$y$. Then, (9) is equivalent to

$$D(\exists x \cdot x = \text{restrict}(\text{dy}, \text{sy})) \vdash \exists x \cdot x = \text{restrict}(\text{dy}, \text{sy}). \quad (10)$$

The goal of (10) is already built from monotonic operators and binders. The hypothesis of (10) is equivalent to

$$(\exists x \cdot \text{dy} \wedge x = \text{restrict}(\text{dy}, \text{sy})) \vee (\forall x \cdot \text{dy} \wedge \neg(x = \text{restrict}(\text{dy}, \text{sy}))).$$

Note that we are unable to directly eliminate D from (9), because that would require us to eliminate D from terms of the form $D(\$y)$.

The following theorem shows how to establish the transition from (9) to (10) in general, provided operator variables are applied only to ordinary variables.

Theorem 7. *Suppose all occurrences of the operator variable $\$f \text{ : } \underline{\nu} \rightsquigarrow \mu$ in the sequent Γ take only ordinary variables as arguments. Extend the underlying signature by the operators $\text{df} \text{ : } \underline{\nu} \rightsquigarrow \mathcal{B}$ and $\text{sf} \text{ : } \underline{\nu} \rightsquigarrow \mu$, and the underlying structure such that $\llbracket \text{df} \rrbracket$ is specified by $\text{WD}(\llbracket \text{df} \rrbracket \underline{x}) \longleftrightarrow \text{WD } \underline{x}$ ³ and $\llbracket \text{sf} \rrbracket$ by $\text{WD}(\llbracket \text{sf} \rrbracket \underline{x}) \longleftrightarrow \text{WD } \underline{x}$. Then Γ is valid w.r.t. the original structure iff $\Gamma[\$f(\$x) := \text{restrict}(\text{df}(\$x), \text{sf}(\$x))]$ is valid w.r.t. the extended structure.*

³ This is achieved by `consts` $\llbracket \text{df} \rrbracket$ `ax_specification` $(\llbracket \text{df} \rrbracket)$ $\text{WD}(\text{df } \underline{x}) \longleftrightarrow \text{WD } \underline{x}$.

Here, $\text{WD } \underline{x}$ abbreviates $\text{WD } x_1 \wedge \cdots \wedge \text{WD } x_{|\underline{x}|}$. A substitution is applied to a sequent by simultaneous application to the hypotheses and the goal.

Proof. Clearly, validity of Γ w.r.t. the original, i.e., unextended, structure implies validity of Γ w.r.t. the extended structure. Validity of Γ w.r.t. the extended structure implies validity of $\Gamma[\$f(\underline{\$x}) := \text{restrict}(\text{df}(\underline{\$x}), \text{sf}(\underline{\$x}))]$ w.r.t. the extended structure, because validity is closed under substitution.

For the converse direction, suppose that $\Gamma[\$f(\underline{\$x}) := \text{restrict}(\text{df}(\underline{\$x}), \text{sf}(\underline{\$x}))]$ is valid w.r.t. the extended structure. We denote the original structure by $(M, \llbracket \cdot \rrbracket)$ and the extended structure by $(M', \llbracket \cdot \rrbracket)$. Hence, by Lemma 3,

$$\llbracket \Gamma \rrbracket[\$f := (\lambda \underline{\$x}. \llbracket \text{restrict}(\text{df}(\underline{\$x}), \text{sf}(\underline{\$x})) \rrbracket)]$$

is valid in M' . By unfolding the definition of $\llbracket \text{restrict} \rrbracket$, we obtain that

$$\llbracket \Gamma \rrbracket[\$f := (\lambda \underline{\$x}. \text{if } \top \llbracket \text{df} \rrbracket \underline{\$x} \text{ then } \llbracket \text{sf} \rrbracket \underline{\$x} \text{ else } \bullet)]$$

is valid in M' . By replacing $\llbracket \text{sf} \rrbracket$ and $\llbracket \text{df} \rrbracket$ by fresh variables dv and sv and universal generalization, we conclude that

$$\begin{aligned} & \forall dv sv. (\forall \underline{\$x}. \text{WD}(dv \underline{\$x}) \longleftrightarrow \text{WD } \underline{\$x}) \\ & \quad \wedge (\forall \underline{\$x}. \text{WD}(sv \underline{\$x}) \longleftrightarrow \text{WD } \underline{\$x}) \\ & \longrightarrow \llbracket \Gamma \rrbracket[\$f := (\lambda \underline{\$x}. \text{if } \top (dv \underline{\$x}) \text{ then } sv \underline{\$x} \text{ else } \bullet)] \end{aligned} \tag{11}$$

is valid in the theory M underlying the *original* structure. We deduce validity of $\llbracket \Gamma \rrbracket$ in M by instantiating dv and sv as follows:

$$\begin{aligned} dv & := (\lambda \underline{\$x}. \text{if } \text{WD } \underline{\$x} \text{ then } (\text{WD}(\$f \underline{\$x}))\uparrow \text{ else } \bullet), \\ sv & := (\lambda \underline{\$x}. \text{if } \text{WD } \underline{\$x} \text{ then } ((\$f \underline{\$x})\downarrow)\uparrow \text{ else } \bullet). \end{aligned} \tag{12}$$

The operator \downarrow takes x of type $\nu\uparrow$ and yields sx of type ν such that $sx\uparrow = x$, if $\text{WD } x$ is true; otherwise, if $\text{WD } x$ is false, $x\downarrow$ is left unspecified.

Note that the premise of (11) is true under the instantiations in (12). It can also be checked that the conclusion of (11) under the instantiations in (12) is equivalent to $\llbracket \Gamma \rrbracket$; that last step relies on the assumption that $\$f$ is applied only to ordinary variables in Γ , because ordinary variables are well-defined. \square

After applying Theorem 7 to eliminate operator variables, it remains to eliminate well-definedness conditions $\text{D}(t)$. If t is a term over Rodin's signature, it is well-known how to do that (see [6]). The procedure in [6] can be easily extended to terms containing restrict and the ad-hoc operators arising from Theorem 7. The result is a soundness proof obligation containing only monotonic operators and binders and not containing operator variables.

To understand the relevance of the machinery described above, we consider the rewrite rules implemented by Rodin as a benchmark. There are still rules for which the soundness proof obligations cannot be expressed, even with the above procedure. These rules fall in at least one of the following two categories:

1. rules containing associative operators with an arbitrary number of arguments; an example is the rule $\$ \varphi_1 \wedge \dots \wedge \$ \psi \wedge \dots \wedge \$ \varphi_n \Rightarrow \$ \psi \sqsubseteq \top$.
2. rules that violate the proviso of Theorem 7. An example is the rule
$$\frac{D(\$y)}{\forall x \cdot x = \$y \Rightarrow \$\varphi(x) \sqsubseteq \$\varphi(\$y)}$$
, because the $\$y$ in $\$\varphi(\$y)$ is not an ordinary variable.

Only five rewrite rules fall into the second category (cf. Sect. B.3).

We admit that our procedure for expressing soundness proof obligations is not straightforward. Rodin’s commitment to monotonicity has a price!

6 Related Work

PVS uses predicate subtypes and dependent types to represent partial functions: a partial function is viewed as total function over a restricted domain. Although the semantics of PVS [17] is presented quite differently from the semantics of Event-B’s logic, we observe the following similarities: if we view type-correctness in PVS as well-definedness in Event-B, then the operators and binders available in PVS are monotonic, operator variables are unavailable, and PVS inference rules are (SW-)sound. Hence, directed rewriting is sound in PVS and the problem and solution concerning reasoning about soundness (cf. Sect. 5) apply.

We are not aware of a paper devising the foundations of term rewriting in PVS. From the prover guide [21, p. 86], we have the impression that PVS implements directed rewrite rules such as $0 * \$x \sqsubseteq 0$. However, the techniques discussed in Sect. 5 seem to be unavailable: we tried to introduce the rule $\$x \in \emptyset \sqsubseteq \perp$ (and some others), but PVS insisted on checking well-definedness of $\$x$ when applying this rule. Therefore we believe that our results can be used to improve term rewriting in PVS.

LPF [10], the logic underlying VDM, is based on SS-semantics. It is therefore sound to apply directed rewrite rules to hypotheses belonging to LPF’s monotonic fragment, but unsound to apply directed rewrite rules to goals. Dawson [7] already defines a version of unconditional directed rewriting for quantifier free LPF; we in particular complement his work by providing evidence that directed rewriting is powerful in practice and by our results on safety. The complications when reasoning about soundness of rewrite rules addressed in Sect. 5 do not arise in LPF, because LPF supports non-monotonic operators.

Monotonicity is an important concept in LCF (see e.g. [18]); hence there is a chance of applying directed rewriting. Conjectures in LCF seem however often to be built from \equiv and \sqsubseteq (in Event-B terminology). The challenge is to reorganize proofs in terms of sequents that have SW-semantics. More research is needed to understand whether directed rewriting has applications in LCF.

Maamria and Butler [11] devise directed rewriting for the untyped fragment of Event-B, i.e., the fragment with exactly one non-boolean type. In their setting, the left- and right-hand sides of rewrite rules are built from operator variables with empty argument types and strict operators. In particular, they do not

support rewrite rules whose left- or right-hand sides involve boolean connectives or binders. We overcome these restrictions and correct a flaw concerning the application of conditional rewrite rules. (The proviso above (4.2) on p. 11 allows one to apply conditional rewrite rules in an unsound way.)

Our research has been greatly inspired by term rewriting in Isabelle [14].

7 Conclusions and Future Work

We have devised the foundations of directed rewriting, a technique that rewrites not only terms to equivalent terms but also ill-defined to well-defined terms. To make concrete statements about the practical impact of directed rewriting, we have focused on a particular logic of partial functions, namely the logic of Event-B. Applications to other logics of partial functions are described in Sect. 6.

As a prerequisite of our investigations, we have defined a semantics for Event-B's logic (see Sect. 3). Soundness of directed rewriting in Event-B, as manifested by Lemmas 4 and 5, is an immediate consequence of our presentation of semantics, and the observations that sequents have SW-semantics (see Sect. 4), and operators and binders are monotonic (see Sect. 4.3). In particular, we have shown how to apply conditional rewrite rules to the arguments of binders, which is currently not supported in Rodin. Our work on semantics has also helped us to spot inconsistency bugs in earlier versions of Rodin's theorem prover and identify dispensable preconditions of rewrite rules (see Sect. B.4).

To understand the practical impact of directed rewriting, we have analyzed the directed rewrite rules implemented by Rodin (see Sect. 4.5). Our conclusion is that directed rewriting makes a significant number of conditional rewrite rules unconditional and thus constitutes an important optimization. Directed rewriting is unsafe in general: it may transform a valid sequent into an invalid one during a backwards proof. This unsafety can however be easily avoided, for reasons explained in Sect. 4.4.

Monotonicity is an important prerequisite for soundness of directed rewriting, but a disturbing restriction when reasoning about soundness of rewrite rules. In Sect. 5 we have shown how to express soundness proof obligations for a practically important class of rewrite rules using only monotonic operators and binders. In particular, we have pointed out how to overcome limitations of Rodin concerning rules that contain boolean connectives or binders.

We are currently integrating Isabelle as an automated theorem prover into Rodin⁴. Our logical embedding of Event-B's logic into HOL serves as a basis. This Isabelle based theorem prover already provides a restricted version of directed rewriting.

Acknowledgements. The author would like to thank several people for their helpful feedback on preliminary versions of this paper: David Basin, Andreas Fürst, Matus Harvan, Thai Son Hoang, Felix Klaedtke, Ognjen Maric, Simon

⁴ http://wiki.event-b.org/index.php/Export_to_Isabelle

Meier, Patrick Schaller, Benedikt Schmidt, and Laurent Voisin. The author is also grateful for the numerous useful suggestions of the anonymous reviewers.

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge (2010)
2. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* 12(6), 447–466 (2010)
3. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory. Kluwer (2002)
4. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. *Acta Inf.* 21, 251–269 (1984)
5. Butler, M., Maamria, I.: Mathematical extension in Event-B through the Rodin theory component (2010), <http://deploy-eprints.ecs.soton.ac.uk/251>
6. Darvas, Á., Mehta, F., Rudich, A.: Efficient well-definedness checking. In: *IJCAR*. LNCS, vol. 5195, pp. 100–115. Springer (2008)
7. Dawson, J.E.: Simulating term-rewriting in LPF and in display logic. In: *Supplementary Proc. of TPHOLs*. pp. 47–62. Australian National University (1998)
8. Gordon, M.J.C., Melham, T.F.: Introduction to HOL. Cambridge (1993)
9. Hindley, J.R., Seldin, J.P.: Lambda-Calculus and Combinators. Cambridge (2008)
10. Jones, C.B., Middelburg, C.A.: A typed logic of partial functions reconstructed classically. *Acta Inf.* 31(5), 399–430 (1994)
11. Maamria, I., Butler, M.: Rewriting and well-definedness within a proof system. In: *PAR*. EPTCS, vol. 43, pp. 49–64 (2010)
12. Mehta, F.: A practical approach to partiality - a proof based approach. In: *ICFEM*. LNCS, vol. 5256, pp. 238–257. Springer (2008)
13. Metayer, C., Voisin, L.: The Event-B mathematical language (2009), <http://deploy-eprints.ecs.soton.ac.uk/11>
14. Nipkow, T.: Term rewriting and beyond - theorem proving in isabelle. *Formal Asp. Comput.* 1(4), 320–338 (1989)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
16. Owe, O.: Partial logics reconsidered: A conservative approach. *Formal Asp. Comput.* 5(3), 208–223 (1993)
17. Owre, S., Shankar, N.: The formal semantics of PVS (1999), <http://pvs.csl.sri.com/papers/csl-97-2/csl-97-2.ps>
18. Paulson, L.C.: Logic and Computation: Interactive Proof with Cambridge LCF. Cambridge (1987)
19. Paulson, L.C.: The foundation of a generic theorem prover. *J. Autom. Reasoning* 5(3), 363–397 (1989)
20. Schmalz, M.: The logic of Event-B (2011), <ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/698.pdf>
21. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS prover guide (2001), <http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf>
22. Silva, R., Butler, M.: Supporting reuse of Event-B developments through generic instantiation. In: *ICFEM*. LNCS, vol. 5885, pp. 466–484. Springer (2009)
23. Wenzel, M.: Type classes and overloading in higher-order logic. In: *TPHOLs*. LNCS, vol. 1275, pp. 307–322. Springer (1997)

A Proofs about Substitutions

The result of applying a type substitution σ to a term is defined by the following conditions:

1. $(x \varepsilon \nu)\sigma = x \varepsilon (\nu\sigma)$,
2. $(f(\underline{t}) \varepsilon \mu)\sigma = f(\underline{t}\sigma) \varepsilon (\mu\sigma)$,
3. $(\$f(\underline{t}) \varepsilon \mu)\sigma = f(\underline{t}\sigma) \varepsilon (\mu\sigma)$,
4. $((Q\underline{x} \cdot \underline{t}) \varepsilon \xi)\sigma = (Q\underline{x}\sigma \cdot \underline{t}\sigma) \varepsilon (\xi\sigma)$.

We adopt the convention that $\underline{t}\sigma$ abbreviates $t_1\sigma, \dots, t_{|\underline{t}|}\sigma$. In 4, we assume that the names of the bound variables \underline{x} are chosen such that (i) different variables in \underline{x} have different names, and (ii) the names of free variables of $(Q\underline{x} \cdot \underline{t})$ differ from the variable names in \underline{x} . Proviso (ii) has the effect that $(\forall x \varepsilon \alpha \cdot x \varepsilon \mathcal{B})[\alpha := \mathcal{B}]$ equals $(\forall y \varepsilon \mathcal{B} \cdot y \varepsilon \mathcal{B})$. Proviso (i) serves a similar purpose.

For the definition of substitutions in HOL, we refer to [8,9]. We denote variable substitutions in HOL by $[\underline{x} := \underline{t}]$ and type substitutions by $[\underline{\alpha} := \underline{\nu}]$.

Proof (of Lemma 2). It is straightforward to show $\llbracket \nu\sigma \rrbracket = \llbracket \nu \rrbracket [\sigma]$ for an Event-B type ν and a type substitution σ .

We prove by structural induction over terms u that $\llbracket u\sigma \rrbracket = \llbracket u \rrbracket [\sigma]$. If u is an ordinary variable, the proof proceeds as follows:

$$\begin{aligned} \llbracket (x \varepsilon \nu)\sigma \rrbracket &= \llbracket x \varepsilon (\nu\sigma) \rrbracket = (x :: \llbracket \nu\sigma \rrbracket) \uparrow = (x :: (\llbracket \nu \rrbracket [\sigma])) \uparrow \\ &= (x :: \llbracket \nu \rrbracket) \uparrow [\sigma] = \llbracket x \varepsilon \nu \rrbracket [\sigma]. \end{aligned}$$

The cases of operators and operator variables proceed similarly and are therefore omitted. If u is of the form $(Q\underline{x} \cdot \underline{t}) \varepsilon \xi$, the proof proceeds as follows:

$$\begin{aligned} &\llbracket ((Q\underline{x} \cdot \underline{t}) \varepsilon \xi)\sigma \rrbracket \\ &= \llbracket ((Q\underline{x}\sigma \cdot \underline{t}\sigma) \varepsilon (\xi\sigma)) \rrbracket \\ &= (\llbracket Q \rrbracket (\lambda \underline{x} :: \llbracket \underline{\nu}\sigma \rrbracket. \llbracket t_1\sigma \rrbracket) \dots (\lambda \underline{x} :: \llbracket \underline{\nu}\sigma \rrbracket. \llbracket t_{|\underline{t}|}\sigma \rrbracket)) :: \llbracket \xi\sigma \rrbracket \uparrow \\ &= (\llbracket Q \rrbracket (\lambda \underline{x} :: (\llbracket \underline{\nu} \rrbracket [\sigma]). \llbracket t_1 \rrbracket [\sigma]) \dots (\lambda \underline{x} :: (\llbracket \underline{\nu} \rrbracket [\sigma]). \llbracket t_{|\underline{t}|} \rrbracket [\sigma])) :: (\llbracket \xi \rrbracket [\sigma]) \uparrow \\ &= ((\llbracket Q \rrbracket (\lambda \underline{x} :: \llbracket \underline{\nu} \rrbracket. \llbracket t_1 \rrbracket) \dots (\lambda \underline{x} :: \llbracket \underline{\nu} \rrbracket. \llbracket t_{|\underline{t}|} \rrbracket))) :: (\llbracket \xi \rrbracket) \uparrow [\sigma] \\ &= \llbracket (Q\underline{x} \cdot \underline{t}) \varepsilon \xi \rrbracket [\sigma]. \end{aligned}$$

The equality between the fourth and fifth line is justified by the definition of type substitutions in HOL. HOL stipulates a similar proviso on the names of bound variables \underline{x} as Event-B. The proviso in HOL is entailed by the proviso in Event-B. \square

Proof (of Lemma 3). As a first step, we consider operator substitutions of the form $[\$x_1 := u_1, \dots, \$x_n := u_n]$. It is straightforward to show that such substitutions satisfy the assertion of the lemma.

Now we prove the general case. Suppose the operator substitution σ is of the form

$$[\$f_1(\underline{\mathbf{x}}^1) := u_1, \dots, \$f_n(\underline{\mathbf{x}}^n) := u_n].$$

We show by structural induction over terms t' that $\llbracket t'\sigma \rrbracket = \llbracket t' \rrbracket \llbracket \sigma \rrbracket$.

The cases that t' is an ordinary variable, or takes the form $f(\underline{t})$ or $\$g(\underline{t})$, where $\$g$ differs from $\$f_i$, for $1 \leq i \leq n$, are straightforward and therefore omitted. If t' is of the form $\$f_i(\underline{t})$, for $1 \leq i \leq n$, the proof proceeds as follows:

$$\begin{aligned}
\llbracket \$f_i(\underline{t})\sigma \rrbracket &= \llbracket u_i[\$x_1^i := t_1\sigma, \dots, \$x^i_{|\underline{\$x}^i|} := t_{|\underline{t}|}\sigma] \rrbracket \\
&= \llbracket u_i[\$x_1^i := \llbracket t_1\sigma \rrbracket, \dots, \$x^i_{|\underline{\$x}^i|} := \llbracket t_{|\underline{t}|}\sigma \rrbracket] \rrbracket \\
&= \llbracket u_i[\$x_1^i := \llbracket t_1 \rrbracket \llbracket \sigma \rrbracket, \dots, \$x^i_{|\underline{\$x}^i|} := \llbracket t_{|\underline{t}|} \rrbracket \llbracket \sigma \rrbracket] \rrbracket \\
&= (\lambda \underline{\$x}^i. \llbracket u_i \rrbracket) (\llbracket \underline{t} \rrbracket \llbracket \sigma \rrbracket) \\
&= (\$f_i \llbracket \underline{t} \rrbracket) \llbracket \sigma \rrbracket \\
&= \llbracket \$f_i(\underline{t}) \rrbracket \llbracket \sigma \rrbracket.
\end{aligned}$$

The step from the first to the second line is justified by our independent proof of the assertion, when the operator variables involved in the substitution have empty argument types.

If t' is of the form $Q\underline{y} \cdot \underline{t}$, the proof proceeds as follows:

$$\begin{aligned}
\llbracket (Q\underline{y} \cdot \underline{t})\sigma \rrbracket &= \llbracket Q\underline{y} \cdot \underline{t}\sigma \rrbracket \\
&= \llbracket Q \rrbracket (\lambda \underline{y}. \llbracket t_1\sigma \rrbracket) \dots (\lambda \underline{y}. \llbracket t_{|\underline{t}|}\sigma \rrbracket) \\
&= \llbracket Q \rrbracket (\lambda \underline{y}. \llbracket t_1 \rrbracket \llbracket \sigma \rrbracket) \dots (\lambda \underline{y}. \llbracket t_{|\underline{t}|} \rrbracket \llbracket \sigma \rrbracket) \\
&= (\llbracket Q \rrbracket (\lambda \underline{y}. \llbracket t_1 \rrbracket) \dots (\lambda \underline{y}. \llbracket t_{|\underline{t}|} \rrbracket)) \llbracket \sigma \rrbracket \\
&= \llbracket Q\underline{y} \cdot \underline{t} \rrbracket \llbracket \sigma \rrbracket.
\end{aligned}$$

The equality of the third and fourth line rests on the assumption in the definition of operator substitutions that no element of \underline{y} occurs free in u_1, \dots, u_n . \square

B Statistics on Rules Implemented by Rodin

Rodin's inference and rewrite rules can be found on the following pages:

http://wiki.event-b.org/index.php?title=Inference_Rules&oldid=8898

http://wiki.event-b.org/index.php?title=Set_Rewrite_Rules&oldid=8523

http://wiki.event-b.org/index.php?title=Relation_Rewrite_Rules&oldid=8891

http://wiki.event-b.org/index.php?title=Arithmetic_Rewrite_Rules&oldid=8892

Our analysis is based on the version of April 5, 2011; we also consider rules proposed for implementation.

B.1 Semantics of Conditions

The documentation clearly distinguishes inference rules from unconditional rewrite rules. Conditional rewrite rules are represented either as inference rules or as

rewrite rules with side-conditions. Therefore, for our analysis we had to change the representation of rules as in the following examples:

$$\begin{array}{l} \text{fin_l_lower_bound_l:} \quad \frac{\text{finite}(\$S)}{\exists n \cdot \forall x \cdot x \in \$S \Rightarrow n \leq x \sqsubseteq \top} \\ \\ \text{ov_setenum_l:} \quad \frac{\$G = \$E}{(\$f \triangleleft \{\$E \mapsto \$F\})(\$G) \sqsubseteq \$F} \\ \\ \text{and} \quad \frac{\neg(\$G = \$E)}{(\$f \triangleleft \{\$E \mapsto \$F\})(\$G) \sqsubseteq \{\$E\} \triangleleft \$f(\$G)} \\ \\ \text{fin_subseteq_r:} \quad \frac{D(\$T) \wedge \$S \subseteq \$T \wedge \text{finite}(\$T)}{\text{finite}(\$S) \sqsubseteq \top} \\ \\ \text{simp_card_setminus_l:} \quad \frac{\text{finite}(\$S)}{\text{card}(\$S \setminus \$T) \sqsubseteq \text{card}(\$S) - \text{card}(\$S \cap \$T)} \end{array}$$

Below is the list of conditional rewrite rules. In total there are 53 such rules; for 15 of them, which are marked with *, the condition could be weakened if conditions φ were interpreted as $\top[\varphi]$. On a closer look, observe that the rules marked with a * are exactly those rules whose conditions contain operator variables that are not contained in the left-hand side.

fun_goal*, neg_in_l, subset_inter, in_inter, notin_inter, fin_l_lower_bound_l, fin_l_lower_bound_r, fin_l_upper_bound_l, fin_l_upper_bound_r, ov_setenum_l (2 rules), ov_l (2 rules), dis_binter_r, dis_setminus_r, sim_rel_image_r, sim_fcomp_r, fin_subseteq_r*, fin_binter_r, fin_kinter_r, fin_qinter_r, fin_setminus_r, fin_rel*, fin_rel_img_r, fin_rel_ran_r, fin_rel_dom_r, fin_fun_dom*, fin_run_ran*, fin_fun_img_r*, fin_fun_ran_r*, fin_fun_dom_r*, fin_lt_0, fin_ge_0, card_interv, card_empty_interv, deriv_le_card, deriv_ge_card, deriv_lt_card, deriv_gt_card, deriv_equal_card, simp_card_setminus_l, simp_card_cprod_l, one_point_l (with \forall), one_point_l (with \exists), simp_funimage_domres*, simp_funimage_domsb*, simp_funimage_ranres*, simp_funimage_ransub*, simp_funimage_setminus*, deriv_dom_totalrel*, deriv_ran_surjrel*, simp_card_setminus, simp_card_setminus_setenum

B.2 Relevance of Directed Rewriting for Rodin

Rodin's proof rules are organized in four categories: inference rules, set rewrite rules, relation rewrite rules, and arithmetic rewrite rules. Table 1 summarizes our observations on unconditional rewrite rules. The column TDCTF counts rules that are truly directed and conditional if restated in a logic of total functions, assuming that this logic of total functions approximates partial functions as underspecified total functions. In fact, there are rules like $(2 * \$x) \div (2 * \$y) \sqsubseteq \$x \div \y that are sound in Event-B's logic and not truly directed. Yet, its classical counterpart $(2*x) \div (2*y) \equiv x \div y$ is unsound, because $2 \div 0$ and $1 \div 0$ may denote different numbers. Such rules are conditional in many logics of total functions

and unconditional in the logic of Event-B; this is however not a consequence of directed rewriting, but a consequence of the fact that there is only one ill-defined value. We therefore do not count such rules in the TDCTF column.

Category	Total	Truly Directed	TDCTF
Inference Rules	0	0	0
Set Rewrite Rules	166	59	0
Relation Rewrite Rules	198	65	12
Arithmetic Rewrite Rules	89	41	23
Total	453	165	35

Table 1. Statistics on unconditional rewrite rules

In the following, we list the unconditional truly directed rewrite rules. The notation “rule_name (+x)” designates the rule with name “rule_name” and the x following rules.

Category Set Rewrite Rules.

simp_multi_and_not, simp_multi_or_not, simp_multi_imp (+1), simp_multi_eqv (+1), simp_multi_equal (+2), simp_type_subseteq, simp_special_subseteq (+7), simp_special_binter, simp_special_bunion, simp_multi_setminus, simp_special_setminus_l (+1), simp_kinter_pow, simp_special_cprod_r (+2), simp_subseteq_compset_l (+4), distri_subseteq_bunion_sing, simp_finite_setenum (+1), simp_finite_qunion, deriv_finite_cprod, simp_finite_upto, simp_finite_lambda, simp_type_in, simp_special_subset_r, simp_multi_subset (+3), def_in_mapsto, def_in_pow1 (+10), deriv_subseteq_setminus_r (+1).

Category Relation Rewrite Rules.

simp_dom_setenum, simp_ran_setenum, simp_type_overl_cprod (+2), simp_special_ranres_r (+1), simp_special_domsup_r (+3), simp_special_ransub_l (+3), simp_special_fcomp, simp_special_bcomp, simp_special_dprod_r (+1), simp_special_pprod_r (+1), simp_special_relimage_r (+1), simp_multi_relimage_cprod_sing (+2), simp_multi_relimage_domsup, simp_special_rel_r (+5), simp_funimage_lambda (+10), simp_funimage_funimage_converse (+2), deriv_fcomp_sing, def_in_domres (+5), def_in_dprod (+1), def_in_reldom (+4), def_in_tinj (+3).

Category Arithmetic Rewrite Rules.

simp_special_mod_0 (+1), simp_min_bunion_sing (+3), simp_card_sing (+1), simp_card_lambda, simp_lit_ge_card_1 (+8), def_equal_min (+1), simp_multi_minus (+10), simp_special_div_0, simp_special_expn_1_l (+8).

In the following, we list the rules we counted in the TDCTF column.

Category Relation Rewrite Rules.

simp_funimage_lambda, simp_in_funimage (+7),
simp_funimage_funimage_converse (+2).

Category Arithmetic Rewrite Rules.

simp_min_bunion_sing (+3), simp_special_equal_card, simp_card_lambda,
simp_lit_ge_card_l (+8), def_equal_min (+1), simp_special_div_0,
simp_special_expn_l_l (+1), simp_multi_div (+2).

Table 2 summarizes our observations on conditional rewrite rules. The inference rule category contains conditional rewrite rules that are represented as inference rules.

Category	Total	Truly Directed
Inference Rules	44	35
Set Rewrite Rules	0	0
Relation Rewrite Rules	7	7
Arithmetic Rewrite Rules	2	0
Total	53	42

Table 2. Statistics on conditional rewrite rules

In the following, we list the truly directed conditional rewrite rules.

Category Inference Rules.

fun_goal, neg_in_l, subset_inter (+6), ov_setenum_l (2 rules), ov_l (2 rules),
fin_subseteq_r (+10), fin_funimage_r (+2), fin_lt_0 (+1), card_empty_interv
(+5).

Category Relation Rewrite Rules.

simp_funimage_domres (+4), deriv_dom_totalrel (+1).

B.3 Soundness Proof Obligations

The method outlined in Sect. 5 cannot be applied to the following 5 rules, because the proviso of Theorem 7 is violated:

one_point_l (version with \forall), one_point_l (version with \exists),
simp_funimage_lambda, simp_compset_equal, simp_in_compset_onepoint.

B.4 Conditional Rules with Dispensable Conditions

For the following rules, the strictness side-condition can be dropped:

`dis_binter_r`, `dis_binter_l`, `dis_setminus_r`, `dis_setminus_l`, `simp_card_setminus_l`,
`simp_card_setminus_r`, `simp_card_cprod_l`, `simp_card_cprod_r`.

In the case of `ov_setenum_l`, the strictness side-condition is necessary for soundness, but can be avoided if the rule is stated as proposed in Sect. [B.1](#). Similar considerations apply for

`ov_setenum_r`, `ov_l`, `ov_r`, `card_interv`, and `card_empty_interv`.