Axiomatizing Reflective Logics and Languages*

Manuel G. Clavel and José Meseguer
Computer Science Laboratory
SRI International, Menlo Park CA 94025

Abstract

The very success and breadth of reflective techniques underscores the need for a general theory of reflection. At present what we have is a wide-ranging variety of reflective systems, each explained in its own idiosyncratic terms. Metalogical foundations can allow us to capture the essential aspects of reflective systems in a formalism-independent way. This paper proposes metalogical axioms for reflective logics and declarative languages based on the theory of general logics [34]. In this way, several strands of work in reflection, including functional, equational, Horn logic, and rewriting logic reflective languages, as well as a variety of reflective theorem proving systems are placed within a common theoretical framework. General axioms for computational strategies, and for the internalization of those strategies in a reflective logic are also given.

1 Introduction

Reflection is a fundamental idea. In logic it has been vigorously pursued by many researchers since the fundamental work of Gödel and Tarski (see the surveys [52, 53]). In computer science it has been present from the beginning in the form of universal Turing machines. Many researchers have recognized its great importance and usefulness in programming languages [54, 51, 60, 56, 23, 19, 30], in theorem-proving [62, 7, 48, 20, 2, 29, 14, 16], in concurrent and distributed computation [28, 41, 46], and in many other areas such as compilation, programming environments, operating systems, fault-tolerance, and databases (see [50] for a recent snapshot of research in reflection).

The very success and extension of reflective ideas underscores the need for conceptual foundations. This need is real enough, because what we have at present are specific *instances* of reflection, each explained in terms of the particular concepts available for it, such as lambda expressions, Horn clauses, objects and metaobjects, and so on. What we do not yet have is a general theory of reflection capable of unifying and interrelating all the instances, and of providing us with general criteria and concepts with which to judge the extent to which a particular system—for example a metacircular interpreter—exhibits some given reflective properties, or falls short of being reflective in certain respects.

We believe that *metalogical* foundations that make the particular logic of choice an easily changeable parameter can be very useful in this regard. The reason for this is that we can then hope to capture in a precise and formalism-independent way the essential

*Supported by Office of Naval Research Contract N00014-95-C-0225, National Science Foundation Grant CCR-9224005, and by the Information Technology Promotion Agency, Japan, as a part of the Industrial Science and Technology Frontier Program "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

features of reflection that intuitively appear to be shared by quite disparate languages and systems.

This paper is a first step in a planned longer-term effort to provide metalogical foundations for reflection in exactly this sense. Although we strongly conjecture that our axioms will be useful for reflection in general, and will apply as well to cases of computational reflection that on first inspection seem less amenable to logical specification, our aim in this paper is more modest. We focus here on the case of reflective logics and reflective declarative languages. In particular, our axioms unify quite different strands of research in reflection, including reflective functional, equational, Horn logic, and rewriting logic programming languages, and work on reflective theorem provers based on different logics.

A key concept in our axiomatic treatment of reflective logics is the notion of a *universal theory*, that is, a theory U that can simulate the deductions of all other theories in a class $\mathcal{C}$ of theories of interest. In particular, if U is one of the theories in the class $\mathcal{C}$, then U can simulate its own metalevel at the object level, and this process can be iterated *ad infinitum*, giving rise to a "reflective tower."

Of course, reflective phenomena admit of degrees, in that some languages may choose to represent only certain metalevel aspects of interest, and may do so in weaker or stronger ways. In our treatment this takes the form of increasingly stronger axiomatizations. Thus, in a weaker axiomatization a universal theory may not belong to the class of theories that it represents, so that the full power of having a reflective tower is lost. Another degree of representational freedom has to do with the metalevel aspects being represented. In a weaker axiomatization, only the entailment relation—holding between the axioms of a theory and the theorems it proves—may be represented, whereas with stronger axioms the actual *proofs* of theorems in the object theories may be represented as well. This gradation of strength in the axioms provides formalism-independent criteria to understand and compare the degrees of reflection exhibited by different logics and declarative languages, even when the corresponding formalisms differ greatly from each other.

The declarative foundations that we propose in this paper can naturally express the structural and behavioral aspects of computational reflection. The point is that, from a declarative perspective, computation *is* deduction. Therefore, the deductive mechanisms specified in the metatheory of a logical language play the role of an operational semantics or abstract machine on which the given logical language can be implemented. If the logic is reflective in our sense, then both the static, structural aspects of its metatheory—for example its theories—as well as the dynamic, behavioral aspects, such as its deductive mechanisms, can be reified. In this way, the logical language and the deductive engine implementing it become causally connected. Since a reflective logic in our sense naturally gives rise to a "reflective tower," it then becomes possible to modify at runtime not only object theories, but also the reified metatheory.

The metalogical foundations that we propose use concepts from the theory of general logics [34] that are recapitulated in Section 2. Section 3 then presents general axioms for reflective logics and languages, and explains how they are satisfied or not by a variety of declarative languages. We discuss declarative reflection for functional, equational, Horn, and rewriting logic languages, and explain the relationship with reflection for Turing machines. In the rewriting logic case, Appendix A explains in detail and illustrates with an example how reflection is achieved. A reflective language is particularly powerful if *strategies* for computing in it can also be represented inside the language; in Section 4 we give a general axiomatization of strategies and discuss specific examples. The paper ends with some concluding remarks in Section 5.

2 Axioms for General Logics and Declarative Languages

A general axiomatic theory of logics should adequately cover all the key ingredients of a logic. These include: a *syntax*, a notion of *entailment* of a sentence from a set of axioms, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. A flexible axiomatic notion of a *proof calculus*, in which proofs of entailments, not just the entailments themselves, are first class citizens should also be included. The theory of general logics [34, 27, 9] is a study of these different ingredients of a logic and their interrelations, and uses those general notions to obtain a general notion of declarative language based on a given logic.

For our present purposes it will be the notions of syntax, of entailment system and of proof calculus proposed in [34] that play a crucial role. We present below in summarized form the axioms characterizing these notions and that of declarative language. The axioms use the language of category theory, but do not require any acquaintance with categories beyond the basic notions of category, functor, and natural transformation.

2.1 Syntax

Syntax can typically be given by a *signature* Σ providing a grammar on which to build *sentences*. For first order logic, a typical signature consists of a list of function symbols and a list of predicate symbols, each with a prescribed number of arguments, which are used to build up the usual sentences. To allow maximum freedom and generality—so that, for example, a graphical language or other representations also counts as syntax—we should not be too particular about the style of signatures that a logic may have or about how sentences are defined relative to a given signature. It is enough to assume that for each logic there is a category **Sign** of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its sentences. We call the pair $(\mathbf{Sign}, sen)$ a *syntax*.

2.2 Entailment systems

For a given signature Σ in **Sign**, *entailment* (also called *provability*) of a sentence $\varphi \in sen(\Sigma)$ from a set of axioms $\Gamma \subseteq sen(\Sigma)$ is a relation $\Gamma \vdash \varphi$ which holds if and only if we can prove φ from the axioms Γ using the rules of the logic. We make this relation relative to a signature.

In what follows, $|\mathcal{C}|$ denotes the collection of objects of a category $\mathcal{C}$.

Definition 1 [34] An *entailment system* is a triple $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ such that

- $(\mathbf{Sign}, sen)$ is a syntax,
- $\vdash$ is a function associating to each $\Sigma \in |\mathbf{Sign}|$ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$ called Σ-*entailment* such that the following properties are satisfied:

1. *reflexivity*: for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
2. *monotonicity*: if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$ then $\Gamma' \vdash_\Sigma \varphi$,
3. *transitivity*: if $\Gamma \vdash_\Sigma \varphi_i$, for all $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$,
4. $\vdash$-*translation*: if $\Gamma \vdash_\Sigma \varphi$, then for any $H : \Sigma \to \Sigma'$ in **Sign** we have $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$. □

Definition 2 [34] Given an entailment system $\mathcal{E}$, its category **Th** of *theories* has as objects pairs $T = (\Sigma, \Gamma)$ with Σ a signature and $\Gamma \subseteq sen(\Sigma)$. A *theory morphism* $H : (\Sigma, \Gamma) \to (\Sigma', \Gamma')$ is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\varphi \in \Gamma$, then $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$.

A theory morphism $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ is called *axiom-preserving* if it satisfies the condition that $sen(H)(\Gamma) \subseteq \Gamma'$. This defines a subcategory $\mathbf{Th}_0$ with the same objects as **Th** but with morphisms restricted to be axiom-preserving theory morphisms. □

Note that we can extend the functor *sen* to a functor on theories by taking $sen(\Sigma, \Gamma) = sen(\Sigma)$.

2.3 Proof calculi

A given entailment system may admit many different proof calculi. For example, in first order logic we have Hilbert style, natural deduction and sequent calculi among others, and the way in which proofs are represented and generated by rules of deduction is different for each of these calculi. It is useful to make proofs relative to a given theory T whose axioms we are allowed to use in order to prove theorems.

A proof calculus associates to each theory T a *structure* $P(T)$ of proofs that use axioms of T as hypotheses. The structure $P(T)$ typically has an *algebraic structure* of some kind so that we can obtain new proofs out of previously given proofs by operations that mirror the rules of deduction of the calculus in question. We need not make a choice about the particular types of algebraic structures that should be allowed for different proof calculi; we can abstract from such choices by simply saying that for a given proof calculus there is a category **Str** of such structures and a functor $P : \mathbf{Th}_0 \longrightarrow \mathbf{Str}$ assigning to each theory T its structure of proofs $P(T)$. Of course, it should be possible to extract from $P(T)$ the underlying set *proofs*(T) of all the proofs of theorems of the theory T, and this extraction should be functorial. Also, each proof, whatever it is, should contain information about what theorem it is a proof of; this can be formalized by postulating a "projection function" π_T (parameterized by T in a natural way) that maps each proof $p \in$ *proofs*(T) to the sentence φ that it proves. Of course, each theorem of T must have at least one proof, and sentences that are not theorems should have no proof. To summarize, a *proof calculus* [34] consists of an entailment system together with:

- A functorial assignment P of a structure $P(T)$ to each theory T.
- An additional functorial assignment of a set *proofs*(T) to each structure $P(T)$.
- A natural function π_T assigning a sentence to each proof $p \in$ *proofs*(T) and such that, for Γ the axioms of T, a sentence φ is in the image of π_T if and only if $\Gamma \vdash \varphi$.

If $p \in proofs(T)$ is a proof of theorem φ, that is, $\pi_T(p) = \varphi$, we will express this by writing the decorated entailment $T \vdash p : \varphi$.

In practice it is quite common to encounter proof calculi of a specialized nature, in which only certain signatures are admissible as syntax, e.g., finite signatures; only certain sentences are allowed as axioms; and only certain sentences (possibly different from the axioms) are allowed as conclusions. The obvious reason for introducing such specialized calculi is that proofs are *more efficient* under the given restrictions. In computer science the choice between an efficient and an inefficient calculus may have dramatic practical consequences. For logic-based declarative languages, such calculi do (or should) coincide with what is called their *operational semantics*. They mark the difference between an intrinsically inefficient theorem prover and an efficiently implementable programming language. These considerations lead to the notion of a *proof subcalculus*, which is just like a proof calculus, except that appropriate restrictions have been imposed as follows:

- a subclass of *admissible signatures* is specified;
- for each admissible signature Σ, a family of sets $\Gamma \subseteq sen(\Sigma)$ called sets of *admissible axioms* is also specified;
- for each admissible signature Σ, a subset $conc(\Sigma) \subseteq sen(\Sigma)$ of sentences called *admissible conclusions* is specified;
- the functorial assignments $P(T)$ and $proofs(T)$, and the natural function π_T are just as those in a proof calculus, but are *restricted* to theories T having admissible signature and axioms, in such a way that π_T maps a proof p to an admissible conclusion, and each admissible conclusion has a proof.

In practice, of course, we are primarily interested in proof calculi and proof subcalculi that are *computationally effective*, i.e., that can in principle be implemented on a computer. This is axiomatized by the notion of an *effective proof subcalculus*; we do not give here the details, which can be found in [34].

2.4 Declarative Languages

In general, given a logic, what does it mean to have a declarative language based on it? We propose the following view:

> A program P in such a language is a theory in the logic. After entering the program P into the machine, the user can ask questions about the program. Such questions, called *queries*, belong to a specified class of sentences in the language of P. When the user submits a query φ, if it is the case that φ is a provable consequence of the axioms in P, then the machine will return a set of *answers* justifying the truth of φ. We can view each of these answers as different *proofs* of the truth of φ. If the query φ is not provable from P, two things can happen: either the machine stops after a finite amount of time with the answer "failure," or otherwise the machine loops forever. Therefore, two things are made equivalent: computation in the machine, and deduction in the logic.

One should of course add that in some pragmatic sense the implementation in the machine should be reasonably *efficient* so that for a broad enough class of applications the language can in fact be used in practice; otherwise such a system should be better described as a *theorem prover*.

Definition 3 below gives an axiomatization—in terms of general logics concepts—of the notion of declarative language just presented; a stronger notion requiring also an initial model semantics can be found in [34, 36]. The definition allows careful selection of the signatures and the finite sets of sentences that can be used as *statements* to make up a program P, and also of those sentences that are admissible as *queries*. The language is also required to have a computationally effective operational semantics given by an effective proof subcalculus.

Definition 3 A *declarative programming language* $\mathcal{L}$ is a 4-tuple $\mathcal{L} = (\mathcal{E}, \underline{Sign}_0, stat, quer)$ with:

1. $\mathcal{E} = (\underline{Sign}, sen, \vdash)$ an entailment system.
2. $\underline{Sign}_0$ a subcategory of $\underline{Sign}$.
3. $stat : \underline{Sign} \longrightarrow \underline{Set}$ a subfunctor of the functor obtained by composing sen with the finite powerset functor, i.e., there is a natural inclusion $stat(\Sigma) \subseteq \mathcal{P}_{fin}(sen(\Sigma))$ for each $\Sigma \in \underline{Sign}$. Each $\Gamma \in stat(\Sigma)$ is called a set of Σ-*statements* in $\mathcal{L}$. A *program* in $\mathcal{L}$ is a theory $P = (\Sigma, \Gamma)$ with $\Sigma \in \underline{Sign}_0$ and $\Gamma \in stat(\Sigma)$.
4. $quer : \underline{Sign} \longrightarrow \underline{Set}$ a subfunctor of the sen functor. The sentences $\varphi \in quer(\Sigma)$ are called the Σ-*queries* of $\mathcal{L}$.

In addition, there is an effective proof subcalculus $\mathcal{O}$ having $\mathcal{E}$ as its underlying entailment system, $\underline{Sign}_0$ as its category of admissible signatures, $stat$ as its axioms, and $quer$ as its conclusions. The effective proof subcalculus $\mathcal{O}$ is not assumed to be unique. Any such $\mathcal{O}$ is called *an operational semantics* for the declarative programming language $\mathcal{L}$. $\square$

This definition encompasses a wide range of logics and languages. For example, most equational, functional, Horn logic, and rewriting logic declarative languages, as well as multiparadigm combinations of these, satisfy the definition's requirements [34, 36].

3 Axiomatizing Reflective Logics and Languages

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. Two obvious metatheoretic notions that can be so reflected are theories and the entailment relation $\vdash$. This leads us to the notion of a universal theory. However, universality may not be absolute, but only relative to a class $\mathcal{C}$ of *representable* theories. Typically, for a theory to be representable at the object level, it must have a finitary description in some way—say, being recursively enumerable—so that it can be represented as a piece of language.

Definition 4 Given an entailment system $\mathcal{E}$ and a set of theories $\mathcal{C} \subseteq |\mathbf{Th}|$, a theory U is $\mathcal{C}$-*universal* if there is a function, called a *representation function*,

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $\varphi \in sen(T)$,

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the entailment system $\mathcal{E}$ is called $\mathcal{C}$-*reflective*. □

Note that in a reflective entailment system, since U itself is representable, representation can be iterated, so that we immediately have a "reflective tower"

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi} \iff U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \;\ldots$$

Another key metatheoretic aspect that can be represented at the object level is the proof theory. This leads to the notion of $\mathcal{C}$-reflective proof calculus and subcalculus.

Definition 5 Given a proof calculus $\mathcal{P}$ and a set of theories $\mathcal{C} \subseteq |\mathbf{Th}|$, a theory U is $\mathcal{C}$-*universal* if there is a function, called a *representation function*,

$$\overline{(\_ \vdash \_ : \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times proofs(T) \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in \mathcal{C}$, $p \in proofs(T)$, $\varphi \in sen(T)$,

$$T \vdash p : \varphi \iff U \vdash \overline{T \vdash p : \varphi}.$$

If, in addition, $U \in \mathcal{C}$, then the proof calculus $\mathcal{P}$ is called $\mathcal{C}$-*reflective*.

The same notions can be made relative to a proof subcalculus $\mathcal{O}$ and a set $\mathcal{C}$ of theories contained in, or equal to, the class of admissible theories of $\mathcal{O}$. We get in this way notions of $\mathcal{C}$-universal theory relative to $\mathcal{O}$ and of $\mathcal{C}$-reflective proof subcalculus. Everything is entirely similar, but in this case the expressions $T \vdash p : \varphi$ have a more restricted range, since $T = (\Sigma, \Gamma)$ must have admissible signature and sentences and belong to $\mathcal{C}$, and the sentence φ must belong to the set $conc(\Sigma)$ of admissible conclusions. In addition, we can get corresponding *effective* notions by requiring the calculus or subcalculus to be effective [34], and the representation map to be computable. □

The main advantage of a reflective proof calculus over a reflective entailment system is that proofs are also represented, so that *proof-checking* becomes then expressible as ordinary deduction in the universal theory. As before, given a reflective proof calculus or subcalculus we again have a "reflective tower"

$$T \vdash p : \varphi \iff U \vdash q : \overline{T \vdash p : \varphi} \iff U \vdash q' : \overline{U \vdash q : \overline{T \vdash p : \varphi}} \;\ldots$$

but now the sentences are decorated by proofs, and at each next level there can be many proofs for the sentence representing the previous level.

Based on the above concepts, we can now give a general definition of reflective declarative language.

Definition 6 A *reflective declarative programming language* is a declarative programming language $\mathcal{L} = (\mathcal{E}, \underline{Sign_0}, stat, quer)$ such that the set $\mathcal{C}$ of its programs contains a program U that is a $\mathcal{C}$-universal theory, for a computable representation function sending pairs (P, φ), with P a program in $\mathcal{L}$ and φ a query for it, to a query for U of the form $\overline{P \vdash \varphi}$, so that we have, for each program $P = (\Sigma, \Gamma)$ and query $\varphi \in quer(\Sigma)$,

$$P \vdash \varphi \iff U \vdash \overline{P \vdash \varphi}.$$

If, in addition, an operational semantics $\mathcal{O}$ for $\mathcal{L}$ is a $\mathcal{C}$-reflective and effective proof subcalculus, so that we have another representation function, and another universal theory U'—possibly distinct from U—such that for each program $P = (\Sigma, \Gamma)$, proof $p \in \mathit{proofs}(P)$, and query $\varphi \in quer(\Sigma)$,

$$P \vdash p : \varphi \iff U' \vdash \overline{P \vdash p : \varphi}$$

then we say that $\mathcal{O}$ is a *reflective operational semantics* for $\mathcal{L}$. □

3.1 Reflective Declarative Languages

The goal of the above definition is to capture key aspects of reflection in a declarative language in a logic-independent way. We now discuss classes of languages that either are indeed reflective in the above sense, or there is strong evidence suggesting that they can be so understood. Given the wide range of languages that we consider, many details are omitted; however, we describe in sufficient detail a universal theory for rewriting logic in Appendix A.

3.1.1 Functional Languages

Here, a program P is a finite collection of recursive function definitions that—after adequate program transformations using the Y combinator (see, for example, [44])—can always be expressed as a very simple theory with signature that of the lambda calculus, and axioms equations of the form

$$f_1 = M_1, \ldots, f_n = M_n$$

with the f_i free variables—understood as *additional constants*—standing for the different functions in the program, and the M_i closed lambda terms standing for their corresponding definitions. Queries are existential formulas of the form $(\exists x)N \rightarrow_{nf} x$ with N a lambda term whose only free variables are among the f_i. Such queries are entailed if only if, after substituting in N the f_is by their corresponding definitions, a normal form N' can be reached, that is, if and only if a normal form N' can be reached for the closed lambda term $\lambda(f_1, \ldots, f_n).N(M_1, \ldots, M_n)$. We can then regard N' as a "witness" *proof* of the query and write

$$P \vdash N' : (\exists x)N \rightarrow_{nf} x.$$

A universal theory U is given by a program with only one function definition, $eval = E$, so that for any program P with function definitions $f_1 = M_1, \ldots, f_n = M_n$ we have

$$P \vdash N' : (\exists x)N \rightarrow_{nf} x \iff U \vdash \overline{N'} : (\exists y)\lambda(f_1, \ldots, f_n).(eval\ \overline{N})(\overline{M_1}, \ldots, \overline{M_n}) \rightarrow_{nf} y.$$

A closed lambda term $eval = E$ and a representation function with exactly these properties have been defined and proved correct by Mogensen [38]. The case of *typed* functional languages can be more subtle, because it may not be possible to typecheck an appropriate *eval* function in the given type discipline; therefore, a universal theory may exist, but outside the class of theories in the language, or it may be necessary to consider *eval* functions with a bound on the number of rewrites to remain within the language (see, for example, [47, 31] for a careful treatment of this problem for the calculus of constructions, [45] for a study of reflection in the polymorphic lambda calculus, and [20, 2, 10] for the treatment of reflection in Nuprl's constructive type theory à la Martin Löf).

Yet another promising recent development in this general area is the use of lambda calculi and monads à la Moggi [39] to give semantics to reflective languages [11, 33].

3.1.2 Equational Languages

There is very strong evidence, thanks to the work on the Refal supercompiler [56, 57, 59] and to related work on partial evaluation of equational programs [6, 49], suggesting that equational languages can be made reflective in our precise axiomatic sense. The careful representation by levels of Refal terms and programs (also represented as terms) [59] has excellent mathematical properties and allows arbitrarily high reflective towers. The Refal supercompiler—a program transformer also written in Refal [56]—can "compile away" extra levels of interpretation and can support very powerful meta-programming and compilation tasks [58]. The metacircular Refal interpreter is virtually a universal theory; however, given that in Refal the computational interpretation is favored over the logical one, and that certain language conventions—for example, the order in which the equations are applied—prevent a semantic account given exclusively in equational logic terms, it appears that some more work is needed to define and prove correct universal theories in our sense for equational logic programming.

A different approach is taken in [40], where an abstract machine for evaluating conditional equational programs is extended with reflective capabilities. The treatment in [40] does not provide a universal theory and seems limited in its reflective power.

3.1.3 Horn Logic Languages

There is a long tradition of metacircular interpreters and meta-programming in Prolog (see the papers collected in [1, 8, 43, 12, 3] and references there) strongly suggesting that a more declarative variant of Prolog can be made reflective in our axiomatic sense. The "vanilla" interpreters presented in the literature, for example in [55], fall short of being a universal theory and lack adequate semantics. A systematic effort to carefully represent metalevel concepts and to give a declarative semantics for Horn logic interpreters using a typed version of the logic has been undertaken by Hill and Lloyd [18]; this work is the theoretical basis of the Gödel language [19]. A metacircular interpreter with properties similar to those of a universal theory is given in [19]. However, the mathematical correctness proofs given in [18] seem to apply to formalizations in which the program P is not represented as a data structure appearing in a goal presented to a universal theory, but is instead represented as an extension of the interpreter by additional atomic clauses of the form $clause(A' \, if \, Q')$ for each clause $A \leftarrow Q$ in P. The work already done in [18] seems a good basis for defining and proving correct a universal theory for Horn logic; however, some extra work seems to be needed to prove the correctness of a universal theory in our sense, since programs should be represented as terms in a goal, and not as extra atomic clauses. Furthermore, more work seems to be needed to better understand how other related approaches in this area such as [22, 17] could perhaps be used to define universal theories.

3.1.4 Rewriting Logic Languages

Rewriting logic [35] is a logic of concurrent action that supports declarative specification and programming of concurrent and distributed systems, including object-oriented ones. Since equational logic and Horn logic are both naturally included in rewriting logic in a conservative way, rewriting logic programming generalizes equational programming and Horn logic programming [36, 26]. A first sketch of the reflective aspects of rewriting logic appeared in [27]. The class of finitely presentable rewrite theories has indeed universal theories in exactly the above axiomatic sense. A detailed description of a universal theory U for rewriting logic, as well as an example illustrating the simulation by U of rewriting in an object rewrite theory T, are given in Appendix A.

Our interest in a conceptual clarification of reflection has been motivated by our current design and implementation work on the Maude rewriting logic language [37]. Two other languages based on rewriting logic have been designed by other researchers [25, 13]. There is also a broader interest in this case due to the fact that rewriting logic can be used as a logical and semantic framework in which a wide range of logics and models of computation—including Horn and equational logic, linear logic, sequent calculi, the lambda calculus, concurrent object-oriented programming, and many others [35, 26]—can be naturally represented. Therefore, because of its reflective properties, rewriting logic seems a good semantic framework to specify and prototype many other reflective logics and computational systems; the example of Turing machines discussed below is a case in point. Further evidence is provided by the recent work of Watanabe [61], and of Malenfant, Dony, and Cointe [21], since in these two papers rewriting is used to formalize computational reflection.

3.1.5 Turing Machines

Turing machines can be naturally viewed as phrase structure grammars, and therefore as string rewriting systems, that is, as rewrite theories *modulo* the associativity equation for a binary string concatenation operator. Therefore, they form a very restricted class $\mathcal{T}$ of finitely presentable theories in rewriting logic [35]. As it is well-known, this class has $\mathcal{T}$-universal theories belonging to it. Therefore, we can view Turing reflection as a very special case of reflection within the semantic framework of rewriting logic.

4 Strategies

A metacircular interpreter may have a fixed strategy, and such a strategy may remain at the metalevel. This will make such an interpreter less flexible, and will complicate formal reasoning about its correctness. Ideally, strategies should also be defined within the same reflective logic, so that they can be represented and can be reasoned about at the object level. In this section we first give axioms for strategies in general, and for strategies internal to a logic; then we discuss the advantages of a reflective logic with internal strategies, as in the case of rewriting logic.

Given a logical theory T, a strategy is a computational way of looking for certain proofs of some theorems of T. This may be done by having a strategy language $S(T)$ associated to T. For example, in theorem provers in the LCF tradition, there is a metalanguage, namely ML, in which "tactics" and "tacticals" can be defined for exactly this purpose [42]. Generally, we can think abstractly of the strategy language $S(T)$ as a computational system in which strategy expressions can be further and further evaluated—in some cases perhaps forever, and sometimes in highly nondeterministic ways—so that the more we evaluate one such expression the more proofs will be *exhibited* in further stages. We can naturally represent such evaluations from a strategy expression E to another E' by labelled transitions $\alpha : E \longrightarrow E'$. If we denote by $\rho(E)$ the set of proofs exhibited by E, then our requirement is that if there is a computation $\alpha : E \longrightarrow E'$, then $\rho(E) \subseteq \rho(E')$. Of course, transitions $\alpha : E \longrightarrow E'$ and $\beta : E' \longrightarrow E''$ should be composable, to yield $\alpha; \beta : E \longrightarrow E''$, and it is always possible to add idle transitions $E : E \longrightarrow E$. Therefore, we can axiomatize the computations of the strategy language as a category $S(T)$.

Definition 7 Given a proof calculus $\mathcal{Q}$, an *external strategy language* for it is a functor $S : \mathbf{Th}_0 \longrightarrow \mathbf{Cat}$, together with a natural transformation $\rho : S \Rightarrow \mathcal{P}_{fin} \circ \mathit{proofs}$, where $\mathcal{P}_{fin} : \mathbf{Set} \longrightarrow \mathbf{Cat}$ is the functor sending each set X to the poset $\mathcal{P}_{fin}(X)$ of finite subsets of X, viewed as a category with arrows the subset inclusions. In addition, the strategy language is required to be *complete* in the sense that for each $p \in \mathit{proofs}(T)$ there is an $E \in S(T)$ such that $p \in \rho(E)$.

An *internal strategy language* for $\mathcal{Q}$ consists of:

- an endofunctor $M : \mathbf{Th}_0 \longrightarrow \mathbf{Th}_0$—that should be thought of as associating to each theory T a "local metatheory" $M(T)$ axiomatizing the strategy expressions for T;
- a functor $V : \mathbf{Str} \longrightarrow \mathbf{Cat}$—that forgets part of the algebraic structure of the algebra of proofs and leaves only the underlying categorical structure; in particular, $V(P(M(T)))$ is the category of computations for the strategy expressions in $M(T)$—and
- a natural transformation $\rho : V \circ P \circ M \Rightarrow \mathcal{P}_{fin} \circ \mathit{proofs}$, that associates to each strategy expression the set of proofs that it exhibits.

$$\begin{tikzcd}
\mathbf{Th}_0 \arrow[r, "M"] \arrow[d, "\mathit{proofs}"'] & \mathbf{Th}_0 \arrow[r, "P"] \arrow[d, phantom, "\Downarrow\rho"] & \mathbf{Str} \arrow[d, "V"] \\
\mathbf{Set} \arrow[rr, "\mathcal{P}_{fin}"'] & & \mathbf{Cat}
\end{tikzcd}$$

In addition, we require such a language to be complete in the above sense. Notice that, by taking $S = V \circ P \circ M$, any internal strategy language can be regarded as an external one.

A $\mathcal{C}$-reflective proof calculus $\mathcal{Q}$ has an *internal strategy language* iff M is now defined as an endofunctor $M : \mathcal{C} \longrightarrow \mathcal{C}$, on the full subcategory of $\mathbf{Th}_0$ determined by $\mathcal{C}$, with V as before, and with ρ now restricted to $\mathcal{C}$.

The definitions of external and internal strategy language can be relativized to a proof subcalculus $\mathcal{O}$ in the obvious manner. We then say that a (reflective) declarative language $\mathcal{L}$ with a (reflective) operational semantics $\mathcal{O}$ *has internal strategies* if there is an internal strategy language for $\mathcal{O}$, where M is required to be an effectively given endofunctor mapping each program T to the program $M(T)$ of strategies for T, and where ρ_T is required to be a computable function extracting the proofs exhibited by each strategy expression. $\square$

We say that a strategy expression $E \in S(T)$ is *Church-Rosser* iff whenever we have transitions $\alpha : E \longrightarrow E'$ and $\beta : E \longrightarrow E''$ there is always an E''' and transitions $\alpha' : E' \longrightarrow E'''$ and $\beta' : E'' \longrightarrow E'''$. We say that a strategy expression E is *sequential* iff there is a (finite or countable) sequence of nonidle transitions $\alpha_n : E_n \longrightarrow E_{n+1}$ with $E_0 = E$, such that for each nonidle transition $\alpha : E \longrightarrow E'$ there exists a unique k such that $\alpha = \alpha_0; \ldots; \alpha_k$. Each sequential strategy expression is obviously Church-Rosser, but the contrary is not true in general. Strategy expressions need not be Church-Rosser. For example, a strategy language can have a nondeterministic choice operator $\_ \oplus \_$ so that an expression $E \oplus E'$ has transitions $\alpha : E \oplus E' \longrightarrow E$, and $\beta : E \oplus E' \longrightarrow E'$. Such a nondeterministic choice operator should then be "opaque," so that no proofs are exhibited until it has disappeared, that is, $\rho(E \oplus E') = \emptyset$.

If a reflective proof calculus, or a reflective declarative language, has an internal strategy language, then the strategies $S(U)$ for the universal theory are particularly important, since they represent at the object level strategies for computing in the universal theory. A *metacircular interpreter* for such a language can then be regarded as the implementation of a particular strategy in $S(U)$, and reasoning about the properties of such an interpreter can then be carried out inside the logic itself.

The class of finitely presentable rewrite theories has universal theories, making rewriting logic reflective. In addition, a variety of internal strategy languages can be defined by effective theory transformations M that can themselves be also defined inside the language of rewriting logic[1]. Then, given a finitely presented rewrite theory $\mathcal{R}$, a good choice for $S(\mathcal{R})$ is the category $\mathcal{T}_{M(\mathcal{R})}(X)$ underlying the free $M(\mathcal{R})$-system [35] on a countable set of variables X. Intuitively, M associates to each rewrite theory $\mathcal{R}$ the rewrite theory specifying how to compute strategy expressions for it, and then $\mathcal{T}_{M(\mathcal{R})}(X)$ is the category of all such strategy expression rewritings. We are currently studying a good choice of M for Maude. Similar ideas about specifying strategies with rewrite rules are also being adopted for the strategies of the ELAN rewriting logic language [24], whose strategy language was initially defined as an external language [25].

In the context of typed lambda calculi, the important advantages of having an internal strategy language has been stressed by several authors. Thus, using reflective capabilities both tactics and decision pocedures can be specified, reasoned about, and executed inside the Nuprl constructive type theory [2, 10]. Similarly, Rueß[32] discusses in detail an elegant

[1] For the definition of a transformation M of this kind in which the strategy expressions actually coincide with multisets of proof expressions see [27]; details for other strategy languages M will appear elsewhere.

approach for endowing the calculus of constructions with internal strategies, as part of his treatment of reflection for such a calculus.

5 Concluding Remarks

This work is a first step within a broader effort to develop a general theory of reflection. The purpose of the concepts presented here is to support reasoning about, and proving properties of, reflective logics and languages in a general and abstract way. The aim is similar to that in abstract model theory [4, 15], where important properties of a logic can be established under general assumptions. Much work remains ahead to further develop and exploit these ideas; in particular, a number of important topics—suggested by this paper, but not addressed in it—should be studied in future, including:

- *Computational reflection for nondeclarative languages.* This is an area in even greater need for theoretical foundations than that of reflective logics and reflective declarative languages. We conjecture that reflective rewriting logic can provide clear and simple semantics for reflective computational systems, as briefly illustrated here by the example of Turing machines and further evidenced by the use of rewriting systems in semantic definitions of reflection by other authors [61, 21]. Since a particularly simple rewriting logic semantics for object-oriented systems, including concurrent ones, already exists [37], developing a rewriting logic semantics for *reflective* object-oriented systems seems a natural and promising research direction.

- *Mappings between reflective systems.* In the theory of general logics, logics are related by mappings preserving the relevant logical structure. These mappings can be very useful for reusing tools and logical components [9]. Similar notions of mapping should be investigated in detail for the reflective case.

- *Modularity.* The fact that theories can be represented at the object level offers excellent opportunities for developing a very powerful, internalized and logic-independent theory of modules and module operations for both specifications and programs. This can lead to fundamental advances in programming methodology.

- *Supercompilation and partial evaluation.* Dramatic improvements in the efficiency of reflective systems can be gained using these techniques. For the most part, they have been developed in a functional programming context. A more general, metalogical understanding of these techniques consistent with the ideas presented here could substantially widen their range of applicability.

- *Reflective logical and semantic frameworks,* in which many different reflective logics, languages, and systems can be naturally represented, can be very useful for executable specification, rapid prototyping, and formal reasoning. A substantial amount of evidence, advocating the suitability of rewriting logic for these purposes already exists [35, 26, 25, 27]. Since rewriting logic is reflective and has internal strategy languages, a more systematic exploitation of reflection in its use as a logical and semantic framework seems both promising and natural. Similar studies can also be useful for other suitable reflective logics. In fact, good evidence already exists about the good properties of Nuprl's constructive type theory as a reflective metalogical framework [5, 2].

Acknowledgements

We cordially thank Steven Eker, David Israel, Narciso Martí-Oliet, Harald Rueß, Johann Schumann, Natajaran Shankar, Mark Stickel, Carolyn Talcott, and Valentin Turchin for their very helpful comments on an earlier version of this paper.

References

[1] H. Abramson and M.H. Rogers, editors. *Metaprogramming in Logic Programming.* MIT Press, 1989.

[2] William E. Aitken, Robert L. Constable, and Judith L. Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical Report, Computer Sci. Dept., Cornell University, 1993; also, lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.

[3] Krzysztof Apt and Franco Turini, editors. *Meta-Logics and Logic Programming.* Logic Programming Series. MIT Press, November 1995.

[4] J. Barwise and S. Feferman (eds.). *Model-Theoretic Logics.* Springer-Verlag, 1985.

[5] D. A. Basin and R. L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[6] Anders Bondorf. A self-applicable partial evaluator for term rewriting systems. In J. Díaz and F. Orejas, editors, *TAPSOFT'89*, pages 81–95. Springer LNCS 352, 1989.

[7] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–185. Academic Press, 1981.

[8] M. Bruynooghe, editor. *Proceedings Second Workshop on Meta-programming in Logic.* K.U.Leuven, Department of Comp.Sc., April 1990.

[9] Maura Cerioli and José Meseguer. May I borrow your logic? (transporting logical structure along maps). To appear in *Theoretical Computer Science*, 1996.

[10] Robert L. Constable. Using reflection to explain and enhance type theory. In *Proof and Computation*, pages 109–144, 1995.

[11] Andrzej Filinski. Representing monads. In *Conference Record POPL '94 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, January 1994.

[12] Laurent Fribourg and Franco Turini, editors. *Logic Program Synthesis and Transformation–Meta-programming in Logic.* LNCS 883. Springer-Verlag, September 1994.

[13] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. To appear in *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

[14] Fausto Giunchiglia, Paolo Traverso, Alessandro Cimatti, and Paolo Pecchiari. A system for multi-level reasoning. In *IMSA'92*, pages 190–195. Information-Technology Promotion Agency, Japan, 1992.

[15] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[16] John Harrison. Metatheory and reflection in theorem proving: a survey and critique. University of Cambridge Computer Laboratory, 1995.

[17] Christopher P. Higgins. Type-safe reflection and encapsulation in logic programming. In this volume.

[18] Patricia Hill and John Lloyd. Analysis of meta-programs. In H. D. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–52. MIT Press, 1989.

[19] Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[20] Douglas J. Howe. Reflecting the semantics of reflected proof. In Peter Aczel, Harold Simmons, and Stanley S. Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1990.

[21] C. Dony J. Malenfant and P. Cointe. A semantics of introspection in a reflective prototype-based language. To appear in *Lisp and Symbolic Computation*.

[22] Marianne Kalsbeek and Yuejun Jiang. *Meta-Logics and Logic Programming*, chapter A Vademecum of Ambivalent Logic, pages 27–55. MIT Press, November 1995.

[23] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[24] C. Kirchner and H. Kirchner. Personal communication. July 1995.

[25] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.

[26] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[27] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

[28] Satoshi Matsuoka, Takuo Watanabe, Yuuji Ichisugi, and Akinori Yonezawa. Object-oriented concurrent reflective architectures. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing*, pages 211–226. Springer LNCS 612, 1992.

[29] Seán Matthews. Reflection in logical systems. In *IMSA'92*, pages 178–183. Information-Technology Promotion Agency, Japan, 1992.

[30] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[31] Harald Rueß. Computational reflection in the calculus of constructions and its application to theorem proving. Manuscript, Universität Ulm, Abt. Künstliche Intelligenz, December 1995.

[32] Harald Rueß. Reflection of formal tactics in a deductive reflection framework. Manuscript, Universität Ulm, Abt. Künstliche Intelligenz, January 96.

[33] Anurag Mendhekar and Daniel P. Friedman. An exploration of relationship between reflective theories. In this volume.

[34] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[35] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[36] José Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Proc. 3rd Intl. Conf. on Algebraic and Logic Programming*, pages 158–200. Springer LNCS 632, 1992.

[37] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[38] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.

[39] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, June 1989.

[40] Masanobu Numazawa, Masahito Kurihara, and Azuma Ohuchi. A reflective language based on conditional term rewriting. Technical report, Division of Systems and Information Engineering, Hokkaido University, Sapporo, Japan, 1996.

[41] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *IMSA'92*, pages 36–47. Information-Technology Promotion Agency, Japan, 1992.

[42] Lawrence Paulson. Tactics and tacticals in Cambridge LCF. Technical Report 39, University of Cambridge, 1983.

[43] A. Pettorossi, editor. *Proceedings Third Workshop on Meta-programming in Logic.* LNCS 649. Springer-Verlag, April 1992.

[44] Simon Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[45] F. Pfenning and P. Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, pages 137–159, 1991.

[46] Luis H. Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *IMSA'92*, pages 107–112. Information-Technology Promotion Agency, Japan, 1992.

[47] Harald Rueß. *Formal Meta-Programming in the Calculus of Constructions.* PhD thesis, Universität Ulm, 1995.

[48] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

[49] David Sherman, Robert Strandh, and Irène Durand. Optimization of equational programs using partial evaluation. In *PEPM'91*, pages 72–82. SIGPLAN Notices, 1991.

[50] Brian Smith and Akinori Yonezawa, editors. *Proc. of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture, Tokyo, November 1992.* Research Institute of Software Engineering, 1992.

[51] Brian C. Smith. Reflection and Semantics in Lisp. In *Proc. POPL'84*, pages 23–35. ACM, 1984.

[52] C. Smorynski. The incompleteness theorems. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 821–865. North-Holland, 1977.

[53] R. M. Smullyan. *Diagonalization and Self-Reference.* Oxford University Press, 1994.

[54] G. L. Steele, Jr. and G. J. Sussman. The art of the interpreter or, the modularity complex. Technical Report AIM-453, MIT AI-Lab, May 1978.

[55] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[56] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[57] Valentin F. Turchin. *REFAL-5, Programming Guide and Reference Manual.* New England Publishing Co., 1989.

[58] Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[59] Valentin F. Turchin and Andrei P. Nemytykh. Metavariables: their implementation and use in program transformation. Technical Report CSc-TR-95-012, City College of CUNY, 1995.

[60] M. Wand and D.P. Friedman. The mystery of the tower revealed. *Lisp and Symbolic Computation*, 1(1):11–38, 1988.

[61] Takuo Watanabe. Towards a foundation of computational reflection based on abstract rewriting (preliminary result). In *IMSA'95*, pages 143–145. Information-Technology Promotion Agency, Japan, 1995.

[62] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

A Reflection in Rewriting logic

This appendix gives the rules of deduction for rewriting logic, defines a universal theory for the class of finitely presentable rewrite theories, and illustrates its use in simulating an object theory by means of an example.

A.1 A Simple Version of Rewriting Logic

A *signature* in rewriting logic is a pair (Σ, E) formed by a ranked alphabet Σ of function symbols and a set E of Σ-equations. Given a signature (Σ, E), *sentences* of the logic are sequents of the form $t \longrightarrow t'$, where t and t' are Σ-terms. A *theory* in this logic, called a rewrite theory, is a triple[2] $T = (\Sigma, E, R)$ with (Σ, E) a signature, and R a set of sequents called the *rewrite rules* of T. For $t, t' \in \mathcal{T}_\Sigma(X)$, where $\mathcal{T}_\Sigma(X)$ denotes the algebra of Σ-terms with variables X, we use the notation $t = t'$ for equations, and write $t =_E t'$ iff $t = t'$ is provable from the equations E using the rules of (unsorted) equational logic.

Given a rewrite theory T, we say that T *entails* a sequent $t \longrightarrow t'$ and write $T \vdash t \longrightarrow t'$ iff $t \longrightarrow t'$ can be obtained by finite application of the following rules of deduction. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $t(x_1, \ldots, x_n) \longrightarrow t'(x_1, \ldots, x_n)$; also $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t.

1. **Reflexivity**. For each $t \in \mathcal{T}_\Sigma(X)$,

$$\frac{}{t \longrightarrow t}$$

[2]In the standard treatment of rewriting logic, rules in a rewrite theory T have labels in a set L and are written $l : t \longrightarrow t'$. We omit labels in the present version to simplify the exposition. All that we say below has a straightforward extension to the labelled case.

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{t_1 \longrightarrow t_1' \quad \ldots \quad t_n \longrightarrow t_n'}{f(t_1, \ldots, t_n) \longrightarrow f(t_1', \ldots, t_n')}$$

3. **Replacement**. For each rewrite rule $t(x_1, \ldots, x_n) \longrightarrow t'(x_1, \ldots, x_n)$ in T,

$$\frac{w_1 \longrightarrow w_1' \quad \ldots \quad w_n \longrightarrow w_n'}{t(\overline{w}/\overline{x}) \longrightarrow t'(\overline{w'}/\overline{x})}$$

4. **Transitivity**.

$$\frac{t_1 \longrightarrow t_2 \qquad t_2 \longrightarrow t_3}{t_1 \longrightarrow t_3}$$

5. **Equality**.

$$\frac{u' =_E u \qquad u \longrightarrow v \qquad v =_E v'}{u' \longrightarrow v'}$$

It is easy to show that the above rules of deduction define an entailment system for rewriting logic. We can choose as signature maps $H : (\Sigma, E) \longrightarrow (\Sigma', E')$ in **Sign** rank-preserving functions $H : \Sigma \longrightarrow \Sigma'$ such that $E' \vdash H(E)$.

A.2 A Universal Theory for Rewriting Logic

In this section we introduce a theory U such that there is a representation function making U a $\mathcal{C}$-universal theory, for $\mathcal{C}$ the class of unconditional and unsorted finitely presentable rewrite theories—that is, theories whose ranked alphabet, set of equations, and set of rules are all finite. Without any essential loss of generality we assume that the syntax of those theories is given by operators and variables that are strings of ASCII characters. We also assume that all such theories have standard parenthesized notation. However, to ease readability, in the particular case of the theory U, we will adopt some extra notational conventions.

We first introduce the ranked alphabet Σ_U of operation symbols of U and briefly explain how a representation function for U can be defined. Then, we give the set E_U of equations, and the set R_U of rules of U.

The operation symbols of Σ_U are as follows:

Σ_0 = ASCII $\cup\{$`--`$\}$, where ASCII denotes the set of ASCII characters.

$\Sigma_1 = \{$ `op{_}`, `var{_}` $\}$

$\Sigma_2 = \{$ `_._`, `_[_]`, `_{_}`, `_,_`, `_[<-_]`, `_=_`, `_=>_`, `_->_`, `_/_`, `_@_`, `_!_` $\}$

$\Sigma_3 = \{$ `<_;_;_>` $\}$

$\Sigma_4 = \{$ `_::<_;_;_>` $\}$

$\Sigma_5 = \{$ `_::<_;_;_;_>` $\}$

To ease readability we adopt the following notational conventions:

1. The appearance of n underbar characters in an n-ary operator of Σ_U, allows us to display the corresponding expressions with mix-fix syntax, adding parentheses when necessary. Thus, for the operator '`_->_`' we write X`->` Y instead of `_->_`(X, Y).

2. The string-forming operators '`_._`', '`_,_`' and '`_!_`' are assumed parsed with left-associating precedence. Thus, `a.b.c` is shorthand for `a.(b.c)`, and similarly, `a,b,c` is shorthand for `a,(b,c)`.

We next define a representation function $\overline{(\_ \vdash \_)}$ for encoding pairs consisting of a finitely presentable rewrite theory T and a sentence in it as sentences in U,

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(U)$$

The definition of $\overline{(\_ \vdash \_)}$ is given in a top-down fashion. This will make clear the intended meaning of most of the operation symbols of Σ_U. Note that, to ease readability, we recursively define the representation of theories, rules, term, etc. using an overloaded function symbol $\overline{(\_)}$.

- For $T \in \mathcal{C}$ a finitely presentable rewrite theory, and $t \longrightarrow t'$ a sentence in T, $\overline{(T \vdash t \longrightarrow t')} = (\overline{T}\,\texttt{@}\,\overline{t} \longrightarrow \overline{T}\,\texttt{@}\,\overline{t'})$.
- For T a rewrite theory $(\Sigma, E, R) \in \mathcal{C}$ such that Var is the finite set of variables appearing in the equations E and rules R, $\overline{T} = \texttt{<}\overline{Var}\texttt{;}\ \overline{E}\texttt{;}\overline{R}\texttt{>}$.
- For Var a set of variables $\{x_1, \ldots, x_n\}$, $\overline{Var} = \overline{x_1}\texttt{,}\ldots\texttt{,}\ \overline{x_n}$; for Var an empty set, $\overline{Var} = \texttt{--}$.
- For E a set of equations $\{e_1, \ldots, e_n\}$, $\overline{E} = \overline{e_1}\texttt{,}\ldots\texttt{,}\ \overline{e_n}$; for E an empty set, $\overline{E} = \texttt{--}$.
- For R a set of rewrite rules $\{r_1, \ldots, r_n\}$, $\overline{R} = \overline{r_1}\texttt{,}\ldots\texttt{,}\ \overline{r_n}$; for R an empty set, $\overline{R} = \texttt{--}$.
- For e an equation $(t = t')$, $\overline{e} = (\overline{t}\ \texttt{=}\ \overline{t'})$.
- For r a rewrite rule $(t \longrightarrow t')$, $\overline{r} = (\overline{t}\ \texttt{=>}\ \overline{t'})$.
- For t a term $f(t_1, \ldots, t_n)$, $f \in \Sigma_n$, $n > 0$, $\overline{t} = \texttt{op\{}\underline{f}\texttt{\}[}\overline{t_1}\texttt{,}\ldots\texttt{,}\overline{t_n}\texttt{]}$; for t a term c, $c \in \Sigma_0$, $\overline{t} = \texttt{op\{}\underline{c}\texttt{\}[--]}$; and for t a term x, $x \in Var$, $\overline{t} = \texttt{var\{}\underline{x}\texttt{\}[--]}$. Note that we have assumed that all operators and variables are strings of ASCII characters.
- For a string of ASCII characters $l = a_1 a_2 \ldots a_n$, $\underline{l} = a_1\,\texttt{.}\,a_2\,\texttt{.}\ldots\texttt{.}\,a_n$.

We next introduce the set E_U of equations,

$$x = (x\texttt{,--}) \qquad x = (x\texttt{!--}) \qquad (x\texttt{=}y) = (y\texttt{=}x)$$

$$x = (\texttt{--,}x) \qquad x = (\texttt{--!}x)$$

$$x\texttt{,}(y\texttt{,}z) = (x\texttt{,}y)\texttt{,}z \quad x\texttt{!}(y\texttt{!}z) = (x\texttt{!}y)\texttt{!}z$$

Therefore, the operation symbols '`_,_`' and '`_!_`' are declared associative with '`--`' their identity element; and the operation symbol '`_=_`' is declared commutative.

We next introduce by groups the set R_U of rules, and explain their intended meaning. To ease readability, variables in the rules appear in *italics*; this is shorthand notation for the convention that all variables are character strings beginning with a quote and having length at least two, so that no ambiguity may ever arise with the ASCII characters themselves, that are constants of Σ_U. We also introduce the notation $t \longleftrightarrow t'$ to indicate a bidirectional rule, that is, a pair of rules $t \longrightarrow t'$ and $t' \longrightarrow t$.

As we shall see, U itself belongs to the class $\mathcal{C}$ of finitely presentable rewrite theories. Therefore, U makes the entailment system of rewriting logic reflective. This means that U reifies the entailment relation, and that the operation symbols, equations, and rules of U can be seen as a specification in rewriting logic of its own rules of deduction, including the congruence, replacement, and equality rules. The reflexivity and transitivity rules of deduction for a theory $T \in \mathcal{C}$ are directly mirrored by the reflexivity and transitivity rules of deduction for U.

Note that rewriting logic has also a proof calculus [35]. By extending the definition of U along the lines of [27], so as to make explicit and reify the proofs built up by the deduction process, one can similarly exhibit a finitely presentable universal theory U' making the proof calculus of rewriting logic $\mathcal{C}$-reflective.

To reify the rule of congruence we will use contexts and (potential) redexes. In fact, our idea is to use contexts and redexes to combine the rules of congruence and replacement. As a result, a step of reified replacement will be taken in any subterm of a reified subject term. In particular, we use the operation symbols '`*`' and '`_[<-_]`' in Σ_U. In rules (1) and (2) below we use these operation symbols to decompose a term $\overline{t}$ to be rewritten into a context and a potential redex. A context is either '`*`' or a term with a '`*`' as one of its arguments. A potential redex is a term. The intended meaning of rules (1) and (2) is to indicate the subterm $\overline{t_1}$ of $\overline{t}$ in which a step of rewriting will be attempted.

$$T\texttt{@}f\texttt{[}x\texttt{]} \quad \longleftrightarrow \quad T\texttt{@*[<-}f\texttt{[}x\texttt{]]} \tag{1}$$

$$T\texttt{@}u\texttt{[<-}f\texttt{[}x\texttt{,}g\texttt{[}x_1\texttt{],}x'\texttt{]]} \quad \longleftrightarrow \quad T\texttt{@}u\texttt{[<-}f\texttt{[}x\texttt{,*,}x'\texttt{]][<-}g\texttt{[}x_1\texttt{]]} \tag{2}$$

Since we have declared the operation symbol '`_,_`' associative and with identity element '`--`', rule (2) can be applied to any argument of f. In fact, an overriding goal in the definition of U is to avoid as much as possible building into it a particular evaluation strategy, so as to allow the greatest possible flexibility in the subsequent definition of such strategies. Therefore, U should not be conceived as a particular interpreter (say left-most inner-most) for rewriting logic. However, with the notion of strategy introduced in Section 4, it is easy to see that many different interpreters can be obtained from U by defining different strategies S for U (details will appear elsewhere).

To reify the rule of replacement we have first to reify the condition for its application. Given a rule $t(x_1, \ldots, x_n) \longrightarrow t'(x_1, \ldots, x_n)$, a replacement can be made in a term t_1 iff $t_1 = t(\overline{w}/\overline{x})$. We use the rule (3) below to *set aside* a matching problem between the left-hand side of a rule and a potential redex. For that we have introduced in Σ_U the operation symbol '`_::<_;_;_;_>`'. The first argument of '`_::<_;_;_;_>`' is a pair formed by a rewrite theory and a term decomposed into a context and a potential redex, to which the matching problem is related; the second and third arguments are the left-hand side of a rule and the potential redex respectively. The fourth argument consists of a pair built up with the operation symbol '`_/_`'. The first element of this pair represents a set of assignments, and the second a set of variables. As we will see below, this argument is needed to handle the case of nonlinear left-hand sides in which a variable can have several occurrences. Note that (3) sets to empty the initial set of assignments, and sets the initial set of variables to the set of variables of the theory. Finally, the fifth argument is the righthand side of the selected rule. As we shall see below, this allows us to continue the reified replacement process without having to keep track of the rule that we have selected. Rule (4) reifies the equality rule treating it in a way entirely similar to replacement.

$$\begin{aligned}&\texttt{<}V\texttt{;}E\texttt{;}R\texttt{,}(t\texttt{=>}t')\texttt{,}R'\texttt{>@}u\texttt{[<-}t_1\texttt{]} \longleftrightarrow\\ &\quad \texttt{<}V\texttt{;}E\texttt{;}R\texttt{,}(t\texttt{=>}t')\texttt{,}R'\texttt{>@}u\texttt{[<-}t_1\texttt{]::<*[<-}t\texttt{];*[<-}t_1\texttt{];--/}V\texttt{;}t'\texttt{>}\end{aligned}\tag{3}$$

$$\begin{aligned}&\texttt{<}V\texttt{;}E\texttt{,}(t\texttt{=}t')\texttt{,}E'\texttt{;}R\texttt{>@}u\texttt{[<-}t_1\texttt{]} \longleftrightarrow\\ &\quad \texttt{<}V\texttt{;}E\texttt{,}(t\texttt{=}t')\texttt{,}E'\texttt{;}R\texttt{>@}u\texttt{[<-}t_1\texttt{]::<*[<-}t\texttt{];*[<-}t_1\texttt{];--/}V\texttt{;}t'\texttt{>}\end{aligned}\tag{4}$$

Note that, since the operation symbol '`_,_`' is associative and has identity '`--`', any rule can be chosen. Therefore, no commitment is made about the strategy of rule selection for a replacement step in the derivation.

The matching problem between two terms $\overline{t}$ and $\overline{t_1}$ will be handled by rules (5) – (7) below. As expected, they will try to come out with a set of assignments A, such that $\overline{t}$ substituted by A is equal to $\overline{t_1}$. The reified matching process can be seen as a process of trying to identify $\overline{t}$ and $\overline{t_1}$ while keeping track of their differences in A. To handle the case of non-linear lefthand sides, we use the pair A/V. The idea is to keep the set of variables in A and the set V of variables not yet assigned disjoint from each other. Note that the initial set V is the set of variables in the theory. When in the reified matching process we reach the base case of matching a variable $\overline{x}$ and a term $\overline{t}$, we consider two cases: rule (6) when $\overline{x}$ is in V, and rule (7) when $\overline{x}$ is in the set of variables in A, because $\overline{x}$ has already been encountered in another occurrence during the matching. Finally, rule (5) makes use of the operation symbols '`*`' and '`_[<-_]`' in a similar way that rules (1) and (2). Therefore, the intended meaning of (5) is to indicate the subterms $\overline{t'}$ and $\overline{t'_1}$ of $\overline{t}$ and $\overline{t_1}$ respectively, in which matching will be further pursued.

$$\begin{aligned}&z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,}g\texttt{[}x_1\texttt{],}x'\texttt{]];}u'\texttt{[<-}f\texttt{[}y\texttt{,}g'\texttt{[}y_1\texttt{],}y'\texttt{]];}A\texttt{/}V\texttt{;}t'\texttt{>} \longleftrightarrow\\ &\quad z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,*,}x'\texttt{]][<-}g\texttt{[}x_1\texttt{]];}u'\texttt{[<-}f\texttt{[}y\texttt{,*,}y'\texttt{]][<-}g'\texttt{[}y_1\texttt{]];}A\texttt{/}V\texttt{;}t'\texttt{>}\end{aligned}\tag{5}$$

$$\begin{aligned}&z\texttt{::<}u\texttt{[<-var}\{x\}\texttt{[--]];}u'\texttt{[<-}t\texttt{];}A\texttt{/}(V\texttt{,}(\texttt{var}\{x\}\texttt{[--]})\texttt{,}V')\texttt{;}t'\texttt{>} \longrightarrow\\ &\quad z\texttt{::<}u\texttt{[<-var}\{x\}\texttt{[--]];}u'\texttt{[<-var}\{x\}\texttt{[--]];}((\texttt{var}\{x\}\texttt{[--]->}t)\texttt{,}A)\texttt{/}(V\texttt{,}V')\texttt{;}t'\texttt{>}\end{aligned}\tag{6}$$

$$\begin{aligned}&z\texttt{::<}u\texttt{[<-var}\{x\}\texttt{[--]];}u'\texttt{[<-}t\texttt{];}(A\texttt{,}(\texttt{var}\{x\}\texttt{[--]->}t)\texttt{,}A')\texttt{/}V\texttt{;}t'\texttt{>} \longrightarrow\\ &\quad z\texttt{::<}u\texttt{[<-var}\{x\}\texttt{[--]];}u'\texttt{[<-var}\{x\}\texttt{[--]];}(A\texttt{,}(\texttt{var}\{x\}\texttt{[--]->}t)\texttt{,}A')\texttt{/}V\texttt{;}t'\texttt{>}\end{aligned}\tag{7}$$

Note again that no commitment is made about the strategy for the reified matching process. However, a matching process can only be terminated using rules (8) and (9) below. Note that rule (8) can be applied at any time in the reified matching process to abort it, but (9) can only be applied when this process has been succesful, that is, when, after a number of applications of the rules (5) – (7) a potential redex $\overline{t_1}$ has been made equal to the lefthand side of a rule $\overline{t}$, and A has become the set of assignments such that $\overline{t}$ substituted by A is equal to $\overline{t_1}$.

$$z\texttt{::<}u\texttt{;}u'\texttt{;}A\texttt{/}V\texttt{;}t'\texttt{>} \longrightarrow z \tag{8}$$

$$z\texttt{::<}u\texttt{;}u\texttt{;}A\texttt{/}V\texttt{;}t'\texttt{>} \longrightarrow z\texttt{::<*[<-}t'\texttt{];}A\texttt{;}t'\texttt{>} \tag{9}$$

An application of rule (9) changes the matching task into a replacement task. For that we have introduced in Σ_U the operation symbol '`_::<_;_;_>`'. The first argument of '`_::<_;_;_>`' is a pair formed by a rewrite theory an a term decomposed into a context an a redex; the second argument is the righthand side $\overline{t'}$ of a rule whose lefthand side has been succesfully matched with the redex; the third argument is the set A of assignments resulting from this matching process. Finally, the fourth argument is a substitution task. As we will see below, this argument will be needed to guarantee the correctness of our reification of the replacement rule. A replacement task is carried out as a two-phase process: the first phase consists in applying the substitution A to $\overline{t'}$, and the second consists in the actual replacement using the result of this substitution.

The application of the substitution A to the term $\overline{t'}$ is performed by rules (10) – (14) below. Unlike the reified matching process, substitution, once initiated, has to be carried on to the end. To indicate that a subterm has been fully substituted, we have introduced in Σ_U the operation symbol '`_{_}`'. Also, to indicate what part of the substitution task remains to be done, we have introduced in Σ_U the operation symbol '`_!_`'. The idea is to reify the substitution task as a `_!_`-sequence of terms, the last one representing the current task. Note that rule (9) chooses $\overline{t'}$ as the current task.

Rule (10) makes use of the operation symbols '`*`' and '`_[<-_]`' in a way similar to rules (1), (2) and (5). Therefore, the intended meaning of (10) is to indicate the subterm $\overline{t'}$ of $\overline{t}$ in which substitution will be further pursued. However, note that a subterm $\overline{t'}$ can be selected only if there is a subtask $\overline{t'}$ in the current task. An application of (10) will not only indicate the subterm $\overline{t'}$ in which substitution will be further pursued, but will also first delete it as a subtask of the current task, and then place it as the new current task. Rule (11) handles the case when the current task has no subtasks. In particular, (11) deletes the current task and changes the current subterm into a fully substituted subterm. In a similar fashion, rules (12) and (13) handle the base cases in the reified substitution process. As expected, (12) deletes the current task and changes the current subterm into a fully substituted subterm. Similar behavior is exhibited by (13), but performing also the actual substitution. Finally, rule (14) recomposes a subterm previously decomposed into a context and a subterm, when the latter has been fully substituted.

$$\begin{aligned} &z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,}g\texttt{[}x_1\texttt{],}x'\texttt{]];}A\texttt{;}w\texttt{!(}f\texttt{[}y\texttt{,}g\texttt{[}x_1\texttt{],}y'\texttt{])]>} \longrightarrow \\ &\quad z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,*,}x'\texttt{]][<-}g\texttt{[}x_1\texttt{]];}A\texttt{;}w\texttt{!(}f\texttt{[}y\texttt{,}y'\texttt{])!(}g\texttt{[}x_1\texttt{])>} \end{aligned} \tag{10}$$

$$z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,}g\texttt{[}x_1\texttt{],}x'\texttt{]];}A\texttt{;}w\texttt{!(}f\texttt{[--])>}\longrightarrow \tag{11}$$
$$z\texttt{::<}u\{\texttt{<-}f\texttt{[}x\texttt{,}g\texttt{[}x_1\texttt{],}x'\texttt{]}\}\texttt{;}A\texttt{;}w\texttt{>}$$

$$z\texttt{::<}u\texttt{[<-op}\{x\}\texttt{[--]];}A\texttt{;}w\texttt{!op}\{x\texttt{[--]}\}\texttt{>}\longrightarrow \tag{12}$$
$$z\texttt{::<}u\{\texttt{<-op}\{x\{\texttt{--}\}\}\texttt{;}A\texttt{;}w\texttt{>}$$

$$z\texttt{::<}u\texttt{[<-var}\{x\}\texttt{[--]];(}A\texttt{,(var}\{x\}\texttt{[--]->}t\texttt{),}A'\texttt{);}w\texttt{!(var}\{x\}\texttt{[--])>}\longrightarrow$$
$$z\texttt{::<}u\{\texttt{<-}t\}\texttt{;(}A\texttt{,(var}\{x\}\texttt{[--]->}t\texttt{),}A'\texttt{);}w\texttt{>} \tag{13}$$

$$z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,*,}x'\texttt{]]}\{\texttt{<-}t\}\texttt{;}A\texttt{;}w\texttt{>}\longrightarrow \tag{14}$$
$$z\texttt{::<}u\texttt{[<-}f\texttt{[}x\texttt{,}t\texttt{,}x'\texttt{]];}A\texttt{;}w\texttt{>}$$

Finally, the actual replacement is handled in the expected way by rule (15) below. Note that (15) can be applied only when the former righthand side of a rule succesfully matched has been fully substituted, that is, when there are no substitution tasks pending.

$$T\texttt{@}u\texttt{[<-}t\texttt{]::<*}\{\texttt{<-}t'\}\texttt{;}A\texttt{;-->}\longrightarrow T\texttt{@}u\texttt{[<-}t'\texttt{]} \tag{15}$$

A.3 An Example of Rewriting in U

Let *Foo* be a rewrite theory with $\Sigma_0 = \{\texttt{a},\texttt{b}\}$, $\Sigma_1 = \{\texttt{h}\}$, $\Sigma_2 = \{\texttt{f}\}$, $Var = \{\texttt{x}\}$, $E = \emptyset$, and $R = \{\texttt{f(a,x)} \longrightarrow \texttt{h(x)}\}$. It is clear that the following holds in rewriting logic,

$$Foo \vdash \texttt{h(f(a,b))} \longrightarrow \texttt{h(h(b))}.$$

We will prove below that

$$U \vdash \overline{Foo \vdash \texttt{h(f(a,b))} \longrightarrow \texttt{h(h(b))}}$$

holds as well, which, when further expanded, becomes

$$U \vdash \overline{Foo}\ \texttt{@}\ \overline{\texttt{h(f(a,b))}} \longrightarrow \overline{Foo}\ \texttt{@}\ \overline{\texttt{h(h(b))}}.$$

We present this proof to illustrate how rewriting in the universal theory U simulates rewriting in an object theory. To ease readability, when $\overline{Foo}$ does not change in a rewriting step, we use it as a shorthand for its fully expanded definition, that is,

$$\overline{Foo} = \texttt{<var\{x\}[--];--;op\{f\}[op\{a\}[--],var\{x\}[--]]=>op\{h\}[var\{x\}[--]]>}$$

In the following we show a rewriting derivation of $\overline{Foo}$ @ $\overline{\mathtt{h(h(b))}}$ from $\overline{Foo}$ @ $\overline{\mathtt{h(f(a,b))}}$.

The first step is to select a redex using rules (1) and (2). In our case, we want to select $\overline{\mathtt{f(a,b)}}$. The label '[$n$]' indicates that the derivation is a replacement step using rule n of U.

$\overline{Foo}$ `@ op{h}[op{f}[op{a}[--],op{b}[--]]]`

$\downarrow$ [1]

$\overline{Foo}$ `@ *[<-op{h}[op{f}[op{a}[--],op{b}[--]]]]`

Note that rule (2) can not be inmediately used for another replacement step. We need first to take several equality steps using the equations of U. The label 'EqU' indicates each of these steps.

$\overline{Foo}$ `@ *[<-op{h}[op{f}[op{a}[--],op{b}[--]]]]`

$\downarrow$ EqU

$\overline{Foo}$ `@ *[<-op{h}[op{f}[op{a}[--],op{b}[--]],--]]`

$\downarrow$ EqU

$\overline{Foo}$ `@ *[<-op{h}[--,op{f}[op{a}[--],op{b}[--]],--]]`

Now we can use rule (2) to select $\overline{\mathtt{f(a,b)}}$.

$\overline{Foo}$ `@ *[<-op{h}[--,op{f}[op{a}[--],op{b}[--]],--]]`

$\downarrow$ [2]

$\overline{Foo}$ `@ *[<-op{h}[--,*,--]][<-op{f}[op{a}[--],op{b}[--]]]`

To simplify the exposition, in the subsequent derivations we will not show explicitly the equality steps taken before or after a replacement step. The label '$EqU, [n]$' will indicate that some equality steps have been taken before and/or after a replacement step using rule n of U.

We now have to select a rule in $\overline{Foo}$. Obviously, we want to select $\overline{\mathtt{f(a,x)} \longrightarrow \mathtt{h(x)}}$. For that we use rule (3). Note that rule (3) also sets aside the matching problem between $\overline{\mathtt{f(a,x)}}$ and $\overline{\mathtt{f(a,b)}}$.

```
<var{x}[--];--;op{f}[op{a}[--],var{x}[--]]=>op{h}[var{x}[--]]> @
        *[<-op{h}[--,*,--]][<-op{f}[op{a}[--],op{b}[--]]]
```

$\downarrow$ [3]

```
<var{x}[--];--;op{f}[op{a}[--],var{x}[--]]=>op{h}[var{x}[--]]> @
        *[<-op{h}[--,*,--]][<-op{f}[op{a}[--],op{b}[--]]]
::<*[<-op{f}[op{a}[--],var{x}[--]]];*[<-op{f}[op{a}[--],op{b}[--]]];
                 --/var{x}[--];op{h}[var{x}[--]]>
```

The context will remain unchanged during the phases of matching and substitution. Therefore, to ease readability we will use *context* as a shorthand for it, that is,

```
context =  *[<-op{h}[--,*,--]][<-op{f}[op{a}[--],op{b}[--]]]
```

Since $\overline{\texttt{f(a,x)}}$ and $\overline{\texttt{f(a,b)}}$ have the same top operator we can use rule (5) to continue the matching process in their subterms. In this way we select $\overline{\texttt{x}}$ from $\overline{\texttt{f(a,x)}}$ and $\overline{\texttt{b}}$ from $\overline{\texttt{f(a,b)}}$.

```
Foo @ context::<*[<-op{f}[op{a}[--],var{x}[--]]];
*[<-op{f}[op{a}[--],op{b}[--]]];--/var{x}[--];op{h}[var{x}[--]]>

↓ [5]

Foo @ context::<*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];*[<-op{f}[op{a}[--],*,--]][<-op{b}[--]];
--/var{x}[--];op{h}[var{x}[--]]>
```

We now can apply rule (6) to identify $\overline{\texttt{f(a,x)}}$ and $\overline{\texttt{f(a,b)}}$ keeping as a result the difference we have found, that is, assigning $\overline{\texttt{b}}$ to $\overline{\texttt{x}}$.

```
Foo @ context::<*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];*[<-op{f}[op{a}[--],*,--]][<-op{b}[--]];
--/var{x}[--];op{h}[var{x}[--]]>

↓ [6]

Foo @ context::<*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];
(var{x}[--]->op{b}[--])/--;op{h}[var{x}[--]]>
```

We can then apply rule (9), which is equivalent to saying that the matching is succesful, and, therefore, that the resulting substitution should be used in a reified replacement step. Note, however, that we did not carry out a similar process of matching in the other subterms of $\overline{\texttt{f(a,x)}}$ and $\overline{\texttt{f(a,b)}}$.

```
Foo @ context::<*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];*[<-op{f}[op{a}[--],*,--]][<-var{x}[--]];
(var{x}[--]->op{b}[--])/--;op{h}[var{x}[--]]>

↓ [9]

Foo @ context::<*[<-op{h}[var{x}[--]]];var{x}[--]->op{b}[--];op{h}[var{x}[--]]>
```

Now we have to carry out the actual reified replacement step. First we have to apply the substitution obtained in the matching process to the reified righthand side of $\overline{\texttt{h(x)}}$. Note that $\overline{\texttt{h(x)}}$ is in fact the current task. Rule (10) selects $\overline{\texttt{x}}$ to continue the substitution process. Therefore, it deletes $\overline{\texttt{x}}$ as a subtask of $\overline{\texttt{h(x)}}$ and replaces it as the current task.

```
Foo @ context::<*[<-op{h}[var{x}[--]]];var{x}[--]->op{b}[--];op{h}[var{x}[--]]>

↓ [10]

Foo @ context::<*[<-op{h}[--,*,--]][<-var{x}[--]];var{x}[--]->op{b}[--];(op{h}[--])!var{x}[--]>
```

Rule (13) performs the actual substitution of the variable $\overline{\texttt{x}}$ by $\overline{\texttt{b}}$. Rule (13) also declares $\overline{\texttt{b}}$ as a fully substituted subterm, and deletes $\overline{\texttt{x}}$ as the current task.

$\overline{\mathit{Foo}}$ `@` *context*`::<*[<-op{h}[--,*,--]][<-var{x}[--]];var{x}[--]->op{b}[--];(op{h}[--])!var{x}[--]>`

$\downarrow$ [13]

$\overline{\mathit{Foo}}$ `@` *context*`::<*[<-op{h}[--,*,--]]{<-op{b}[--]};var{x}[--]->op{b}[--];op{h}[--]>`

Now we can use rule (14) to recompose $\overline{\texttt{h(b)}}$ before continuing with the next current substitution task.

$\overline{\mathit{Foo}}$ `@` *context*`::<*[<-op{h}[--,*,--]]{<-op{b}[--]};var{x}[--]->op{b}[--];op{h}[--]>`

$\downarrow$ [14]

$\overline{\mathit{Foo}}$ `@` *context*`::<*[<-op{h}[op{b}[--]]];var{x}[--]->op{b}[--];op{h}[--]>`

In this case, however, the current substitution task has no other subtasks. Therefore, we can apply rule (11) to declare $\overline{\texttt{h(b)}}$ as a fully substituted subterm, and to delete the empty current task.

$\overline{\mathit{Foo}}$ `@` *context*`::<*[<-op{h}[op{b}[--]]];var{x}[--]->op{b}[--];op{h}[--]>`

$\downarrow$ [11]

$\overline{\mathit{Foo}}$ `@` *context*`::<*{<-op{h}[op{b}[--]]};var{x}[--]->op{b}[--];-- >`

Note that now we have no other substitution tasks pending. Therefore, we can use rule (15) to finally perform the actual reified replacement step. Since rule (15) will change *context* we can not use it any more as a shorthand.

$\overline{\mathit{Foo}}$ `@ *[<-op{h}[--,*,--]][<-op{f}[op{a}[--],op{b}[--]]]::`
`<*{<-op{h}[op{b}[--]]};var{x}[--]->op{b}[--];-- >`

$\downarrow$ [15]

$\overline{\mathit{Foo}}$ `@ *[<-op{h}[--,*,--]][<-op{h}[op{b}[--]]]`

To finish our rewriting derivation of $\overline{\mathit{Foo}}$ `@` $\overline{\texttt{h(h(b))}}$ from $\overline{\mathit{Foo}}$ `@` $\overline{\texttt{h(f(a,b))}}$, we just have to recompose $\overline{\texttt{h(h(b))}}$ using in the opposite directions rules (2) and (1). The label '$[-n]$' indicates that the derivation is a replacement step using from right to left the bidirectional rule n of U.

$\overline{\mathit{Foo}}$ `@ *[<-op{h}[--,*,--]][<-op{h}[op{b}[--]]]`

$\downarrow$ $[-2]$

$\overline{\mathit{Foo}}$ `@ *[<-op{h}[op{h}[op{b}[--]]]]`

$\downarrow$ $[-1]$

$\overline{\mathit{Foo}}$ `@ op{h}[op{h}[op{b}[--]]]`