A Comprehensive Description of Consistent Document Engineering

Jan Scheffczyk
Uwe M. Borghoff
Peter Rödig
Lothar Schmitz

Report 2003-03
December 2003

University of the Federal Armed Forces Munich

Department of

COMPUTER SCIENCE

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Abstract

When a group of authors collaboratively edits interrelated documents, consistency problems occur almost immediately. Current document management systems (DMS) provide useful mechanisms such as document locking and version control, but often lack consistency management facilities. If at all, consistency is "defined" via informal guidelines, which do not support automatic consistency checks.

In this paper, we propose to use explicit *formal* consistency rules for heterogeneous repositories that are managed by traditional DMS. Rules are formalized in a variant of first-order temporal logic. Functions and predicates, implemented in a full programming language, provide complex (even higher-order) functionality. A static type system supports rule formalization, where types also define (formal) document models. In the presence of types, the challenge is to smoothly combine a first-order logic with a useful type system including subtyping. In implementing a *tolerant* view of consistency, we do not expect that repositories satisfy consistency rules. Instead, a novel semantics precisely pinpoints inconsistent document parts and indicates *when*, *where*, and *why* a repository is inconsistent.

Speed is a key issue in our approach towards tolerating inconsistencies. We, therefore, developed efficient techniques for consistency rule evaluation. Our strategy is known from databases: (1) static analysis characterizes and simplifies consistency rules and (2) at run-time rules are evaluated incrementally. The major differences to databases are that we consider informal documents and explicitly allow inconsistencies. Consequently, we lack formal update descriptions and cannot rely on consistency prior to updates.

Our major contributions are

1. the use of explicit formal rules giving a precise (and still comprehensible) notion of consistency,
2. a static type system securing the formalization process,
3. a novel semantics pinpointing inconsistent document (parts) precisely,
4. efficient techniques for consistency rule evaluation, and
5. a design of how to automatically check consistency for document engineering projects that use existing DMS.

We have implemented a prototype of a consistency checker. Applied to real world content, it shows that our contributions can significantly improve consistency in document engineering processes.

1 Introduction

Larger works of writing – e.g., books, technical documentations, or software specifications – contain many documents[1] that are collaboratively and concurrently edited by a number of authors. Mainstream DMS store documents in repositories and provide version control, rights management etc. Usually, authors aim to produce an overall consistent work, i.e., certain relations between the documents are maintained. These relations, however, are mostly implicit and vague, e.g., "Links inside documents must have a valid target." In order to achieve consistency, authors have to spend much time re-reading and revising their own and related documents. Worse, each check-in to the repository potentially invalidates consistency. Larger companies define guidelines and policies for writing; but still, a *human* reviewer is required to enforce them. What prevents automatic checks is that guidelines are implicit or at least informal. Yet recent proposals for managing XML documents appear to neglect these shortcomings [45].

We, therefore, propose the use of explicit *formal* consistency rules to manage consistency in heterogeneous repositories. *Strict* rules must be adhered to, whereas *weak* rules may be violated. In addition, each rule has a priority, which allows to gauge the impact of an inconsistency. Rules may restrict how documents evolve in time – hence we employ *temporal* logic. Since rule design is a complex task, a static type system helps to define syntactically well-formed consistency rules. Being aware that check-ins potentially violate consistency rules, we replace traditional boolean semantics by a novel semantics, which pinpoints inconsistencies within documents.

This provides automatic checks indicating precisely *when*, *where*, and *why* inconsistencies occur and opens the road to *tolerating* inconsistencies – a necessity in many areas [12, 24]. For example, if documents evolve at different rates enforcing consistency may cause deadlocks. Consistency rules may be too strict for some purposes – an inconsistency may indicate an exception or a design alternative. Finally, the impact of an inconsistency can be low compared to the costs of resolving it.

Speed is a key to user acceptance. After a check-in to the repository, authors want to know almost immediately whether this check-in is accepted and how it meets the consistency rules. Therefore, we need *efficient* tech-

[1]We use the term "document" informally here, being aware that documents do not need to follow a formal document model. Using XML, however, we gain several advantages.

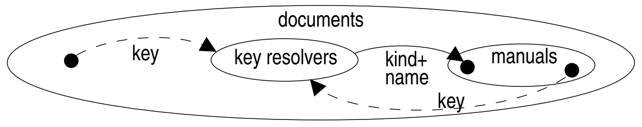


Figure 1: Example repository

niques for evaluating first-order linear temporal formulae against a heterogeneous repository while retaining our tolerant semantics. We improve efficiency by two measures: The first is to reduce checking complexity by static analysis that is performed prior to actually evaluating consistency rules. The second is to reduce the complexity of the checking algorithm itself during evaluation. Static rule analysis attempts to decrease the number of rules to be re-evaluated at a check-in by "localizing" and filtering, and to decrease the static computational complexity of a rule by rewriting. Our evaluation algorithm dynamically reduces quantifier domains. Since we allow inconsistencies in previous repository states we employ an incremental algorithm that makes heavy use of previous consistency reports.

The following running example illustrates the formal part of our paper. In practice we come across far more complex examples of the same kind, e.g., in [20]. Documents linking to the current version of another document and persistent URLs [36] impose closely related problems.

Example 1 Assume we want to archive manuals over a long period of time. Documents (and manuals) reference manuals through a key – see Fig. 1. Since names and kinds of manuals may change over time, we need key resolvers mapping keys to their semantics, e.g., manual kind and name. There may exist many key resolvers, the actual names of which are hidden from authors. To ensure consistency we require that (1) (two-step) links are valid and (2) names and kinds of manuals are invariant over time. For example, a referenced key k is invalid if no resolver contains k, or a resolver maps k to a manual that does not exist, or a resolver maps k to an existing manual m but m's kind is different from the kind of k's resolver entry. □

This paper[2] is organized as follows: Sect. 2 shows how we integrate our work into document engineering processes that are based on DMS. Sect. 3 describes how consistency rules are formalized. Sect. 4 sketches our static type system. Sect. 5 shows our tolerant semantics and how it helps to precisely detect inconsistencies. We address the issue of efficient rule evaluation in Sect. 6. Notes on our implementation and results from our experiments can be found in Sect. 7. In Sect. 8 we discuss some related work. A summary and directions for future research are given in Sect. 9.

[2]Important parts of this report are published as [32] and [33].

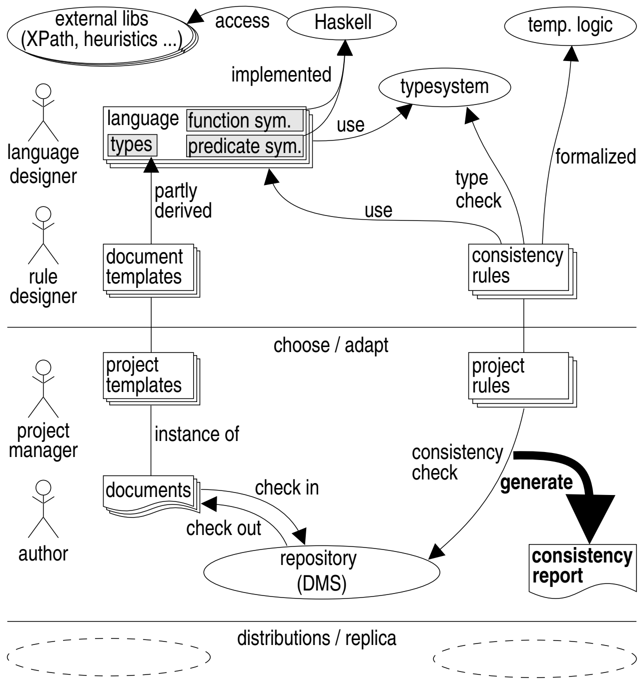


Figure 2: System overview (ovals mark fixed components; rectangles mark customizable components)

2 Using Consistency Rules in Document Engineering

We consider formalizing consistency rules a complex process. In order to handle this additional complexity, we divide formalization into different tasks and supply tools to every stakeholder (see Fig. 2).

From a repository, authors typically check out working copies of documents, modify them, and finally check them in again. Among other things, a classic DMS manages concurrent check-ins, author rights, version control, and repository backup and distribution. We design our consistency checker in a way that makes only few assumptions about the DMS. We require only

1. a facility to access past and present document versions,
2. a locking mechanism that prevents check-ins during a consistency check, and
3. that the DMS signals the documents added or changed by an update.

Note that we make no assumptions about the document model of the DMS, such that our extensions also apply to revision control systems like CVS [6].

A *rule designer* formalizes consistency rules. Rules define what it means for the repository to be consistent – they reflect wishes from the "administrator" perspective. We check the repository for consistency w.r.t. the rules formalized at given events, e.g., document check-in or the end of a development phase. Checking the repository for consistency generates a *consistency report* to which we can react in various ways. On violation of a weak rule the system could inform those authors who have checked out (now) inconsistent documents. If, however, a strict rule is violated the system will reject the check-in in question. In addition, the rule designer creates templates for documents. These will be DTDs or Schemas if XML is used as document format.

Consistency rules use function and predicate symbols from a domain specific language. This makes the rules completely independent of concrete document formats, which can be changed without affecting rules. A static type system ensures that rules are well-typed w.r.t. the language used and that rules are well-formed first-order formulae. If, e.g., a predicate symbol $=$ requires two arguments of the same type the careless application of $=$ to a number and a string must be rejected because it is meaningless. Thus, without accessing repository data, our static type system decides on the syntax level whether a consistency rule can possibly make sense.

A *language designer* declares valid symbols and their types in a signature, and implements symbol semantics in the statically typed functional programming language Haskell[3] [26]. Haskell provides access to other libraries via a foreign function interface [7]. Such libraries offer sophisticated functionality, e.g., for parsing documents or heuristics for semantic content analysis [31]. Formal document types (usually corresponding to document templates) can be expressed by record or variant types. This makes our approach independent of any particular document format and facilitates heterogeneous repositories. For our running example, a record type `ResD` defines the formal structure of key resolvers. If XML is used as document format, document types and parser function implementations can be derived from XML DTDs [42]. Note that complex programming tasks are hidden from the rule designer who only needs simple first-order logic.

[3]For example, the language designer would declare $=$ to have the polymorphic type $\forall\alpha.\alpha \times \alpha$ (α denotes a type variable, $\times$ separates argument types). A Haskell function then defines the meaning of $=$, e.g., via instances of the type class `Eq`.

For specific projects, the *project manager* chooses consistency rules and document templates. In some cases adaptions will be necessary, e.g., the document templates get another company logo or some rules are weakened. Of course, major changes in the document templates cause adaptions to the corresponding formal document types, which may involve further changes in the language. But typically most projects require only layout related adaptions.

In the rest of this paper we shall concentrate on the work of the rule designer and the language designer, and on the generation of comprehensive and precise consistency reports in particular.

3 Formalizing Consistency Rules

Our examples show that we need a very expressive language to formalize consistency rules. We require a temporal component as well as complex functions and predicates. Besides being precise rule design should be comfortable: Once defined, we want to reuse functions and predicates as often as possible. This is why we have decided to formalize consistency rules in a *full* first-order temporal predicate logic.[4] We allow polymorphic (even higher-order) functions and predicate to facilitate comfort and reuse. The challenge is, therefore, to smoothly combine a first-order logic with higher-order functions and predicates. Our type checker, sketched in Sect. 4, provides significant support for this.

3.1 Overview

Consistency rules are statements about repository states, which we call timestamps because they represent given points in time. Similar to our architecture, our abstract syntax consists of two parts: (1) the rule designer expresses consistency rules in a first-order temporal logic; (2) the language designer declares symbols (that can be used in rules) in a signature and implements symbol semantics in Haskell.

Rule designers formalize consistency rules in a variant of two-sorted temporal first-order predicate logic with linear time and equality [1, 15]. The two-sorts approach to temporal logic introduces a new temporal sort Time. To each non-temporal predicate or function symbol a timestamp is added. We call these symbols partially temporal. Fully temporal predicate or function symbols

[4]We employ a first-order logic only because based on our experiments we have not seen the need for higher-order logic.

(1) At each time links must be valid:
"At any time t we have for all documents x at t that for all their referenced keys k there exists a key definition d (in one of the resolvers) for k and there exists a manual m with name and kind as defined by d."

$$\begin{aligned}\phi_1 = \;& \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\ & \exists\, d \in \texttt{concatMap}(t, \texttt{kDefs}, \texttt{repResDs}(t)) \bullet k = \texttt{key}(d) \wedge \\ & \left(\begin{array}{l} \exists\, m \in \texttt{repManDs}(t) \bullet \\ \texttt{id}(m) = \texttt{kId}(d) \;\wedge\; \texttt{kind}(m) = \texttt{kKind}(d) \end{array}\right)\end{aligned}$$

(2) Names and kinds of manuals are invariant over time:
"At all times t_1 and t_2 we have for manuals m_1 at t_1 that if t_1 is smaller than or equal to t_2 there exists a manual m_2 at t_2 that has the same name and kind as m_1."

$$\begin{aligned}\phi_2 = \;& \forall\, t_1 \in \texttt{repStates} \bullet \forall\, t_2 \in \texttt{repStates} \bullet \\ & \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet t_1 \leq t_2 \Rightarrow \\ & \left(\begin{array}{l} \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{id}(m_1) = \texttt{id}(m_2) \;\wedge\; \texttt{kind}(m_1) = \texttt{kind}(m_2) \end{array}\right)\end{aligned}$$

Figure 3: Example consistency rules
(for function and predicate symbols see Fig. 4)

Type definitions in **T**

`String`			strings
$[\alpha]$	$=$	$\{[], (:) : \alpha \times [\alpha]\}$	lists
`Doc`	$=$ `Doc`	$\{$`id`$:$`String`$,$ `time`$:\mathsf{Time}\}$	documents
`ManD` $<_{\mathsf{S}}$ $\{$`Doc`$\}$	$=$ `Man`	$\{$`kind`$:$`String`$\}$	manuals
`ResD` $<_{\mathsf{S}}$ $\{$`Doc`$\}$	$=$ `Res`	$\{$`kDefs`$:[$`KDef`$]\}$	key resolvers
`KDef`	$=$ `KDef`	$\{$`key`$,$ `kId`$,$ `kKind`$:$`String`$\}$	key definition

Predicate symbol definitions in **P**

$=^*$	:	$\forall\alpha.\mathsf{Time} \times \alpha \times \alpha$	equality
$\leq$	:	$\mathsf{Time} \times \mathsf{Time}$	timestamp ordering

Function symbol definitions in **F**

`concatMap`	:	$\forall\alpha,\beta.\mathsf{Time} \times (\mathsf{Time} \times \alpha \to [\beta]) \times [\alpha] \to [\beta]$	Haskell `concatMap`
`repDs`	:	$\mathsf{Time} \to [$`Doc`$]$	get documents
`repManDs`	:	$\mathsf{Time} \to [$`ManD`$]$	get manuals
`repResDs`	:	$\mathsf{Time} \to [$`ResD`$]$	get resolvers
`refs`*	:	$\mathsf{Time} \times$ `Doc` $\to [$`String`$]$	referenced keys
`repStates`	:	$[\mathsf{Time}]$	all repository states

Figure 4: Example types and symbols

only have temporal arguments (and results). Quantifiers iterate over variables of the sort Time, too. We use the two-sorts approach because (1) timestamp variables make temporal logic more expressive, (2) rule designers do not need to learn temporal connectives, and (3) the introduction of types makes the two-sorts approach straightforward. In our setting Time is a type.

Example 2 The rule designer defines consistency rules like those shown in Fig. 3. If a partially temporal symbol does not depend on time we can omit its temporal parameter (see, e.g., `refs`(x) in ϕ_1). We call such applications "unsaturated." Predicate symbols are written in infix notation for convenience.

We formalize rule ϕ_1 by first quantifying over all states in the repository, provided by `repStates`. Then, for each state t we need the current documents x, obtained from `repDs`(t), and the referenced keys k therein, computed via `refs`(x). Since `refs` is independent of time we omit its temporal parameter. For every k there must exist a key definition d such that the key defined by d (computed via `key`(d)) equals the referenced key k. Furthermore, there must exist a manual m whose name equals the identifier mentioned by d (we get the current manuals via `repManDs`(t)). In addition, the kind of m must equal the kind mentioned by d. We get the current key definitions d via `concatMap`$(t,$ `kDefs`$,$ `repResDs`$(t))$. This computes the current resolver documents (`repResDs`(t)) and extracts the key definitions from the resolver documents. So finally, d iterates over all key definitions inside all resolvers. Essentially, `concatMap` behaves like a universal quantifier here. Assume we wanted to neglect case when comparing manual kinds. Then we simply use a different predicate symbol, say $\equiv$. This is beyond the possibilities of XLink.

In rule ϕ_2 we quantify twice over time because we have to relate different versions of manuals, where one version (at state t_1) is "older" than the other (at state t_2). $\square$

The rules in Fig. 3 appear more complex than the vague "ideas" in our introductory example. Formal rules are much more *precise* than informal guidelines and give no room for misinterpretations. We consider this an important feature of formalization: When formalizing consistency rules we have to decide what we really want. In our experience this contributes to a common understanding of what consistency actually means. This is vital for any collaborative work.

Example 3 For the rules formalized the language designer defines symbols and types like those shown in Fig. 4, where $<_{\mathsf{S}}$ denotes an explicit subtype relation (called *subtype axiom*) between two types. Partially temporal symbols that do not really depend on time are marked with $*$. They are subject to unsaturated application.

The variant list type $[\alpha]$ is declared as usual in functional programming – the variant constructors are $[]$ (empty list) and $(:)$ (cons). The record type `Doc` stands for a formal document, holding a name (of type `String`) and a check-in time (of type Time). That way we distinguish different document versions. We require each document type to be a subtype of `Doc`. The record type `ResD` resembles the key resolver structure (see Fig. 7 in Sect. 5 for an example). `ResD` inherits all record labels from its supertype `Doc`. Our notion of subtyping resembles XML Schema subtyping via extension and restric-

ϕ, ψ	$::= p(e_1, \dots, e_n)$	atomic formula
	$\mid \phi \cdot \psi$	junction, $\cdot \in \{\vee, \wedge, \Rightarrow\}$
	$\mid \neg\phi$	negation
	$\mid Q\ x \in e \bullet \phi$	quantified formula, $Q \in \{\forall, \exists\}$
e	$::= x$	variable
	$\mid s$	symbol, $s \in \{f, l, k\}$
	$\mid s(e_1, \dots, e_n)$	application
	$\mid K_R\{\overline{l_i = e_i}\}$	record construction
	$\mid \mathsf{case}(e_0, e, \{\overline{k_i} \to \overline{s_i}\})$	variant deconstruction
	$\mid \mathsf{case}(e_0, e, V, \{\overline{k_i} \to \overline{s_i}\})$	variant deconstruction for V
	$\mid e :: \tau$	type annotation

Figure 5: Formal consistency rule syntax

tion. Record and variant type definitions induce new symbols (labels and constructors, respectively), which do not depend on time. For example, `ResD` induces `kDefs`* : Time $\times$ `ResD` $\to$ [`Item`].

Function symbols starting with `rep` provide access to documents inside the repository at a given time. For a document d, `refs`$^*(d)$ returns all referenced keys. It is independent of time since documents already contain their check-in time. `concatMap`(t, f, xs) applies $f(t, .)$ to each member in xs and concatenates the result lists. Due to their optional temporal parameter record labels can serve as parameter for `concatMap`, e.g., in `concatMap`$(t,$ `kDefs`$,$ `repResDs`$(t))$. $\square$

Next, we shall introduce the formal means required for the example above.

3.2 Abstract Syntax

Fig. 5 summarizes the abstract syntax of consistency rules – the rule designer toolset. More often than not, formulae are standard first-order formulae. A quantifier Q introduces a bound variable x, restricted by a term e that evaluates to the domain of x. We use terms for quantifier restrictions to easily identify variable domains. We call closed formulae *consistency rules* and non-empty finite rule sets *rule systems*. Typically, a rule contains additional metadata, e.g., weakness annotations and its priority.

Our notion of terms corresponds to Nordlander's Haskell extension O'Haskell [22], which extends the Haskell type system with *subtypes*. Thus, we regard function symbols f, record labels l, and variant constructors k as terms, too. In contrast to O'Haskell, we let an explicit record constructor K_R construct a record of type R in order to guide type inference. Both case constructs take an additional temporal argument e_0, fixing the time of evaluation. An optional variant type constructor V uniquely identifies the type of e, thus avoiding ambiguities that may arise from subtyping.

In the logic used here it is *undecidable* whether a formula is satisfiable. This, however, is no issue in our setting because rules are evaluated against concrete repositories. What we are really interested in are concrete inconsistency pointers generated from complex consistency rules! We, therefore, sacrifice decidability for expressivity. For some applications it might be interesting to know whether the rules are satisfiable or whether some rules are implied by others. Both questions reduce to the implication problem in predicate logic, which has been proven to be undecidable (see, e.g., [19]).[5] Thus, we only perform analyses that detect some cases of rule contradiction and implication [8], but not all of them.

The language designer declares valid symbols for rule definition in a *signature*. Each symbol has a unique name. Partially temporal predicate symbols in $\dot{\mathbf{P}}$ have a timestamp as first parameter; their meaning may change over time. Fully temporal predicate symbols in $\ddot{\mathbf{P}}$ have only temporal parameters. A similar distinction holds for function symbols. We require a predicate symbol $\leq$, interpreted by a *total* ordering relation to facilitate linear time logic. We further require a function symbol $*$, which is "plugged" into unsaturated applications prior to type checking. The type structure $\Omega(\mathbf{T})$ contains all types properly constructed from the type constructors in $\mathbf{T}$ (see below). In contrast, the temporal type structure $\ddot{\Omega}(\mathbf{T}) \subseteq \Omega(\mathbf{T})$ contains only temporal types. These are constructed from Time using non-atomic type constructors in $\mathbf{T}$, e.g., [Time], [[Time]] $\in \ddot{\Omega}(\mathbf{T})$. The type constructor set $\mathbf{T}$ also contains document type constructors, resembling document templates in the repository. A subtype theory $\mathbf{S}$ contains subtype axioms about record and variant types, respectively. Record type constructor definitions R induce a record constructor K_R and a label environment $\prod_{\widehat{R}}$ containing record labels and their types. For example, the label environment $\prod_{\widehat{\texttt{ResD}}}$ contains `kDefs`* : Time$\times$`ResD` $\to$ [`Item`], `id`* : Time$\times$`ResD` $\to$ `String`, and `time`* : Time $\times$ `ResD` $\to$ Time. Variant type constructor definitions V induce a constructor environment $\sum_{\widehat{V}}$ containing variant constructors; e.g., the constructor environment $\sum_{\widehat{[\_]}}$ contains $[]^*$: Time $\to [\alpha]$ and $(:)^*$: Time $\times\, \alpha \times [\alpha] \to [\alpha]$.

Definition 1 *A* signature $\Sigma = (\mathbf{T}, \mathbf{S}, \prod, \mathbf{K}, \sum, \mathbf{P}, \mathbf{F})$ *contains:*[6]

- *a type constructor set* $\mathbf{T}$ *(we require a unary list type constructor* $[\_] \in \mathbf{T}$*);*
- *a subtype theory* $\mathbf{S}$ *holding subtype axioms over* $\Omega(\mathbf{T})$*;*

[5]There exist decidable subsets in temporal predicate logic [15]. A restriction to these subsets, however, would severely limit the range of applications using our approach.

[6]$\uplus$ denotes disjoint union, $:$ denotes a "has type" relation.

T	$::=$	A_0, R_n, V_n	type constructors (atomic, record, variant)
τ_g, ρ_g	$::=$	α, β	ground type variable
	$\mid$	$T_n\ \tau_{g_1} \dots \tau_{g_n}$	constructed type
	$\mid$	Time	temporal type
	$\mid$	Top	supertype of all types
τ, ρ	$::=$	τ_g	ground type
	$\mid$	ν	type variable
	$\mid$	$\tau_1 \times \ldots \times \tau_n \rightarrow \tau_g$	function type
τ_p	$::=$	$\tau_1 \times \ldots \times \tau_n$	predicate type
D, C	$::=$	$\{\bar{\tau} \leq \bar{\rho}\}$	subtype constraints
σ	$::=$	$\forall \bar{\alpha}.(\tau \mid C)$	function type scheme
σ_p	$::=$	$\forall \bar{\alpha}.(\tau_p \mid C)$	predicate type scheme

Figure 6: Formal type syntax

- *a record type label environment* $\prod_{\widehat{R}} \in \prod$ *and a record constructor* $K_R \in \mathbf{K}$ *for each record type constructor* $R \in \mathbf{T}$*;*
- *a variant type constructor environment* $\sum_{\widehat{V}} \in \sum$ *for each variant type constructor* $V \in \mathbf{T}$*;*
- *predicate symbols* $p : \tau_p \in \mathbf{P}$*; where*
  $\mathbf{P} = \dot{\mathbf{P}} \uplus \ddot{\mathbf{P}}$,
  $\dot{\mathbf{P}} = \bigcup \dot{\mathbf{P}}_{\mathsf{Time} \times \tau_1 \times \ldots \times \tau_n} (\tau_i \in \Omega(\mathbf{T}))$*, and*
  $\ddot{\mathbf{P}} = \bigcup \ddot{\mathbf{P}}_{\tau_1 \times \ldots \times \tau_n} (\tau_i \in \ddot{\Omega}(\mathbf{T}))$, $\leq\ : \mathsf{Time} \times \mathsf{Time} \in \ddot{\mathbf{P}}$;
- *function symbols* $f : \tau \in \mathbf{F}$*; where*
  $\mathbf{F} = \dot{\mathbf{F}} \uplus \ddot{\mathbf{F}}$,
  $\dot{\mathbf{F}} = \bigcup \dot{\mathbf{F}}_{\mathsf{Time} \times \tau_1 \times \ldots \times \tau_n \rightarrow \tau_g}\ (\tau_i, \tau_g \in \Omega(\mathbf{T}))$*, and*
  $\ddot{\mathbf{F}} = \bigcup \ddot{\mathbf{F}}_{\tau_1 \times \ldots \times \tau_n \rightarrow \tau_g}\ (\tau_i, \tau_g \in \ddot{\Omega}(\mathbf{T}))$, $* : \mathsf{Time} \in \ddot{\mathbf{F}}$.

$\square$

Our static type system is based on functional programming languages. Many ideas carry over from [23]. Fig. 6 summarizes our type language. Type constructors $T \in \mathbf{T}$ have a fixed arity. The type Time models discrete repository states.

We apply some restrictions to ensure that (higher-order) functions result only in ground types (τ_g) and thus the type system supports first-order logic properties: we do not quantify over functions. In contrast to most functional programming languages, we restrict ourselves to full application of a type constructor to argument types[7] and full application of functions and predicates to argument terms. To simplify notation, we use uncurried syntax in function and predicate types. Furthermore, we distinct between ground types τ_g and general types τ to ensure that the result type of a function is not a function type.

As usual, function and predicate type schemes (σ and σ_p) are quantified over all their type variables. Note the distinction between ground type variables α and general type variables ν. General type variables ν are reserved for type inference only, whereas ground type variables can be used by the language designer to define polymorphic functions. This restriction is necessary to guarantee that polymorphic functions result in a ground type.

As we already have seen, subtypes prove practical for document management: we can easily model document subsets from our example via subtypes. Subtyping rules carry over from [23]. Below, the rule SubConst defines a subtype relationship between instances $\theta\tau$ and $\theta\rho$[8] if there exists a subtype axiom $\tau <_{\mathbf{S}} \rho$ in the subtype theory $\mathbf{S}$. As usual, the subtype relation $\leq$ is transitive. For other subtyping rules see [23].

$$\text{SubConst}\quad \frac{\tau <_{\mathbf{S}} \rho}{C \vdash_{\mathbf{S}} \theta\tau \leq \theta\rho} \qquad\qquad \text{SubTrans}\quad \frac{C \vdash_{\mathbf{S}} \tau \leq \rho \quad C \vdash_{\mathbf{S}} \rho \leq \tau'}{C \vdash_{\mathbf{S}} \tau \leq \tau'}$$

We have seen that rule design is a complex task. Higher-order functions can simplify this task. But if used without caution they can result in subtle evaluation errors. Thus we provide the rule designer with a *type checker* that helps him to decide whether the rules formalized can make sense.

4 Typing Consistency Rules

The motivation behind typing consistency rules is that only for well-typed terms and formulae we can give a reasonable semantics. Since we allow language designers to define complex domain specific functions and predicates the rule designer's task of writing meaningful consistency rules becomes more complicated and must be supported. Our example rules in Fig. 3 are well-typed. If, however, we erroneously omit the function symbol `key` within $k = \texttt{key}(d)$ then a type checker ought to warn us that there is no subtype relationship between k's type `String` and d's type `KDef`. The type checker we have developed supports basic subtyping and higher-order functions, and guarantees first-order properties on the formula level.

Our combined type checking and type inference algorithm assigns a monomorphic type to each term. Monomorphism is important since it allows to treat types like primitive sorts, which can be omitted. In conjunction with the restrictions we put on function types this retains compatibility with traditional first-order logic and supports *simple* set theoretic semantics. As usual, we separate well-typedness rules from the type inference algorithm. Both are extensions to [23].

[7]Partial type constructor application would require a kind system similar to that of Haskell.

[8]θ denotes a type substitution.

Prior to type checking we run a preprocessor that augments each unsaturated application of a partially temporal symbol with a temporal parameter $*$. Well-typedness rule judgments for terms follow the pattern $C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} e : \sigma$ and read: "In the subtype theory $\mathbf{S}$, under subtype constraints C, variable assumptions Γ, and symbol assumptions Δ the term e has the type scheme σ." Rule judgments for formulae $C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} \phi$ ensure that ϕ is well-typed. The context Δ holds the types of all symbols in a signature Σ. Γ holds the types for variables introduced by quantifiers.

Next, we introduce some of the typing rules to demonstrate derivation of well-typedness.

$$\text{TypSymApp}\quad \frac{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} s : \tau_1 \times \ldots \times \tau_n \rightarrow \tau_g \qquad C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} e_i : \tau_i}{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} s(e_1, \ldots, e_n) : \tau_g}$$

The well-typedness rule TypSymApp models valid function symbol application. If a symbol s has the function type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_g$ and each term e_i $(i \in \{1 \ldots n\})$ has the type τ_i then we can apply s to e_1 through e_n and $s(e_1, \ldots, e_n)$ has type τ_g. Above we use $e_i : \tau_i$ to abbreviate $e_1 : \tau_1 \ldots e_n : \tau_n$.

$$\text{TypSub}\quad \frac{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} e : \tau \qquad C \vdash_{\mathbf{S}} \tau \leq \tau'}{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} e : \tau'}$$

The well-typedness rule TypSub covers subsumption due to subtyping. If a term e has type τ and τ is a subtype of τ' (inferred through subtyping rules) then e also has type τ'.

$$\text{TypQuant}\quad \frac{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} e : [\tau] \qquad C,\ \Gamma \cup \{x : \tau\},\ \Delta \vdash_{\mathbf{S}} \phi}{C,\ \Gamma,\ \Delta \vdash_{\mathbf{S}} Q\ x \in e \bullet \phi}$$

Well-typedness rules for formulae are straightforward. The rule TypQuant corresponds to TypLet in [23] except that the term e, defining the domain for x, must have a *list* type. Since we understand e as a domain, e must have some kind of "container" type.[9] If e has the type $[\tau]$ and the formula ϕ is well-typed under Γ, extended by the new variable assumption $x : \tau$, then the quantified formula $Q\ x \in e \bullet \phi$ is well-typed. Without loss of generality, we allow only *rectified* formulae, in which each quantifier binds a different variable. Thus Γ cannot already contain an assumption for x. Fig. 18 (in App. B) shows all typing rules for terms and formulae, where we adopt abbreviations from [23].

The following example illustrates the application of the above rules. In rule ϕ_1 we would like to derive the type `String` for the term $\mathtt{id}(m)$.

Example 4 First, our preprocessor extends $\mathtt{id}(m)$ to $\mathtt{id}(*, m)$. To infer $\mathtt{id}(*, m) : \tau_g$ the rule TypSymApp requires: $\mathtt{id} : \tau_1 \times \tau_2 \rightarrow \tau_g$, $* : \tau_1$, and $m : \tau_2$. From our example signature Δ derives $* : \mathtt{Time}$ and $\mathtt{id} : \mathtt{Time} \times \mathtt{Doc} \rightarrow \mathtt{String}$. Quantifiers push the types of their bound variables into Γ such that $m : \mathtt{ManD} \in \Gamma$. Due to subtyping we also obtain $m : \mathtt{Doc}$ (TypSub). Finally, we have $\mathtt{id}(*, m) : \mathtt{String}$. $\square$

[9]We use lists instead of sets because lists are more familiar to Haskell programmers – our language designers.

Our type inference algorithm detects whether a consistency rule is well-typed. With each term the algorithm tries to associate a monomorphic type. Although Nordlander's type inference algorithm is *incomplete* [23] we use it because it is fast and easy to comprehend. The algorithm does not involve sophisticated constraint solving, as proposed by many other authors (see, e.g., [37, 28]). Whereas complete subtype inference is NP-hard [17], Nordlander's quasi-linear algorithm is almost as fast as standard Hindley-Milner type inference. The major source for the algorithm's incompleteness is partial application of terms, which we exclude. Furthermore, we use explicit record constructors and an annotated `case` construct, providing further guidance.

Applied to a formula ϕ, our type inference algorithm returns a type substitution θ that should instantiate all type variables in ϕ to *monomorphic* types, which is not always possible for well-typed formulae (see below). Fig. 19 (in App. B) shows our type inference algorithm. Rule judgments for terms follow the pattern

$$\overset{\downarrow}{C},\ \overset{\uparrow}{\Gamma},\ \overset{\uparrow}{\Delta} \Vdash_{\mathbf{S}} \overset{\uparrow}{e} : \overset{\downarrow}{\theta}\ (\overset{\uparrow}{\sigma})$$

where an up-arrow denotes an input to the algorithm, whereas a down-arrow denotes an output. The above rule judgment reads: "Given variable assumptions Γ, symbol assumptions Δ, a subtype theory $\mathbf{S}$, an expression e, and an *expected* type scheme σ, return subtype constraints C and a substitution θ, instantiating σ." Rule judgments for formulae omit σ. Type variables ν are fresh, i.e., they do not occur in the derivation before.

Record constructors and `case` annotations determine the type of a constructed record and deconstructed variant, respectively. This is essential for subtypes that have exactly the same components as their supertypes, resulting in similar label and constructor environments, respectively.

$$\text{ChkQuant}\quad \frac{\begin{array}{c} C,\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} e : \theta([\nu]) \qquad C',\ \Gamma \cup \{x : \theta\nu\},\ \Delta \Vdash_{\mathbf{S}} \phi,\ \theta' \\ \theta'' \Vdash_{\mathbf{S}} C' \setminus C'_{\Gamma} \end{array}}{\theta''(C \cup C'_{\Gamma}),\ \Gamma,\ \Delta \Vdash_{\mathbf{S}} Q\ x \in e \bullet \phi,\ \theta'' \circ \theta'}$$

Our *monomorphic* rule ChkQuant differs from the polymorphic ChkLet rule in [23]. A quantified formula is well-typed if its domain term e has a list type $[\nu]$ and we can infer well-typedness of the body ϕ under the assumption that the bound variable x has the type $\theta\nu$. The type substitution θ (returned from inferencing

e's type) instantiates ν. Similar to [23] we solve new subtype constraints that reference type variables free in Γ.[10] For solving subtype constraints we essentially use the algorithm from [23]. Without generalizing the type of the bound variable x we allow the body ϕ to instantiate the type of x to a monomorphic type. This is not possible in Nordlander's polymorphic ChkLet rule. For example, in $\forall\, t \in [] \bullet \forall\, x \in [] \bullet \texttt{repDs}(t) = x$ the type of t is instantiated to Time and the type of x is instantiated to $[\mathsf{Doc}]$ purely by the body $\texttt{repDs}(t) = x$.

A lot of background support is provided by our tools to ensure that only well-typed rules are evaluated. We consider type checking a key ingredient in our framework because syntactical well-typedness of consistency rules is a vital precondition for their evaluation, which we discuss next.

5 Evaluating Consistency Rules

In this section we show a semantics that allows to pinpoint the trouble spots making a repository inconsistent w.r.t. a rule system. We do not restrict ourselves to finding out whether a repository fails to satisfy certain rules. Instead, we want to know *when*, *where*, and *why* documents in the repository are inconsistent.

Classic set theoretic semantics provides a boolean result only and is, therefore, not sufficient for our purposes. Evaluation of a predicate logic formula, however, assigns concrete values to quantified variables. The basic idea behind our tolerant semantics is to exploit these assignments to indicate inconsistencies. We have designed our evaluation algorithm to provide compact information that precisely characterize inconsistencies. Our tolerant semantics follows xlinkit [21], which calculates a set of links, each containing a consistency flag and a value set that makes the formula true and false, respectively. A major difference is that we also return those atomic formulae that make a rule true or false. Authors need a more detailed report when consistency rules grow large and only some of many predicates falsify a rule.

5.1 Overview

For each consistency rule we generate a *consistency report* containing a boolean result (representing boolean truth semantics) and a diagnosis set. A diagnosis $(\mathsf{C}, as, ps_t, ps_f)$ reads: "The processed formula is satisfied (Consistent) for variable assignments as due to true atomic formulae ps_t and false atomic formulae ps_f." A diagnosis $(\mathsf{IC}, as, ps_t, ps_f)$ indicates that the processed formula is violated (InConsistent). The assignment as maps variables to concrete values, e.g., timestamps or documents. Due to temporal quantification the variable assignment as tells *when* and *where* a rule is violated. The sets ps_t and ps_f describe *why* a rule is violated.

[10] C_Γ returns subtype constraints that do not reference type variables free in Γ.

State 1	doc.txt	`... as shown in manual kaA1c3 ...`
	keys.xml	`<kDef key="kaA1c3" kId="man1.xml" kKind="technical Manual"/>`
	man1.xml	`<man kind="technical Manual"> ...`
State 2	man1.xml	`<man kind="field Manual"> ...`

Figure 7: Example repository check-ins

Generated report for rule ϕ_1 ($\mathcal{R}_A[\![\phi_1]\!][]$) :

$$\left(\mathsf{False}, \left\{\left(\begin{array}{l}\mathsf{IC},\\ \left\{\begin{array}{l} t \mapsto 2, x \mapsto \{\mathsf{id} = \mathsf{doc1.txt}, \mathsf{time} = 1\},\\ k \mapsto \mathsf{kaA1c3}\end{array}\right\},\\ \emptyset,\\ \{\texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right)\right\}\right)$$

Generated report for rule ϕ_2 ($\mathcal{R}_A[\![\phi_2]\!][]$) :

$$\left(\mathsf{False}, \left\{\left(\begin{array}{l}\mathsf{IC},\\ \left\{\begin{array}{l} t_1 \mapsto 1, t_2 \mapsto 2,\\ m_1 \mapsto \left\{\begin{array}{l}\mathsf{id} = \mathsf{man1.xml}, \mathsf{time} = 1,\\ \mathsf{kind} = \mathsf{technical\ Manual}\end{array}\right\}\end{array}\right\},\\ \{t_1 \leq t_2\},\\ \{\texttt{kind}(m_1) = \texttt{kind}(m_2)\}\end{array}\right)\right\}\right)$$

Figure 8: Example generated reports

Example 5 For brevity we omit Haskell sources of function and predicate symbol implementations. Fig. 7 shows a small repository containing a documentation doc1.txt, a key resolver keys.xml, and a manual man1.xml the kind of which changed during the transition from state 1 to state 2. This introduces two inconsistencies: State 2 is inconsistent w.r.t. rule ϕ_1 because the link kaA1c3 is invalid at state 2. States 1 and 2 are inconsistent w.r.t. rule ϕ_2 because the kind of man1.xml has changed.

The automatically generated reports shown in Fig. 8 reflect these inconsistencies. The report for rule ϕ_1 lacks assignments to d and m – the repository is inconsistent w.r.t. ϕ_1 for all possible assignments to d and m, respectively. The report lacks the atomic formulae $k = \texttt{key}(d)$ and $\texttt{id}(m) = \texttt{kId}(d)$ – the key resolver contains kaA1c3, but its definition is inconsistent. This means that manuals m with the correct name were found but that they have the wrong kind: The first step of the link is consistent, the second step is inconsistent. The report for rule ϕ_2 shows that for the manual man1.xml at state 1 no manual at state 2 could be found with the same kind. The report lacks $\mathtt{id}(m_1) = \mathtt{id}(m_2)$ because there exists a manual man1.xml at state 2 – only its kind is wrong.

We can react in various plausible ways to the above reports; e.g., we might change the key resolver to point at a field manual or inform the author of man1.xml about the inconsistencies. □

Next, we present our algorithm that generates consistency reports.

5.2 Generating Reports Automatically

As usual, we interpret temporal formulae in *first-order temporal structures* $A = (TL, I)$, consisting of (1) a timeline TL interpreting temporal types, fully temporal predicate symbols, and fully temporal function symbols; and (2) a function I associating a time w with a non-temporal structure $\dot{A} = I(w)$ interpreting non-temporal types, partially temporal predicate symbols, and partially temporal function symbols. Neglecting their temporal parameter, record labels and variant constructors have the usual built-in semantics: A record label selects a field from a record, a variant constructor builds a variant. The domain of Time is the set of natural numbers $\mathbb{N}$, representing repository states. We interpret $\leq$ via the natural "less equal" ordering.

Formally, a consistency report is defined as follows:

Definition 2 *A* consistency report $(b, \mathbb{D})$ *contains a boolean value* b *(representing boolean truth semantics) and a set of diagnoses* $\mathbb{D}$*. A* diagnosis $(c, \mathrm{as}, \mathrm{ps}_t, \mathrm{ps}_f) \in \mathbb{D}$ *contains a consistency flag* $c \in \{\mathsf{C}, \mathsf{IC}\}$*, a* variable assignment as*, and sets of atomic formulae* ps_t *and* ps_f*.* □

The function $\mathcal{R}_A\llbracket\phi\rrbracket\eta$ generates the consistency report for a formula ϕ w.r.t. the temporal structure A and the variable assignment η – see Fig. 9. Fig. 17 (in App. A) shows auxiliary functions used by $\mathcal{R}$. Initially, $\mathcal{R}$ is applied to an empty variable assignment.

Depending on their truth value we push atomic formulae into ps_t and ps_f, respectively. We store the complete application to identify predicate symbols that occur more than once in a formula. For a conjunction $\phi \wedge \psi$ we retain the report of subformulae that are responsible for the final truth value – this shortens our reports. If ϕ and ψ have the same truth value we compute the cartesian product of their reports via the auxiliary function $\otimes$. Otherwise, we retain the report of the *false* subformula, because the false subformula is already sufficient for making the conjunction false. For disjunctions we use a similar approach but join reports via $\oplus$.

$$
\begin{array}{rcll}
\mathcal{R}_{(TL,I)}\llbracket p(e_0, e_1, \ldots, e_n)\rrbracket\eta & & (p \in \dot{\mathbf{P}}) & \\
 & = & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_0, e_1, \ldots, e_n)\}, \emptyset)\}) & \\
 & & \quad \text{if } (\mathcal{V}_A\llbracket e_1\rrbracket\eta, \ldots, \mathcal{V}_A\llbracket e_n\rrbracket\eta) \in p^{\dot{A}} & \\
 & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_0, e_1, \ldots, e_n)\})\}) & \\
 & & \quad \text{otherwise} & \\
\text{where } \dot{A} & = & I\,(\mathcal{V}_{TL}\llbracket e_0\rrbracket\eta) & \\[1ex]
\mathcal{R}_{(TL,I)}\llbracket p(e_1, \ldots, e_n)\rrbracket\eta & & (p \in \ddot{\mathbf{P}}) & \\
 & = & (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_1, \ldots, e_n)\}, \emptyset)\}) & \\
 & & \quad \text{if } (\mathcal{V}_{TL}\llbracket e_1\rrbracket\eta, \ldots, \mathcal{V}_{TL}\llbracket e_n\rrbracket\eta) \in p^{TL} & \\
 & & (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_1, \ldots, e_n)\})\}) & \\
 & & \quad \text{otherwise} & \\[1ex]
\mathcal{R}_A\llbracket \neg\phi\rrbracket\eta & = & \overline{\mathsf{flip}}(\mathcal{R}_A\llbracket\phi\rrbracket\eta) & \\[1ex]
\mathcal{R}_A\llbracket \phi \wedge \psi\rrbracket\eta & = & r_\phi \otimes r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) \\
 & & r_\phi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{False} \\
 & & r_\psi & \text{if } \mathsf{fst}(r_\psi) = \mathsf{False} \\
\text{where } r_\phi & = & \mathcal{R}_A\llbracket\phi\rrbracket\eta;\ r_\psi \ = \ \mathcal{R}_A\llbracket\psi\rrbracket\eta & \\[1ex]
\mathcal{R}_A\llbracket \phi \vee \psi\rrbracket\eta & = & r_\phi \oplus r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) \\
 & & r_\phi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{True} \\
 & & r_\psi & \text{if } \mathsf{fst}(r_\psi) = \mathsf{True} \\
\text{where } r_\phi & = & \mathcal{R}_A\llbracket\phi\rrbracket\eta;\ r_\psi \ = \ \mathcal{R}_A\llbracket\psi\rrbracket\eta & \\[1ex]
\mathcal{R}_A\llbracket \phi \Rightarrow \psi\rrbracket\eta & = & \mathcal{R}_A\llbracket\neg\phi \vee \psi\rrbracket\eta & \\[1ex]
\mathcal{R}_A\llbracket \forall\, x \in e \bullet \phi\rrbracket\eta & = & \overline{\oplus}(F) & \text{if } F \neq \emptyset \\
 & & \overline{\oplus}(T) & \text{if } T \neq \emptyset \\
 & & (\mathsf{True}, \emptyset) & \text{otherwise} \\
\text{where } F & = & \{\mathsf{push}(x \mapsto v, \mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) \mid & \\
 & & \quad v \in \mathcal{V}_A\llbracket e\rrbracket\eta \text{ and} & \\
 & & \quad \mathsf{fst}(\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) = \mathsf{False}\} & \\
T & = & \{\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v] \mid & \\
 & & \quad v \in \mathcal{V}_A\llbracket e\rrbracket\eta \text{ and} & \\
 & & \quad \mathsf{fst}(\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) = \mathsf{True}\} & \\[1ex]
\mathcal{R}_A\llbracket \exists\, x \in e \bullet \phi\rrbracket\eta & = & \overline{\oplus}(T) & \text{if } T \neq \emptyset \\
 & & \overline{\oplus}(F) & \text{if } F \neq \emptyset \\
 & & (\mathsf{False}, \emptyset) & \text{otherwise} \\
\text{where } T & = & \{\mathsf{push}(x \mapsto v, \mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) \mid & \\
 & & \quad v \in \mathcal{V}_A\llbracket e\rrbracket\eta \text{ and} & \\
 & & \quad \mathsf{fst}(\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) = \mathsf{True}\} & \\
F & = & \{\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v] \mid & \\
 & & \quad v \in \mathcal{V}_A\llbracket e\rrbracket\eta \text{ and} & \\
 & & \quad \mathsf{fst}(\mathcal{R}_A\llbracket\phi\rrbracket\eta[x \mapsto v]) = \mathsf{False}\} &
\end{array}
$$

Figure 9: Rule evaluation algorithm
(for auxiliary functions see Fig. 17 in App. A)

For a formula $\forall\, x \in e \bullet \phi$ we first evaluate the domain term e giving a list of values. For each value v in this list we compute ϕ's reports w.r.t. the assignment extension $\eta[x \mapsto v]$. If ϕ is violated for an assignment extension then the boolean value of the corresponding report is false. In this case we push the current variable assignment $x \mapsto v$ into the variable assignments of each diagnosis in ϕ's report. Finally, $\overline{\oplus}$ joins the resulting reports in F. If F is empty then ϕ is satisfied for any assignment extension. So we join the reports in T, containing the reports of satisfied ϕ *without* $x \mapsto v$. The new variable assignment is superfluous here because ϕ is true for *every* $\eta[x \mapsto v]$. In a consistency report, we understand omitted variable assignments as universally quantified. For existentially quantified formulae we use

state	repository changes	rules	IC	CPU time
1	`txt`: 1n, `key`: 1n, `man`: 9n	ϕ_1, ϕ_2	0	5.23 sec.
2	`txt`: 1c, 4n	ϕ_1, ϕ_2	4	13.48 sec.
3	`man`: 2c	ϕ_1, ϕ_2	14	18.95 sec.
4	`man`: 2c	ϕ_1, ϕ_2	19	26.38 sec.
5	`txt`: 1c	ϕ_1, ϕ_2	21	33.40 sec.
6	`txt`: 1c	ϕ_1, ϕ_2	26	41.15 sec.
7	`key`: 1n	ϕ_1, ϕ_2	31	47.11 sec.
8	`man`: 1n	ϕ_1, ϕ_2	35	57.34 sec.
9	`key`: 1c	ϕ_1, ϕ_2	36	67.14 sec.

Figure 10: Brute force checking performance

a similar approach. The rôles of T and F are, however, "reversed" because an existentially quantified formula is already satisfied if its child formula is true for *one* assignment extension.

Note that variable assignments inside diagnoses can only contain quantified variables. So we must be careful when replacing quantifiers by applications of `concatMap`.

The evaluation function $\mathcal{V}_A[\![e]\!]\eta$ returns the value of a term e w.r.t. a temporal structure A and a variable assignment η. We calculate values as usual in two-sorted temporal logics; for brevity we omit a formal definition. Applications of partially temporal function symbols $f(e_0, e_1, \ldots, e_n)$ first evaluate the temporal parameter e_0, select its associated non-temporal structure $\dot{A}$, and apply the interpretation of f in $\dot{A}$ to the evaluations of e_1 through e_n.

We demonstrate the algorithm above by showing part of the report generation for rule ϕ_1.

Example 6 In the rule ϕ_1 the topmost quantifier $\forall\, t \in \texttt{repStates} \bullet \phi$ first evaluates `repStates` to $[1, 2]$ and calls $\mathcal{R}$ with two variable assignments, $\mathcal{R}_A[\![\phi]\!][t \mapsto 1]$ and $\mathcal{R}_A[\![\phi]\!][t \mapsto 2]$. The former results in a true report $(\mathsf{True}, \emptyset)$. The latter reports the following inconsistency:

$$\left(\mathsf{False}, \left\{\left(\begin{array}{l}\mathsf{IC},\\ \left\{\begin{array}{l} x \mapsto \{\texttt{id} = \mathsf{doc1.txt}, \texttt{time} = 1\}, \\ k \mapsto \mathsf{kaA1c3}\end{array}\right\},\\ \emptyset,\\ \{\texttt{kind}(m) = \texttt{kKind}(d)\}\end{array}\right)\right\}\right)$$

We push the assignment $t \mapsto 2$ into the above report via $\mathsf{push}(t \mapsto 2, \mathcal{R}_A[\![\phi]\!][t \mapsto 2])$. Since F contains exactly one report $\overline{\oplus}(F)$ results in the final report, shown in Fig. 8. □

We have implemented our consistency checker into the revision control system darcs [30]. Evaluating our example brute force is by far not satisfactory. Fig. 10 sketches the development of a "toy" repository, which we shall use throughout.[11] The second column shows how many documents were added (n) or changed (c) during a state transition. We use `man` for manuals, `key` for key resolvers, and `txt` for other text documents. The third column contains the rules evaluated. Inconsistencies found are summarized in the fourth column. The final column shows the CPU time needed for the consistency check. To great extent CPU time depends on the repository state because brute force checking evaluates every repository state.

[11]Tests are performed on a Dell X200 laptop; 800 MHz PIII CPU.

In order to make our approach viable it is absolutely necessary to develop methods that decrease evaluation time.

6 Speeding up Consistency Checking

We have seen that it is infeasible to check consistency rules brute force. In this section we consider static and dynamic methods to decrease the computational complexity of evaluating first-order consistency rules. For simplicity we shall neglect the individual computational complexity of functions and predicates. We designed our methods to make as few as possible assumptions to the underlying DMS. We only require (1) a repository to access past documents, (2) a locking mechanism that avoids updates during a consistency check, and (3) that the DMS signals the documents added or changed by an update. Since we do not require a specific document model our techniques also apply to revision control systems, such as CVS or darcs.

6.1 Static Analysis

Static analysis tries to localize and simplify consistency rules *before* they are evaluated against a repository. It is performed almost exclusively on a syntactical level. *Localizing* a rule means to associate it with the set of those documents that can possibly affect it – thus reducing the number of rules to be re-evaluated at a repository check-in. A rule is *simplified* by minimizing its quantifier nesting and thus reducing the static evaluation time complexity. We shall only sketch our methods here because they are straightforward adaptions to techniques known from databases [9, 14].

When the DMS signals that the repository has changed, we want to re-evaluate only those consistency rules that might be affected by these updates. With each rule we associate a set of affected documents because we also want to extend revision control systems, such as

CVS, that lack a formal document model.[12] Rule localization depends on appropriate metadata for functions and predicates. Static Haskell code analysis helps the language designer to add these metadata. Then, a straightforward algorithm associates a consistency rule with a document set. For example, it associates the rule ϕ_1 with all text documents and all XML documents (`*.txt|*.xml`), and the rule ϕ_2 with manuals only (`man*.xml`). That is because `repDs` accesses text documents and XML documents (`*.txt|*.xml`), `repManDs` accesses manuals only (`man*.xml`), `repResDs` accesses key resolvers only (`key*.xml`), and `refs` accesses the documents of its second argument only.

The computational cost to evaluate a consistency rule depends on the deepest quantifier nesting, which is minimal if the scope of each quantifier is minimal. Such formulae are called *miniscope* [43]. Consider the following rule ϕ_1':

$$\begin{aligned}\phi_1' = \;& \forall\, t \in \texttt{repStates} \bullet \forall\, x \in \texttt{repDs}(t) \bullet \forall\, k \in \texttt{refs}(x) \bullet \\ & \exists\, d \in \texttt{concatMap}(t, \texttt{kDefs}, \texttt{repResDs}(t)) \bullet \\ & k = \texttt{key}(d) \wedge \\ & \left(\exists\, m \in \texttt{repManDs}(t) \bullet \begin{array}{l} \texttt{id}(m) = \texttt{kId}(d) \wedge \\ \texttt{kind}(m) = \texttt{kKind}(d) \end{array} \right)\end{aligned}$$

Here, the existential quantifier over m was moved into the conjunction, which is sound because $k = \texttt{key}(d)$ does not contain m. Since the existential quantifier has a smaller scope now, ϕ_1' can be evaluated faster than ϕ_1. Let e.g., each domain have a cardinality of n and each atomic formula evaluate in constant time t, then evaluating ϕ_1 costs $n^5 \cdot 3t$ while evaluating ϕ_1' costs only $n^4 \cdot t + n^5 \cdot 2t$. Due to our simple example the speedup is not very impressive. For more complex rules we experienced greater speedups.

We adapt the techniques in [8] to convert our rules to miniscope. Miniscoping also removes implications, pushes negations into formulae, and "flattens" nested conjunctions and disjunctions. Our incremental evaluation algorithm benefits from these simplifications.

Miniscoping the rule ϕ_2 removes the implication by a disjunction and pushes the universal quantifier over m_1 into the right hand side of this disjunction:

$$\begin{aligned}\phi_2' = \;& \forall\, t_1 \in \texttt{repStates}^* \bullet \forall\, t_2 \in \texttt{repStates}^* \bullet \\ & \neg(t_1 \leq t_2) \vee \\ & \left(\begin{array}{l} \forall\, m_1 \in \texttt{repManDs}(t_1) \bullet \exists\, m_2 \in \texttt{repManDs}(t_2) \bullet \\ \texttt{id}(m_1) = \texttt{id}(m_2) \;\wedge \texttt{kind}(m_1) = \texttt{kind}(m_2) \end{array} \right)\end{aligned}$$

Let us return to our toy repository: How does static analysis improve evaluation time? Fig. 11 shows some benefits, especially when the second rule is not re-evaluated. Improvements achieved by miniscoping can

[12]In more sophisticated DMS we could also define document classes (so-called stereotypes) and associate these classes with consistency rules.

state	repository changes	rules	IC	CPU time
1	`txt`: 1`n`, `key`: 1`n`, `man`: 9`n`	ϕ_1, ϕ_2	0	4.44 sec.
2	`txt`: 1`c`, 4`n`	ϕ_1	4	9.55 sec.
3	`man`: 2`c`	ϕ_1, ϕ_2	14	15.25 sec.
4	`man`: 2`c`	ϕ_1, ϕ_2	19	20.40 sec.
5	`txt`: 1`c`	ϕ_1	21	23.46 sec.
6	`txt`: 1`c`	ϕ_1	26	27.98 sec.
7	`key`: 1`n`	ϕ_1	31	32.50 sec.
8	`man`: 1`n`	ϕ_1, ϕ_2	35	44.66 sec.
9	`key`: 1`c`	ϕ_1	36	45.34 sec.

Figure 11: Improvements by static analysis

be seen in states where both rules are re-evaluated. The results are, however, unsatisfactory: Still rule evaluation time depends on the repository state. One of the major reasons is that we access documents at *previous* repository states. Usually, the DMS rebuilds these documents step by step using state transition descriptions, e.g., diffs or patches. Next, we discuss dynamic methods, which attempt to avoid accessing past document versions.

6.2 Incremental Evaluation

The major goals of incremental evaluation are: (1) access only current document versions and (2) as far as possible access changed or new documents only. A very simple strategy would be the following: Static rules quantify only once over all repository states and lack calculations of time, e.g., ϕ_1. Thus we can copy the report from the previous evaluation and evaluate the rule w.r.t. the new repository state only. This, of course, breaks down for temporal rules (e.g., ϕ_2), which relate different repository states.

We, therefore, developed a general technique that also applies to temporal rules. As usual, it is most challenging to balance the speedup against the space needed for storing auxiliary information. Our approach needs only few additional information (see Sect. 6.4). Instead it exploits previous reports to re-evaluate rules only w.r.t. a part of the documents they affect. Our strategy is as follows:

1. Keep the old report from the last evaluation.
2. If possible re-evaluate a rule only on new or changed domain values. For old domain values copy the relevant part from the old report.

A fundamental prerequisite to incremental evaluation is that the consistency report of a rule remains constant if its quantifier domains remain constant. This requires that the result of a function or predicate depends on its parameters only – a feature called *referential transparency* in functional programming. In general, Haskell guarantees referential transparency. Since in our approach language designers use Haskell to define symbol semantics, the strategy above is sound.

A notable exception is the "unsafe" function `repStates`, which is not referentially transparent because it reads the number of already performed check-ins directly from the repository (see Sect. 6.4).

In order to support incremental evaluation, the language designer associates with each function symbol and each predicate symbol those parameters it actually depends on. For example, the partially temporal function symbol `refs` does not depend on its temporal parameter. Within rules a straightforward algorithm marks subformulae and quantifier bounding terms with the variables they actually depend on. Below, we have marked our example rules ϕ_1' and ϕ_2' where variables appear as superscripts. ($*$ means that a subformula or quantifier bounding term contains `repStates`, i.e., it is unsafe.)

$$\begin{aligned}\phi_1' = {} & \forall^{*}\ t \in \texttt{repStates}^{*} \bullet \forall^{t}\ x \in \texttt{repDs}(t)^{t} \bullet \\ & \forall^{t,x}\ k \in \texttt{refs}(x)^{x} \bullet \\ & \exists^{t,k}\ d \in \texttt{concatMap}(t, \texttt{kDefs}, \texttt{repResDs}(t))^{t} \bullet \\ & k =^{k,d} \texttt{key}(d) \wedge^{t,k,d} \\ & \left(\begin{array}{l} \exists^{t,d}\ m \in \texttt{repManDs}(t)^{t} \bullet \texttt{id}(m) =^{d,m} \texttt{kId}(d) \wedge^{d,m} \\ \qquad\qquad\qquad\qquad\qquad \texttt{kind}(m) =^{d,m} \texttt{kKind}(d) \end{array} \right)\end{aligned}$$

$$\begin{aligned}\phi_2' = {} & \forall^{*}\ t_1 \in \texttt{repStates}^{*} \bullet \forall^{*}\ t_2 \in \texttt{repStates}^{*} \bullet \\ & \neg(t_1 \leq^{t_1,t_2} t_2)^{t_1,t_2} \vee^{t_1,t_2} \\ & \left(\begin{array}{l} \forall^{t_1,t_2}\ m_1 \in \texttt{repManDs}(t_1)^{t_1} \bullet \\ \exists^{t_2,m_1}\ m_2 \in \texttt{repManDs}(t_2)^{t_2} \bullet \\ \texttt{id}(m_1)\ {}^{m_1 \underline{\rightharpoonup} m_2}\ \texttt{id}(m_2) \wedge^{m_1,m_2} \\ \texttt{kind}(m_1)\ {}^{m_1 \underline{\rightharpoonup} m_2}\ \texttt{kind}(m_2) \end{array} \right)\end{aligned}$$

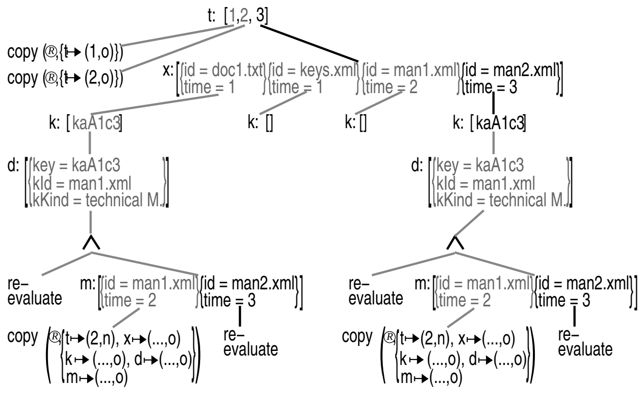
In order to narrow quantifier domains we partition them into four sets: *new* contains new values, *chg* contains changed values, *del* contains values that have been deleted, and *old* contains values that remained constant. Finally, we extend variable assignments, which map variables to values, to also mark variables as new and old, respectively.

The central idea behind our incremental evaluation algorithm is to *re-evaluate a subformula only if it depends on a variable marked as new* in the current assignment or if it contains an unsafe function or predicate. In case a subformula depends only on old variables and contains only referentially transparent functions and predicates, we copy part of the old report and abort re-evaluation of this subformula.

Before detailing the formalism we illustrate our incremental strategy with an example:

Example 7 Consider a check-in of a new manual man2.xml at state 3 also referencing the key kaA1c3:

```
<man kind="field M.">
  ... <see>kaA1c3</see> ... </man>
```



t: [1,2, 3]
copy (®,{t↦(1,o)})
copy (®,{t↦(2,o)})
x: [{id = doc1.txt, time = 1} {id = keys.xml, time = 1} {id = man1.xml, time = 2} {id = man2.xml, time = 3}]
k: [kaA1c3]
k: []
k: []
k: [kaA1c3]
d: [{key = kaA1c3, kId = man1.xml, kKind = technical M.}]
d: [{key = kaA1c3, kId = man1.xml, kKind = technical M.}]
∧
∧
re–evaluate
m: [{id = man1.xml, time = 2} {id = man2.xml, time = 3}]
copy (®,{t↦(2,n), x↦(...,o), k↦(...,o), d↦(...,o), m↦(...,o)})
re–evaluate
re–evaluate
m: [{id = man1.xml, time = 2} {id = man2.xml, time = 3}]
copy (®,{t↦(2,n), x↦(...,o), k↦(...,o), d↦(...,o), m↦(...,o)})
re–evaluate

Figure 12: Incremental evaluation of rule ϕ_1'

We have to re-evaluate both rules. Fig. 12 shows how our incremental algorithm evaluates rule ϕ_1'. In the tree vertices represent conjunctions, disjunctions, or quantifier domains where old values are grey and new values are black. A path from the root to a leaf represents an assignment.

During evaluation of rule ϕ_1' we first calculate t's domain, which results in the sets $\mathit{new} = \{3\}$, $\mathit{chg} = \emptyset$, $\mathit{old} = \{1, 2\}$, and $\mathit{del} = \emptyset$. Since t's subformula depends only on t and is referentially transparent we can abort re-evaluation for values in *old*. Instead, we copy relevant states from the old report ®, which results in a true report $(\mathsf{True}, \emptyset)$ for $t \mapsto 1$ and the following report for $t \mapsto 2$ (assignments in reports lack variable markers):

$$\left(\begin{array}{l} \mathsf{False}, \\ \left\{ \left(\begin{array}{l} \mathsf{IC}, \{x \mapsto \{\texttt{id} = \mathsf{doc1.txt}, \texttt{time} = 1\}, k \mapsto \mathsf{kaA1c3}\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\} \end{array} \right) \right\} \end{array} \right)$$

The report above lacks the binding $t \mapsto 2$, which is removed while copying. When we finally evaluate t's quantifier the binding for t is "pushed" in again. Note that the domain of k contains only old values when it is evaluated for $x \mapsto \{\texttt{id} = \mathsf{doc.txt}, \texttt{time} = 1\}$ because the bounding term $\texttt{refs}(x)^{x}$ depends on the old variable x only (see Sect. 6.4). Re-evaluating ϕ_1 for $t \mapsto 3$ results in:

$$\left(\begin{array}{l} \mathsf{False}, \\ \left\{ \begin{array}{l} \left(\begin{array}{l} \mathsf{IC}, \{x \mapsto \{\texttt{id} = \mathsf{doc1.txt}, \texttt{time} = 1\}, k \mapsto \mathsf{kaA1c3}\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\} \end{array} \right), \\ \left(\begin{array}{l} \mathsf{IC}, \{x \mapsto \{\texttt{id} = \mathsf{man2.xml}, \texttt{time} = 3\}, k \mapsto \mathsf{kaA1c3}\}, \\ \emptyset, \{\texttt{kind}(m) = \texttt{kKind}(d)\} \end{array} \right) \end{array} \right\} \end{array} \right)$$

Pushing the bindings $t \mapsto 2$ and $t \mapsto 3$ into the reports above results in the final report:

$$\left(\begin{array}{l}\mathsf{False},\\ \left\{\begin{array}{l}\left(\begin{array}{l}\mathsf{IC},\left\{\begin{array}{l}t\mapsto 2, x\mapsto\{\mathtt{id}=\mathsf{doc1.txt},\mathtt{time}=1\},\\ k\mapsto \mathsf{kaA1c3}\end{array}\right\},\\ \emptyset,\{\mathtt{kind}(m)=\mathtt{kKind}(d)\}\end{array}\right),\\ \left(\begin{array}{l}\mathsf{IC},\left\{\begin{array}{l}t\mapsto 3, x\mapsto\{\mathtt{id}=\mathsf{doc1.txt},\mathtt{time}=1\},\\ k\mapsto \mathsf{kaA1c3}\end{array}\right\},\\ \emptyset,\{\mathtt{kind}(m)=\mathtt{kKind}(d)\}\end{array}\right),\\ \left(\begin{array}{l}\mathsf{IC},\left\{\begin{array}{l}t\mapsto 3, x\mapsto\{\mathtt{id}=\mathsf{man2.xml},\mathtt{time}=3\},\\ k\mapsto \mathsf{kaA1c3}\end{array}\right\},\\ \emptyset,\{\mathtt{kind}(m)=\mathtt{kKind}(d)\}\end{array}\right)\end{array}\right\}\end{array}\right)$$

Due to our strategy we copied some part of the old report during evaluation of the *new* repository state ($t \mapsto 3$). Notice, however, that bindings to the current state 3 are lifted to the previous state 2 because the old report cannot contain a binding $t \mapsto 3$.

Evaluation of rule ϕ_2' proceeds similar to the evaluation of rule ϕ_1'. Fig. 13 shows the effectiveness of our incremental evaluation strategy for *temporal* consistency rules. The final report for rule ϕ_2' at state 3 shows that state 1 is inconsistent with the states 2 and 3:

$$\left(\begin{array}{l}\mathsf{False},\\ \left\{\begin{array}{l}\left(\begin{array}{l}\mathsf{IC},\left\{\begin{array}{l}t_1\mapsto 1, t_2\mapsto 2,\\ m_1\mapsto\left\{\begin{array}{l}\mathtt{id}=\mathsf{man1.xml},\mathtt{time}=2,\\ \mathtt{kind}=\mathsf{technical\ M.}\end{array}\right\}\end{array}\right\},\\ \{t_1\leq t_2\},\{\mathtt{kind}(m_1)=\mathtt{kind}(m_2)\}\end{array}\right),\\ \left(\begin{array}{l}\mathsf{IC},\left\{\begin{array}{l}t_1\mapsto 1, t_2\mapsto 3,\\ m_1\mapsto\left\{\begin{array}{l}\mathtt{id}=\mathsf{man1.xml},\mathtt{time}=2,\\ \mathtt{kind}=\mathsf{technical\ M.}\end{array}\right\}\end{array}\right\},\\ \{t_1\leq t_2\},\{\mathtt{kind}(m_1)=\mathtt{kind}(m_2)\}\end{array}\right)\end{array}\right\}\end{array}\right)$$

□

6.3 An Incremental Evaluation Algorithm

In this section we formalize our algorithm – an incremental variant of brute force evaluation shown in Sect. 5.2.

The function $\mathcal{DV}_A[\![e]\!]\eta$ calculates the domain e of a quantified formula w.r.t. a temporal structure A and an assignment η. It results in the four sets *new*, *chg*, *old*, and *del* (see Sect. 6.4). In contrast, $\mathcal{V}_A[\![e]\!]\eta$ performs classic value computation of terms outside domains.

Our incremental report generator $\mathcal{IR}_A^b[\![\phi]\!]\eta$ (see Fig. 14) is initially applied to an empty assignment. The boolean parameter b determines whether the evaluated subformula is immediately below a quantifier. Only then it is sound to copy part of the old report because conjunctions and disjunctions combine reports non-trivially. Since miniscoping flattens conjunctions and disjunctions this restriction has only low impact. Also due to miniscoping, we can neglect negations of non-atomic formulae. This simplifies our algorithm.

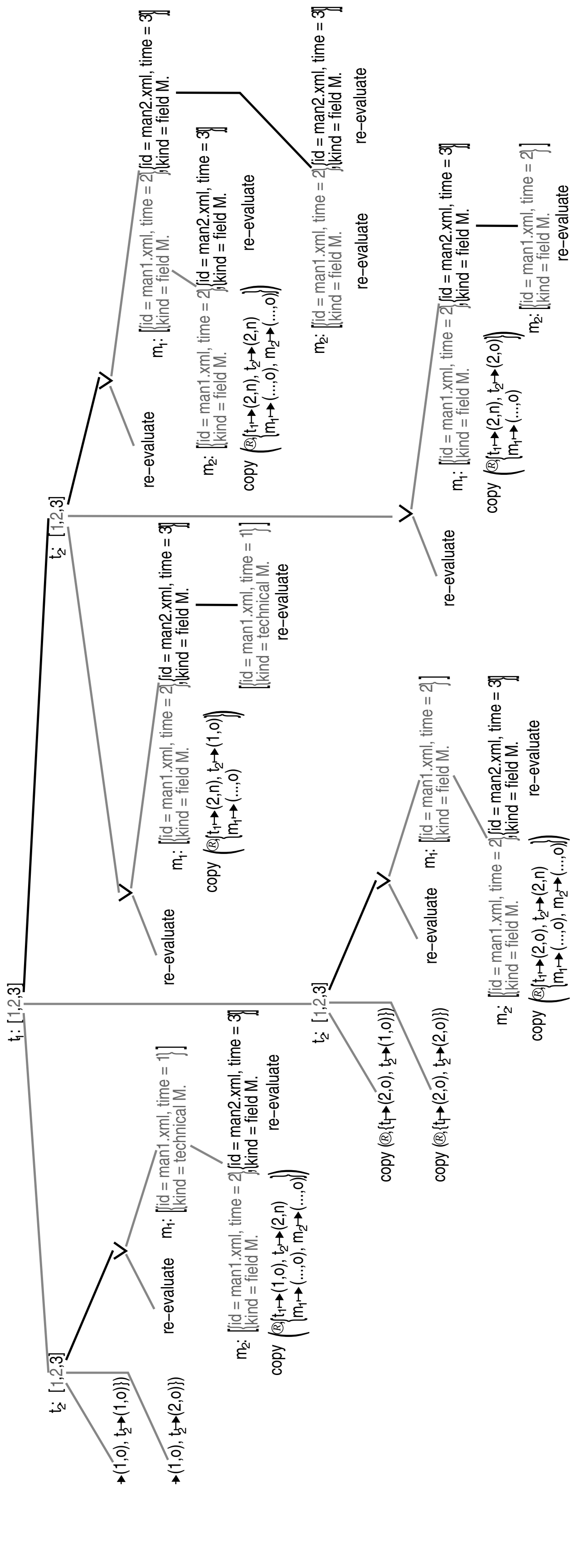


Figure 13: Incremental evaluation of rule ϕ_2'

$$
\begin{array}{l}
\mathcal{IR}^b_{(TL,I)}\llbracket p(e_0, e_1, \ldots, e_n)^{xs}\rrbracket\eta \qquad (p \text{ partially temporal}) \\
\qquad = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_0, e_1, \ldots, e_n)\}, \emptyset)\}) & \text{if } (\mathcal{V}_A\llbracket e_1\rrbracket\eta, \ldots, \mathcal{V}_A\llbracket e_n\rrbracket\eta) \in p^{\dot{A}} \\ (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_0, e_1, \ldots, e_n)\})\}) & \text{otherwise} \end{cases} \\
\quad \text{where } \dot{A} = I\,(\mathcal{V}_{TL}\llbracket e_0\rrbracket\eta)\,;
\end{array}
$$

$$
\begin{array}{l}
\mathcal{IR}^b_{(TL,I)}\llbracket p(e_1, \ldots, e_n)^{xs}\rrbracket\eta \qquad (p \text{ temporal}) \\
\qquad = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ (\mathsf{True}, \{(\mathsf{C}, \emptyset, \{p(e_1, \ldots, e_n)\}, \emptyset)\}) & \text{if } (\mathcal{V}_{TL}\llbracket e_1\rrbracket\eta, \ldots, \mathcal{V}_{TL}\llbracket e_n\rrbracket\eta) \in p^{TL} \\ (\mathsf{False}, \{(\mathsf{IC}, \emptyset, \emptyset, \{p(e_1, \ldots, e_n)\})\}) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{IR}^b_A\llbracket \phi \wedge^{xs} \psi\rrbracket\eta = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ r_\phi \otimes r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) = \mathsf{False} \\ r_\phi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{False} \\ r_\psi & \text{if } \mathsf{fst}(r_\psi) = \mathsf{False} \\ (\mathsf{True}, \emptyset) & \text{otherwise} \end{cases} \\
\quad \text{where } r_\phi = \mathcal{IR}^{\mathsf{False}}_A\llbracket\phi\rrbracket\eta;\ r_\psi = \mathcal{IR}^{\mathsf{False}}_A\llbracket\psi\rrbracket\eta
\end{array}
$$

$$
\begin{array}{l}
\mathcal{IR}^b_A\llbracket \phi \vee^{xs} \psi\rrbracket\eta = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ r_\phi \oplus r_\psi & \text{if } \mathsf{fst}(r_\phi) = \mathsf{fst}(r_\psi) = \mathsf{False} \\ (\mathsf{True}, \emptyset) & \text{otherwise} \end{cases} \\
\quad \text{where } r_\phi = \mathcal{IR}^{\mathsf{False}}_A\llbracket\phi\rrbracket\eta;\ r_\psi = \mathcal{IR}^{\mathsf{False}}_A\llbracket\psi\rrbracket\eta
\end{array}
$$

$$
\mathcal{IR}^b_A\llbracket (\neg\phi)^{xs}\rrbracket\eta = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ \overline{\mathsf{flip}}(\mathcal{IR}^b_A\llbracket\phi\rrbracket\eta) & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\mathcal{IR}^b_A\llbracket \forall^{xs}\ x \in e \bullet \phi\rrbracket\eta \\
\qquad = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ \overline{\oplus}(F) & \text{if } F \neq \emptyset \\ (\mathsf{True}, \emptyset) & \text{otherwise} \end{cases} \\
\quad \text{where } (\mathit{new}, \mathit{chg}, \mathit{old}, \mathit{del}) = \mathcal{IV}_A\llbracket e\rrbracket\eta \\
\qquad rs = \big\{\big(v, \mathcal{IR}^{\mathsf{True}}_A\llbracket\phi\rrbracket\,\eta[x \mapsto (v, \mathsf{o})]\big) \mid v \in \mathit{old}\big\} \cup \\
\qquad\qquad \big\{\big(v, \mathcal{IR}^{\mathsf{True}}_A\llbracket\phi\rrbracket\,\eta[x \mapsto (v, \mathsf{n})]\big) \mid v \in \mathit{new} \cup \mathit{chg}\big\} \\
\qquad F = \{\mathsf{push}(x \mapsto (v, \mathsf{o}), r) \\
\qquad\qquad \mid (v, r) \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{IR}^b_A\llbracket \exists^{xs}\ x \in e \bullet \phi\rrbracket\eta \\
\qquad = \begin{cases} \mathsf{copy}(\text{®}, \eta) & \text{if } \mathsf{notEval}(b, xs, \eta) \\ (\mathsf{False}, \emptyset) & \text{if } T = F = \emptyset \\ (\mathsf{True}, \emptyset) & \text{if } T \neq \emptyset \\ \overline{\oplus}(F) & \text{otherwise} \end{cases} \\
\quad \text{where } (\mathit{new}_0, \mathit{chg}, \mathit{old}_0, \mathit{del}) = \mathcal{IV}_A\llbracket e\rrbracket\eta \\
\qquad \mathit{new} = \begin{cases} \mathit{new}_0 \cup \mathit{old}_0 \cup \mathit{chg} & \text{if } \mathit{del} \neq \emptyset \text{ or } \mathit{chg} \neq \emptyset; \\ \mathit{new}_0 & \text{otherwise} \end{cases} \\
\qquad \mathit{old} = \begin{cases} \emptyset & \text{if } \mathit{del} \neq \emptyset \text{ or } \mathit{chg} \neq \emptyset; \\ \mathit{old}_0 & \text{otherwise} \end{cases} \\
\qquad rs = \{\mathcal{IR}^{\mathsf{True}}_A\llbracket\phi\rrbracket\,\eta[x \mapsto (v, \mathsf{o})] \mid v \in \mathit{old}\} \cup \\
\qquad\qquad \{\mathcal{IR}^{\mathsf{True}}_A\llbracket\phi\rrbracket\,\eta[x \mapsto (v, \mathsf{n})] \mid v \in \mathit{new}\} \\
\qquad T = \{r \mid r \in rs \text{ and } \mathsf{fst}(r) = \mathsf{True}\} \\
\qquad F = \{r \mid r \in rs \text{ and } \mathsf{fst}(r) = \mathsf{False}\}
\end{array}
$$

Figure 14: Incremental evaluation algorithm for annotated rules (for auxiliary functions see Fig. 17 in App. A)

For readability we introduce a global variable ®, which represents the old report.

For every formula $\mathsf{notEval}(b, xs, \eta)$ determines whether it appears immediately below a quantifier, it depends only on variables xs marked as old in the current assignment η, and contains referentially transparent functions and predicates only. In this case we copy the relevant states from the old report ®.[13] (copy replaces bindings to the current repository state with bindings to the previous repository state.)

The most important case is an existentially quantified formula. Due to miniscoping all quantifiers occur positively in a formula. So existential quantifiers cannot appear "disguised" as negated universal quantifiers. We first compute the domain, resulting in the four sets new_0, chg, old_0, and del. The general idea is to mark the values in new and chg as new (n) and the values in old as old (o). Then the subformula is evaluated for possible assignment extensions to these values ($\eta[x \mapsto (v, \mathsf{n})]$ and $\eta[x \mapsto (v, \mathsf{o})]$, respectively). Alas this is only sound if no values were changed or deleted in the domain, i.e., $\mathit{chg} = \mathit{del} = \emptyset$. Naturally, an existentially quantified formula can become false just from deleting values in its domain. Adding new values to the domain cannot falsify an existentially quantified formula. Consequently, if either chg or del contains values we must mark *every* domain value as new in the assignment extensions. So the worst cases for incremental evaluation are changed or deleted values in domains of existential quantifiers.

We could overcome the restrictions above by adding explicit information to the reports, which values in the domain of an existential quantifier made a rule consistent or inconsistent. We would really like to do so but we experienced that reports become extremely large when we stored these additional values. Consequently, our tradeoff is to avoid storing these additional information and accept that evaluation of existential quantifiers can be slower than evaluation of universal quantifiers. Universal quantifiers do not suffer from changed or deleted domain values because our reports already store the values that make a rule inconsistent – they pinpoint inconsistencies.

Let us return to our toy repository. What does incremental evaluation buy? Fig. 15 shows the performance of our consistency checker using both static analysis and incremental evaluation. Now, evaluation time depends rather on the changed content than on the repository state. Notice that except for states 3 and 4 every

[13]A state in the old report may contain an assignment more detailed or more general than η. Therefore, we copy all states containing assignments that subsume or are subsumed by η. An assignment as_1 subsumes an assignment as_2 if each variable binding in as_1 also occurs in as_2, where we neglect variable markers.

state	repository changes	rules	IC	CPU time
1	`txt`: 1n, `key`: 1n, `man`: 9n	ϕ_1, ϕ_2	0	4.47 sec.
2	`txt`: 1c, 4n	ϕ_1	4	2.33 sec.
3	`man`: 2c	ϕ_1, ϕ_2	14	6.45 sec.
4	`man`: 2c	ϕ_1, ϕ_2	19	6.58 sec.
5	`txt`: 1c	ϕ_1	21	2.68 sec.
6	`txt`: 1c	ϕ_1	26	2.53 sec.
7	`key`: 1n	ϕ_1	31	3.09 sec.
8	`man`: 1n	ϕ_1, ϕ_2	35	4.01 sec.
9	`key`: 1c	ϕ_1	36	3.61 sec.

Figure 15: Overall performance using static analysis and incremental evaluation

evaluation is faster than the initial evaluation although the repository grows. The simple initial strategy from Sect. 6.2 cannot achieve this.

6.4 Technical Notes

In this section, we discuss some of the rather technical details we came across while implementing our approach.

We interface the repository through some Haskell functions (see Sect. 7.1) to get the timestamps of already performed check-ins (`repStates`), get documents at a given check-in time, and parse documents via a Haskell function. The repository guarantees that, except for `repStates`, interface functions are referentially transparent, although they involve IO. The interface function `repStates` cannot be referentially transparent because it reads the timestamps of already performed check-ins directly from the repository, and thus its result will change between (but not during) consistency checks. Straightforward source code analysis determines whether a function or predicate defined by the language designer is referentially transparent, i.e., it does not call `repStates`.

A simple approach to incremental evaluation would require to store *every* quantifier domain from the previous evaluation. This is, however, not feasible because quantifier domains may become unexpectedly large as we do not control their vocabulary. For example, a quantifier might iterate over the complete document content.

Instead of storing domains, we *memoize* certain functions that occur in quantifier bounding terms. This reduces the space needed for storing intermediate data while maximizing the benefits for incremental evaluation. We distinguish functions w.r.t. their result type. Storing timestamps needs only few space. We also consider documents by exploiting a natural property of a DMS: A document can be identified by its name and check-in time only. Since we only need to know whether a document has been touched by a check-in we store the "Doc-part" of each function the result type of which is a subtype of [Doc]. Thus we neglect additional record labels, such as `kind` in ManD. In summary, we memoize each function that occurs in a quantifier bounding term and has either [Time] or a subtype of [Doc] as result type. In our example these are: `repStates`, `repDs`, `repResDs`, and `repManDs`. Since we memoize functions instead of storing domains the domain of d in ϕ_1 is also incrementally evaluated.

We calculate domains via $\mathcal{IV}$. If $\mathcal{IV}$ encounters an applied function symbol returning a result of either type [Time] or a subtype of [Doc] it calculates the four lists *new*, *chg*, *old*, and *del* from the current result and the stored result from the previous evaluation. These lists are propagated up the term structure. Therefore, some restrictions apply to quantifier bounding terms. We only permit functions that treat each list member separately, e.g., `concatMap`.

We apply $\mathcal{IV}$ also to bounding terms that lack memoized functions, e.g., `refs`(x) in ϕ_1. If such a term depends on variables marked as new the current assignment or it is unsafe the lists *old*, *chg*, and *del* are empty and *new* contains the complete domain. If, otherwise, such a term only depends on variables marked as old in the current assignment and it is referentially transparent $\mathcal{IV}$ returns all domain values in *old* while *new*, *chg*, and *del* are empty. This is sound because the domain cannot have changed due to referential transparency.

7 Implementing Consistency Rules

This section sketches our implementation and presents some lessons learned from our experiments.

7.1 Prototype Implementation

How do we perform a consistency check in practice? Our consistency checker is implemented in Haskell. Currently, we interface the revision control system darcs [30]. Assume an author checks in a document. Then, darcs first performs some tests to ensure that the changes submitted may be applied to the repository. If this test succeeds darcs calls our consistency checker via a system call. We use a system call because of its simplicity. During the run of our consistency checker the repository is locked, i.e., no check-ins can be performed. In case all strict rules are satisfied our consistency checker finishes normally and darcs acknowledges the check-in. Otherwise, the check-in is rejected.

Our consistency checker reads the rules to check from the project description supplied by the project manager. The next step is to type-check the rules against the functions and predicates they use. Usually, the language used is not changed between consistency checks. In this case we *dynamically* link auxiliary Haskell modules to the running consistency checker (the modules have been compiled during a previous consistency check). These Haskell modules contain implementations of functions, predicates, and types. Haskell lacks subtyping. We, therefore, resolve subtype relationships through (1) coercion functions, which coerce a subtype to its supertype, and (2) marshalling functions, which convert a Haskell value into a value as required by our consistency checker. Now, we can call the functions supplied by the language designer in order to check the repository for consistency. Consistency checking uses the algorithm from Sect. 5.2. If, however, the language used has changed since the last consistency check we generate the above Haskell modules and compile them by the Haskell compiler GHC. The GHC also type-checks functions and predicates implemented by the language designer. All this is performed in the background.

As already mentioned we make only few assumptions about the underlying DMS or revision control system. We have developed a simple repository interface, which must be instantiated for a specific repository type, e.g., darcs or CVS. The interface consists of five Haskell functions (`Rep` denotes a Haskell type representing the repository):[14]

- `repStates :: Rep -> [Time]` returns all states of a given repository.
- `repHeadState :: Rep -> Time` returns the repository head, i.e., the current state.
- `repDocs :: Rep -> Time -> String -> [Doc]` returns the documents in the repository that are current at a given time and match a regular expression. For example, `repDocs repo 2 "*.xml"` returns all XML documents current at state 2.
- `repFolders :: Rep -> Time -> String -> [Folder]` behaves similarly to `repDocs` but returns directories instead.
- `parseDoc :: Rep -> Doc -> (String -> a) -> Maybe a` accesses a given document in the repository. The third parameter is a function that parses the document content and converts it into an appropriate Haskell data structure. For XML documents these parser functions can be generated.
- `repChanges :: ClockTime -> Rep -> [String]` returns a list of document names and folder names that have been changed since the last evaluation, determined by the first parameter.

[14]Due to dynamic linking we cannot use a Haskell type class here. Instead we provide an abstract data type `Rep`.

Except for `repChanges` the interface functions above can be used by the language designer. Our tools generate the repository argument and provide it as a top-level Haskell function `repo`. The interface function `repChanges` is essential for filtering consistency rules.

7.2 Lessons Learned

In order to test the feasibility and the practical relevance of our approach, we currently run a project together with sd&m[15] – a well established software company in Munich, Germany. We formalize consistency rules for sd&m's analysis modules [38, 34] – a *document*-based approach to software specification. Applied to an example specification our consistency checker precisely identifies inconsistencies such that developers can concentrate on their real work: the specification should reflect the requirements of the system specified. Due to incremental consistency checking our prototype performs at reasonable speed.

Prior to formalization consistency rules were described in natural language only and were distributed over all analysis modules. Rules were imprecise, thus imposing misinterpretations and contradictions. Worse, software engineers had to spend a lot of time just to ensure formal consistency between the documents of a specification.

Formalizing the rules and collecting them into a rule system resulted in a much better and precise understanding of what consistency means. Furthermore, we learned to understand the consistency rules themselves better. Some rules were dropped because they failed to reflect what was actually required from a software engineering point of view. Subtypes played an important rôle: they significantly decreased the number of types and functions declared. This simplified formalization. We also learned to appreciate our type checker, which has pinpointed a lot of formalization errors especially when polymorphic higher-order functions were used.

There is also some bad news. We had to spend much time in formalizing the rules and in defining appropriate types, functions, and predicates. Still, formalization is

[15]see www.sdm.de

a complex task. We are, however, convinced that the benefits of our approach outweigh the costs.

We have as yet not tested our consistency checker "in the wild." This will be certainly one of the next steps.

8 Related Work

Because managing consistency is a fundamental problem a huge body of related work exists; here we identify research areas that are closely related to our approach. In contrast to our approach, most of the other approaches *enforce* consistency.

Programming language environments [29] evaluate semantic rules of the underlying programming language on abstract syntax trees of source code documents. Later work on software engineering environments [39] provides consistency checks across different documents. Rules are, however, limited to a subset of a (non-temporal) first-order logic.

Various database systems [9] employ integrity constraints. Our approach shares some ideas with Thémis [4]: Higher-order complexity is encapsulated in functions and thus hidden from first-order rules. Recent works on semistructured databases [5] and integrity constraints for web sites [11] use *decidable* subsets of first-order logic. We need more expressive power as our examples have shown.

At first sight, we might use the DMS metadata database and allow violations of database integrity constraints. The database "corset" is, however, too strict and limited to document metadata only. Complex rules, e.g., "referenced sections should keep similar over time," require to inspect document *content* via information retrieval techniques [44]. Metadata impinging consistency may change, which would require to adapt the database schema. For smaller projects the database approach appears too heavy weight. Of course, our database-independent approach can still use databases for fast metadata access.

In software engineering [12, 24] tolerating inconsistency is considered more profitable than enforcing consistency. Many-valued logics help to model inconsistencies and their consequences [10]. We argue, however, that classic logic is easier to understand for users and that our approach is powerful enough to manage consistency. Our priority levels for rules can be considered as a coarse grained approach to many-valued logic.

The idea to compute inconsistency diagnoses rather than just detecting that an inconsistency has occurred is not new. It has been explored in the context of knowledge bases [41] and for analyzing consistency between documents [21]. While for knowledge bases decidability of the implication problem is crucial, we find closer relationships in the context of *distributed* documents. xlinkit [21] statically checks distributed documents against user-defined consistency rules and implements tolerant consistency. Rules are formalized in an untyped non-temporal first-order logic employing user-defined predicates the semantics of which is implemented in a Javascript "blackbox." This hinders the reuse of already available algorithms. We share many ideas with xlinkit but use a *repository*, which has a lot of advantages: (1) DMS and repositories are widely used and provide useful management mechanisms, e.g., document locking; (2) Internet access to a repository already supports distribution; and (3) history information (already stored by DMS) is a precondition for temporal consistency rules and incremental evaluation. Furthermore, we employ a sophisticated type system that helps to define meaningful consistency rules. We argue that DMS-managed repositories support collaborative distributed work and should be extended to provide consistency management.

As far as document types are concerned, we might also have used finite tree automata [40]. We decided not to do so because our type system is closer to the type system of Haskell – the programming language used by language designers.

Incremental programming languages [18, 46] provide a uniform approach to incremental computation. They attempt to minimize redundant computation provided that a program runs repeatedly on slightly different inputs. To produce the current results of a program run, incremental computation uses previous results, differences between previous inputs and current inputs, and auxiliary information. We can regard quantified formulae as "functions" having the quantifier domain as input, which is reduced during incremental evaluation.

From its origin, the database community has been striving for efficient algorithms that check integrity constraints [27, 9, 35, 14], maintain views [13], and optimize queries [2]. Usually, these approaches employ the following techniques: *Compile-time* analysis simplifies integrity constraints and identifies update types that might violate constraints. *Run-time* approaches use previous results to avoid re-computation. In the context of first-order logic they reduce quantifier domains.

In general, we follow the database community approach but lack a formal database schema, formalized updates, and consistency prior to updates. This makes our approach to incremental evaluation more complicated. We cannot benefit from constraint subsumption because this is undecidable in our rule language; miniscoping performs only some very simple subsumption analysis. Localizing rules roughly corresponds to using update information in [14]. In order to gain the fine granularity needed for database approaches we parse documents. Consequently, our approach would lack efficiency if we implemented only strict rules. Tolerating inconsistencies, however, allows to defer evaluation of weak rules and to acknowledge or prohibit a check-in after the strict consistency rules have been evaluated. In (temporal) databases the problem of efficiently storing and managing historical database states arises. In our setting, historical repository states are already stored and managed by the DMS.

Recently, incremental evaluation techniques were used by the XML community to maintain consistency w.r.t. user defined rules [3, 16]. The consistency checking toolkit xlinkit [25] uses static analysis to filter consistency rules relevant to document changes and a treediff algorithm to determine document parts that have to be re-checked. Its tolerant view of consistency distinguishes xlinkit from other approaches and makes it closer to our approach. By allowing distribution and avoiding a history-aware repository, xlinkit cannot implement temporal consistency rules and lacks several incremental techniques we can benefit from.

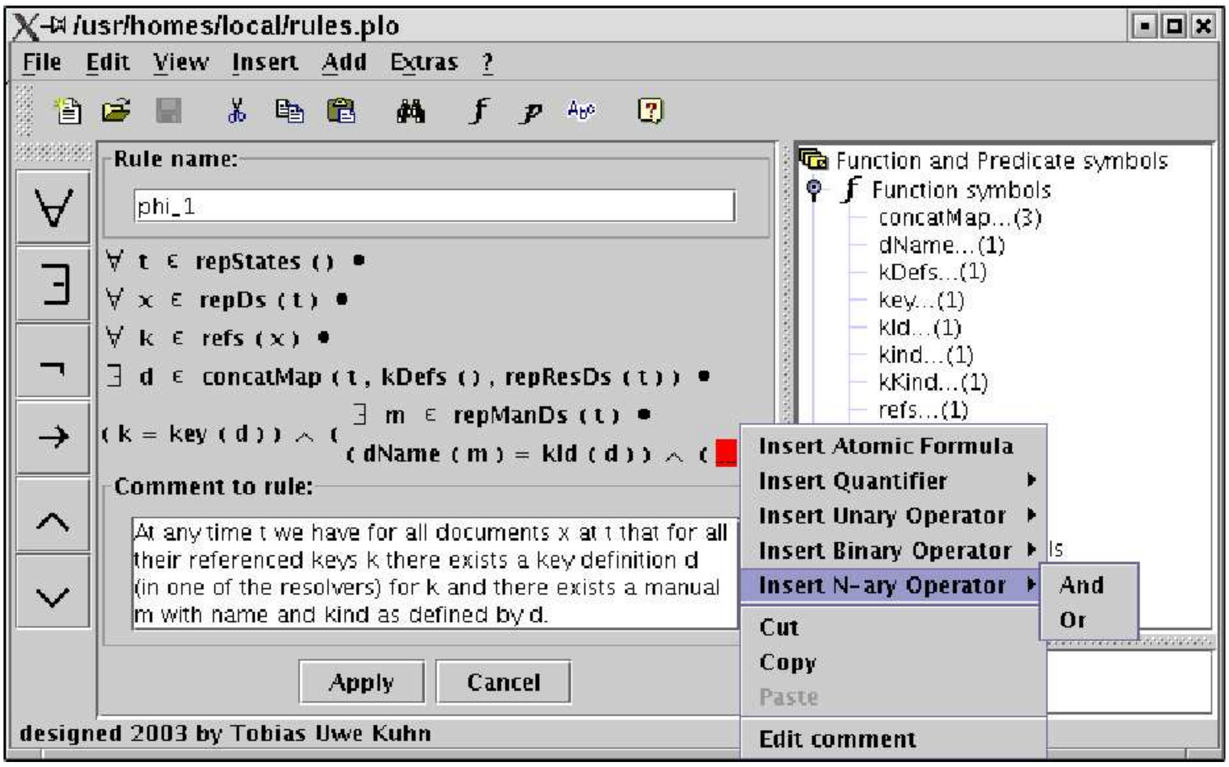
The incremental approaches above show that incremental computation comes with a cost, which is in general hard to analyze: storage for previous results and history information, lookup costs, computing differences of inputs. The notion that incremental evaluation is "often" cheaper than computation from scratch can, therefore, only be regarded as a heuristic. This leaves open the question when to use or avoid incremental computation, which highly depends on the actual application. In general, database performance analyses show that the smaller the changes through updates are and the less expressive the constraint language is, the better performs incremental computation. Our experiments have shown that we achieved a reasonable tradeoff between the costs for incremental evaluation and the speedup gained.

9 Conclusion and Outlook

We have presented an approach to flexible consistency management in heterogeneous repositories by explicit formal consistency rules. Formalization provides a common understanding of consistency, which is vital for any collaborative document engineering process. We have sketched how our approach can be integrated into existing DMS. A rule designer defines domain specific rules in a full first-order temporal logic with linear time. In contrast to decidable subsets, full temporal first-order logic provides an expressive rule designer toolset and supports formalizing practically relevant consistency rules. Higher-order complexity is hidden from the rule designer; it is encapsulated in functions and predicates defined by the language designer. Our type system significantly supports the formalization process. In the presence of subtyping records and variants provide expressiveness to describe document structures, comparable to the XML Schema standard. Automatically generated consistency reports precisely pinpoint inconsistencies within documents w.r.t. the rules defined. This allows for flexible inconsistency handling strategies. We have implemented our consistency checker interfacing the revision control system darcs [30].

We also have shown how first-order consistency rules can be evaluated efficiently by employing incremental techniques. Exploiting domain knowledge supplied by the language designer, we statically analyze consistency rules to (1) associate with each rule a document set the rule depends on, and (2) miniscope rules to reduce their static evaluation time complexity. At run-time we re-evaluate a rule only on new and changed documents if possible. It turned out that Haskell's referential transparency provides fundamental support for the soundness of our techniques. We provided concrete performance measurements proving that static analysis combined with incremental evaluation provides significant speedup compared to brute force evaluation. We conjecture that our incremental evaluation algorithm could be of value in other research areas as well.

We omit our concrete rule syntax, which is XML-based. Currently, a graphical user interface is developed (see Fig. 16) that greatly simplifies formalizing and maintaining consistency rules. We consider this an important ingredient for ensuring user acceptance. The correctness of rules covers another aspect, not yet dealt with in this paper: Do the rules formalized reflect the rule designer's intentions? For this purpose we plan to re-translate formal consistency rules into natural language. The translation can be compared to the initial intentions. Finally, just pinpointing inconsistencies is only a first step towards flexible consistency management. We are developing strategies to suggest compact and reasonable repair actions, in order to resolve inconsistencies. These repair actions will be generated from enriched consistency reports.



Rule name:

phi_1

```
∀ t ∈ repStates ( ) •
∀ x ∈ repDs ( t ) •
∀ k ∈ refs ( x ) •
∃ d ∈ concatMap ( t , kDefs ( ) , repResDs ( t ) ) •
                      ∃ m ∈ repManDs ( t ) •
( k = key ( d ) ) ∧ (
                      ( dName ( m ) = kId ( d ) ) ∧ (
```

Comment to rule:

At any time t we have for all documents x at t that for all their referenced keys k there exists a key definition d (in one of the resolvers) for k and there exists a manual m with name and kind as defined by d.

Figure 16: Graphical User Interface showing the formalization of rule ϕ_1 from Fig. 3

We are confident that using formal consistency rules in conjunction with a semantics that pinpoints inconsistencies can significantly improve consistency management in document engineering processes. Early tests in the field of software engineering confirm this assumption.

Acknowledgements

We would like to thank Christiane Stutz from sd&m for many fruitful discussions and practical examples, Wolfram Kahl for valuable suggestions concerning the formal basics, and Tobias Uwe Kuhn for developing the GUI.

References

[1] S. Abiteboul, L. Herr, and J. van den Bussche. Temporal versus first-order logic to query temporal databases. In *ACM Symp. on Principles of Database Systems*, pages 49–57, Montreal, Canada, 1996. ACM.

[2] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Trans. on Database Systems (TODS)*, 20(2):111–148, 1995.

[3] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Proc. of PLAN-X: Programming Language Technologies for XML*, 2002.

[4] V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4(3):493–518, 1995.

[5] P. Buneman, W. Fan, and S. Weinstein. Path consistency rules in semistructured databases. *Journal of Computer and System Sciences*, 61(2):146–193, 2000.

[6] P. Cederqvist et al. Version management with CVS, 2002. see www.cvshome.org/docs/manual/.

[7] M. Chakravarty. The Haskell foreign function interface 1.0, addendum to the Haskell 98 report, 2003. see www.cse.unsw.edu.au/~chak/haskell/ffi/.

[8] D. de Champeaux. Subproblem finder and instance checker, two cooperating modules for theorem provers. *Journal of the ACM*, 33(4):633–657, 1986.

[9] M. e Silva. Dynamic integrity constraints definition and enforcement in databases: a classification

framework. In *Proc. of the IFIP TC-11 Working Group 11.5 1st Working Conf. on Integrity and Internal Control in Information Systems*, pages 65–87, 1997.

[10] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *23rd Int. Conf. on Software Engineering (ICSE-01)*, Toronto, Canada, 2001. IEEE.

[11] M. Fernández, D. Florescu, A. Levy, and D. Suciu. Verifying integrity constraints on web sites. In *IJCAI*, pages 614–619, 1999.

[12] A. Finkelstein. A foolish consistency: technical challenges in consistency management. In *Proc. of the 11th Int. Conf. on Database and Expert Systems Applications (DEXA)*, pages 1–5, London, UK, 2000. Springer.

[13] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of data*, pages 157–166. ACM Press, 1993.

[14] A. Gupta, Y. Sagiv, J. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 45–55, Minneapolis, MS, 1994. ACM.

[15] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 2000.

[16] B. Kane, H. Su, and E. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proc. of the 4th Int. Workshop on Web Information and Data Management*, pages 1–8. ACM Press, 2002.

[17] P. Lincoln and J. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. of the 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 293–304. ACM, 1992.

[18] Y. Liu. Efficient computation via incremental computation. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 194–203, 1999.

[19] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth & Brooks/Cole Advanced Books & Software, 3rd edition, 1987.

[20] Minister of Defence. *ZDv 90/1 – Dienstvorschriften der Bundeswehr.* Department of Defence, 1983. (English: Service Manual Rules for the Federal Armed Forces).

[21] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. In *ACM Trans. on Internet Technology*, 2001.

[22] J. Nordlander. *Reactive Objects and Functional Programming.* Chalmers Tekniska, Högskola, 1999.

[23] J. Nordlander. Polymorphic subtyping in O'Haskell. In *APPSEM Workshop on Subtyping and Dependent Types in Programming*, 2000.

[24] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.

[25] C. Perez-Arroyo, C. Nentwich, W. Emmerich, and A. Finkelstein. Scaling consistency checking. submitted, 2003. see www.cs.ucl.ac.uk/staff/nentwich/publications/scalingchecking.html.

[26] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge Univ. Press, 2003. see also haskell.org/onlinereport/.

[27] D. Plexousakis. *On the Efficient Maintenance of Temporal Integrity in Knowledge Bases.* PhD thesis, 1996.

[28] F. Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170(2):153–183, 2001.

[29] T. Reps. *Generating language-based Environments.* MIT, 1984.

[30] D. Roundy. Darcs: David's advanced revision control system, 2003. see www.abridgegame.org/darcs/.

[31] G. Salton. Automatic indexing and abstracting. pages 42–80, 1988.

[32] J. Scheffczyk, U. Borghoff, P. Rödig, and L. Schmitz. Consistent document engineering. In *Proc. of the 2003 ACM Symp. on Document Engineering*, Grenoble, France, 2003. to appear.

[33] J. Scheffczyk, U. Borghoff, P. Rödig, and L. Schmitz. Efficient (in-)consistency management for heterogeneous repositories. In *4th Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, Lübeck, Germany, 2003. to appear.

[34] J. Scheffczyk, C. Stutz, U. Borghoff, and J. Siedersleben. Konsistente Software-Spezifikationen. submitted, 2003. (English: Consistent software specifications).

[35] R. Seljée and H. de Swart. Three types of redundancy in integrity checking; an optimal solution. In *Proc. of Data and Knowledge Engineering*, pages 135–151, 1999.

[36] K. Shafer, S. Weibel, E. Jul, and J. Fausey. Introduction to persistent uniform resource locators, 1996. see purl.oclc.org/OCLC/PURL/INET96.

[37] G. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2-3):197–226, 1994.

[38] C. Stutz, J. Siedersleben, D. Kretschmer, and W. Krug. Analysis beyond UML. In *Proc. of RE'02: IEEE Joint Int. Requirements Engineering Conf. 2002*, pages 215–222, Essen, 2002. IEEE.

[39] The GOODSTEP Team. The GOODSTEP project: general object-oriented database for software engineering processes. In *Proc. of the 2nd Int. Workshop on Database and Software Engineering — 16th Int. Conf. on Software Engineering (ICSE'94)*, 1994.

[40] A. Tozawa. Towards static type checking for XSLT. In *Proc. of the 2001 ACM Symp. on Document engineering*, pages 18–27. ACM, 2001.

[41] A. Vermesan and F. Coenen, editors. *Validation and Verification of Knowledge Based Systems - Theory, Tools and Practice, Papers from EUROVAV '99, 5th European Symp. on Validation and Verification of Knowledge Based Systems, Oslo, Norway.* Kluwer, 1999.

[42] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the 4th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 1999. ACM.

[43] H. Wang. Toward mechanical mathematics. 4(1):2–22, 1960.

[44] R. Wilkinson and A. Smeaton. Automatic link generation. *ACM Computing Surveys*, 31(4es), 1999.

[45] R. Wong and N. Lam. Managing and querying multi-version XML data with update logging. In *Proc. of the 2002 ACM Symp. on Document engineering*, pages 74–81. ACM, 2002.

[46] D. Yellin and R. Strom. Inc: a language for incremental computations. *ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, 13(2):211–236, 1991.

A Auxiliary Functions for Rule Evaluation

$$\begin{aligned}
\wp(s) &= \text{power set of the set } s\\
\mathbb{V} &= \text{set of all variables}\\
\mathbb{D} &= \text{set of all diagnoses}\\
\mathbb{A} &= \text{set of all assignments}\\
\mathbb{B} &= \{\mathsf{True}, \mathsf{False}\}
\end{aligned}$$

decompose pairs

$$\begin{aligned}
\mathsf{fst}((x,y)) &= x\\
\mathsf{snd}((x,y)) &= y
\end{aligned}$$

flip a diagnosis

$$\begin{aligned}
\mathsf{flip} &: \mathbb{D} \to \mathbb{D}\\
\mathsf{flip}\left((\mathsf{C}, as, ps_t, ps_f)\right) &= (\mathsf{IC}, as, ps_t, ps_f)\\
\mathsf{flip}\left((\mathsf{IC}, as, ps_t, ps_f)\right) &= (\mathsf{C}, as, ps_t, ps_f)
\end{aligned}$$

$\overline{\text{flip}}$ a consistency report

$$\begin{aligned}
\overline{\mathsf{flip}} &: (\mathbb{B} \times \wp(\mathbb{D})) \to (\mathbb{B} \times \wp(\mathbb{D}))\\
\overline{\mathsf{flip}}\,((b, ds)) &= (\neg b, \{\mathsf{flip}(d) \mid d \in ds\})
\end{aligned}$$

join diagnoses

$$\begin{aligned}
\mathsf{join} &: \mathbb{D} \times \mathbb{D} \to \mathbb{D}\\
\mathsf{join}\Big((c, as, ps_t, ps_f), (c', as', ps'_t, ps'_f)\Big)&\\
&= (c, as \cup as', ps_t \cup ps'_t, ps_f \cup ps'_f)
\end{aligned}$$

push an assignment into a report

$$\begin{aligned}
\mathsf{push} &: \mathbb{A} \times (\mathbb{B} \times \wp(\mathbb{D})) \to \mathbb{B} \times \wp(\mathbb{D})\\
\mathsf{push}\,(x \mapsto v, (b, ds)) &= \left(b, \left\{ \begin{array}{l} (c, \{x \mapsto v\} \cup as, ps_t, ps_f) \\ \mid (c, as, ps_t, ps_f) \in ds \end{array} \right\}\right)
\end{aligned}$$

cartesian product of reports

$$\begin{aligned}
\otimes &: (\mathbb{B} \times \wp(\mathbb{D})) \times (\mathbb{B} \times \wp(\mathbb{D})) \to (\mathbb{B} \times \wp(\mathbb{D}))\\
(b, ds) \otimes (b', ds') &= (b \wedge b', \{\mathsf{join}(d, d') \mid d \in ds, d' \in ds'\})
\end{aligned}$$

join reports

$$\begin{aligned}
\oplus &: (\mathbb{B} \times \wp(\mathbb{D})) \times (\mathbb{B} \times \wp(\mathbb{D})) \to (\mathbb{B} \times \wp(\mathbb{D}))\\
(b, ds) \oplus (b', ds') &= (b \vee b', ds \cup ds')
\end{aligned}$$

$\overline{\text{join}}$ a report set via folding ($\mathbf{S} \neq \emptyset$)

$$\begin{aligned}
\overline{\oplus} &: \wp(\mathbb{B} \times \wp(\mathbb{D})) \to (\mathbb{B} \times \wp(\mathbb{D}))\\
\overline{\oplus}(\mathbf{S}) &= \mathsf{fold}(\oplus, s, \mathbf{S}'), \text{ where } \mathbf{S}' = \mathbf{S} \uplus \{s\}
\end{aligned}$$

fold a set (f symmetric and associative)

$$\begin{aligned}
\mathsf{fold} &: \forall\alpha.(\alpha \times \alpha \to \alpha) \times \alpha \times \wp(\alpha) \to \alpha\\
\mathsf{fold}(f, x, \emptyset) &= x\\
\mathsf{fold}(f, x, (\{x'\} \uplus xs)) &= \mathsf{fold}(f, f(x, x'), xs)
\end{aligned}$$

copy relevant part of a report

$$\begin{aligned}
\mathsf{copy} &: (\mathbb{B} \times \wp(\mathbb{D})) \times \mathbb{A} \to (\mathbb{B} \times \wp(\mathbb{D}))\\
\mathsf{copy}((b, ds), \eta) &= \begin{array}{ll} (\mathsf{True}, \emptyset) & \text{if } ss = \emptyset \\ (\mathsf{False}, ss) & \text{otherwise} \end{array}\\
\text{where } ss &= \{(c, as', ps_t, ps_f) \mid (c, as, ps_t, ps_f) \in ds \text{ and}\\
&\qquad (\eta \sqsubseteq as \vee as \sqsubseteq \eta)\}\\
as' &= \{x \mapsto (v, m) \mid x \mapsto (v, m) \in as \text{ and } x \mapsto \_ \notin \eta'\}\\
\eta' &= \{x \mapsto (v', m) \mid x \mapsto (v, m) \in \eta\}\\
v' &= \begin{array}{ll} v - 1 & \text{if } v = \mathsf{current\ State} \\ v & \text{otherwise} \end{array}
\end{aligned}$$

determine whether to re-evaluate a subformula

$$\begin{aligned}
\mathsf{notEval} &: (\mathbb{B} \times \wp(\mathbb{V}) \times \mathbb{A}) \to \mathbb{B}\\
\mathsf{notEval}(b, xs, \eta) &= \begin{array}{ll} \mathsf{True} & \text{if } b = \mathsf{True} \text{ and } news = \emptyset \\ & \quad\text{and } xs \neq \{*\} \\ \mathsf{False} & \text{otherwise} \end{array}\\
\text{where } news &= \{x \mid x \in xs \text{ and } x \mapsto (\_, \mathsf{n}) \in \eta\}
\end{aligned}$$

first assignment subsumes second assignment (preorder)

$$\begin{aligned}
\sqsubseteq &: (\mathbb{A} \times \mathbb{A}) \to \mathbb{B}\\
as \sqsubseteq as' &= x \mapsto (v, \_) \in as \Rightarrow x \mapsto (v, \_) \in as'
\end{aligned}$$

Figure 17: Auxiliary functions for rule evaluation

B Typing Consistency Rules

TypVar

$$\frac{}{C,\ \Gamma \cup \{x : \sigma_g\},\ \Delta \vdash_{\mathsf{S}}\ x : \sigma_g}$$

TypSym

$$\frac{}{C,\ \Gamma,\ \Delta \cup \{s : \sigma\} \vdash_{\mathsf{S}}\ s : \sigma}$$

TypSymApp

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ s : \tau_1 \times \ldots \times \tau_n \rightarrow \tau_g \quad C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e_i : \tau_i}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ s(e_1, \ldots, e_n) : \tau_g}$$

TypStruct

$$\frac{\begin{array}{c}\prod_{\widehat{R}} = \{l_i : \forall\bar{\alpha}.\mathsf{Time} \times R\ \bar{\alpha} \rightarrow \tau_{g_i}\} \\ C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e_i : [\bar{\rho}/\bar{\alpha}]\tau_{g_i}\end{array}}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ K_R\{\overline{l_i = e_i}\} : [\bar{\rho}/\bar{\alpha}](R\ \bar{\alpha})}$$

TypCase

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e_0 : \mathsf{Time} \quad C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : [\bar{\rho}/\bar{\alpha}](V\ \bar{\alpha}) \\ \sum_{\widehat{V}} = \{k_i : \forall\bar{\alpha}.\tau_{g_1} \times \ldots \times \tau_{g_{n_i}} \rightarrow V\ \bar{\alpha}\} \\ C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ s_i : [\bar{\rho}/\bar{\alpha}](\tau_{g_1} \times \ldots \times \tau_{g_{n_i}}) \rightarrow \tau_g\end{array}}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \mathsf{case}(e_0, e, \{\overline{k_i \rightarrow s_i}\}) : \tau_g}$$

TypCase$V$

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e_0 : \mathsf{Time} \quad C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : [\bar{\rho}/\bar{\alpha}](V\ \bar{\alpha}) \\ \sum_{\widehat{V}} = \{k_i : \forall\bar{\alpha}.\tau_{g_1} \times \ldots \times \tau_{g_{n_i}} \rightarrow V\ \bar{\alpha}\} \\ C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ s_i : [\bar{\rho}/\bar{\alpha}](\tau_{g_1} \times \ldots \times \tau_{g_{n_i}}) \rightarrow \tau_g\end{array}}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \mathsf{case}(e_0, e, V, \{\overline{k_i \rightarrow s_i}\}) : \tau_g}$$

TypAnno

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \tau \quad \mathsf{tv}(\tau) = \emptyset}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ (e :: \tau) : \tau}$$

TypGen

$$\frac{C \cup D,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \tau \quad \bar{\alpha} \notin \mathsf{tv}(C) \cup \mathsf{tv}(\Gamma)}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \forall\bar{\alpha}.\tau | D}$$

TypInst

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \forall\bar{\alpha}.\tau | D \quad C \vdash_{\mathsf{S}}\ [\bar{\tau}/\bar{\alpha}]D}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : [\bar{\tau}/\bar{\alpha}]\tau}$$

TypSub

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \tau \quad C \vdash_{\mathsf{S}}\ \tau \leq \tau'}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : \tau'}$$

TypPred

$$\frac{}{C,\ \Gamma,\ \Delta \cup \{p : \sigma_p\} \vdash_{\mathsf{S}}\ p : \sigma_p}$$

TypPredApp

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ p : \tau_1 \times \ldots \times \tau_n \quad C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e_i : \tau_i}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ p(e_1, \ldots, e_n)}$$

TypNot

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \phi}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \neg\phi}$$

TypBin

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \phi \quad C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \psi}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ \phi \cdot \psi}$$

TypQuant

$$\frac{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ e : [\tau] \quad C,\ \Gamma \cup \{x : \tau\},\ \Delta \vdash_{\mathsf{S}}\ \phi}{C,\ \Gamma,\ \Delta \vdash_{\mathsf{S}}\ Q\ x \in e \bullet \phi}$$

Figure 18: Well-typedness rules for terms and formulae

ChkVar

$$\frac{\begin{array}{c}\bar{\beta} = \mathsf{tv}(\tau) \quad C = \{\beta_i \leq \nu_i\}^{\beta_i \in (\mathsf{tv}(\tau) \setminus \bar{\alpha})} \\ \theta \Vdash_{\mathsf{S}}\ \{[\bar{\nu}/\bar{\beta}]\tau \leq \tau'\}\end{array}}{\theta C,\ \Gamma \cup \{x : \forall\bar{\alpha}.\tau\},\ \Delta \Vdash_{\mathsf{S}}\ x : \theta(\tau')}$$

ChkSym

$$\frac{\theta \Vdash_{\mathsf{S}}\ \{[\bar{\nu}/\bar{\alpha}]\tau \leq \tau'\}}{\emptyset,\ \Gamma,\ \Delta \cup \{s : \forall\bar{\alpha}.\tau\} \Vdash_{\mathsf{S}}\ s : \theta(\tau')}$$

ChkSymApp

$$\frac{\begin{array}{c}C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e_i : \theta_i(\nu_i) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ s : \theta(\theta_1\nu_1 \times \ldots \times \theta_n\nu_n \rightarrow \tau)\end{array}}{C \cup \bigcup \theta C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ s(e_1, \ldots, e_n) : \theta(\tau)}$$

ChkStruct

$$\frac{\begin{array}{c}\prod_{\widehat{R}} = \{l_i : \forall\bar{\alpha}.\mathsf{Time} \times R\ \bar{\alpha} \rightarrow \tau_{g_i}\} \\ \theta \Vdash_{\mathsf{S}}\ \{[\bar{\nu}/\bar{\alpha}](R\ \bar{\alpha}) \leq \rho_g\} \quad \rho_{g_i} = \theta[\bar{\nu}/\bar{\alpha}]\tau_{g_i} \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e_i : \theta_i(\rho_{g_i}) \quad \theta' \Vdash_{\mathsf{S}}\ \{\theta_i\rho_{g_i} \leq \rho_{g_i}\}\end{array}}{\bigcup \theta' C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ K_R\{\overline{l_i = e_i}\} : \theta'\theta(\rho_g)}$$

ChkCase

$$\frac{\begin{array}{c}\sum_{\widehat{V_j}} = \{k_{j,i} : \forall\bar{\alpha}.\tau_{g_{j,i,1}} \times \ldots \times \tau_{g_{j,i,n_{j,i}}} \rightarrow V_j\ \bar{\alpha}\} \\ C_0,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e_0 : \theta_0(\mathsf{Time}) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e : \theta(\nu) \\ \theta' \Vdash_{\mathsf{S}}\ \{\theta\nu \leq [\bar{\nu}/\bar{\alpha}](V_j\ \bar{\alpha})\} \text{ for exactly one } j \\ \theta_i \Vdash_{\mathsf{S}}\ \{\nu_i \leq [\bar{\nu}/\bar{\alpha}](\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}}) \rightarrow \rho_g\} \\ \rho_i = \theta_i\nu_i \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ s_i : \theta'_i(\rho_i) \quad \theta'' \Vdash_{\mathsf{S}}\ \{\theta'_i\rho_i \leq \rho_i\}\end{array}}{\begin{array}{l}\theta'\theta_0(C_0 \cup C \cup \bigcup C_i),\ \Gamma,\ \Delta \Vdash_{\mathsf{S}} \\ \qquad\qquad \mathsf{case}(e_0, e, \{\overline{k_i \rightarrow s_i}\}) : \theta''\theta'\theta_0(\rho_g)\end{array}}$$

ChkCase$V$

$$\frac{\begin{array}{c}\sum_{\widehat{V}} = \{k_i : \forall\bar{\alpha}.\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}} \rightarrow V\ \bar{\alpha}\} \\ C_0,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e_0 : \theta_0(\mathsf{Time}) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e : \theta(\nu) \quad \theta' \Vdash_{\mathsf{S}}\ \{\theta\nu \leq [\bar{\nu}/\bar{\alpha}](V\ \bar{\alpha})\} \\ \theta_i \Vdash_{\mathsf{S}}\ \{\nu_i \leq [\bar{\nu}/\bar{\alpha}](\tau_{g_{i,1}} \times \ldots \times \tau_{g_{i,n_i}}) \rightarrow \rho_g\} \\ \rho_i = \theta_i\nu_i \\ C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ s_i : \theta'_i(\rho_i) \quad \theta'' \Vdash_{\mathsf{S}}\ \{\theta'_i\rho_i \leq \rho_i\}\end{array}}{\begin{array}{l}\theta'\theta_0(C_0 \cup C \cup \bigcup C_i),\ \Gamma,\ \Delta \Vdash_{\mathsf{S}} \\ \qquad\qquad \mathsf{case}(e_0, e, V, \{\overline{k_i \rightarrow s_i}\}) : \theta''\theta'\theta_0(\rho_g)\end{array}}$$

ChkAnno

$$\frac{\mathsf{tv}(\tau) = \emptyset \quad C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e : \theta(\tau) \quad \theta' \Vdash_{\mathsf{S}}\ \tau \leq \tau'}{\theta C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ (e :: \tau) : \theta'(\tau')}$$

ChkPred

$$\frac{\theta \Vdash_{\mathsf{S}}\ \{[\bar{\nu}/\bar{\alpha}]\tau_p \leq \tau_p'\}}{\theta C,\ \Gamma,\ \Delta \cup \{p : \forall\bar{\alpha}.\tau_p\} \Vdash_{\mathsf{S}}\ p : \theta(\tau_p')}$$

ChkPredApp

$$\frac{\begin{array}{c}C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e_i : \theta_i(\nu_i) \\ C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ p : \theta(\theta_1\nu_1 \times \ldots \times \theta_n\nu_n)\end{array}}{C \cup \bigcup \theta C_i,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ p(e_1, \ldots, e_n),\ \theta}$$

ChkNot

$$\frac{C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ \phi,\ \theta}{C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ \neg\phi,\ \theta}$$

ChkBin

$$\frac{C_1,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ \phi,\ \theta_1 \quad C_2,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ \psi,\ \theta_2}{\theta_1\theta_2(C_1 \cup C_2),\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ \phi \cdot \psi,\ \theta_1 \circ \theta_2}$$

ChkQuant

$$\frac{\begin{array}{c}C,\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ e : \theta([\nu]) \quad C',\ \Gamma \cup \{x : \theta\nu\},\ \Delta \Vdash_{\mathsf{S}}\ \phi,\ \theta' \\ \theta'' \Vdash_{\mathsf{S}}\ C' \setminus C'_\Gamma\end{array}}{\theta''(C \cup C'_\Gamma),\ \Gamma,\ \Delta \Vdash_{\mathsf{S}}\ Q\ x \in e \bullet \phi,\ \theta'' \circ \theta'}$$

Figure 19: Combined type inference and type checking algorithm