universite
PARIS-SACLAY

An Ontology Framework for Formal Libraries

Conception et Implantation d'un Environnement d'Ontologie pour des Bibliothèques Formelles

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Graduate School : Informatique et sciences du numérique. Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire Méthodes Formelles** (Université Paris-Saclay, CNRS, ENS Paris-Saclay), sous la direction de **Burkhart WOLFF**, Professeur des universités et le co-encadrement d'**Idir AIT-SADOUNE**, Professeur des universités

Thèse soutenue à Paris-Saclay, le 12 juillet 2024, par

Nicolas Méric

Composition du jury

Membres du jury avec voix délibérative

Prénom NOM Titre, Affiliation	Président ou Présidente
Prénom NOM Titre, Affiliation	Rapporteur & Examinateur / trice
Prénom NOM Titre, Affiliation	Rapporteur & Examinateur / trice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice

THESE DE DOCTORAT

NNT : 2024UPASG038

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: titre (en français)..
Mots clés: 3 à 6 mots clefs (version en français)

Résumé: Les ontologies de documents, c'est-à-dire une forme traitable par la machine de la structure des documents et du discours des documents, jouent un rôle crucial dans la structuration du lien entre des notions sémantiques et des documents contenant du texte informel. Dans de nombreuses disciplines scientifiques comme la médecine ou la biologie, les ontologies permettent l'organisation des articles de recherche et leur accès automatisé. Les champs des mathématiques et de l'ingénierie représentent un défi particulier avec des documents qui contiennent à la fois des éléments textuels formels et informels avec une structure complexe de liens mutuels et de dépendances de divers types. Les textes peuvent contenir des formules (qui devraient au moins être correctement typées et si possible sémantiquement cohérentes avec les définitions formelles) et des définitions formelles fondées sur des conventions de nommage qui devraient refléter les explications informelles. Un accès approfondi aux parties formelles de ce type de documents implique un cadre capable de prendre en compte des langages logiques typés, ce qui nécessite d'aller beaucoup plus loin que les langages ontologiques existants comme, par exemple, OWL [1, 2].

Le point de départ de ce travail est le langage ontologique ODL implanté dans Isabelle/DOF [3, 4, 5]. Profondément intégré dans l'assistant de preuve Isabelle/HOL et son interface *PIDE*, il permet à la fois le développement d'ontologies typées et de documents contenant des définitions, de la documentation et des preuves formelles pour des textes mathématiques et d'ingénierie formelle. Les ontologies génèrent des théories de méta-informations et les utilisent pour la validation de contraintes à respecter lors de l'édition des documents. L'implantation originelle prenait en charge des liens vers les termes dans des définitions et des preuves, mais pas les liens entre les termes, ni l'ajout de méta-informations structurées *à l'intérieur et entre* des formules ou des preuves. Le mécanisme de validation des méta-informations était réduit à des solutions artisanales. En outre, des entités formelles comme les définitions ou les lemmes ne pouvaient pas être référencées en tant qu'entités ontologiques. Ces fonctionnalités sont néanmoins essentielles à un certain nombre d'applications pour l'échange d'informations (semi-)formelles entre des prouveurs interactifs et automatisés d'une part, et pour des techniques de recherches *sémantiques* avancées axées sur la connaissance dans ces documents d'autre part.

Cette thèse surmonte ces limitations : Isabelle/DOF est étendu par un mécanisme de description et d'évaluation des méta-informations à l'intérieur du niveau des termes et des objets de preuve. Il est donc conçu pour fournir une intégration « plus profonde » dans des textes de bibliothèques mathématiques formelles, pour lesquelles les entités formelles peuvent être abstraites et devenir des éléments de document. Les entités formelles représentant des termes, des définitions ou des théorèmes peuvent être référencées et utilisées comme des objets ontologiques de première classe. La nouvelle prise en charge du polymorphisme pamamétré par des classes de type permet de généraliser les concepts ontologiques. Ces nouvelles fonctionnalités d'Isabelle/DOF peuvent exprimer non seulement des liens entre du texte informel et des concepts ontologiques formels mais aussi entre des entités formelles ou informelles en tant qu'éléments de documents, permettant ainsi d'affiner le lien entre des éléments de documents et la connaissance dans des textes mathématiques et d'ingénierie. Un mécanisme de réification dans Isabelle/DOF permet d'attacher des méta-données ontologiques à des objets de preuve pour ajouter de la connaissance sur la structure de scripts de preuves et les tactiques de preuve associées pour prouver un théorème. Cela pourrait être approprié pour des techniques d'importation/exportation de preuves entre assistants de preuve [6, 7].

Title: titre (en anglais)...

Keywords: 3 à 6 mots clefs (version en anglais)

Abstract: Document ontologies, i. e., a machine-readable form of the structure of documents as well as the document discourse, play a key role in structuring the link between semantic notions and documents containing informal text. In many scientific disciplines such as medicine or biology, ontologies allows the organization of research papers and their automated access. There are particular challenges in the field mathematics and engineering where documents contain both formal and informal text-elements with a complex structure of mutual links and dependencies of various types. Texts may contain formulas (which should be at least type-correct and if possible semantically consistent with the formal definitions) and formal definitions based on naming conventions that should reflect informal explanations. A deeper access into the formal parts of this type of documents involves a framework that can cope with typed, logical languages, which requires going substantially further than existing ontological languages like, for example, OWL [1, 2].

The starting point of this work was the ontological language ODL implemented in Isabelle/DOF [3, 4, 5]. Deeply integrated into the interactive theorem proving system Isabelle/HOL and its front-end *PIDE*, it allows both the development of typed ontologies and of documents containing definitions, documentation, and formal proofs. The ontologies generate *theories* of meta-data and use them for the validation of constraints which were enforced during document editing. The original implementation supported links to terms in definitions and proofs, but not links between terms, nor the addition of structured meta-information *inside and between* formulas or proofs. The validation mechanisms for meta-information were limited to hand-crafted solutions. Furthermore, formal entities like definitions or lemmas could not be referenced as ontological entities. These features are nevertheless essential for a number of applications when it comes to the exchange of (semi-)formal information between interactive and automated provers on the one hand and to advanced "semantic", knowledge-oriented search techniques in these documents on the other hand.

This thesis overcomes these limitations: Isabelle/DOF is extended by a description and an evaluation mechanism of meta-information inside the level of terms and proof-objects. It is therefore designed to provide a "deeper" integration into formal mathematical library texts, where formal entities can be abstracted and become document elements. Formal entities representing terms, definitions or theorems can be referenced and used as first class ontological objects. The new support of type classes parameterized polymorphism allows to generalize ontological concepts. These new features in Isabelle/DOF can express not only links between informal text and formal ontological concepts but also between formal or informal entities as elements of a document, thus improving the linking between document elements and *knowledge* in mathematical and engineering texts. A reification mechanism in Isabelle/DOF allows to attach ontological meta-data to proof objects to add knowledge about the structure of the proof scripts and associated proof tactics used to prove a theorem. It should be relevant for import/export techniques of proofs between theorem provers [6, 7].

Contents

Chapter 1

Introduction

1.1 Context

Documents have greatly evolved in recent history with the digitization of modern society. Document authors wish to define specific format and generalize over similar text, restrict access control for specific document parts using security models, declare advanced linking mechanism between document elements, target particular subjects like certifications, and so on.

This lead to the development of document languages that combine literal text and program source code to gain control at every steps of the document generation, whether it is parsing, computation, or rendering and presentation. This approach is computation-driven, in the sense that the main purpose of the document language development is motivated by the decision to follow the standard document generation pipeline.

Another approach is to have semantic notions emerging from document elements and allow for logical reasoning about them. This approach is logic – resp. semantics driven and aims to extract knowledge from reasonably well-structured informal "raw" texts.

With both these approaches arise questions on a formal foundation and the interest of this foundation regarding its expressiveness and adequacy to the author wishes. Furthermore, among documents in general and libraries in particular, formal libraries distinguish themselves when considering semantic concepts. They come with *formal* and *informal* content that can commingle. The formal concepts and notions require specific handling to give them adequate semantics and structure; mathematical concepts need to be formalized to form a sound foundation to build upon, and their semantics should also have a formal representation to allow advanced handling by users. For example, the notion of definition should have a formal representation in semantics to offer sound foundations for other semantic concepts to build upon.

The document calculus [8], a formal model based on the system F typing system chooses the computational approach and addresses the formal foundation issue. Its authors also state that it offers foundations for future programming language research and provides examples addressing the expressiveness like the possibility to model higher-level features like references and reforestation. Using the document calculus as a foundation for formal libraries implies that each formal concept needs to be defined from scratch to formalize the higher-level concepts like theorems and definitions: a logic should be implemented, with axioms and inference rules as its basis, or specific measures might be required if system F is used directly and a decidable type inference is wished, design and implementation of conservative extension should be done to offer a sound system for higher-level concepts like theorems, and so on. Basically a lot of technology should be developed from scratch implying a large overhead to have a sound foundation for formal libraries encoded inside the document calculus. Using the document calculus also has repercussions on the expressiveness. For example the formalization of the linking between document elements like references is already possible, so it should be possible to define other links and give them formal semantics. But to express these formal semantics, we might need to define new formal concepts and extend the document calculus. The overhead issue arises once again, indirectly this time.

With the semantic and logical approach, the knowledge extraction is a crucial prerequisite for any form of document handling. The extraction could be represented abstractly as a linking between formal and informal content where literal text is the latter and formalized concepts giving semantics are the former. Numerous research efforts summarized under the labels “semantic web” or “data mining” developed technologies using this abstract representation for advanced search, classification, “semantic” validation and “semantic” merge. For these technologies, a key role in structuring this linking is played by ontologies (also called “vocabulary” in semantic web communities), i. e., a machine-readable form of the structure of documents as well as the document discourse. Such ontologies can be used for scientific discourse underlying scholarly articles, the conversion, and integration of semi-formal content, for advanced semantic search in mathematical libraries as well as documentation in various engineering domains. In other words, ontologies generate the meta-data necessary to annotate raw text allowing their “deeper analysis”, in particular inside mathematical formulas or equivalent formal content such as programs, UML-models, etc.

For the use case of formal libraries, ontologies play a dual role: we will distinguish *document ontologies* from *domain ontologies*. The former are oriented towards meta-information used as a document language, i. e., the representation aspects of a target type-setting technology (e.g., LaTeX, HTML). Document ontologies have the same purpose as the document calculus, but are more permissive: the raw text is considered as a document element and any advanced manipulation of it is done using attached meta-data. Features like references and reforestation can be implemented using document ontologies. When it comes to domain ontologies, they are oriented towards a specific knowledge domain, like mathematics or engineering, to represent formal entities, whether these entities are document elements or attached meta-data elements that give formal semantics. We can see here that the knowledge representation language is left open. Traditional semantic web ontologies will use languages like description logics to define the Web Ontology Language (OWL), or other languages presented in the related works.

This leads to another path than the development from scratch when using the document calculus. Indeed with ontologies, and contrary to the document calculus, it seems easier to reuse existing technology with its own knowledge representation language, that might even already have a formalization of advanced concepts that could be made available to domain ontologies for formal libraries. Interactive theorem provers are such technologies: they define and formalize concepts like definitions and theorems and can prove mathematical assertions using these concepts. Interactive theorem provers offer a full ecosystem from their design as logical frameworks to large formal libraries developed over the years. And to be sound these formal libraries are developed on top of primitives specified in the logical framework, adopting the conservative extension approach. As logical frameworks for proofs they are of great value for logical reasoning about document elements specifically for formal libraries. When using the document calculus as a foundation, it means that a technology with some of the features of a theorem prover need to be developed to have the same soundness but the proof aspect is out of scope, i. e. having an environment to prove assertions. Indeed writing theorem provers is generally considered a really difficult challenge, and most of them are the result of large amount of research and development efforts redoubled in the formalization of large mathematical and engineering libraries. Paulson, who designed and develops the Isabelle theorem prover, concluded his handbook article on theorem prover design [9] with the edifying sentence: “Don’t write a theorem prover. Try to use someone else’s”.

Using a theorem prover for the semantic and logical approach may take several flavors. An existing ontology language can be used, then a thin layer needs to be designed that will translate concepts defined in the theorem prover language from and to the ontology language: by developing this thin layer, existing technologies are reused and the task seems feasible. But it implies that the formalism of both the languages should have the same expressiveness. An ontology language like OWL that is based on description logics, less expressive than First-Order Logic (FOL), will not be able to express every concepts of Coq, an interactive theorem prover based on the calculus of inductive constructions in which higher-order concepts can be defined. Isabelle/DOF [4], a Document Ontology Framework, chose another original approach. The key idea was to develop the framework on top of a theorem prover, to take advantage of such a technology and the ecosystem it offers, and to implement ontologies represented inside its logical language. Isabelle/DOF specifies an Ontology Definition Language (ODL) using the logical language of the Isabelle theorem prover and as such is deeply integrated within the theorem prover. It also generates a theory of meta-data in the logical language of Isabelle, and attaches these meta-data defined in ODL to informal raw text written in the Isabelle document format, creating a link between informal and formal content. It has two main benefits. Firstly, ODL can be used as the ontology language underlying the semantic notions and the theorem prover logical language can be used for logical reasoning about literal text. We can reuse existing technologies and focus on formal library specificities. Secondly, as formal concepts like definitions and theorems already exist directly as formal objects in the logic or at least as abstract concepts in Isabelle, it may be possible to declare them as elements of an ontology so that they are used for semantic validation and advanced handling.

Isabelle/DOF is developed with document and domain ontologies in mind: A previous work [10] was particularly interested in domain ontologies concerning software developments targeting certifications (such as CENELEC 50128 [11] or Common Criteria [12]) and played a part in shaping the design of Isabelle/DOF. Certifications of safety or security-critical systems, albeit responding to the fundamental need of the modern society of trustworthy numerical infrastructures are particularly complex and expensive. A major reason for this is that distributed labor as occurring in practice requires that complex documents composed of artifacts from analysis, design, coding, and verification must preserve coherence under permanent changes. Moreover, certification processes impose a strong need of traceability within the global document structure. Last but not least, modifications and updates of a certified product usually result in a complete restart of the certification activity since the impact of local changes can usually not be mechanically checked and must be done by manual inspection. Isabelle/DOF turned out to be a good candidate to model certification documents. It offers an answer to the particular challenge to the syncing of informal and formal content for formal libraries: it supports links inside informal text to terms in definitions and proofs. But links between terms are not supported, nor the addition of structured meta-information *inside and between* formulas or proofs. The validation mechanisms for meta-information are limited to hand-crafted solutions. Furthermore, formal entities like definitions or lemmas could not be referenced as ontological entities. These features are nevertheless essential for a number of applications when it comes to the exchange of (semi-)formal content between interactive and automated provers on the one hand and to advanced “semantic”, knowledge-oriented search techniques in these documents on the other hand. The goal of this thesis is to extend

Isabelle/DOF to capture the particularities of formal libraries. To achieve that an important redesign of the implementation is necessary and novelties must be developed.

1.2 Contribution

The original version of Isabelle/DOF (called Isabelle/DOF 1.0 here) was designed with the linking between informal text and formal ontological entities in mind, and was influenced by certification documents it targeted. With attached ontological content, informal text becomes an identifiable element that can be referenced and a part of the document structure. Certification standards specify the process and technical requirements for a specific goal, the development of software for programmable electronic systems for use in railway control and protection applications in the case of CENELEC 50128. This process and requirements materialize through the production of specifications written in documents. So the structure of the certification documentation expresses the specifications that were added to the documentation. Isabelle/DOF 1.0 comes with the concept of monitor to check and enforce the structure of a document. A document containing the specifications expressed with ontologized informal text can be checked to assert that it is complete with respect to a certification using a monitor enforcing its structure.

But Isabelle/DOF 1.0 lacks expressiveness to talk about its own concepts and hence to enable technologies pervasive in ontologies such as advanced semantic search and validation. An evaluation environment compatible with Isabelle/DOF 1.0 concepts is also required to allow this execution. Isabelle/DOF 1.0 can express formal concepts like definitions and theorems ontologically, but the ontological concept can not be linked to the abstract notion specified in the Isabelle theorem prover. Isabelle/DOF 1.0 can use an Isabelle definition in the logical context but can not talked about Isabelle definitions as a specific group of formal entities: formal concepts are not identifiable elements in a document. It also lacks the possibility to attach the abstract concept of rule to entities and individuals represented in the ontology language to make this rule part of a validation process.

To overcome these limitations, The notion of term contexts is introduced. Then to capture new semantics for the linking between informal or formal entities, Isabelle/DOF 1.0 ODL is extended to support parametric polymorphism. Both these features allow a deeper analysis of Isabelle/HOL where ontological meta-data can be attached to proof objects. All of these new features lead to a new version of Isabelle/DOF rebranded as Isabelle/DOF 2.0.

Term Contexts The Isabelle theorem prover uses typed λ-terms as a syntactic presentation for expressions, formulas, definitions, and rules. Rather than using a classical programming language, the concept of deep ontologies led us to use HOL itself and generate the checking-code for references to λ-terms inside λ-terms via reflection and reification techniques. In particular, this paves the way for a new context type called *term contexts*. As a consequence, syntactic categories that represents higher level concepts like definitions and theorems can *contain* ontological references inside their formulas to entities. Furthermore ontological references can be used in ontological definitions, whether it is a reference to a formal or an informal abstract concept in the ontology. This allows to reconsider the linking in Isabelle/DOF 2.0: it does not just combine documents as informal text with formal ontological concepts, it reconsiders the document that now consists of document elements and can combine them to capture new semantics.

Polymorphism for Parametric Ontological Classes Isabelle possesses a type-class polymorphic type system. Isabelle/DOF 2.0 now also supports type-class polymorphism for its ontological classes. Using the new view on the linking in Isabelle/DOF 2.0 allowed by term-contexts, polymorphic algebraic structures can serve as the foundation to define novel semantic concepts for the linking within formal libraries and then be constrained using type-classes: advanced notions like *provenance* or security models can be expressed and used as a basis for a full management system of formal libraries archives like the AFP.

Deep Isabelle/DOF 2.0 Concepts like types, terms, theorems are defined at the Isabelle engine top-level and do not exist in Isabelle/HOL as logical objects. These objects are considered as meta-types in Isabelle/HOL and could be defined as types in a metalogic. Then, using this metalogic with the help of term-contexts and polymorphic ontological classes, they could be referenced in ontological classes. We propose a mechanism to reify meta-types into HOL using Isabelle metalogic, hence to allow a full integration of meta-types in Isabelle/DOF 2.0 ontologies. New possibilities like proof terms annotations may be used as the basis to facilitate the translation of proof terms between Isabelle/HOL and other theorem provers.

1.3 Structure of the Thesis

This thesis is composed of four chapters.

The first chapter sets out the foundations upon which our work is built. It presents the different layers bottom-up from the underlying logical framework to the higher-level Isabelle/DOF 1.0 ontology framework, and associated key concepts. It also presents the ontology examples extracted from existing Isabelle/DOF 2.0 ontologies that will be used to explain our work along this thesis, and the related works.

The second chapter introduces extensions of Isabelle/DOF 1.0 ODL and the ontology-controlled environment. The concept of term-contexts, making ontological references pervasive in Isabelle/DOF 2.0, and class invariants are set out and extension to Isabelle/DOF 1.0 structural constraint mechanism is explained. These extensions are deeply integrated in Isabelle/HOL. It allows to combine term-contexts support and Isabelle/HOL programming language properties to define a query mechanism for advanced search. Finally we show the utility of class invariants as first class citizen. They become part of an ontology and the Isabelle theorem prover can help prove properties on ontologies involving these invariants.

The third chapter explains the new parametric polymorphism support in ontological classes. Ontological concepts defined using polymorphism are used to present diverse semantics: the linking in Isabelle/DOF 2.0 can capture the notion of provenance from database communities reconsidered as information on the document elements in Isabelle/DOF 2.0. A basic security model for integrated documents using Isabelle/DOF 2.0 new properties is presented.

The fourth chapter suggests a methodology to introduce formal ontological meta-data into generated proof objects using a metalogic of Isabelle/HOL [13]. These meta-data could be formalized information on the proof document structure to help in theorem reconstruction when translating proof terms between theorem provers.

The last chapter summarizes the achievements made in this thesis and sketches future extensions.

Chapter 2

Background

2.1 Introduction

Isabelle/DOF, an Isabelle componenet that we extend, is deeply integrated into the Isabelle platform. In this chapter we introduce the Isabelle theorem coceived as an “LCF style prover” and the presentation of its layered structure ranging from the generic Isabelle/Pure framework to the higher-level Isabelle/HOL based on the Higher-Order Logic (HOL), up to the Isabelle/Isar framework that adds a proof-checking environment. Then we set out the concept of axiomatic type classes, the order-sorted polymorphic type system of Isabelle, which is used extensively by Isabelle theories. Afterwards, we present Isabelle/DOF itself, with its Ontology Definition Language (ODL) and its ontology-controlled editing environment. Next, we give a brief overview of Isabelle IDE which recently added support for an incremental document preparation, and concepts of the Isabelle/Pure API heavily used internally when extending Isabelle/DOF. After, we explain the techniques for term evaluation in Isabelle, used together by the new implementation of Isabelle/DOF to generate ontological concepts, and present the ontological examples that will be used along the development of this thesis. These examples are already defined using the extensions of Isabelle/DOF. We conclude with a presentation of related works.

2.2 The LCF Approach

“The Logic for Computable Functions (LCF) approach aims at implementing the inference rules of a logical calculus within a proof kernel that has the exclusive right to create theorems” [14]. Edinburgh LCF pioneered this approach and Milner chose as its basis LCF that Scott proposed. Edinburgh LCF was influential in the techniques it introduced like "working in a theory hierarchy, and the central role of a functional programming language, ML." [14].

LCF-style systems define the abstract ML type of theorems, *thm*, and inference rules of the logical calculus are implemented as ML functions operating on the concrete representation of values of type *thm*. The values of type *thm* are propositions: for the judgment $\Gamma \vdash_\Theta \varphi$ which means that the proposition φ holds for assumptions Γ in a particular theory Θ, φ is a value of type *thm* and ML type checker guarantees the creation of the proposition of type *thm* using inference rules. Thus, there is no need to store the proofs of theorems and only the propositions are kept as values of the abstract type *thm*.

Edinburgh LCF introduced the notion of proofs tactics, i. e. instructions that transforms *thm*s and thus construct eventually proofs, that lead to a polysemic definition of *proof* that could be [14]:

- formal deductions of theorems from axioms using the inference rules of a logical calculus;
- executable code written using tactics or other primitives, expressing the search for such deductions.

Terms like proof objects and proof terms (or proof scripts) are sometimes used to distinguished the former from the latter: we'll stick to these but if the sense is not inferable from the context.

2.3 The Isabelle/Pure Framework

Isabelle [15] was designed as a generic theorem prover for interactive reasoning in a variety of logical calculi. So Isabelle can be thought as an instance of the traditional LCF approach, but where the abstract type *thm* formalizes a logical framework or meta-logic, in which other formalism can later be encoded. At its core is the Pure logic [16], that offers a reusable infrastructure to introduce different calculi, called objects logics, by specifying their syntax and natural deduction inference rules, notably, First-Order Logic (FOL), Zermelo-Fraenkel set theory, and Church's Higher-Order Logic (HOL).

The Pure logic, also called Isabelle/Pure, is an intuitionistic fragment of HOL of simply-typed schematically polymorphic λ-terms, consisting in three main layers. In the following, we introduce Isabelle/Pure's logical entities relevant in our context.

The layer of types τ is defined by:

$$\tau ::= \kappa\ \tau_1\ \dots\ \tau_k \mid \alpha$$

where κ are type constructors with a fixed arity like the constructor of *list*, *prop* is a nullary type constructor for the type of meta level truth values, α is a set of types variables like $'a$, $'b$, and $\alpha \Rightarrow \beta$ denotes the binary type constructor for the total function space from α to β. .

Terms t are represented by

$$t ::= t_1\ t_2 \mid \lambda x{::}'\tau.\ t \mid x{::}'\tau \mid c$$

where:

- c is a set of constant symbols like *min* or *max*.
- x is the set of variable symbols like A, B. The notation $x{::}'\tau$ is a type assertion in the language which means "x is required to have the type $'\tau$". Note that the syntactic categories x and α are disjoint: $'x$ is a possible type variable. Variables in the scope of a λ-operator are called bound variables, all others are free variables.
- $\lambda x.\ t$ is called a λ-abstraction, like for example the identity function $\lambda x.\ x$. A λ-abstraction forms a scope for the variable x.
- $t_1\ t_2$ is called an application.

The term layer adds some rules like the introduction of the application and λ-abstraction:

$$\frac{\Gamma \vdash t :: \tau \Rightarrow \sigma \quad \Gamma \vdash u :: \tau}{\Gamma \vdash t\ u :: \sigma} \qquad \frac{\Gamma,\ x :: \tau \vdash t :: \sigma}{\Gamma \vdash \lambda x :: \tau.\ t :: \tau \Rightarrow \sigma}$$

Proofs are abstract derivations and the resulting propositions are terms of the distinguished type *prop* [17], containing:

- implication $P \Longrightarrow Q$, using the built-in meta-level implication constant ($\Longrightarrow$),
- and universal meta quantification $\bigwedge x.\ P\ x$, to capture the idea "x must not occur free in the assumptions" for quantifier rules.

with additional primitive inference rules like introduction and elimination of the meta implication and the universal meta quantification:

$$\frac{\Gamma \vdash p : \bigwedge x :: \tau.\varphi \quad \Gamma \vdash t :: \tau}{\Gamma \vdash p \cdot t : P\{x \mapsto t\}} \qquad \frac{\Gamma,\ x :: \tau \vdash p : \varphi}{\Gamma \vdash \lambda x :: \tau.\ p : \bigwedge x :: \tau.\ \varphi}$$

$$\frac{\Gamma \vdash p : \varphi \Longrightarrow \psi \quad \Gamma \vdash q : \varphi}{\Gamma \vdash p \bullet q : \psi} \qquad \frac{\Gamma,\ h : \varphi \vdash p : \psi \quad \Gamma \vdash \psi : prop}{\Gamma \vdash \lambda h : \varphi.\ p : \varphi \Longrightarrow \psi}$$

where φ and ψ are terms of type *prop*, $\cdot$ denotes the application of propositions to terms, $\{x \mapsto y\}$ denote substitutions, application of a substitution is written as $u\{x \mapsto y\}$, and $\bullet$ the application of propositions to propositions. $\lambda x :: \tau.\ p$ is the abstraction over a term variable x of typ τ, and $\lambda h : \varphi.\ p : \varphi$ the abstraction over a proof variable h where h is a proof of the proposition φ. Concluded proofs become theorems of abstract ML type *thm*.

Conceptually types are implicit: terms are associated to types by type inference rules, similar to the Hindley-Milner type system. $'a \Rightarrow {}'b$ captures the idea of a term depending on a term, $\bigwedge x.\ P\ x$ the idea of proofs depending on terms and $P \Longrightarrow Q$ proofs depending on proofs, giving the structure of the logic.

Primitive inferences mostly serve foundational purposes. The main reasoning mechanisms of Isabelle/Pure operate on nested natural deduction rules expressed as formulas, using $\bigwedge$ to bind local parameters and $\Longrightarrow$ to express entailment.

By default the abstract type *thm* in Isabelle is conceptually equivalent to a proposition of type *prop*, meaning in the implementation that a term-value of a proposition term can be extracted from a theorem object of abstract type *thm*.

Deviating from the LCF approach, Isabelle can optionally provide an explicit proof object: A judgment $\Gamma \vdash_\Theta \varphi$ can be extended to $\Gamma \vdash_\Theta \psi : \varphi$ where ψ is a proof for the proposition φ, with ψ and φ represented as λ-terms. In the implementation, the proposition φ of type *prop* and the proof ψ of abstract ML type *proof* are extracted from the theorem object.

2.4 The Isabelle/Isar Framework

"Isabelle/Isar [18, 19] is a generic framework on top of Isabelle/Pure for developing formal mathematical documents with full proof checking. Definitions, statements and proofs are organized as theories." [20] Isabelle declares the abstract ML type *theory* alongside the type of theorems. "Type *theory* makes available the initial theory of the logical framework and allows for its extension as an acyclic graph of application theories." [14] A theory is a document whose concrete syntax looks like this:

```
theory Test
imports Main
begin
definition constant = term
theorem name:statement ⟨proof⟩
end
```

"The framework syntax emerges from three main syntactic categories: commands of the top-level Isar engine (covering theory and proof elements), methods for general goal refinements (analogous to traditional "tactics"), and attributes for operations on facts (within a certain context)." [21] The interaction between Isabelle and the user happens through these syntactic categories.

For example, the **definition** command:

Example 1 *Definition of the K combinator*

definition $K :: {}'a \Rightarrow {}'b \Rightarrow {}'a$ **where** $K\ x\ y \equiv x$

updates the theory by adding the constant K and the theorem $K\ x\ y \equiv x$ to the logical context:

consts $K :: {}'a \Rightarrow {}'b \Rightarrow {}'a$
theorem K_def: $K\ x\ y \equiv x$

The free variables x and y are implicitly generalized. Then the formal content added by this definition can be used to prove a property on this other definition:

Example 2 *Definition of the S combinator*

definition S **where** $S\ x\ y\ z \equiv x\ z\ (y\ z)$

For that the **lemma** command can be used:

lemma $S\_id : S\ K\ K\ x \equiv id\ x$
  unfolding $K\_def\ S\_def\ id\_def$ **by** $(rule\ reflexive)$

The theorem is stated as a proposition, accompanied by proof text that builds the corresponding derivation. In that proof, proof commands and proof methods are used and perform certain reasoning steps. Here, we unfold the definitions of the theorem $K\ x\ y \equiv x$, $S\ x\ y\ z \equiv x\ z\ (y\ z)$ and $id = (\lambda x.\ x)$ with the command **unfolding** and conclude the proof using the *rule* method to introduce the rule of reflexivity $x \equiv x$.

The theory is updated and the logical context contains the new theorem:

theorem S_id: $S\ K\ K\ x \equiv id\ x$

The Isabelle/Isar framework also defines a language for structural proofs, Isar, to help the one looking at the proof term to understand what is being proved at a given point. Inside a proof written using Isar, proof methods are still available.

Markup commands like ***chapter***, ***section***, etc. allow to introduce text structurally into documents with ***text*** dedicated to plain text internally accepting markdown-like structure like items and enumeration. The ***ML*** command evaluates the given text as ML source:

```
ML‹
fun factorial n =
  if n <= 1 then 1
  else factorial (n−1) ∗ n;

val t = 24 = factorial 4
›
```

assigns to t the ML value `true`. These commands are directly declared in Isabelle/Pure. Isabelle/Isar also reuses concepts defined in Isabelle/Pure like anti-quotations. Anti-quotations are semantic macros used in Isabelle commands embedding content referring to formal entities of the logic. They enable some degree of consistency-checking between a text or an ML source. This "formal content" inside a text or an ML command is checked within the current theory. For example:

```
text‹
@{term λx. x} and term‹λx. x› are the same term.
As for the formula @{term dist_safe = sqrt(d−a·(Δt)^2)}...
...and we see that the constant const‹K› is of type typ‹'a ⇒ 'b ⇒ 'a››
```

`@{`*term*`"`$\lambda x.\ x$`"}` or its short form `\<^`*term*`>`‹$\lambda x.\ x$› (pretty printed as ***term***‹$\lambda x.\ x$› in the example) introduces a term in a text block and makes $\lambda x.\ x$ appear in the final document output. Functions like $dist_{safe}$, *sqrt*, etc. have to be defined in the signature of the logical context or background theory of this formula. Isabelle comes with a couple of hand-programmed anti-quotations to reference formal entities like definitions using @{*const* ...} and types with @{*typ* ...}, but also other higher-level concepts like the abstract ML datatype *thm* with @{*thm* ...}, or *proof* with @{*prf* ...}

With this ***ML*** command:

```
ML‹val mk_S = @{const S ('a, 'b, 'c)}›
```

the @{*const S* ('*a*, '*b*, '*c*)} ML anti-quotation first checks that the constant *S* exist within the theory and then initializes *mk_S* as the representation in ML of the constant *S* in the logic.

Application theories contain definitions, proofs, ML-code, and text elements that commingle and form an integrated document.

2.5 Isabelle/HOL

"Isabelle/HOL [22] is based on Higher-Order Logic, a polymorphic version of Church's Simple Theory of Types. HOL can be best understood as a simply-typed version of classical set theory. The logic was first implemented in Gordon's HOL system [23]. It extends Church's original logic [24] by explicit type variables (naive polymorphism) and a sound axiomatization scheme for new types based on subsets of existing types." [20] It is an implementation of HOL as an extension of Isabelle/Pure: types of Isabelle/Pure are identified with types in Isabelle/HOL, taking advantage of the default type-inference mechanism of Isabelle/Pure. Isabelle/HOL introduces a distinguished Boolean type *bool* with constants *True* and *False* denoting truth values in connection with the usual logical connectives like $(\wedge)$, $(\longrightarrow)$, $\neg$, as well as the object logical quantifiers $\forall$ and $\exists$. In contrast to FOL, quantifiers may range over arbitrary types, including total functions $f :: 'a \Rightarrow 'b$. Isabelle/HOL is centered around extensional equality $(=)::'a \Rightarrow 'a \Rightarrow bool$. Extensional equality means that two functions f and g are equal if and only if they are point-wise equal. Isabelle/HOL is more expressive than FOL, since among many other things, induction schemes can be expressed inside the logic. To define new concepts, Isabelle/HOL provides *specification constructs* through the syntactic categories of commands. It defines new commands for a more comfortable interface to the user to define derived definition schemes which internally are mapped down to primitive ones [25].

The primitive way of introducing new types in Isabelle/HOL is the **typedef** specification construct. The declaration:

```
typedef 'a dlist = {xs::'a list. distinct xs}
  morphisms list_of_dlist Abs_dlist
proof
  show [] ∈ {xs. distinct xs} by simp
qed
```

defines the *dlist* type of distinct lists as a subset of *list* type restricted by the invariant *distinct xs* that uses the primitive recursive function *distinct* defined in the *HOL.List* theory. The command generates a proof obligation that the type is inhabited. The newly introduced type is accompanied by a pair of morphisms to relate it to the representing set over the old type. The explicit **morphisms** specification allows to provide alternative names. In our case the *list_of_dlist* constant will translate a *dlist* to a *list* and *Abs_dlist* will abstract a *list* to a *dlist*.

To define new induction schemes, Isabelle/HOL provides the **datatype** specification, which will generate a set of logical rules when introducing the *list* datatype:

```
datatype 'a list = Nil ([]) | Cons 'a 'a list (infixr # 65)
```

The **datatype** command allows for the specification of concrete mixfix annotations: for *Cons a* (*Cons b Nil*), the user may use the alternative [*a*, *b*].

Isabelle/HOL also offers the **primrec** construct to define primitive recursive functions over datatypes:

primrec *append* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixr** @ *65*) **where**
append_Nil: [] @ *ys* = *ys* |
append_Cons: (*x*#*xs*) @ *ys* = *x* # *xs* @ *ys*

The specification constructs **function** and **fun** defines functions by general wellfounded recursion with pattern matching where **fun** adds automated proof support for pattern matching and termination. For example:

fun *successively* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
successively P [] = *True* |
successively P [*x*] = *True* |
successively P (*x* # *y* # *xs*) = (*P x y* ∧ *successively P* (*y*#*xs*))

Here **primrec** is unusable because the pattern matching P [x] of the second equation is not a primitive pattern of the *list* datatype.

2.6 Axiomatic Type Classes

Isabelle possesses an order-sorted polymorphic type system [26, 27] (see [20], pp 250, for an introduction). This feature emerged from the genericity of Isabelle, whose support of polymorphic type variable could not be allowed in a logical framework. Indeed "some types are intrinsic to the framework itself and other types might be unsuitable to a particular object-logic. In FOL, the type of x in $\forall x.\ \varphi\ x$ must not involve functions or Booleans" [14] (recall the *bool* type introduced by Isabelle/HOL).

Order-sorted polymorphism introduces a hierarchy of sorts on types. Sorts add a further level of logical expressions : a sort s is an intersection of finitely many type classes C: s ::= $C_1 \cap \ldots \cap C_n$ where C_1, ... C_n are collection of types. This means that we can not only express genericity on a meta-level, but also that type-variables α, β, γ, ... can be constrained in a judgement α :: C where C is a type *class*.

The judgement α::{C_1,...,C_n} is a notation for α::s, where s is a sort. When combined with axioms, axiomatic type classes can be understood as an abstract interface to types, imposing operations and properties over them. For example, the type class constraint α::*order* may constrain the possible instances of α to those who possess an ordering operation *ord* :: $\alpha \Rightarrow \alpha \Rightarrow$ *bool* which must satisfy the properties of reflexivity, transitivity and anti-symmetry. When declaring that a concrete type, say *int*, is an instance of *order*, which is to say that the judgement *int*::*order* holds, this results in the obligation to define *ord* by a concrete operation, say $\leq$ :: *int* $\Rightarrow$ *int* $\Rightarrow$ *bool*, and proof obligations that this definition satisfies the required properties on *ord*-types. Thus, it is possible to define once and for all the operation *sort* :: (α::*order*) *list* $\Rightarrow$ (α::*order*) *list*, prove a bunch of theorems over it, and inherit them for the special case *sort* :: *int list* $\Rightarrow$ *int list* automatically, without reproving them.

Isabelle/Pure provides a syntax for axiomatic types class:

```
Isabelle
class semigroup_add =
  fixes plus :: 'a ⇒ 'a ⇒ 'a  (infixl + 65)
  assumes add_assoc: (a + b) + c = a + (b + c)
```

The *class* command introduces a *semigroup_add* type class that specifies a (+) operation using the **fixes** statement and an axiom *add_assoc* as a property for this operation using the **assumes** statement. The operation is lifted to a constant + :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'a* in the logical context where *'a* is constrained by the type class (*'a*::*semigroup_add*) and a theorem *add_assoc*: $(a + b) + c = a + (b + c)$ where free variables a, b, and c are implicitly generalized and also constrained by the type class due to the association of the (+) operation to the *semigroup_add* type class.

Isabelle provides the *instantiation* command:

```
Isabelle
instantiation bool :: semigroup_add
begin
```

to make the *bool* type an instance of the *semigroup_add* type class. The instantiation consists in first the specification of the operation and second proving that it satisfies the axiom. The specification of the (+) operation is given by a definition:

```
Isabelle
definition plus_bool: a + b = (a ∨ b)
```

Then the proof environment is introduced by the *instance* command:

```
Isabelle
instance proof
  fix a b c :: bool
  show a + b + c = a + (b + c)
by (induct a) (auto simp: plus_bool) qed
```

The proof uses *plus_bool* that was just defined to prove the *add_assoc* axiom still holds for the *bool* type.

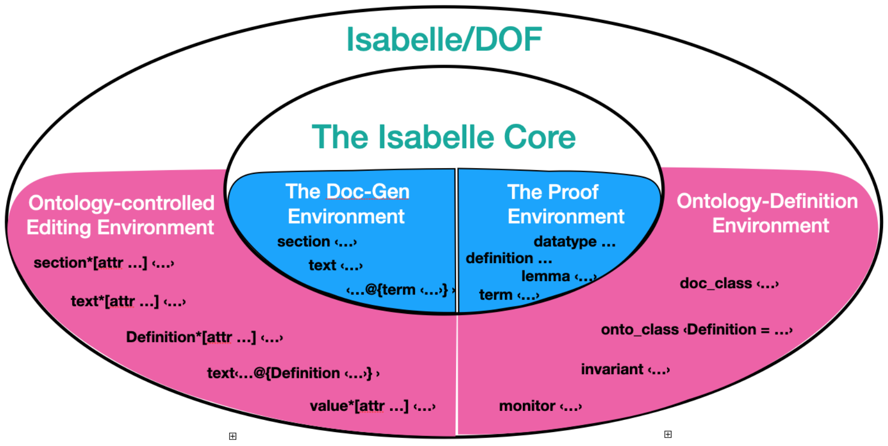


Figure 2.1: The Ontology Environment Isabelle/DOF.

2.7 The Isabelle/DOF Framework

Isabelle/DOF [10] extends Isabelle/HOL core (see Figure 2.1) by a number of constructs, allowing for the specification of formal ontologies (righ-hand side); additionally, it provides documentation constructs (left-hand side) for text-, definition-, term-, proof-, code-, and user-defined elements that enforce document conformance to a given ontology.

Isabelle/DOF[1] is a new kind of ontological modeling and document validation tool. In contrast to conventional languages like OWL and development environments such as Protégé [2], it brings forward the concept of *deep ontologies*, i. e., ontologies represented inside a logical language such as HOL rather than a conventional programming language like Java. Deep ontologies generate theories of strongly typed meta-information specified in HOL-theories allowing both efficient execution **and** logical reasoning about meta-data. They generate their own particular form of *anti-quotations* to be used inside ML-code and texts. Deeply integrated into the Isabelle ecosystem [28], thus permitting continuous checking and validation, they also allow for ontology-aware navigation inside large documents with both formal and informal content.

Isabelle/DOF provides a strongly typed Ontology Definition Language (ODL) (Figure 2.1, right-hand side). ODL provides the usual concepts of ontologies as they are often implemented. To draw a parallel with OWL and OWL API [29]:

- entities are defined using the concept of *document class* or *domain class* (using the `doc_class` and `onto_class` commands, respectively),

[1]The official releases are available at https://zenodo.org/record/6810799, the developer version at https://git.logicalhacking.com/Isabelle_DOF/Isabelle_DOF.

- data properties will be *attributes* specific to classes (attributes might be initialized with default values), and
- a special link, the reference to a super-class, establishes an *is-a* relation between classes.

The following command:

Example 3 *Definition of an author*

```
Isabelle
doc_class author₀ =
 name :: string
```

introduces the ontological concept of $author_0$ with its *name* attribute. Semantically, classes have a strong similarity to HOL's *extensible records* [20] used to represent them logically. The types of attributes are HOL-types, and concrete values for attributes are just HOL-terms, i.e. λ-terms over the signature of imported HOL-theories. Here the attribute type is specified as a *string*, a type declared in the *HOL.String* theory file. A default value *''Church''* for the *name* attribute can be specified:

```
Isabelle
doc_class author =
 name :: string <= ''Church''
```

Using the *is—a* relation, a localized author can extend the ontological class author:

Example 4 *Definition of a localized author*

```
Isabelle
doc_class loc_author = author +
 localization :: string
```

The *loc_author* will inherit the *name* attribute from the superclass *author*.

As a consequence of the logical representation, for each class, there will be a HOL type for the instances of a class. Therefore, class definitions allow for formal *links* to and between ontological concepts:

```
Isabelle
doc_class introduction =
   authored_by :: author set
```

The *author* class is used inside the *introduction* class to type its *authored_by* attribute. Since ODL specification elements are just another kind of command in Isabelle/HOL, they can be arbitrarily mixed with standard HOL specification constructs like inductive datatype definitions:

```
Isabelle
datatype role = developer | verifier | validator

onto_class requirement =
  long_name   ::string option
  is_concerned::role set
```

In our example, the enumeration for *role*s is used by a simplified version of CENELEC's requirement to enforce a separation of author groups in a process (see section 2.11).

The underlying Isabelle/DOF theory provides types for Isabelle types, terms, and theorems as well as specific means to denote them when referring to them in meta-data.

To define instances of these ontological concepts, Isabelle/DOF, as a document centric framework, extends Isabelle **text**-command using Isabelle/Pure ML API and defines the **text∗**-command to add ontological concepts to informal text. This command is part of the ontology-controlled editing environment (Figure 2.1, left-hand side):

$$\texttt{text}*[label::cid,\ attrib\_def_1,\ldots,attrib\_def_n]\langle\ldots\ annotated\ text\ \ldots\ \rangle$$

The ontological definition is given between [...]-brackets, where *cid* is an identifier of an ontological class introduced in an ontology, with attributes $attrib\_def_1,\ldots,attrib\_def_n$ belonging to this class defined in ODL. For example:

```
Isabelle
text∗[church::loc_author,
       name=''Alonzo Church'',
       localization=''Paris'']‹ Author description ...›
```

defines the *church* instance of the ontological class *loc_author* with the attributes *name* and *localization* and associates it to the text inside the ‹...›-brackets. Here the *name* attribute might be omitted and the default value *''Church''* will then be used. When declaring an instance, unspecified attributes are possible and leads to instance definitions strongly typed but left undefined for evaluation. Later, attribute values of instances can be updated using the **update_instance∗** command:

```
Isabelle
update_instance∗[church::loc_author, localization:=''Madrid'']
```

Now the *localization* attribute-value of the *church* instance will be the *string ''Madrid''*. The text part of a **text∗** command can also be left empty giving the basic syntax to define instances of ontological concepts not associated with text. Additional ontological markup commands like **chapter∗**, **section∗**, etc., equivalent to Isabelle/Isar markup commands, allow to make structural presentation concepts in document ontological elements, using the same [...]-brackets.

Isabelle/DOF *ML∗*-command extends Isabelle *ML*-command to annotate code in the same way.

Using Isabelle/DOF the abstract notion of definition can be specified and give semantics to an informal explanation of safety using annotation:

```
text∗[safe::Definition, name=''safety'']‹Our system is safe if the following holds ...›
```

When declaring the *Definition* class, anti-quotations are generated, allowing to refer to the *safe* instance in another text element:

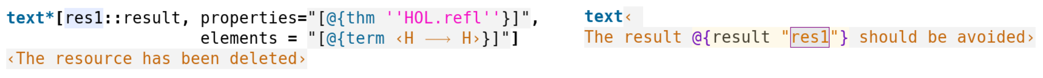
```
text∗[use_safe]‹As stated in @{Definition ‹safe›}, ...›
```

where Isabelle/DOF checks on-the-fly that the reference *safe* is indeed defined in the document and has the right type (it is not an *Example*, for example), generates navigation information (i.e. hyperlinks to *safe* as well as the ontological description of *Definition* in the Isabelle IDE) as well as specific documentation markup in the generated PDF document, e.g.:

As stated **in** *Def. 3.11* (*safety*), ...

where the underline may be blue because the layout description configured for this ontology says so. Moreover, this is used to generate an index containing, for example, all definitions. It should be noted that the *Definition* class defines an ontological concept of definition and then can be used to associate meta-data to an informal definition as text, but no linking to the concept of definition specified with the Isabelle/Isar command is possible, as only Isabelle/Isar definition instances are exposed as formal entities.

Isabelle/DOF implements a feature to handle meta-level datatypes like *thm*, *typ*, *term*. They are declared axiomatically and can be specified using anti-quotations like @{*thm* ...}, @{*typ* ...}, @{*term* ...} inside a λ-term. Lists of theorems of type *thm* can then be used as attribute-value for ontological class instances:



```
text*[res1::result, properties="[@{thm ''HOL.refl''}]",
                    elements = "[@{term ‹H ⟶ H›}]"]
‹The resource has been deleted›
```

```
text‹
The result @{result "res1"} should be avoided›
```

(a) A Text-Element as Result. (b) Referencing a Result.

Figure 2.2: Referencing a Result.

```
Isabelle
onto_class result =
  elements  :: term list
  properties :: thm list

text*[res1::result, properties=[@{thm ''HOL.refl''}],
                    elements = [@{term ‹H ⟶ H›}]]
‹The resource has been deleted›
```

The *elements* attribute of the *res1* instance is a list with one element, the term $H \longrightarrow H$, specified using the *term* anti-quotation, and its *properties* attribute a list with only the reflexivity axiom specified using the *thm* anti-quotation. These anti-quotations serve as abstract references: they are checked but remain empty syntactic categories.

Figure 2.2 shows an ontological annotation of a result and its referencing via an antiquotation @{*result res1*}; the latter is generated from the above class definition. Undefined or ill-typed references were rejected, and the high-lighting displays the hyperlinking which is activated with a click. The class-definition of *result* and its documentation is also just a click away.

As Isabelle/DOF is based on the idea of “deep ontologies”, a logical representation for an instance is generated, i. e. a λ-term, which is used to *represent* those meta-data. For this purpose, Isabelle/DOF uses Isabelle/HOL’s record support [20].

For the *loc_author* example, this means that the *church* instance is represented by:

- the record term $(\!|name = ''Alonzo\ Church'',\ localization = ''Paris''|\!)$ and the corresponding record type $(\!|name{::}string,\ localization{::}string|\!)$,
- while the resulting selectors were written *name* $(r{::}loc\_author)$, *localization* $(r{::}loc\_author)$ where r is the record term.

Isabelle/DOF implements the concept of monitor classes [28], which are classes that may refer to other classes via a regular expression in an **accepts** clause. Semantically, monitors introduce a behavioral element into ODL and enforce the structure in a document. For example, with the following definition:

```
Isabelle
doc_class title  =
  short_title :: string option <= None
doc_class author =
  name :: string <= ''Church''

doc_class text_document =
  style :: string
  accepts (title ~~ ⦃author⦄⁺)
```

we specify that a document must start with a *title* element and then must have at least one *author*.

Isabelle/DOF also proposes a syntax for class invariant:

```
Isabelle
doc_class author =
 name :: string
 invariant name_nempty :: λσ. name σ ≠ ''''
```

to declares invariants but they can currently only be specified in ML.

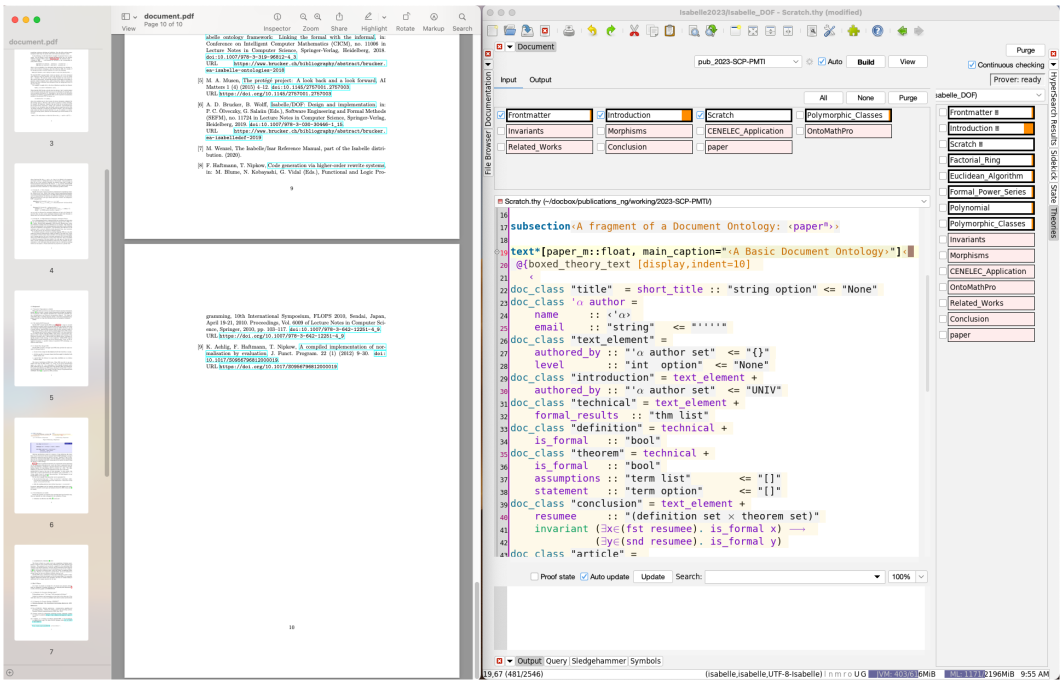
2.8 The Isabelle IDE

Wenzel introduced the document-oriented Prover IDE (PIDE) approach in [30, 31, 32, 33]. The main principles of PIDE are as follows [14]:

- the prover supports document edits an markup reports natively. Interaction works via protocol commands that take regular prover commands as data (e.g. definition, theorem).
- The editor connects the physical world of editor input events and GUI painting to the mathematical document-model of the prover.

"Isabelle/Jedit is the main application of the PIDE framework and the default user-interface for Isabelle." [34], based on the jEdit text editor. Recently the support of interactive document preparation was added to PIDE, and is accessible via the Isabelle/jEdit Document panel.

A screenshot of the editing environment is shown in Figure 2.3. It supports incremental continuous PDF generation which improves usability. Currently, the granularity is restricted to entire theories (which have to be selected in the document panel). The response times can not (yet) compete with a Word- or Overleaf editor, though, which is mostly due to the checking and evaluation overhead (the



abelle ontology framework: Linking the formal with the informal, in: Conference on Intelligent Computer Mathematics (CICM), no. 11006 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2018. doi:10.1007/978-3-319-96812-4_3.
URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelle-ontologies-2018

[5] M. A. Musen, The protégé project: A look back and a look forward, AI Matters 1 (4) (2015) 4–12. doi:10.1145/2757001.2757003.
URL https://doi.org/10.1145/2757001.2757003

[6] A. D. Brucker, B. Wolff, Isabelle/DOF: Design and implementation, in: P. C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods (SEFM), no. 11724 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2019. doi:10.1007/978-3-030-30446-1_15.
URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelledof-2019

[7] M. Wenzel, The Isabelle/Isar Reference Manual, part of the Isabelle distribution. (2020).

[8] F. Haftmann, T. Nipkow, Code generation via higher-order rewrite systems, in: M. Blume, N. Kobayashi, G. Vidal (Eds.), Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings, Vol. 6009 of Lecture Notes in Computer Science, Springer, 2010, pp. 103–117. doi:10.1007/978-3-642-12251-4_9.
URL https://doi.org/10.1007/978-3-642-12251-4_9

[9] K. Aehlig, F. Haftmann, T. Nipkow, A compiled implementation of normalisation by evaluation, J. Funct. Program. 22 (1) (2012) 9–30. doi:10.1017/S0956796812000019.
URL https://doi.org/10.1017/S0956796812000019

Scratch.thy (~/docbox/publications_ng/working/2023-SCP-PMTI/)

```

subsection‹A fragment of a Document Ontology: ‹paperᵐ››

text*[paper_m::float, main_caption="‹A Basic Document Ontology›"]‹
 @{boxed_theory_text [display,indent=10]
   ‹
doc_class "title"  = short_title :: "string option" <= "None"
doc_class 'α author =
    name      :: ‹'α›
    email     :: "string"   <= "''''"
doc_class "text_element" =
    authored_by :: "'α author set"  <= "{}"
    level       :: "int  option"   <= "None"
doc_class "introduction" = text_element +
    authored_by :: "'α author set"  <= "UNIV"
doc_class "technical" = text_element +
    formal_results  :: "thm list"
doc_class "definition" = technical +
    is_formal   :: "bool"
doc_class "theorem" = technical +
    is_formal   :: "bool"
    assumptions :: "term list"         <= "[]"
    statement   :: "term option"       <= "[]"
doc_class "conclusion" = text_element +
    resumee     :: "(definition set × theorem set)"
    invariant (∃x∈(fst resumee). is_formal x) ⟶
              (∃y∈(snd resumee). is_formal y)
doc_class "article" =
```

Figure 2.3: A Screenshot while editing this Paper in Isabelle/DOF with Preview.

turnaround of this chapter is about 20 s). However, we believe that better parallelization and evaluation techniques will decrease this gap substantially for the most common cases in future versions.

2.9 The Isabelle/Pure API

In the following we present several concepts of the Isabelle/Pure framework. Most of these concepts emerge at every level of Isabelle, and they are implicitly or explicitly used to build and extend the frameworks, and some are only accessible via Isabelle/Pure ML API. We also give hints on how the new version of Isabelle/DOF uses Isabelle/Pure ML API.

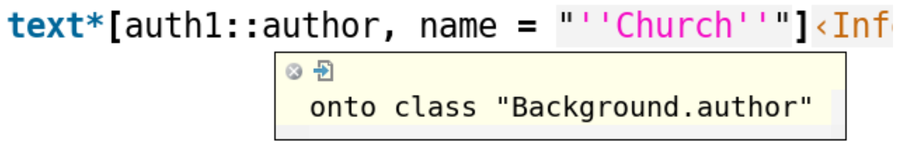
2.9.1 Contexts

From [21], "A logical context represents the background that is required for formulating statements and composing proofs. It acts as a medium to produce formal content, depending on earlier material (declarations, results etc.)". We used the image of a judgment $\Gamma \vdash_\Theta \varphi$ to describe derivations from inference rules in section 2.2, where the proposition φ is derivable from assumptions Γ in a particular theory Θ.

The same image can describe Isabelle/Pure derivations where Θ holds global declarations of polymorphic type constructors, term constants, and axioms, and Γ covers locally fixed type variables, term variables, and hypotheses. "Isabelle/Isar elevates Θ to a theory context and Γ to a proof context, both supporting arbitrarily typed data, being introduced by the user at compile time" [35]. Then using this generic data management, an object logic like Isabelle/HOL (in its implementation of Higher-Order Logic) adds specific components for automated reasoning like a classical reasoner and derived specification mechanisms like recursive functions. Isabelle/DOF extends Isabelle/HOL in the same way and acts as a component for document ontologies. It uses the Isabelle/Pure ML API to extend Isabelle syntax and documentation support, but also to offer the same level of integration in the IDE; and it represents its notions using Isabelle/HOL object logic. Depending of the notions it introduces and the implementation of the Isabelle/Isar framework, it has to deal with either a theory context, a proof context or the disjoint sum of either, a generic context; and it uses them to update the logical context data.

2.9.2 Name Spaces

"A name space manages a collection of long names, together with a mapping between partially qualified external names and fully qualified internal names (in both directions)" [21]. Name spaces use bindings that specify details about the prospective long name and the original source position in a document. Separate name space exist for each kind of formal entity, fact (theorems, lemmas, etc.), constant, type constructor, type class and implement the fact that for example, the syntactic categories c of constants and α of type variables are disjoint. It is the mechanism that will allow to fully integrate Isabelle/DOF notions in Isabelle/HOL and the IDE. By using name spaces for Isabelle/DOF concepts, the Isabelle engine can infer which kind of Isabelle/DOF entity a name refers to, and Isabelle/HOL and Isabelle/DOF entities can commingle. For example, the concept of ontological class instance previously only used internal data to present this logical entity to the user. But it was impossible to define two instances with the same name within an integrated document of application theories as they were considered as global entities, even so these entities were defined in two different theory files. More generally, Isabelle/DOF did not use name spaces for any formal entity, even in its core principle like anti-quotations. Using name spaces for ontological class data allow a deeper integration in Isabelle. Name spaces also carry original positions extracted from the bindings and can generate markup for reporting to the IDE. Name spaces are central to the extension of Isabelle/DOF to support term anti-quotations and term-contexts. Isabelle/DOF data model implementation was entirely rewritten to use name spaces.



```
text*[auth1::author, name = "''Church''"]‹Inf
onto class "Background.author"
```

Figure 2.4: *author* Pop-Up indicates that it is an Instance of the Ontological Class Formal Entity.

2.9.3 Markups

To fully integrate Isabelle/DOF in the IDE, and allow code navigation and pop-up information, Isabelle/DOF uses Isabelle/Pure ML API to carry code position and markup. "The standard way to provide the required position markup for input works via the outer syntax parser wrapper *Parse.inner_syntax*, which is already part of Parse.typ, Parse.term, Parse.prop" [21]. It means that Isabelle uses the content stored in name spaces entities to generate the reporting to the IDE. Isabelle/DOF uses a mixture of positions transported through the code from the parsed token and markups generated from name spaces to report to the IDE and allow code navigation and pop-up information for Isabelle/DOF entities. For example with the following definition:

```
Isabelle
text*[auth1::author,
    name = ''Church'']‹Informal description of the author @{author ‹auth1›}›
```

we get source code navigation and pop-up for the ontological class *author*. In Figure 2.4, when hovering over *author* with the mouse, we get the formal entity information of the class in the pop-up.

2.9.4 Configuration Options

A configuration option is a named optional value of some basic type (Boolean, integer, string) that is stored in the context [21]. Isabelle/DOF uses configuration options to parameterize ontology definition declarations, change the integration in the IDE, modify behavioral elements of the ontology language, etc.

2.10 Term-Evaluations in Isabelle

Besides the powerful, but relatively slow rewriting-based proof method *simp*, there are basically two other techniques for the evaluation of terms:

- evaluation via reflection into ML [36] (*eval*), and
- normalization by evaluation [37] (*nbe*).

The former is based on a nearly one-to-one compilation of datatype specification constructs and recursive function definitions into ML datatypes and functions. The generated code is directly compiled by the underlying ML compiler of the Isabelle platform. This way, pattern-matching becomes natively compiled rather than interpreted as in the matching process of *simp*. Aehlig et al [37] are reporting on scenarios where *eval* is five orders of magnitude faster than *simp*. In special applications, e.g., the verification of security protocols, *eval* can reduce the running time from several hours to a few seconds, compared to *simp* [38]. However, *eval* is essentially restricted to ground terms. *nbe* is not restricted to ground terms but lies in its efficiency between *simp* and *eval*.

2.11 Ontology Examples

ODL can be used to specify both document and domain ontologies; in the sequel, we will discuss the example *paper*^{m} in Figure 2.5 for the former and *cenelec*^{m} in Figure 2.6 for the latter. For the purpose of the presentation, we chose these fragments from existing ontologies of the Isabelle/DOF distribution. The ontology *paper*^{m} introduces document classes which are typical for the structure of a mathematical paper framing the canonical sequence "definition-theorem-proof". Attributes like *short_title* were typed with HOL types from the Isabelle library, and default values like *None* for class-instances can be declared. ODL can refer to any predefined type from the HOL library, e. g., *string*, *int* as well as parameterized types, e. g., *option*, *list*. Isabelle/DOF now also supports polymorphic variables in these types in order to support class schemata (see chapter 4. For example, in *Document_Ontology_Example.affiliation*, the precise format specification is left open due to the fact that publishers like Elsevier or ACM have very different requirements to represent them; thus, polymorphism is a means to increase reusability by abstraction.

Isabelle

```
doc_class title  = short_title :: string option <= None
doc_class affiliation =
 journal_style :: 'α
doc_class author =
   affiliations :: 'α affiliation list
   name         :: string
   email        :: string   <= ''''
   invariant ne_name :: name σ ≠ ''''
doc_class text_element =
   authored_by :: ('α author) set  <= {}
   level       :: int  option  <= None
   invariant authors_req :: authored_by σ ≠ {}
   and       level_req   :: (level σ) ≠ None ∧ the (level σ) > 1
doc_class introduction = text_element +
   authored_by :: ('α author) set  <= UNIV
   invariant author_finite :: finite (authored_by σ)
doc_class technical = text_element +
   formal_results  :: thm list
doc_class definition = technical +
   is_formal   :: bool
doc_class theorem = technical +
   assumptions :: term list        <= []
   statement   :: term option       <= None
doc_class conclusion = text_element +
   resumee     :: (definition set × theorem set)
   invariant is_form :: (∃ x∈(fst (resumee σ)). definition.is_formal x) ⟶
                    (∃ y∈(snd (resumee σ)). is_formal y)
doc_class article =
   style_id    :: string   <= ''LNCS''
   accepts (title ~~ ⦃author⦄⁺ ~~ ⦃introduction⦄⁺
            ~~ ⦃⦃definition ~~ example⦄⁺ || theorem⦄⁺ ~~ ⦃conclusion⦄⁺)
```

Figure 2.5: A Basic Document Ontology: paper^m

As for the domain ontology fragment *cenelec*^{m} shown in Figure 2.6, the definition proceeds as an extension of the *paper*^{m} ontology providing elements such as introductions, conclusions, formal and informal definition elements, etc. Note that *cenelec*^{m} is a very condensed version of some aspects of *one* artefact out of 27 in CENELEC EN 50128, the *software requirements specification* (SRS). As our model of the standard is 70 kB long and contains 1800 LOC, we will restrict ourselves to present zooms into certain chosen aspects in this thesis. The model heavily uses class invariants (see section 3.3) to offer an environment where document certifications are dynamically checked along the development of the documentation triggering errors or warnings to assist in the fulfillment of the CENELEC EN 50128 certification requirements.

Since ODL specification elements are just another kind of command in Isabelle, they can be arbitrarily mixed with standard HOL specification constructs like inductive datatype definitions — in our example, the enumeration for *role*'s, a simplified version of CENELEC's requirement to enforce a separation of author groups in a process. The Isabelle/DOF command ***class_synonym*** introduces equivalent names for classes; it also generates a ***type_synonym*** for the types induced by the ODL class, but is aware of the implicit type variables.

Note that the concept of *definition* appears in both ontologies. This is a consequence of the fact that this entity and similar common rhetoric constructions, like *assumption* or *consequence*, appear in many domains with slightly different meanings and document representations; a mathematician may have a different understanding of these terms than a lawyer or an engineer. ODL support of name spaces allow for a separation of these.

2.12 Related Works

In the introduction, ontologies are presented through the prism of documents. More generally, Gruber suggests the following definition: "In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse." But the methodology used to define the ontology language to formally represent an ontology highly influences the design of the language. For example, semantic networks [39] and frames systems [40] influenced the definition of Description Logics: the notion of individuals is reused to refine their definition as a formalism "that represents the

```
Isabelle
imports paper^m
onto_class 'a SR = 'a requirement +
  formal_definition :: thm list
class_synonym 'a safety_requirement = 'a SR

onto_class 'a FR = 'a requirement +
  formal_definition :: thm list
class_synonym 'a functional_requirement = 'a FR

onto_class 'a definition = 'a text_element +  — terminological definition.
  is_formal :: bool

datatype ass_kind = informal | semiformal | formal

onto_class 'a assumption = 'a text_element +
  assumption_kind :: ass_kind <= informal


onto_class 'a ASS = 'a assumption +
  is_concerned :: role set <= UNIV
class_synonym 'a application_constraint = 'a ASS

onto_class 'a EC = 'a assumption +
  assumption_kind :: ass_kind <= formal
class_synonym 'a exported_constraint = 'a EC
onto_class 'a SRAC = 'a EC +
  assumption_kind :: ass_kind <= formal
     formal_repr      :: thm list
class_synonym 'a safety_related_application_constraint = 'a SRAC

datatype kind = expert_opinion | argument | proof

onto_class 'a result = 'a technical +
 evidence :: kind
 property :: 'a theorem list <= []
 invariant has_property :: evidence σ = proof ⟷ property σ ≠ []
```

Figure 2.6: A Basic Domain Ontology: cenelec^{m}

knowledge of an application domain (the “world”) by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description).” [41] Another example in [42] follows the methodolgy of using an ontology for controlling the semantics of a conceptual model, proposed by [43]. The authors define K-DTT, a two-layered language based on the calculus of construction with inductive types and the Coq language using types classes. The ontological layer is then shaped by a reference ontology using a methodology that associates universals to general entities and particulars to instances, i. e. individiuals. The concepts of an ontology are represented in the underlying constructive logic. The authors also argue that Object-Oriented (OO) languages or the formalism of conceptual graphs lack expressiveness for logical or contextual reasoning, taking the example of F-Logic [44] as a tentative to add some logic to OO languages. Isabelle/DOF’s underlying ontology definition language ODL tries to offer the best of both world, expressiveness of both OO languages, like with F-logic, and logical background for reasoning, like with K-DTT. It has many similarities with F-Logic [44] and its successors Flora-2 [45], ObjectLogic — with *Ontoroker*[46][2] as a commercial ObjectLogic implementation —, and Ergo [47]. Isabelle/DOF also defines an OO language and shared features include object identity, complex objects, inheritance, polymorphic types, query methods, and encapsulation principles. Flora-2 adds a reification mechanism where statements inside ${...} are reified and made accessible, similar to constants name for definitions in Isabelle/HOL and to ontologized term in Isabelle/DOF using the ***term****∗* command (see section 3.2). Specific features like negation as failure in F-logic or defeasible reasoning in Ergo should be implementable using HOL functions and then could integrate queries as specified in Isabelle/DOF (see section 3.5). Motivated by the desire for set-theoretic modeling, F-logic and its successors possess syntax for some higher-order constructs but bases themselves on first-order logic; this choice limits the potential for user-defined data-type definitions and proofs over classes significantly. Originally designed for object-oriented databases, F-Logic became mostly used in the area of the *Semantic Web*. In contrast, Isabelle/DOF represents an intermediate layer between the logic HOL and its implementing language ML (having its roots as a meta-language for theorem provers). This “in-between”, where ontological concepts, like with K-DTT, are represented in the logic, and where a Isabelle/DOF ontology defines a theory of meta-data, allows for both executability and logical reasoning over meta-data generated to annotate formal terms and texts (see section 2.7 and section 3.2).

Other knowledge modeling languages originally targeting the semantic web are available like RDF [48] and OWL [49], and come with a query language, SPARQL [50], for the former and a rule language, SWRL [51], and a query lan-

[2]https://www.semafora-systems.com/ontobroker-and-ontostudio-x

guage, SQWRL [52], for the latter. These modeling languages are limited to a subset of first-order logic (using description logics formalism) or first-order logic (for OWL-Full) and the query languages provide query capabilities. Reasoners (for instance Pellet [53] or FaCT++ [54]) have been proposed to extend reasoning capabilities, and the ontology development platform Protégé [2] was extended with the PROMPT plugin [55] to address the ontology matching problem [56] with ontology mapping and merging features. But it lacks an advanced proof environment and a strongly type logic to define mapping function that allows statements like invariants preservation (see section 3.6). WSML [57] is the latest offspring of semantic web languages for handling semantic web services. WSML is based on different logical formalisms, namely, Description Logics, First-Order Logic and Logic Programming. Indeed "the most basic and least expressive variant is WSML-Core, which is based on DLP [58] as a least common denominator for description logic formalisms on the one hand and logic programming and rule-based systems on the other hand." [59] Then WSML-DL extends WSML-core towards the Description Logics paradigm, WSML-Flight extends WSLM-core with such features as meta-modeling, constraints and non-monotonic negation and is based on F-logic and WSML-Rule is an extension of WSML-Flight in the direction of Logic Programming. Finally WSML-Full aims at reuniting extensions under the same First-Order umbrella. "Specification of the semantics of WSML-DL and WSML-Full constitutes future work." [57]

The ISO 13584 (PLIB) Standard series [60] defines a model and an exchange format for digital libraries of technical components. A ontology model has been defined with a logical model specified in [61] and a PLIB ontology was proposed in [62]. Both the logical model and the PLIB ontology are specified using the EXPRESS language which became an ISO standard [63, 64]. EXPRESS, seen as an OO language share many similarities with ODL. EXPRESS can declare rules, i. e. specifications of one or more constraints on or between entity values, similar to invariants in Isabelle/DOF. EXPRESS is similar to programming languages such as Pascal but unlike K-DTT or ODL it does not rely on a logic for reasoning. Further formal specification and meta-programming were nevertheless investigated [65] which should help reasoning on the structural schema of knowledge.

In the introduction, we have defined a domain ontology as knowledge of a specific domain formally represented in the modeling language, ODL in our case. [66] refines the definition of a domain ontology using three criteria (an ontology should be formal, consensual and have the capability to be referenced) and distinguishes it from a conceptual model that is context dependent. Following this definition, ontologies defined in Isabelle/DOF should be able to fullfill the criteria: they are based on a formal theory, and we consider the concensual and capability to be referenced criteria as a technical limitation. Domain ontologies defined in Isabelle/DOF already satisfy common consensual criterion requirements: context definition is enriched by ontological concepts as they are logical objects and separation between concept definition and data structure prescription is allowed by using Isabelle/HOL proof environment with built-in specification constructs, and Isabelle/DOF ODL language with polymorphism and axiomatic type classes support. Then domain ontologies in Isabelle/DOF could become fully consensual with multiple instantiation requirement depending on the serialization capabilities and an associated technology allowing them to be updated online. And they could have the capability to be referenced by associating a unique identifier (whether it is a URI or a UUID) to every ontological concepts.

[67] focus on engineering domain and emphasizes its particularity when it comes to system design models: the authors argue that system design models are richer than document oriented models targeting the semantic web. We argue otherwise in the introduction when considering formal library texts. Document ontologies targeting mathematical and engineering domain require modeling languages able to represent formal concepts defined in a logical language. The defined engineering domain ontology will be used to annotate the formal library document in the later case and the designed system model in the former. Both will require contextual properties and associated proofs to validate the document or the system design model. Then the authors notably show how to define and exploit domain ontologies in Event-B [68] to enrich design models where the definition of the linking between ontological concepts and entities in the design model is represented by annotations formalized as relation definitions. This offer a very parametric notion of the linking. Isabelle/DOF takes another approach where the linking is specified by extended syntactic categories, i. e. Isabelle/DOF commands, and specific ontological concepts are attached as meta-data. The definition of the linking might require more work with Isabelle/DOF when new declared entities must also be considered as document elements.

The idea of annotations as relations serve as the basis of several approaches interested in engineering domain to enrich design models with properties and constraints. A first approach [69] aims at using an event-B context to model an ontology and then extending the context to instantiate the ontology. Using this approach, a methodology was proposed [70] to generate Event-B models from OWL ontologies and the resulting architecture was then extended [71] to describe engineering ontologies that can be found in ontologies description languages like OntoML [72], an ontology markup language, part of PLIB.

Another approach is used in [73] where a framework is defined in which generic components are developed using an ontology modeling language formalized as an Event-B theory [74], and specific components formalize a system behavior, leading to reusable domain-specific knowledge ontologies defining data types. Then domain ontologies are instantiated using an Event-B context and the instances properties and related constraints are attached to system design models using annotations defined as typing relations.

Regarding the use of formal methods to formalize standards, Mendil et al. [75] use the framework for formalizing standard conformance through formal modeling of standards as ontologies. The proposed approach was exemplified on the ARINC 661 standard. In another case, the Event-B method was proposed by Fotso et al. [76] for specifications of the hybrid ERTMS/ETCS level 3 standard, in which requirements are specified using SysML/KAOS goal diagrams. The latter were translated into Event-B, where domain-specific properties were specified by ontologies. These works are essentially interested in expressing ontological concepts in a formal method but do not explicitly deal with the formalization of invariants defined in ontologies. The question of ontology matching is not addressed. Another work along the line of certification standard support is Isabelle/SACM [77], which is a plug-in into Isabelle/DOF in order to provide specific support for the OMG Structured Assurance Case Meta-Model. The use of Isabelle/SACM guarantees well-formedness, consistency, and traceability of assurance cases, and allows a tight integration of formal and informal evidence of various provenance.

Let us now move away from domain ontology. Abstract reasoning about formal framework objects while still working within the logical language of the frameworks, i. e. introspective reasoning[3], is most commonly achieved by defining a meta-model. Riviere et al. propose an Event-B-based modeling framework, EB4EB [79], that allows for the explicit manipulation of Event-B features using meta modeling concepts. Using Event-B theories, objects of the Event-B framework like the design model that specify an event-transition system or theorems are represented using deep embedding, i. e. data types, and can be instantiated using deep or shallow embedding to exploit this framework, like defining a trace semantics to validate proof obligations formalized as Event-B operators that check the behavior of the design model [80]. METACoq [81] is an ambitious project that targets meta-programming in Coq, that is “writing programs (in a meta-language) that produce or manipulate programs (written in an object language).” For that, in addition to reified terms and typing judgements METACoq also reifies Coq commands like *Definition* and *Lemma* using a monad. METACoq is used by other projects like ConCert [82] to define a framework for smart contract verification. It is worth noting that Coq proofs are already objects of the formalism and theorems are constants. Dedukti [6] and its successor Lambdapi [7], are logical

[3]We use this term instead of the “reflective reasoning” vocabulary to not mislead the reader so he does not confuse it with a common acceptation of the term *reflection* in computer science since [78].

frameworks using the expressive $\lambda\Pi$-calculus modulo theory logical language that can be used as a proof checker. Many tools have been developped to translate proofs of interactive theorem provers to Dedukti. Some efforts to use Dedukti as a meta-programming language[4] were made, where they were interested in the normal form of some terms. Abstract reasoning in Isabelle/DOF is not so much interested in meta-programming than in adding information to objects of the logic. So it can follow the conservative extension approach: abstract objects like terms and types are reified using the metalogic and as others objects like definitions are formalized as λ-terms, they are already objects of the logic that can be annotated.

Isabelle/DOF is tuned towards systems with a document-centric view on code and semi-formal text as is prevailing in proof-assistants. Not limited to, but currently mostly used as a *document*-ontology framework, it has similarity with other documentation generation systems such as `Javadoc` [83, 84], `Doxygen` or `ocamldoc` [85, chap. 19]. These systems are usually external tools run in batch-mode over the sources with a fixed set of structured comments similar to Isabelle/DOF's antiquotations. In contrast, our approach foresees freely user-definable anti-quotations, which are in the case of references automatically generated. Furthermore, we provide a flexible and highly configurable LaTeX backend. Indeed Isabelle/DOF is interested in the presentation aspect: ontological concepts can be associated with presentation requirements and specifications. Isabelle/DOF can ontologize document elements like figures and slides, by adding meta-data to informal but structured text and linking the structured element to the LaTeX backend for presentation.

Obvious future applications for supporting the link between *formal* and *informal* content, i.e. between *information* and *knowledge*, consist in advanced search facilities in mathematical libraries such as the Isabelle Archive of Formal Proofs [86]. The latter passed the impressive numbers of 730 articles, written by 450 authors at the beginning of 2023. Related approaches to this application are a search engine like `http://shinh.org/wfs` which uses clever text-based search methods in many formulas, which is, however, agnostic of their logical context and of formal proof. Related is also the OAF project [87] which developed a common ontological format, called OMDoc/MMT, and six *export* functions from major interactive theorem prover systems into it. Limited to standard search techniques on this structured format, the approach remains agnostic on logical contexts and an in-depth use of typing information.

[4]https://github.com/Deducteam/Dedukti/tree/meta

Chapter 3

Term-Contexts and Isabelle/DOF Extension

3.1 Introduction

As Isabelle offers a document-centric view to the *formal* theory development process, this led to strong documentation generation mechanisms over the years, using built-in text and code anti-quotations. Isabelle commands serve as an interface to initialize and update the logical context and setup the environment within an integrated document, that is the acyclic graph of application theories. For example, a **`text`** command will setup the syntax, the parsers, the markup information, the annotations, the document elements, and so on, i. e. not only the logical elements but also the environment that may be involved when writing informal text. We call this setup a *context*, as an instance of the environment and logical context of a theory that is later refined as a theory context or a proof context. The **`text`** command introduces a *text*-context where its content, as informal text, can also have checked *text* anti-quotations and other user-defined macros like abstracted latex macros. The *ML*-command introduces its own *ML*-context to allow ML code to be parsed, interpreted and enrich the background theory within the same integrated document, and supports its own checked *ML* anti-quotations.

The Isabelle/DOF commands **`text*`** and *ML** can already be used to refer to our ontological concepts whether it is an ontological class, an ontological class instance, or an ontological attribute. Isabelle/DOF also offers references to some meta-level objects when inside a λ-term, like theorems and types, but these anti-quotations are empty syntactic categories and not ratable.

These features are not sufficient to allow advanced data handling pervasive in ontologies like classification and semantic validation. The ontological concepts should be made available inside λ-terms so they can reference each other instances.

Then an instance will be able to use other instances as an attribute-value. Furthermore, ontological and meta-level concepts should be ratable. Then advanced data handling become concrete. From these considerations emerge the concept of *term*-context, i. e. annotations to be made inside ratable λ-terms.

In this chapter, first we explain how term-contexts were implemented in Isabelle/DOF and how it can be integrated in the Isabelle framework by abstracting over Isabelle/Isar commands. Formal elements of formal libraries can now be reconsidered as document elements and become components in ontology classification and semantic validation. Then we explain the class invariant implementation which shares similarities with SWRL [51], with the important difference that they are made first citizen inside HOL object logic. The next section describes the extension of Isabelle/DOF monitors to work with term-contexts and expose relevant structural content for the definition of the linking in a document. Afterwards, we show how the combination of term-contexts and Isabelle/HOL offer a framework for advanced search inside Isabelle/DOF ontologies similar to SQWRL [52]; with Isabelle/DOF we are not limited to semantic queries, and we illustrate how Isabelle as a theorem prover can be used for semantic validation by proving morphisms on ontologies.

3.2 Term-Contexts

Term-contexts are introduced by the mean of Isabelle/DOF commands that extends Isabelle commands. Their content is not text nor ML code but λ-terms. They come in two kinds: *proper* commands that update the logical context or help structuring a document, and *diagnostic* commands that print, check or evaluate λ-terms.

For instance:

Example 5 *Examples of* `term*` *and* `value*` *commands with term anti-quotations*

```
Isabelle
term*‹ @{thm ''HOL.refl''}›
value*‹ @{thm ''HOL.refl''} = @{thm ''HOL.sym''} ›
```

The `term*` and `value*` Isabelle/DOF diagnostic commands are equivalent to Isabelle `term` and `value` commands respectively and extend the editing environment of Isabelle/DOF with new documentation constructs (recall Figure 2.1). They both parse and type-check a λ-term, and the latter also compiles and executes the λ-term. Their content is λ-terms that may comprise *term* anti-quotations treated by a refined process involving some or all of the following steps:

- *parsing* and *typechecking* of the term in HOL context. The logical context may be a theory context, a proof context, or a generic context, depending on the kind of the command,
- ontological *validation* of the term:
  - the arguments of term anti-quotations are parsed and checked,
  - checks resulting from ontological invariants were applied,
- *generation of markup information* for the navigation in the IDE,
- *elaboration* of term anti-quotations: depending on the anti-quotation specific elaboration function, the anti-quotations containing references were replaced by the object they refer to, and
- *evaluation*: HOL expressions were compiled and the result executed.

As we want to be able to have advanced treatment like classification, evaluation of an object referenced by a term anti-quotation should return a result, whether it is a reference to a meta-type or an ontological instance.

In the previous implementation, meta-types were declared as abstract entities making their evaluation impossible. Indeed, Isabelle code evaluation uses the *term_of* axiomatic type class to make λ-terms allowed type instances for the evaluation process. But Isabelle/DOF meta-types were declared as axiomatic types using **typedecl**: this command just declares a new type constructor and does make it a *term_of* type class instance. Using the **datatype** command, newly defined meta-types can be evaluated:

```
Isabelle
datatype thm = Isabelle_DOF_thm string (@{thm _})
```

The datatype *thm* defines a type but also declares the term anti-quotation as a constructor (*Isabelle_DOF_thm*) that takes a *string* argument and adds the notation @{*thm* _} where _ is a placeholder for the argument. The **datatype** command also declares its type as an instance of the *term_of* type class making terms of type *thm* ratable.

In Example 5, the **term∗** command parses and type-checks this λ-term as usual; logically, as we just saw, the @{*thm ''HOL.refl''*} term anti-quotation is predefined by Isabelle/DOF as a constant *Isabelle_DOF_thm*. The validation will check that the string *''HOL.refl''* is indeed a reference to the theorem in the HOL-library, notably the reflexivity axiom. The type-checking of **term∗** will infer *thm* for this expression. Now, if we look at the **value∗** command: this time the type-checking will infer *bool* for this expression and then will replace each term anti-quotation by a constant representing a symbolic reference to a theorem; code-evaluation will compute *False* for this command. Note that this represents a kind of referential equality, not a “very deep” ontological look into the proof objects (in our standard configuration of Isabelle/DOF). Further, there is a variant of **value**∗, called **assert**∗, which additionally checks that the term-evaluation results in *True*.

For the ontological instances, the integration to the evaluation process can reuse the elaboration process just defined.

First, ontological instances should generate ratable values. We saw in section 2.7 that unspecified attributes make ontological instances left undefined and then not ratable. Also, in Isabelle/DOF the logical representation of an instance, i. e., the record value, can be updated after its declaration using the **update_instance**∗ command. The implementation of both ODL and the editing environment need to be updated. For ODL, the idea is to generate a record value that is sufficiently defined to be processed by the code evaluation but still compatible with Isabelle/DOF instance update mechanism. For that Isabelle/DOF now uses a combination of term evaluation techniques in order to evaluate the instance record value. For example:

```
Isabelle
text∗[church1::loc_author]‹ Author description ...›
```

The *church1* instance declaration of the *loc_author* defined in Example 4 does not specify any attribute-value. Then it inherits the default value for its *name* attribute from its *Background.author* superclass but the *localization* attribute is left unspecified. Isabelle/DOF will define this attribute as a term variable. The record value of the *church1* instance is:

$(\!|name = {''Church''},$
$localization = Background\_loc\_author\_localization\_Attribute\_Not\_Initialized|\!)$

where its *localization* attribute-value

$Background\_loc\_author\_localization\_Attribute\_Not\_Initialized$

is a term variable whose type is *string*. To obtain this result, we use a generic way to generate record value that integrates well with Isabelle/DOF. The record package used internally by Isabelle/DOF generates HOL constants when declaring a record. Among them, the *make* constant allows to declare a record value in a functional way by taking each attributes as arguments in their declaration order. A first default record value is generated with only term variables as attributes using the *make* constant. This way an instance declared without any attributes specified and without default value inherited from its class hierarchy is ratable. For example:

```
Isabelle
doc_class loc_author₀ = author₀ +
 localization :: string

text*[church2::loc_author₀]‹ Author description ...›
```

Here the attributes of the *church2* instance are left unspecified and no default value were specified in the class hierarchy: neither $author_0$ defined in Example 3 nor loc_author_0 specify a default value for the *name* attribute. The record value generated is:

$$(\!| name = Invariants\_loc\_author_0\_name\_Attribute\_Not\_Initialized, \\ loc\_author_0.localization = \\ \quad Invariants\_loc\_author_0\_localization\_Attribute\_Not\_Initialized |\!)$$

where each attribute-value is a well typed term variable. This record value is ratable using *nbe*. Then if default values were specified in the class hierarchy, they are used to update the record value reusing Isabelle/DOF update record mechanism as if its was the result of the Isabelle/DOF `update_instance*` command for example. Then specified attributes in the instance declaration are used to once again update the record value. Record value with attributes left unspecified are now always ratable using nbe. And if the record value is totally specified, then it is considered a ground term and *eval* is used.

The implementation of the generation of record values exposed a flaw in Isabelle/DOF implementation. By default Isabelle/DOF allows to overwrite attributes for example to specify new default value:

```
Isabelle
doc_class author₁ = Background.author +
 name :: string <= ''Alonzo Church''

text*[church3::author₁]‹ Author description ...›
```

Here the $author_1$ class overwrites the *name* attribute of the *author* class and specify a new default value *''Alonzo Church''*. But for the logical representation, the *name* attribute is still an attribute of the *Background.author* class, so a way to distinguished both the classes should be found to allow distinct logical representation. The previous Isabelle/DOF implementation used a cumulative tagging system. The internal value for an instance would be:

$$(\!|(tag_0 = tv_0,\ attrib0_0 = v0_0,\ ...,\ attrib0_n = v0_n), \\ (tag_0 = tv_0,\ tag_1 = tv_1,\ attrib1_0 = v1_0,\ ...,\ attrib1_n = v1_n), \\ ...,\ (tag_0 = tv_0,\ tag_1 = tv_1,\ ...,\ tag_n = tv_n,\ attribn_0 = vn_0,\ ...,\ attribn_n = vn_n)|\!)$$

where $attrib0_0 = v_0$, ..., $attrib0_n = v_n$ are attributes $attrib0_0 ... attrib0_n$ of the super class with their values $v0_0 ... v0_n$. Superclass attributes are tagged using the attribute tag_0 with value tv_0. $attrib1_0 = v1_0$, ..., $attrib1_n = v1_n$ are attributes $attrib1_0 ... attrib1_n$ of the first subclass with their values $v1_0 ... v1_n$. The subclass attributes are tagged with $tag_0 = tv_0$, $tag_1 = tv_1$, and so on. When a subclass only overwrites attributes of its superclass, *tag* attributes are used to distinguished them internally. For example the internal value of the record value for the *church3* instance was:

$$\llparenthesis author\_tag\_attribute = tv_0,\ name = ''Alonzo\ Church'', \\ author\_tag\_attribute = tv_0,\ author_1.tag\_attribute = tv_1 \rrparenthesis$$

We can see the issue: The differentiation relies on all the class hierarchy of internal tag attributes. As the $author_1$ class does not declare any new attribute, the differentiation between the class is done using the second *author_tag_attribute* and the $author_1.tag\_attribute$ attributes. This lead to bloated record values and difficulties to generate them when using the *make* constant.

The new concept of *virtual* class is introduced to allow a clean generation of record values. Now the $author_1$ class, as it does not define any new attribute but only overwrites attributes from its class hierarchy, is considered a virtual class internally. Record values generation of virtual classes are treated specifically: only the tag attribute of this class is used to distinguish it from its superclass. For example, the internal record value of the *church3* instance is now:

$$\llparenthesis author\_tag\_attribute = tv_0,\ name = ''Alonzo\ Church'', \\ author_1\_tag\_attribute = tv_1 \rrparenthesis$$

So virtual classes exist logically as separate class to generate ratable record values and for the user they are fully ontological class. The introduction of virtual classes and the new mechanism of record values generation allows Isabelle/DOF class instances to always be ratable.

Once ontological instances are ratable, ontological classes should generate their own term anti-quotations. Term anti-quotations for ontological classes are declared as constants in Isabelle/HOL and are typed with the type of the logical record representation. For example:

```
Isabelle
value*‹ @{loc_author ''church1''}›
```

The **value*** command will replace the term anti-quotation by the record value of the *church1* instance of type *loc_author*. The record value is ratable because its type is an instance of the *term_of* type class. Indeed this type was declared by the record package used internally that instantiates the type properly. Ontological class term anti-quotations integrate fully in Isabelle/HOL by reusing the concept of name spaces.

Let's suppose that we are writing a theory that inherits two ontologies from different theories. These ontologies define ontological concepts using the same name but with different semantics. Isabelle/DOF will generate name spaced term anti-quotations so the user can use both definition concepts at the same time. For example:

```
Isabelle
value*‹ @{Theory1.definition ''Theory3.def1''} =
        @{Theory2.definition ''Theory4.def1''}›
```

The ontology in *Theory1* and the one in *Theory2* both define the concept of *definition*. The first term anti-quotation refers to the *def1* instance declared in *Theory3* as an instance of the ontological class *definition* declared in *Theory1* and the second to the *def1* instance of the *Theory4* theory instantiating the *definition* of *Theory2*. In the standard configuration of Isabelle/DOF, the **value*** command will result in *False* because both the instances have different types.

Name spaced anti-quotations reproduce for term-contexts the concept of name space pervasive in Isabelle. But contrary to anti-quotations in text-contexts and ML-contexts, term anti-quotations are terms defined in HOL as constants and exist in the logical context. So both unqualified and qualified anti-quotations must be declared and then internally mapped to the formal entity when they are elaborated. To reference instances of the class *Theory1.definition* declared in the *Theory1* theory, two term anti-quotations are generated: @{*definition* ...} and @{*Theory1.definition* ...}. When in the theory *Theory1* where the *definition* ontological class is declared, both term anti-quotations reference the same formal entity. When in a theory that imports the theory *Theory1* where the class was declared, whenever a new *definition* ontological class is declared, @{*definition* ...} will reference an instance of the new class, and @{*Theory1.definition* ...} an instance of the class declared in the imported *Theory1* theory. Internally the name space of the ontological class formal entities is used to map the term anti-quotation to the right formal entity.

With ratable record values of ontological class instances and term anti-quotations for ontological concepts and meta-types, we are now able to add and evaluate meta-data of abstract concepts.

The interaction in Isabelle/Isar happens through commands. But this syntactic category can be used for other purpose like abstract objects definition. In fact, the **text*** command of Isabelle/DOF does exactly that: it associates ontological concepts as meta-data to informal text. It means that the command is now considered as an abstract object. Meta-data are the abstract representation in the HOL object logic of a document element, the abstract object in Isabelle/DOF that is an Isabelle/Isar command. In the same way, **value*** as a command used for evaluation semantically amounts to the outcome of some evaluated λ-term, and **term*** is semantically a λ-term document element in Isabelle/DOF. In Isabelle/HOL, the equivalent commands are used for diagnostic, so they do not influence the logical context or the document structure. But in Isabelle/DOF, as they become document elements objects, they can play a role in the document treatment, and meta-data information associated to these commands are also associated to the logical meaning of those commands. Both `value*` and `term*` can extend the ontology-controlled editing environment. To become objects that can be referenced in ontological definitions, `term*` and `value*` support the [...] option to be associated with an ontological class and become ontological instances. This option is not only a prerequisite to support the linking between informal and formal information, as with the `text*` command, it also allows to make commands document elements and associate these document elements with their content, whether it is formal or informal.

Extending Isabelle/Isar commands by adding ontological definitions can be used in a systematic manner to define new abstract objects in Isabelle/DOF: this is an abstraction over commands pattern. For example, the commands:

```
Isabelle
doc_class oterm =
  is_formal   :: bool
doc_class otheorem =
  is_formal   :: bool
  assumptions :: oterm list        <= []
  statement   :: term option       <= None

term*[reflterm::oterm]‹s = t›

text*[secreq::otheorem,
      assumptions = [@{oterm ‹reflterm›}],
      statement = Some @{term ‹P s ⟹ P t›}]
‹The ontological notion @{otheorem ‹secreq›} ...›
```

define the instances *reflterm* and *secreq*. The *assumptions* attribute of the *otheorem* class instance *secreq* is a list with a single element, the instance *reflterm* referenced using a term anti-quotation. We can link the formal term referenced by the id *reflterm* to an informal representation of a theorem, the instance *secreq*. Semantically, *oterm* class instances are equivalent to the concept of definition in Isabelle/HOL. As definitions in Isabelle/HOL are declared as constants that can be referenced, the *oterm* class is not of great interest but with Isabelle/DOF we can abstract a step further. For that, higher level objects like definitions need special care. Consider the following commands:

```
Isabelle
doc_class odef =
  is_formal   :: bool
doc_class otheorem2 =
  is_formal   :: bool
  assumptions :: odef list        <= []
  statement   :: term option       <= None

definition*[reflterm2::odef] reflterm2 where ‹reflterm2 s t ≡ s = t›

text*[secreq2::otheorem2,
      assumptions = @{instances_of ‹odef›},
      statement = Some @{term ‹P s ⟹ P t›}]
‹The assumptions term- ‹assumptions @{otheorem2 ‹secreq2›}› of the ontoligical
notion @{otheorem2 ‹secreq2›} ...›
```

The newly defined Isabelle/DOF command ***definition**** extends the standard Isabelle/Pure ***definition*** command. Using the abstraction over commands pattern, ***definition**** can have attached meta-data and becomes an abstract object in Isabelle/DOF, that is a document element in Isabelle/DOF. Meta-data give semantics to this abstract object linked to the formal content of the definition that exists in the logical context. Now we can have references to definitions using this command: new term anti-quotations can be defined to semantically amount to a list of *odef* definitions, like @{*instances_of* ‹*odef*›}. Here, by using ontology meta-data, we can reference abstract concepts like list of definitions of the same type, unlike in Isabelle/HOL where only definitions instances as formal entities can be referenced. The *reflterm2* definition becomes not only a constant in Isabelle/HOL but also a document element that can be used for advanced handling like linking formal definitions existing as formal entities in Isabelle/HOL to informal document element like *secreq2*. Definition objects that exist in the logical context as constants are enriched using ontology meta-data. Isabelle/DOF now allows advanced handling over higher-level formal concepts like definitions.

The attentive reader may have noticed that for these examples the term anti-quotations were not only used in the command content but also in its ontological definition: the ontological definition is also made aware of the term-context. The [...] option of each command augments the logical context by adding ontological elements, so a command, when elaborating the term anti-quotation, may need to be aware of its own ontological information. The is done by adding ontological elements to the logical context before parsing the command content. The text content of the *secreq2* instance contains a text anti-quotation *term_*. This anti-quotation is also aware of the term-context and allows to specify the term *otheorem2.assumptions* @{*otheorem2* ''*secreq2*''} that contains a term anti-quotation that references its own ontological definition.

The elements referenced using term anti-quotations may also used term anti-quotations. The *otheorem2.assumptions* attribute-value of the *secreq2* instance is specified using the term anti-quotation @{*instances_of* ‹*odef*›} leading to nested term anti-quotations in the *term_* text anti-quotation. Because expressions with nested term anti-quotation can become quite complex, a double strategy is chosen: the elaboration of the term anti-quotation uses an eager evaluation to add only term-values with fully resolved term anti-quotations in the logical context, but terms with non-elaborated term anti-quotations can be made available for debugging via a lazy evaluation using a configuration option.

The meta-type term anti-quotation @{*thm* ...} allows to refer to theorems but meta-data can not be attached to it and then its use for advanced handling is limited. Ontological commands **lemma***, **theorem***, etc. were added to Isabelle/DOF editing environment to have abstract theorems objects that can take part as document elements into ontological definitions.

In Isabelle/DOF formal concepts used by formal libraries are no more simply abstract entity instances that may be referenced, they become fully integrated document elements with formal content, and can take part in ontological definitions. The first concept that comes in mind for formal libraries may be references to ontologized formal concepts like definitions, using the **definition*** command, or theorems using **lemma*** **theorem***, etc. But by using ontologies, the semantics of the concept is left open: any kind of document treatment is possible and it can be formally expressed using ontological definitions.

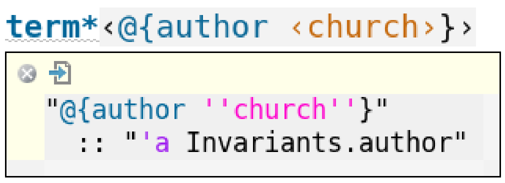
The resolution of each term anti-quotations follows the same pattern we just presented, but each concept it refers to needs special care with regard to markup. Term anti-quotations are designed using markup and binding to integrate fully in the IDE. For example the @{*otheorem2* ''*secreq2*''} term anti-quotation is made clickable to allow navigation in the IDE and basic markup information will pop-up when the pointer passes over. Dynamic information along the elaboration process is also integrated in the IDE.

We illustrate this integration in the IDE using some class instances of the *paper*^{m} and *cenelec*^{m} ontologies defined with the **text*** command, as in Figure 3.1.

In the instance *intro1*, the term antiquotation @{*author* ‹*church*›}, or its equivalent notation @{*author* ''*church*''}, denotes the instance *church* of the class *author*, where ''*church*'' is a HOL-string referring to an author text element in the global context. As explained earlier, one can also reference a class instance in a **term*** command as a term anti-quotation. In the command **term***‹@{*author* ‹*church*›}› the term @{*author* ‹*church*›} is type-checked (see Figure 3.2).

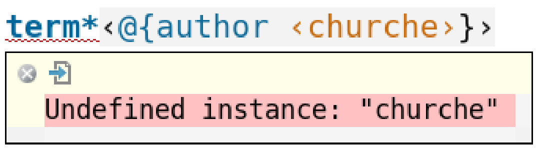
```
                                                                      Isabelle
text*[church::author, email=‹church@lambda.org›]‹... text›

text*[intro1::introduction, authored_by={@{author ‹church›}},
                    level=Some 0]‹... text›

text*[safety::theorem, assumptions = [@{term ‹s = t›}],
                   statement = Some @{term ‹t = s›}]‹... text›
text*[security::theorem, assumptions = [@{term ‹t = s›}],
                    statement = Some @{term ‹P s ⟹ P t›}]‹... text›

text*[proof1::'a result, evidence = proof,
                  property=[@{theorem ‹safety›}, @{theorem ‹security›}],
                  level = Some 2,
                  authored_by = {@{author ‹church›}}]‹... text›
text*[proof2::'a result, evidence = proof,
                  property=[@{theorem ‹security›}, @{theorem ‹safety›}],
                  level = Some 2,
                  authored_by = {@{author ‹church›}}]‹... text›
```

Figure 3.1: Some Instances referring to Figure 2.5.



```
term*‹@{author ‹church›}›
"@{author ''church''}"
  :: "'a Invariants.author"
```

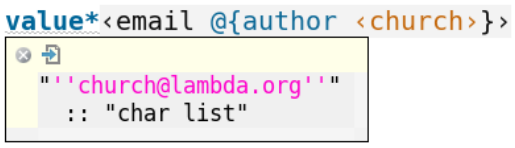
(a) Here, *church* is an existing Instance.



```
term*‹@{author ‹churche›}›
Undefined instance: "churche"
```

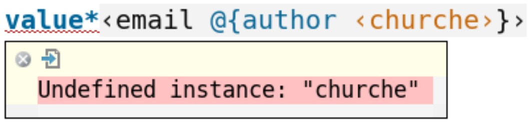
(b) The Instance *churche* is not defined.

Figure 3.2: Type-Checking of Antiquotations in a Term-Context.



```
value*‹email @{author ‹church›}›
"''church@lambda.org''"
  :: "char list"
```

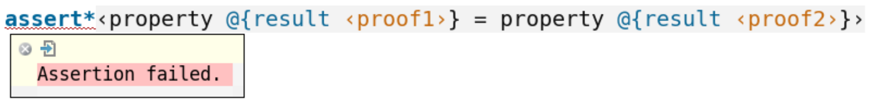
(a) The Evaluation succeeds.



```
value*‹email @{author ‹churche›}›
Undefined instance: "churche"
```

(b) The Evaluation fails.

Figure 3.3: Evaluation of Antiquotations in a Term-Context.



```
assert*‹property @{result ‹proof1›} = property @{result ‹proof2›}›
Assertion failed.
```

Figure 3.4: Evaluation of an Attribute of two Class Instances.

The command **value**∗‹*email* @{*author* ‹*church*›}› validates @{*author* ‹*church*›} and returns the attribute-value of *email* for the *church* instance, i. e. the HOL-string *''church@lambda.org''* (see Figure 3.3).

Referential equality is the standard interpretation for the meta-types, that is objects considered a types in Isabelle's metalogic. But for other concepts like ontological classes, the structural equality is directly possible. In the case of class instances, since term anti-quotations are basically uninterpreted constants, class instances can be compared logically. The assertion in the Figure 3.4 fails: the *property* attribute of class instances *proof1* and *proof2* is not equivalent because the lists sorting differs. When **assert**∗ evaluates the term, the term anti-quotations @{*theorem* ‹*safety*›} and @{*theorem* ‹*security*›} are checked against the global context such that the strings ‹*safety*› and ‹*security*› denote existing *theorem* class instances.

Abstraction over Isabelle/Isar commands pattern reuse already existing abstract entities with formal or informal content and make them document elements, and the introduction of the term-context allows to integrate document elements into ontological concepts using term anti-quotations. The is the first building block to define a framework for formal libraries.

3.3 Invariants

In semantic web communities, the limited expressivity of OWL drove diverse efforts, particularly with respect to properties (attributes in Isabelle/DOF). Rules languages [88, 89] were proposed like SWRL [51] whose purpose is a sound rule language by extending OWL with Horn-like rules. SQWRL [52] is a query language built on SWRL and uses its semantics foundation. SQWRL queries can be stored in OWL ontologies. Used together with OWL, SWRL and SQWRL offer an expressive environment for advanced search and classification. Isabelle/DOF was extended with the same idea in mind to use the same foundation for rules and queries. The counterpart of SWRL in Isabelle/DOF are class invariants. Originally restricted to checking function in ML, this limited their integration in an ontology definition and their usage with the Isabelle/Isar framework for semantic validation. The idea is to make invariants first class citizen in the HOL object logic; then reuse the Isabelle/Isar framework and Isabelle/DOF term-context and abstraction over commands pattern to match SQWRL features and do more. Indeed, as first class citizen, invariants are now parts of the theory of an ontology, and can become elements in proofs scripts. Through the support of term-contexts, term anti-quotations can also be used when specifying invariants constraints using the common HOL syntax.

To become first class citizen, class invariants are reified as follows: The invariant is declared as a λ-term with a reserved placeholder symbol for the future class instance it will be checked against, then a λ-abstraction is applied and updated as an equivalence over the name of the invariant and finally declared as a constant using a definition. Invariants are now a fully integrated part of an ontology definition and will play a key role in advanced handling like semantic validation and semantic merge. Indeed, like class attributes invariants are also inherited by subclasses. They were introduced by the keyword **invariant** in a class definition (recall Figure 2.5). The *ne_name* invariant of the *author* class, where the placeholder is the σ symbol, checks that the *name* attribute of an *author* instance is not an empty *string*. The term *name* $\sigma \neq []$ is first abstracted to obtain $\lambda\sigma.$ *name* $\sigma \neq []$. Then the generated equivalence gives:
ne_name $\equiv \lambda\sigma.$ *name* $\sigma \neq []$

The equivalence is used as the λ-term of a definition whose representation in Isabelle/Isar is:

```
definition ne_name where ne_name σ ≡ name σ ≠ ''''          Isabelle
```

In the same way, the *authors_req* invariant defined by the term *authored_by* $\sigma \neq \{\}$ is equivalent to the definition:

```
definition authors_req where authors_req σ ≡ authored_by σ ≠ {}          Isabelle
```

It enforces that a *text_element* instance has at least one author. Following the constraints proposed in [10], one can specify that any instance of a *result* class finally has a non-empty property list, if its *kind* is *proof* (see the *has_property* invariant). The *is_form* invariant specify the relation between the sets of *definition* and *theorem* document elements for the *resumee* attribute of the conclusion class and forces a *theorem* to be tagged as formal if its related *definition* also is. By relying on the implementation of extensible records in Isabelle/HOL [20], one can reference an instance attribute using its selector function. For example, in the *is_form* invariant, *resumee* σ denotes the *resumee* attribute-value of the future *conclusion* class instance.

Class invariants support constraints on the superclasses attributes using record package properties. To explain how this works, we first need to explain how the *is—a* class relation is done. Internally it uses the *is—a* relation of the record package that defines the notion of fixed and schematic records, supported at the level of terms and types. We already saw fixed records in section 2.7. A schematic record is represented by the record term

$(\!| x = a,\ y = b,\ \ldots = m |\!)$

and the corresponding record type

$(\!| x :: A,\ y :: B,\ \ldots :: M |\!)$

where "…" is a notation for possibly further fields, called the *more* part. "Fixed records are special instances of record schemes, where "…" is properly terminated by the $() :: unit$ element. In fact, $(\!| x = a,\ y = b |\!)$ is just an abbreviation for $(\!| x = a,\ y = b,\ \ldots = () |\!)$." [20] "For convenience, $(\alpha_1, \ldots, \alpha_m)\ t$ is made a type abbreviation for the fixed record type $(\!| c_1 :: \sigma_1, \ldots, c_n :: \sigma_n |\!)$ of a record t, likewise is $(\alpha_1, \ldots, \alpha_m, \zeta)\ t\_scheme$ made an abbreviation for $(\!| c_1 :: \sigma_1, \ldots, c_n :: \sigma_n, \ldots :: \zeta |\!)$." [20]

If a record t' of type $(\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_scheme$ extends a record t of type $\psi\ t\_scheme$, then $(\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_ext$ is an abbreviation for ψ, meaning $(\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_scheme$ and $((\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_ext)\ t\_scheme$ are the same type. We can see the *is—a* relation emerging from the schematic record type. A constant of type $(\alpha_1, \ldots, \alpha_m, \zeta)\ t\_scheme \Rightarrow \beta$ accepts terms of type $((\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_ext)\ t\_scheme$ as arguments, i. e., a term of type $(\alpha_1, \ldots, \alpha_m, \zeta)\ t'\_scheme$ where t' extends t.

To support constraints on superclasses attributes, the type of the λ-term elements of the invariants are updated: types of subterms like selectors of superclasses are rewritten to the schematic type of the current class.

Let's see an example:

```
Isabelle
doc_class 'a loc_author = 'a author +
  localization :: string
  invariant ne_email :: email σ ≠ ''''
```

The *loc_author* class is a subclass of the *author* class defined in Figure 2.5, and specify the constraint in its class invariant *ne_email* that the *email* attribute of its instances must not be empty. In the *ne_email* invariant the σ symbol, as a placeholder for the future instance of the class *loc_author*, is of type *'a loc_author*. But the selector *email* that returns the *email* attribute value of the class *loc_author* is generated when declaring the class *author*, so its type is *'a author_scheme* $\Rightarrow$ *string*. The type of the subterm σ and the type of the subterm *email* are updated to *'a loc_author_scheme*. As the type *'a loc_author_scheme* is the same as the type (*'a loc_author_ext*) *author_scheme*, and also an instance of the type scheme *'a author_scheme*, the *email* selector accepts the future instance represented by the σ symbol as an argument and the overall λ-term is well typed.

Using type rewriting, attributes selectors of any class in the class hierarchy of the current class may appear in a class invariant. Term rewriting also enables the support of invariants constraints on attributes of attributes, i. e., on attributes of ontological classes whose type is another ontological class. For example:

```
Isabelle
doc_class 'α text_element =
   authored_by :: 'α author set  <= {}
   level        :: int  option  <= None
   invariant authors_req :: authored_by σ ≠ {}
   and        level_req   :: the (level σ) > 1
   and        ne_emails :: ∀ x ∈ authored_by σ. email x ≠ ''''
```

We have added the *ne_emails* invariant to the *text_element* class that declares a constraint on the *email* attribute of the elements of its *authored_by* attribute which is a *set* of *author* classes.

Invariants are also aware of the term-context and term anti-quotations are allowed in their declaration. Here is a small example that extends the previous one:

```
Isabelle
text*[church::'a author, email=‹church@lambda.org›]‹›

text*[scott::'a loc_author, email=‹scott@domain.org›]‹›

doc_class 'α text_element =
   authored_by :: 'α author set  <= {}
   level        :: int  option  <= None
   invariant authors_req :: authored_by σ ≠ {}
   and        level_req   :: the (level σ) > 1
   and        ne_emails :: ∀ x ∈ authored_by σ. email x ≠ ''''
   and       n_church :: ∀ x ∈ authored_by σ. email x ≠ email @{author ‹church›}

text*[text_el1::'α text_element,
     authored_by={@{loc_author ‹scott›}}]‹›
```

The *n_church* invariant enforces each *email* attribute-value of *authored_by* attribute elements of the *text_el1* instance to be inequal to the *email* attribute of the *church* instance, referenced in the invariant using the term anti-quotation @{*author* ‹*church*›}.

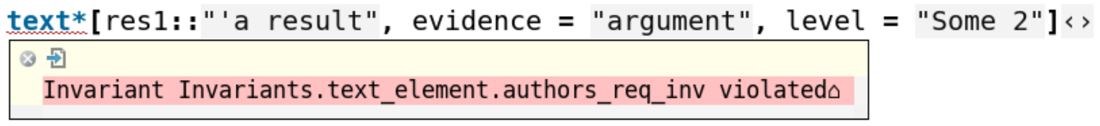
The value of each attribute defined for the instances is checked at run-time against their class invariants. Recall that classes also inherit the invariants from their super-classes. As the *result* class defined in Figure 2.6 is a subclass of the *text_element* class, it inherits its invariants. In Figure 3.5, we attempt to specify a new instance *res1* of this class However, the invariant checking triggers an error because the *authors_req* invariant forces the *authored_by* attribute to be a non-empty set and as its value was not set in the *res1* instance definition, *res1* inherits the default value from the *text_element* class which is the empty set.

The invariant checking mechanism uses configuration options to let the user choose the kind of checking he wishes. Basic proof tactics can be enabled involving **unfolding** commands, most commonly needed when invariants involve inductive predicates over algebraic structures, and the *auto* tactic. To make the choice with configuration options possible, an errors handling cascade is used: artificial terms and theorems are constructed to fill the gap along the cascade and shape the mechanism, leading to choices parametererized by configuration options to change the behavior of the checking. First the invariant is evaluated using the standard evaluation mechanism with *nbe* or *eval*. It might fail mainly due to a *Wellsortedness* error. This opens the next step in the errors cascade and introduce the proof tactics checking. This checking is not the first one used as it relies on *simp*, slower but more powerful than *nbe* or *eval*. At this step **unfolding** commands and the *auto* tactic are used. But *simp* generates theorems and not terms, so to integrate the errors cascade and allow the checking mechanism to go to the next step, artificial trivial theorems are generated to lure the type system.

For example:

```
Isabelle
text*[introduction1::'a introduction,
      authored_by = {@{author ‹church›}}]‹›
```

When declaring the *introduction1* ontological instance, the default checking (the evaluation using *nbe* or *eval*) of the *author_finite* invariant defined in the *introduction* class will fail. The evaluation of a λ-term that uses the *finite* inductive predicate requires its argument to be an instantiation of the *finite* axiomatic type class, but the argument, the attribute-value {@{*author* ′′*church*′′}} of the *introduction1* ontological instance does not instantiate the *finite* type class. So, the invariant is passed to the proof tactics method and then checked: the invariant definition is unfolded and the *auto* tactic finishes the proof. Then an artificial theorem is generated and serves as pattern matching in the error cascade to validate



```
text*[res1::"'a result", evidence = "argument", level = "Some 2"]‹›
Invariant Invariants.text_element.authors_req_inv violated
```

Figure 3.5: Inherited Invariant Violation.

the invariant.

Invariant declarations can lead to confusing behavior even with basic specification. The choice has been made to be permissive in definition specifications and rather give some feedback when a declaration triggers an error. With the *introduction1* instance declaration, if the definition of the *level_req* invariant were to be the term $\mathit{1} < \mathit{the}\ (\mathit{level}\ \sigma)$ instead of the term $\mathit{level}\ \sigma \neq \mathit{None} \wedge \mathit{1} < \mathit{the}\ (\mathit{level}\ \sigma)$, a *Match* exception would be triggered. As the *level* attribute-value is not specified in the *introduction1* instance, the default value *None* inherited from the *text_element* class is used and the invariant checking mechanism triggers a *Match* exception: indeed the $\mathit{1} < \mathit{the}\ (\mathit{level}\ \sigma)$ will be reduced to $\mathit{1} < \mathit{the}\ \mathit{None}$ which is well typed but not ratable. This exception comes with contextual information to help debugging.

Class invariants in Isabelle/DOF are fully integrated in Isabelle whether it is inside HOL object logic as first class citizens or in the IDE using errors cascade handling and configuration options.

Before explaining how an equivalent to SQWRL is shaped, another important feature of Isabelle/DOF, monitor classes, should be available for advanced search and validation of documents due the structure they enforce.

3.4 Monitors Extension

Monitors classes, in addition to enforcing the structure in a document, also generate traces of ontological classes involved in the monitor *accept* clause. They could be of substantial value as information for formal libraries, for example when helping to define best practices to shape their content. So making monitor traces available for advanced document handling by using the term-context is an added-value. It involves several steps:

- enhance the structural configuration possibilities
- make traces ratable
- make the term-context aware of monitors traces using term anti-quotations

The structural constraint that could be enforced were informally specified but never fully implemented. The idea is to have two clauses for the monitor classes, the **accepts** clause that specifies the order, specified by a regular expression, in which classes instances have to appear in the document, and a **rejects** clause with a list of ontological classes. This allows specifying ranges of admissible instances along the class hierarchy:

- a superclass in the reject list and a subclass in the accept expression forbids instances superior to the subclass, and
- a subclass S in the reject list and a superclass T in the accept expression allows instances of superclasses of T to occur freely, instances of T to occur in the specified order and forbids instances of S.

A class is considered a subclass of itself in the following.

The **accepts** clause regular expression is compiled via an implementation of the Functional-Automata of the AFP [90] into a deterministic automaton. So each time a class instance is declared in the monitor we might advance one step in the automaton depending on the algorithm explained just above. To allow the structural checking to be configurable using configuration options, the first accepted class *fa* for the current class instance ι declared in the monitor at the step φ is extracted from the alphabet of classes α available at this step φ in the automaton, that is at this specific step in the regular expression of the **accepts** clause. *fa* is the most direct super class of ι in α. In the same way, the first rejected class *fr* of the current instance is extracted from the reject list ϱ: this is the most direct superclass of ι in ϱ. Then cases on the fact that *fa* and *fr* are inhabited or not are checked to allow configuration options to be used to parameterize the structural constraint.

Example 6 *Monitor example with* **accepts** *and* **rejects** *clauses*

```
Isabelle
doc_class mon_head =
  tmhd :: int
doc_class mon_A = mon_head +
  tmA :: int
doc_class mon_B = mon_A +
  tmB :: int
doc_class mon_C = mon_A +
  tmC :: int
doc_class mon_D = mon_B +
  tmD :: int
doc_class mon_E = mon_D +
  tmE :: int

doc_class monitor_M =
  tmM :: int
  rejects mon_A
  accepts mon_head ~~ mon_B ~~ mon_C

open_monitor*[monitor_M0::monitor_M]

text*[mon_A0::mon_A]‹›
text*[mon_head0::mon_head]‹›
text*[mon_E0::mon_E]‹›
text*[mon_C0::mon_C]‹›

close_monitor*[monitor_M0]
```

In Example 6, the **accepts** clause of the *monitor_M* monitor class uses the sequence constructor ~~ of regular expressions to enforce the structure of the document that must start with an instance of the *mon_head* class or one of its subclasses, followed by an instance of the *mon_B* or one of its subclasses, and so on, and the **rejects** clause rejects instances of the *mon_A* or one of its subclasses at each step in the sequence. *mon_A0* is the first ontological instance declared in the monitor instance *monitor_M0*, opened using the `open_monitor*` command. So we are at the first step φ_0 in the automaton of the **accepts** clause regular expression of the monitor class *monitor_M*: the alphabet α of accepted classes at this step is composed of the sole class *mon_head*. *mon_A0* is an instance of an instance of *mon_A*: The *fa* of *mon_A0* is *mon_head*, as it is a superclass of *mon_A0* and *mon_head* is in α. But the *fr* of *mon_A0* is *mon_A* as it is the sole class in ϱ and *mon_A* is a superclass of *mon_A* (recall we consider a class to be a subclass of itself). So the instance is rejected by the algorithm, and we stay at the step φ_0 in the automaton. Both *fa* and *fr* are inhabited, giving access to a specific choice parameterized by the configuration option to trigger an error or a warning in the IDE for this rejected instance. Next we look at the instance *mon_head0*. This time the instance is accepted at the step φ_0 because its class *mon_head* is a superclass of the rejected class *mon_A*, and we move to step φ_1 in the automaton. Here α is composed of one class, the class *mon_B*. *mon_E0* is an instance of the class *mon_E*. The *fa* of *mon_E0* is *mon_B* at step φ_1 but its *fr* is empty as the *mon_E0* class is not a subclass of *mon_A*. This gives access to another choice for another configuration option. *mon_E0* is accepted and we move to the final step of the automaton. At the end the monitor is closed by the `close_monitor*` command.

As monitors may span several theory files, they were extended to supports name spaces. It means that monitors clauses may refer to equivalent ontological concepts with different semantics already defined in other theories and mix them. For example:

```
Isabelle
doc_class monitor_M =
  tmM :: int
  rejects Theory1.mon_A
  accepts test_monitor_head ~~ test_monitor_B ~~ test_monitor_C
```

The new **rejects** clause definition will reject instances of the ontological class *mon_A* defined in the theory file *Theory1*, but still accepts *mon_A* instances defined in the current theory in the previous example as *mon_A* is a subclass of *mon_head*.

To constrain further the structure, invariants can be specified when opening the monitor instance, when closing it, or at each step an instance is declared inside the monitor. For now this type of invariants needs to be specified in ML but through traces generation and computation they can be exposed and add valuable information to the context. The computation of traces computation are possible at the ML level, but to have computable traces at the Isabelle/Isar level, they are reified. For that **rejects** and **accepts** clauses must generate ratable objects for the logical context. In the previous implementation, logical objects were declared as constants but were not computable. In Example 6, the clause **rejects** *mon_A* generated a constant *mon_A* (recall that syntactic categories like constants and types are disjoint, so *mon_A* can be a constant and a type) using the equivalent of the `consts` command in Isabelle/Isar, and an abstract type *doc_class* for their type. But this declaration does not generate code and leads to traces that are not ratable. The new generation mechanism redefines the abstract type as a datatype with a type constructor. We saw that the `datatype` command make terms ratable. Then the same mechanism as the invariant reification is used to make the constant ratable: the constant is generated as an HOL string of type *doc_class* using the datatype constructor, and lifted to a regular expression in HOL as an *Atom* of the language giving a constant of type *doc_class rexp*. This constant is declared as a definition that generates an associated theorem automatically added to the logical context, and adds a code equation for the code generation. This gives a fully computable constant. Next the constants list of the classes declared in the clauses are reified as HOL lists, and added as an hidden attribute to the instance. At the end, we get a fully computable *trace* class-attribute accessible by a classical selector of the ontological class, but it is still not possible to reference a trace attribute of a specific instance for further treatment.

We need to make the term-context aware of the abstract concept of an instance trace: A term anti-quotation is generated that will checks that the instance name argument is indeed a monitor class instance. Its elaboration could be left to the interpretation. In the current implementation, to allow an easy evaluation of *rexp* λ-terms and keep legibility in the output trace attribute, term anti-quotations are elaborated as HOL string lists. For example when evaluating the trace attribute of the *monitor_M0* monitor instance using the @{*trace_attribute ''monitor_M0''*} term anti-quotation just after the declaration of the *mon_A0* instance, that is when we are still inside the monitor, we get the list [*''mon_A''*], and when the monitor is closed, we obtain the list [*''mon_A''*, *''mon_head''*, *''mon_E''*, *''mon_C''*].

3.5 Queries in Isabelle/DOF

Evaluation of terms in Isabelle/Isar is done using the ***value*** diagnostic command, and ***value****∗* in Isabelle/DOF. Also, Isabelle/HOL implements commands like ***primrec*** and ***fun*** offering a logical language similar to functional programming languages. The combination of term anti-quotations elaborated to objects they refer to and Isabelle/HOL functional programming language capabilities paves the way for a new mechanism to query the "current" instances presented as a HOL *list*. Using functions defined in HOL, arbitrarily complex queries can therefore be defined inside the logical language on document objects defined through abstraction over Isabelle/Isar commands and evaluated using ***value****∗*. Contrary to SQWRL, queries in Isabelle/DOF do not need to support a *select* operator, as any ratable λ-terms can be the content of the ***value****∗* command and every concepts defined in Isabelle/DOF are represented as λ-terms in HOL using reification mechanisms. ***value****∗* is aware of the term-context so all the term anti-quotations defined in the previous sections are at disposal in the logical context. Thus, to get the property list of the *result* class instances, it suffices to process this meta-data via mapping the *property* selector over the *result* class:

```
value*‹map (property) @{instances_of ‹'a result›}›                Isabelle
```

Analogously we can define an arbitrary filter function, for example the HOL *filter* definition on lists:

```
fun filter:: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list                     Isabelle
  where filter P [] = []
         | filter P (x # xs) = (if P x then x # filter P xs else filter P xs)
```

to get the list of the *result* class instances whose *evidence* is a *proof*:

```
value*‹filter (λσ. evidence σ = proof) @{instances_of ‹'a result›}›   Isabelle
```

Queries on other Isabelle/DOF concepts like monitor classes are possible as their generated traces are also presented as a *list* of *string*. For this monitor specification:

```
doc_class doc_monitor =                                            Isabelle
  ok :: unit
  accepts ⟦introduction⟧ ~~ ⦃result⦄⁺ ~~ ⟦conclusion⟧
```

one can define an *is—in* function in HOL to check the trace of a document fragment against a regular expression:

```
definition word_test :: 'a list ⇒ 'a rexp ⇒ bool (infix is—in 60)   Isabelle
 where  w is—in rexp ≡  DA.accepts (na2da (rexp2na rexp)) (w)

definition example_expression
 where example_expression
          ≡ ⦃⌊''introduction''⌋ || ⌊''result''⌋ || ⌊''conclusion''⌋⦄*

  value*‹ (map fst @{trace_attribute ''monitor1''}) is—in example_expression ›
```

Here, the term anti-quotation @{*trace_attribute ''monitor1''*} denotes the instance trace of *monitor1*. It is checked against the regular expression *example_expression*. *example_expression* is compiled using *is—in* into a deterministic automaton. On the latter, the above acceptance test is still reasonably fast.

This example show the benefit of defining queries inside the logical language. Isabelle/HOL commands like `definition` and `function` are reused to develop personalized queries, giving the power of programming languages. The queries themselves are λ-terms, so their checking can be delegated to the Isabelle engine and through abstraction over the `value*` command, Isabelle/DOF can be used to generate meta-data to annotate them and define an ontology of reusable queries associated to other ontology domains. The queries meta-data could even be proofs over the evaluation outcome, using term anti-quotations that reference theorems specified with `lemma*` or `theorem*` commands.

Contrary to SQWRL, in Isabelle/DOF we are not limited to queries outcome checking. Indeed Isabelle/DOF generates a theory of meta-data that can be used inside Isabelle/HOL environment to prove properties on ontologies.

3.6 Proving Morphisms on Ontologies

The Isabelle/DOF framework does not assume that all documents refer to the same ontology. Each document may even build its local ontology without any external reference. It may also be based on several reference ontologies (e. g., from the Isabelle/DOF library). Making a relationship between a local ontology and reference ontologies is a way to express that the content referencing a local ontology is not "far away" from a domain reference ontology.

Since ontological instances possess *representations inside the logic*, the relationship between a local ontology and a reference ontology can be represented by a formalized morphism. More precisely, the instances of local ontology classes may be mapped via conversion functions to one or several other instances belonging to another ontology. Since an instance representation as well as the conversion functions are constructed inside HOL, it is possible to prove formally once and for all that the morphism preserves the invariants for all meta-data. This means that morphisms may provably be injective, surjective, bijective, and thus describe abstract relations between ontologies.

To illustrate invariance preservation of a morphism, we zoom into the *paper*^{m} example where *author*s for specific journals were defined. The example addresses the common problem that publishers require slightly different meta-data which might be a nuisance for an author when addressing a paper to a different journal. Our zoom refines the concept of *author* locally:

```
Isabelle
doc_class title  = short_title :: string option <= None
doc_class affiliation =
   journal_style :: 'α
doc_class author =
   affiliations :: 'α affiliation list
   firstname    :: string
   surname      :: string
   email        :: string   <= ''''
   invariant ne_fsnames :: firstname σ ≠ '''' ∧ surname σ ≠ ''''
```

We specialize authors in the following:

```
Isabelle
doc_class acm_author = acm author +
  orcid :: int
  footnote :: string
doc_class elsevier_author = elsevier author +
  short_author :: string
  url          :: string
  footnote     :: string
```

Each class inherits the *affiliations* attribute from the *author* class and defines a list of *affiliations* as specified by the journal. In our example, *elsevier_author* and *acm_author* implement the specification of an Elsevier article or of an ACM article respectively. Also, each class inherits the *author name* attribute and the *ne_name* invariant that enforces its *name* to be non empty.

As a local ontology, it may have different meanings and document representations when compared to *paper*^{m}, which "live" together in the same document but in different name-spaces. This ontology defines a specific *author* class *elsevier_author* that implements an Elsevier article author. It inherits the *firstname* and *surname* attributes from the local *author* class. It also inherits the *ne_fsnames* invariant that requires that *firstname* and *surname* are non-empty.

Using this ontology we are now able to update Elsevier article authors from the local ontology to ACM article authors from the reference ontology, for example. And if all the specification of an Elsevier article were to be defined in our local ontology, we would be able to convert the meta-data of an entire Elsevier article to an ACM article. To update an Elsevier article author, we define a relationship between the local ontology and the *paper*^{m} ontology using conversion functions (also called *mapping rules* in the ATL framework [91] or in the EXPRESS-X language [65]). These rules are applied to define the relationship between one class of the local ontology to one or several other class(es) described in our *paper*^{m} ontology. In our case, our morphism is represented by three conversion functions, addressing the conversion of base-data, the affiliation and finally the authors. The base-data conversion of the Elsevier enumeration type in the local ontology to the ACM enumeration in the reference ontology is defined as follows:

Isabelle

```
doc_class elsevier =
  organization :: string
  address_line :: string
  postcode :: int
  city :: string

definition elsevier_to_acm_morphism :: elsevier ⇒ acm
                                        (_ ⟨acm⟩_elsevier [1000]999)
   where σ ⟨acm⟩_elsevier = ⦇ tag_attribute = 1::int,
                          position = ''no position'', institution = organization σ,
                          department = 0, street_address = address_line σ,
                           city = elsevier.city σ, state = 0, country = ''no country'',
                          postcode = elsevier.postcode σ ⦈
```

The more high-level conversions concerning the affiliation is detailed as:

Isabelle

```
definition
elsevier_aff_to_acm_aff_morphism :: elsevier affiliation ⇒ acm affiliation
                                                (_ ⟨acm'_aff⟩_elsevier'_aff [1000]999)
   where σ ⟨acm_aff⟩_elsevier_aff = ⦇ tag_attribute = 1::int,
          journal_style = (affiliation.journal_style σ) |> (λx. x ⟨acm⟩_elsevier) ⦈
```

where (|>) is simply a reverse application combinator. The top-level conversion for the author looks as follows:

Isabelle

```
definition acm_name where acm_name f s = f @ '' '' @ s

definition elsevier_author_to_acm_author_morphism
                :: elsevier_author ⇒ acm_author
                   (_ ⟨acm'_auth⟩_elsevier'_auth [1000]999)
  where σ ⟨acm_auth⟩_elsevier_auth = ⦇ tag_attribute = 1::int,
                                  affiliations = (author.affiliations σ)
                                                 |> map (λx. x ⟨acm_aff⟩_elsevier_aff) ,
                                  name = acm_name (firstname σ) (surname σ),
                                  email = author.email σ, orcid = 0,
                                  footnote = elsevier_author.footnote σ ⦈
```

These definitions specify how *affiliation* and *elsevier_author* meta data representations are mapped to *affiliation* and *acm_author* objects as defined in $paper^m$. The *acm_author name* attribute-value is derived from the *elsevier_author firstname* and *surname* attributes using a parsing function *acm_name* that follows the ACM journal author specification. This mapping shows that the structure of a local (user) ontology may be arbitrarily different from the one of a standard ontology it references to.

In order to support morphisms, we implemented a high-level syntax for this:

```
onto_morphism (elsevier_author) to acm_author ..
```

where the ".." stands for a standard proof attempt consisting of unfolding the invariant predicates and a standard *auto* proof. This syntax also cover more general cases such as :

onto_morphism $(A_1, \ldots, A_n)$ **to** X_i **and** $(D_1, \ldots, D_m)$ **to** Y_j

were tuples of instances belonging to classes $(A_1, \ldots, A_n)$ can be mapped to instances of another ontology.

After defining the mapping rules, we have to deal with the question of invariant preservation. The following nearly trivial proof for a simple but typical example is shown below:

```
Isabelle
lemma elsevier_inv_preserved :
  ne_fsnames_inv σ ⟹ ne_name_inv (σ ⟨acm_auth⟩elsevier—auth)
  unfolding ne_fsnames_inv_def ne_name_inv_def
          elsevier_author_to_acm_author_morphism_def
  by (simp add: combinator1_def acm_name_def)
```

After unfolding the invariant and the morphism definitions, the preservation proof is automatic. The advantage of using the Isabelle/DOF framework compared to approaches like ATL or EXPRESS-X is the possibility of formally verifying the *mapping rules*, i. e., proving the preservation of invariants once and for all rather than converting data and then relying on a post-hoc check.

3.7 Conclusion

Abstracting over Isabelle/Isar commands lifts abstract formal concepts like definitions and theorems to Isabelle/DOF. Using term-contexts with term anti-quotations, they become document elements and can take part in the definition of the linking inside a formal document for structural validation using monitors or when proving morphisms on ontologies for semantic validation. Inside queries defined in HOL they are used for advanced search and the queries outcome can serve as a testing framework to help in the validation of a certification, for example. Class invariants in Isabelle/DOF are first class citizen in HOL and as

Isabelle/DOF is defined in HOL, they can be components of a Isabelle/DOF ontology. Also, queries in Isabelle/DOF are λ-terms, and as such they can become ontological objects through abstraction over the ***term**** command. Both can be embedded in the ontology and produce an integrated ontology where ontological concepts, invariants and queries commingle and are reconsidered as generic concepts to broaden the notion of an ontology.

Chapter 4

Parametric Polymorphic Classes for Ontologies

4.1 Introduction

Isabelle/DOF initial goal was to define an ontology language to help in structuring the linking between formal and informal content. The linking was thought as a coupling between informal text and formalized concepts represented in HOL object logic.

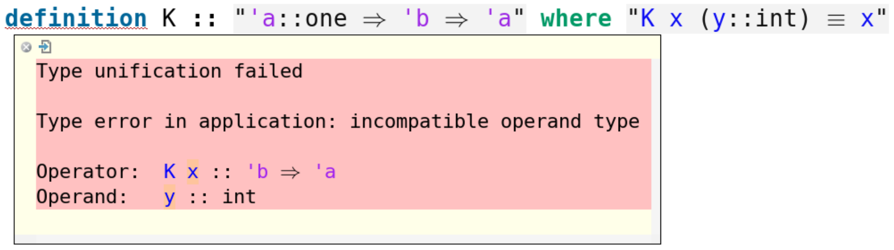
The new approach of term-context adds the possibility to use term anti-quotations inside ontological declarations using abstraction over commands pattern for commands that extend the editing environment on the one hand, and inside ontological definitions for default values of ***`doc_class`*** and ***`onto_class`*** commands in ODL on the other hand. This lead to a new meaning for the linking itself. Now ontological concepts can embed meta-data of informal and formal content reconsidered as document elements and these meta-data can contain references to other document elements. The linking is extended from a linking between document elements and their ontological definitions to a linking between document elements. We are able to specify semantics for the linking between documents elements by means of ontological meta-data.

This coincides with the overall objective of this work which is to express the machine-checkable linking between document element objects in formal libraries, whether they represent formal or informal content. The first version of Isabelle/DOF was implemented using non-polymorphic records to represent ontological classes logically. The idea was that the HOL-types pool in the main HOL library could be extended using HOL theories in the AFP to capture new semantic linkings as ontologies are developed. But would polymorphic support in Isabelle/DOF be somehow beneficial for formal libraries?

A first use that comes in mind is a pragmatic one: we mention in section 2.11 the reusability by abstraction. We would like to emphasize that, even if it is a very pragmatic example, it should not be underestimate as it reduces the burden when developing ontologies, increases their legibility, and simplifies their usage by giving a modular framework when extending them. In Figure 2.5, the ontology is modular in the sense that the format specification of the *affiliation* class is left open to the user, so it can be specified later reusing another ontology or defining its own definition of the journal style. In section 3.4, we propose an elaboration of monitors traces by representing them as HOL string lists, and add that the elaboration could be left to the interpretation. Because the trace attribute term anti-quotation is implemented in ML, giving the possibility to users to choose the elaboration of term anti-quotations would require extra work, but this is the same idea: leave interpretation open to the user. Could the linking between document elements be left open? And how to constrain ontological concepts to conform to some properties but let the user choose the final implementation? Axiomatic type classes offer exactly the type of abstract interface to answer these issues. If Isabelle/DOF were to support parametric polymophism (polymorphism parameterized by axiomatic type classes), the linking represented by meta-data could be left open to the interpretation. First, a user will have the possibility to choose the abstract representation of the concept through the type selection of the ontological class attributes, even when types are not ground, for example by using abstract algebraic structure. So he will be able to specify further an abstract definition by enforcing its structure and then constrain its properties using axiomatic type classes. Secondly, choosing how to evaluate the ontological instances of the concepts becomes feasable. The types can be left abstract until instances are evaluated, and then the user can choose between different evaluations according to the semantics he wants to give to the ontological concepts when in a specific context of interpretation.

Another consideration is that substantial HOL theories make extensive use of axiomatic type classes, like the Analysis library of Isabelle/HOL `HOL-Analysis` [92]. To use concepts defined in these libraries, Isabelle/DOF should support axiomatic type classes. The support of parametric polymorphism in Isabelle/DOF will not be beneficial to formal libraries only but to any document, regardless of its content.

In this chapter, we explain how we use the original manner Isabelle treats terms and variables to facilitate type inference as a basis to offer parametric polymorphism to ontological classes and maintain the usual experience when developing ontologies in Isabelle/Jedit editor. Then we show through the notion of provenance that emerged in data-base communities how parametric polymorphic classes increase Isabelle/DOF expressive power and allow to define new semantics for the



```
definition K :: "'a::one ⇒ 'b ⇒ 'a" where "K x (y::int) ≡ x"

Type unification failed

Type error in application: incompatible operand type

Operator:  K x :: 'b ⇒ 'a
Operand:   y :: int
```

Figure 4.1: The Type Unification Process triggers an Error.

linking between document elements in an ontology. Finally, we explain how this new polymorphic property is able to capture the notion of security models to offer an access-control model for the integrated document model promoted by the Isabelle framework.

4.2 Polymorphism Implementation

The syntactic category of commands in Isabelle often allows the specification of types in two ways: in the main type specification and inside the specification itself. In Example 1, the ***definition*** command specifies the type of the K combinator whereas in Example 2, the type of the S combinator is inferred from the ***definition*** content. The type specification can be composed of type variables with attached sorts and ground types, and types variables with sorts and ground types can also be attached to terms inside the ***definition*** content. Isabelle will check that the type specification matches the types attached to terms inside the ***definition*** content. For example:

definition $K :: {'a}{::}one \Rightarrow {'b} \Rightarrow {'a}$ **where** $K\ x\ (y{::}int) \equiv x$

will trigger an error explaining that the type unification has failed (see Figure 4.1). It comes from the type specification ${'a} \Rightarrow {'b} \Rightarrow {'a}$ that does not match the type of the content where y is a term variable with the ground type *int*, and not a type variable. The following definition will not fail:

definition $K :: {'a}{::}one \Rightarrow int \Rightarrow {'a}$ **where** $K\ x\ (y{::}int) \equiv x$

The specification of the *one* axiomatic type class for the ${'a}$ type variable does not trigger an error because its corresponding term variable x in the ***definition*** content does not come with a type specification, so the most general type for x will be inferred from the ***definition*** content and then during the type unification process, the generated term $\mathit{Polymorphic\_Classes.K}\ x\ y \equiv x$ will have an x term variable of type $'a{::}one$.

"The Pure logic represents the idea of variables being either inside or outside the current scope by providing separate syntactic categories for *fixed variables* (e.g. x) vs. *schematic variables* (e.g. $?x$)." [21] To facilitate type inference, Isabelle treats type variables with additional care. "In principle, term variables depend on type variables, which means that type variables would have to be declared first. For example, a raw type-theoretic framework would demand the context to be constructed in stages as follows: $\Gamma = \alpha\colon \mathit{type},\ x\colon \alpha,\ a\colon A(x_\alpha)$." [21] Isabelle will first fix term variables x "without specifying a type and the first occurrence of x in a specific assigns its most general type, which is then maintained consistently in the context. The above example becomes $\Gamma = x\colon \mathit{term},\ \alpha\colon \mathit{type},\ A(x_\alpha)$, where type α is fixed *after* term x, and the constraint $x :: \alpha$ is an implicit consequence of the occurrence of x_α in the subsequent proposition." [21]

It is implemented through the separation of the parsing and the checking of terms. The parsing will generate type constraints for terms variables with dummy types "_" assigned, and then the checking will check the validity of the types constraints against the type specification (the type specified with the ***definition*** command, for example) and will infer the final type of the terms variables.

The Isabelle/Isar commands for the record package allow for the same specification of types. To not disorient users, parametric polymorphic classes implementation in Isabelle/DOF should also support this. With ontological classes, it is even required as internally they use the record package for their logical representation. The ***record*** command:

record $(\alpha_1, \ldots, \alpha_m)\ t = \tau + c_1 :: \sigma_1 \ \ldots\ \ c_n :: \sigma_n$

defines a record where the types variables of τ, the optional parent record, and σ_i, the types of the fields, need to be covered by the (distinct) parameters $\alpha_1, \ldots, \alpha_m$, i. e., $\alpha_1, \ldots, \alpha_m$ parameters are mandatory.

The support for parametric polymorphic classes in Isabelle/DOF is two fold: on the one hand, support in ontological classes definition through the ***doc_class*** and ***onto_class*** commands of ODL, on the other hand support in the abstraction over Isabelle/Isar commands pattern, i. e., support by Isabelle/DOF commands that extend Isabelle/Isar commands like ***text****, ***definition****, ***lemma****, ***theorem****, etc., i. e. the ontology-controlled editing environment. For the abstraction over commands pattern, the behavior is similar to Isabelle/Pure ***definition*** command, but for ontological classes, the behavior is similar to a mixture of the record package commands and the ***definition*** command.

The implementation relies on the separation of parsing and checking in Isabelle: a bit like the markups support in the IDE described in subsection 2.9.3, parsed terms are carried through the code until they have to be checked. A subsidiary goal was to extend the support of term anti-quotations inside default values of ontological class declarations.

A polymorphic ontological class definition intertwines the covering of distinct parameters with the unification process for default values. For example, with this class definition:

```
Isabelle
doc_class 'a::one relation_one =
  rel :: 'a <= 1
```

First, the covering is checked: the *'a* parameter declared with the class *one* is covered by the type of the *relation_one.rel* attribute. As the type of the attribute is specified without a class, then the type inference will infer the most general type *'a::one* for the type of the *relation_one.rel* attribute.

Then the type unification process will infer the type *'a::one* for the default value *1::'a* of the *relation_one.rel* attribute: the term *1::'a* is a constant of type *'a::one* and this type is consistent with the type *'a::one* inferred for the *relation_one.rel* attribute.

To check the validity of the default values types, a pattern involving type unification over *prop* terms is used. First the default term-value is parsed to generate the type constraints, then term anti-quotations in this parsed term are elaborated. Next, a *prop* parsed term is generated as a meta-equivalence (an Isabelle/Pure equivalence) over the name of the attribute lifted as a free variable term and the parsed term, and finally the *prop* term is checked to trigger the type unification process. For the type unification process to fully type the term using type inference, the *prop* is first typed with dummy types and then its type is inferred from the context updated with the type of the attribute and the type constraints in the default term value using the internal checking mechanism. So the parsed *prop* becomes a fully typed and checked term. If the checking is successful, then the default value is considered well typed.

In the end, we obtain well types default values and dynamic checking with errors triggered and reported in the IDE.

The following declaration:

```
Isabelle
doc_class 'a::one relation_one =
  rel :: 'a::zero <= 1
```

will fail and trigger an error because the sort constraint of the *relation_one.rel* attribute type, consisting of only the *zero* class, is inconsistent with the parameter *'a::one* constraint, consisting of only the *one* class.

The following declaration:

```
doc_class 'a relation_one =                                         Isabelle
  rel :: 'a <= 1::int
```

will also fail. The default value of the *relation_one.rel* attribute specifies a type constraint for the *1::'a* constant: it must be of type *int*. Internally, the default value is parsed giving the term (*_type_constraint_::int⇒int*) *1::_*. The *_type_constraint_* function is generated during the parsing and is used to enforce the type constraint over the *1::'a* constant, as specified in the default value *1*. Then the following *prop* term is generated: *rel::_* ≡ (*_type_constraint_::int⇒int*) *1::_* or its curryfied variant (≡) *rel::_* ((*_type_constraint_::int⇒int*) *1::_*). The *relation_one.rel* attribute is then added to the context as a term variable typed with the type specified in the class declaration, i. e., the type variable *'a*, and the type unification process tries to infer the type of the *prop* term. It finds in the context the *relation_one.rel* free term variable with a fixed type *'a* and infer the type *'a* ⇒ *prop* for the term (≡) *relation_one.rel*. It will also infer the term *1* from the term (*_type_constraint_::int⇒int*) *1::_*. Then it tries to infer the type of the full term but fails because the argument of (≡) *relation_one.rel*, the term *1*, is not of type *'a* and an error is triggered and reported in the IDE. The declaration must be updated for the type unification to succeed:

```
doc_class relation_one =                                            Isabelle
  rel :: int <= 1::int
```

Here, the parameter becomes useless as the *relation_one.rel* attribute type is now ground, so the covering is not necessary.

The type unification over *prop* term pattern is used whenever a term in Isabelle/DOF commands needs to be checked and well typed.

Regarding the editing environment, the polymorphic ontological instance definition is simpler. We saw in section 3.2 that when an ontologial class instance is declared, a default record value is generated with only term variables as attributes. This record value checking also uses the *prop* term pattern: the default record value constructed using the *make* constant is only parsed and a meta-equivalence term is constructed using the record value. The meta-equivalence is checked in a context updated with the type of the class. If the checking succeeds, the default record value is extracted from the *prop* term giving a fully typed and checked default record value.

For the following instance:

Example 7 *Declaration of an instance that specifies its type*

```
text*[one_inst::int relation_one]‹›                                 Isabelle
```

The default record value of the *one_inst* instance will be:

⦇ *one_rel_Attribute_Not_Initialized* :: *int* ⦈

The term variable *one_rel_Attribute_Not_Initialized* was typed as an *int*. As a default value is specified for the *relation_one.rel* attribute in the *relation_one* class, it is used to update the record value of the *one_inst* instance. Like in the **definition** command, the type declaration in the **text∗** command is a type specification for the class instance: the *int relation_one* is used when updating the instance record value with the default value. The default value of the *relation_one.rel* attribute is the constant *1*::*'a* that is considered by Isabelle to also denotes the integer *1*, so the type unification will infer the record value:

⦇ *1* :: *int* ⦈

The record value has a fixed type that is used to infer the type of the record value updated with the parsed terms obtained from the default values declared in the class, the attributes specified in the instance declaration using Isabelle/DOF commands or with the **update_instance∗** command.

The instance:

```
Isabelle
text∗[one_inst2::int relation_one, rel = 2]‹›
```

is well typed as the value *2*::*'a* of type *'a* can be used as a *int*, the type declared in the type specification *int relation_one* of the *one_inst2* instance.

Keeping the support of term anti-quotations in polymorphic Isabelle/DOF commands was first thought to be a simple task. Term anti-quotations of meta-types like *thm* reference fully-typed constant representing symbolic references in the standard configuration of Isabelle/DOF. At first glance, having term anti-quotations with ground types seems like a good behavior to integrate them with polymorphic ontological classes, because the type of a term anti-quotation will help the dynamic checking in the IDE.

For example:

```
Isabelle
doc_class 'a::one relation_thm =
  rel :: 'a <= @{thm ‹refl›}
```

The declaration of the *relation_thm* fails and the type unification process triggers an error explaining that the type *thm* of the default value is not compatible with the default type *'a* of the *relation_one.rel* attribute.

Term anti-quotations were first thought as references to be used inside commands content and specification: so they were supposed to reference checkable and ratable elements. For example the evaluation of the *relation_one.rel* attribute of the *one_inst* instance in Example 7:

```
Isabelle
value*‹rel @{relation_one ‹one_inst›}›
```

will return the *int 1*.

But with the support of parametric polymorphism, term anti-quotations can also be references to ontological class instances that have attributes whose types are polymorphic.

So the element referenced by the term anti-quotation should take part in the type unification process over *prop* pattern when a term anti-quotation is used inside Isabelle/DOF commands specification.

For example with these declarations:

```
Isabelle
doc_class 'a embed_int =
embedded :: 'a

text*[embed_inst::'a::numeral embed_int, embedded = 4]‹›

doc_class 'a relation_one_ant =
  rel :: 'a <= embedded @{embed_int ‹embed_inst›}
```

the term anti-quotation @{*embed_int ''embed_inst''*} is a reference to the *embed_inst* instance and the default value of *relation_one_ant* class should be reduced to the term *4::'a::numeral*. Also the type of the *relation_one.rel* attribute is *'a::type* where *type* is the most general class assigned by default to type variable. With the current implementation of term anti-quotations, the type unification fails. Indeed, the term anti-quotation references a record value whose type is *'a embed_int*, so still a polymorphic type, but its type variable *'a* is fixed. This allows the checking and the evaluation of λ-terms with term anti-quotations with **term*** and **value*** for example, but for the polymorphism this fixed type variables will conflict with other fixed type variables added to the context for the type unification process. For the *relation_one_ant* class declaration, the fixed type variable *'a::numeral* of the term @{*embed_int ''embed_inst''*} will conflict with the fixed type variable *'a::type* of the *relation_one.rel* attribute.

Term anti-quotations should behave like constants declared with the **definition** command. We saw in section 2.4 that a **definition** command generates a constant and a theorem. Types variables of this theorem are generalized as schematic variables, allowing the type unification to instantiate the term of the definition with the right type. For example:

```
Isabelle
definition one_inst where one_inst ≡ 1
```

the definition *one_inst* will generate the constant *one_inst* of type *'a::one* and the theorem *one_inst_def*: *one_inst* ≡ *1::?'a::one* where the type of *1::'a* is generalized as a schematic variable. When evaluating the term:

```
value‹(λx. x::int) one_inst›                                    Isabelle
```

first the following term is generated using the type constraint: *one_inst*. Then the evaluation will substitute the constant *one_inst* with the term it references and instantiate the term by replacing its schematic variable with the new type of the constant, i. e., *int*. Finally **value** return *1*.

As we saw in section 3.2 the record value of class instances is constructed starting from a default record value with only free term variables for the record fields. But it also implicitly carries the type of each attribute that is later used by the type checking when updating the record with default values declared in the class or with attribute-values specified in the instance declaration with **text*** for example. With the support of polymorphic term anti-quotation, the record-value now plays two roles: on the one hand it has to have fixed type variables so it can be used to store well typed default values declared in the class and to construct well typed record values when declaring an instance, and on the other hand it has to have schematic variables so that when used as the elaboration of term anti-quotations appearing inside λ-terms of Isabelle/DOF commands, for example in default values of class attributes, they can take part in the type unification process when the default value is evaluated before it is stored for a later usage.

Having fully integrated polymorphic term anti-quotations implies major changes to the current implementation of Isabelle/DOF, and is not yet finalized. In the meantime, a quick workaround can be used by embedding a term anti-quotation in a definition using the Isabelle/DOF command **definition***.

Parametric polymorphic ontological classes allow to express new concepts that will integrate well with queries written using Isabelle/HOL functional programming capabilities. The relational algebra of relational databases through algebraic structures and the definition of queries share similarities with expressing the linking between informal and formal document elements and advanced search that Isabelle/DOF targets. An interesting field of research in data-base communities is the *data provenance* [93, 94] that describe origins of data and its history, i. e., the process by which it arrived in a database. Diverse algebraic structures have been proposed to express the provenance of data used in results of queries or virtual tables (cf. [95, 96]) and among them polynomials [97].

In the following we show how parametric polymorphic classes can capture the notion of provenance and how it relates to the linking in Isabelle/DOF.

4.3 Modeling “Provenance” in ODL

The provenance semiring [97] introduced the algebraic structure of semirings as a mathematical framework to cover diverse notions of provenance, such as the *Boolean* provenance [95] and the *why* provenance [93], or the notion of a security model [98].

The Boolean provenance is captured by the Boolean semiring $(\{\bot, \top\}, \bot, \top, \vee, \wedge)$ and tells if a tuple exists when a subdatabase is selected: tuples are annotated by a Boolean function over $\mathcal{B}$, the finite set of Boolean events, a function of the form: $(\mathcal{B} \rightarrow \{\bot, \top\}) \rightarrow \{\bot, \top\}$. The valuation denotes a possible world of the database.

The Boolean provenance is understood in Isabelle/DOF in the same way. The document, composed of document element-objects generated using the abstraction over command pattern, can be considered as a database and the tuples as document elements, and Boolean provenance as meta-data give information of the document elements to consider.

Defining the Boolean semiring $(\{\bot, \top\}, \bot, \top, \vee, \wedge)$ in Isabelle/DOF is straightforward by making the *bool* type an instance of the axiomatic type class *semiring*. First we instantiate the *bool* type:

Isabelle

```
instantiation bool :: semiring
begin

definition plus_boolean_def: i + j = (i ∨ j)

definition times_boolean_def: i ∗ j = (i ∧ j)

instance
  proof
  fix i j k :: bool have (i ∨ j ∨ k) = (i ∨ (j ∨ k)) by simp
  then show i + j + k = i + (j + k)
    unfolding plus_boolean_def by simp
  have (i ∨ j) = (j ∨ i) by auto
  then show i + j = j + i
    unfolding plus_boolean_def by simp
  have (i ∧ j ∧ k) = (i ∧ (j ∧ k)) by simp then show i ∗ j ∗ k = i ∗ (j ∗ k)
    unfolding times_boolean_def by simp
  have ((i ∨ j) ∧ k) = ((i ∧ k) ∨ (j ∧ k)) by auto
  then show (i + j) ∗ k = i ∗ k + j ∗ k
    unfolding plus_boolean_def times_boolean_def by simp
  have (k ∧ (i ∨ j)) = ((k ∧ i) ∨ (k ∧ j)) by auto
  then show k ∗ (i + j) = k ∗ i + k ∗ j
    unfolding plus_boolean_def times_boolean_def by simp
qed

end
```

We provide definitions for the operations (+) and (∗) to match the boolean constants (∨) and (∧) and prove the associative, commutative and distributive properties. Then we define the *0*::*'a* and *1*::*'a* of the *bool* type in the Boolean semiring where *0*::*'a* is ⊥, i. e., *False* and *1*::*'a* is ⊤, i. e., *True*:

Isabelle
```
instantiation bool :: zero
begin
definition zero_bool_def:
    0 = False
instance ..
end

instantiation bool :: one
begin
definition one_bool_def:
    1 = True
instance ..
end
```

Now we can declare a default provenance class that will embed the abstract notion of Boolean provenance and make queries on the document over some Boolean events to learn which document elements are to be considered:

Isabelle
```
doc_class 'a b_relation =
  rel :: 'a::semiring
```

Then we can extend the *paper*^{m} ontology so that a *Polymorphic_Classes.text_element* has a new *b_text_element.relation* attribute (for legibility, we prefix the new defined classes with *b_*, mimicking the Boolean provenance class definition):

Isabelle
```
doc_class ('α, 'β) b_text_element =
    relation:: 'β b_relation
    authored_by :: 'α author set  <= {}
    level       :: int  option  <= None
    invariant authors_req :: authored_by σ ≠ {}
    and       level_req   :: (level σ) ≠ None ∧ the (level σ) > 1

doc_class ('α, 'β) b_introduction = ('α, 'β::semiring) b_text_element +
    authored_by :: ('α author) set  <= UNIV

doc_class ('α, 'β) b_conclusion = ('α, 'β::semiring) b_text_element +
    resumee     :: (('α, 'β::semiring) definition set × ('α, 'β::semiring) theorem set)
    invariant is_form :: (∃x∈(fst (resumee σ)). definition.is_formal x) ⟶
                (∃y∈(snd (resumee σ)). is_formal y)
```

and we can declare some instances:

```
Isabelle
text*[relb1::bool b_relation, rel=b1]‹›

text*[relb2::bool b_relation, rel=b2]‹›

text*[relb3::bool b_relation, rel=b3]‹›

text*[ch::elsevier author, name=''Church'']‹›
text*[sc::elsevier author, name=''Scott'']‹›

text*[intro1::(elsevier, bool) b_introduction,
       relation=@{b_relation ‹relb1›}, authored_by={@{author ‹ch›}, @{author
‹sc›}}]‹›

text*[intro2::(elsevier, bool) b_introduction,
       relation=@{b_relation ‹relb2›}, authored_by={@{author ‹ch›}, @{author
‹sc›}}]‹›

text*[intro3::(elsevier, bool) b_introduction,
     relation=@{b_relation ‹relb3›}, authored_by={@{author ‹ch›}}]‹›

text*[conclu1::('a, bool) b_conclusion, relation=@{b_relation ‹relb1›}]‹›

text*[conclu2::('a, bool) b_conclusion, relation=@{b_relation ‹relb2›}]‹›
```

The *b_text_element.relation* relational attribute can be used to select between two different document content flavors: the document elements can be embedded in a monitor instance and an ML invariant can be written to select elements using the value of the *b_text_element.relation* attribute, or a query can be made over the *Polymorphic_Classes.text_element* instances to filter over the *b_text_element.relation* attribute-value:

```
Isabelle
value*‹@{instances_of ‹(elsevier, bool) b_introduction›}
       |> filter (λx. relation x = @{b_relation ‹relb1›})›
```

This way we select the *b1* version of the document. With further work a wrapper for the parser could be written that understands this type of query and generate flavors of the document targeting different requirements.

We can also know in which *Polymorphic_Classes.introduction* class instances a specific author is declared by projecting over an author name. First we define a small definition to get a pair composed of the name of the author and the list of *b_text_element.relation* attribute-values:

```
Isabelle
definition bool_proj where
  bool_proj auth_name l  = l |> filter (λx. ∃ y ∈ authored_by x. name y =
auth_name)
                              |> map (rel) |> (λx. Pair auth_name x)
```

The *bool_proj* definition is then used to filter the *Polymorphic_Classes.introduction* instances whose *sec_text_element.authored_by* attribute contains the author *''Scott''*:

```
Isabelle
value*‹@{instances_of ‹(elsevier, bool) b_introduction›}
     |> bool_proj ''Scott''›
```

The query returns only the two *b_relation* record value of the instances *Polymorphic_Classes.intro1* and *Polymorphic_Classes.intro2* as *Polymorphic_Classes.intro3* does not declare the *sc* author in its *sec_text_element.authored_by* attribute.

So one can choose between *Polymorphic_Classes.intro1* and *Polymorphic_Classes.intro2* to select the introduction he wants for his document to have *''Scott''* as an author.

The polymorphic type of the *relation_one.rel* attribute of the *b_relation* class can be shaped using algebraic structures to capture other semantics, similarly to the provenance semiring that uses polynomials to capture the notion of *how provenance* [97]. The how provenance explains how a query result has been computed.

To capture an equivalent notion of how provenance in Isabelle/DOF, we also use polynomials, more precisely multivariate polynomials. Recall that a multivariate polynomial structure $(A,X,*,+)$ over a set of coefficients A and a set of indeterminates X forms wrt. the multiplicative and additive operations a polynomial semiring. As a consequence, polynomials such as:

$$P(x) = a_n * x^n + ... + a_1 * x^1 + a_0$$

or even more generally:

$$P'(x_1...x_m) = \sum a_{i_1 i_2 ... i_m} x_1^{i_1} x_2^{i_2} ... x_m^{i_m}$$

can be represented in this structure and will have a normal form which permits their comparison via partial orders. Moreover, multivariate polynomials are substitutive, i. e. an equality like $x_1 = P(x_2)$ can be used to eliminate the variable x_1 in P' and to compute again a normal form. A set of such equalities allows therefore to reduce multivariate polynomials to one based only on a subset of base multivariates.

In our context, we will mostly use $\mathbb{Z}$ or $\mathbb{N}$ for the coefficients A; the indeterminates can be interpreted by instances id’s or arbitrary labels used to define groups over them. An interpretation of the coefficients $\mathbb{B}$ and the $(*)$-operator by the logical conjunction $(\wedge)$ and the $(+)$ by the disjunction $(\vee)$ leads to a collapse of the polynomials to disjunctive normal-forms (DNF) which can be easily interpreted by "this concept depends on the concept (x_1 and x_2) or x_3", for example. More general interpretations than the latter allow for expressing weights on the these dependencies. Isabelle/DOF support of parametric polymorphism opens a new path to use multivariate polynomials to express the linking between a document element and others where the indeterminates $x_1^{i_1}, x_2^{i_2}, \ldots x_m^{i_m}$ will capture some kind of document element references, the coefficients $a_{i_1 i_2 \ldots i_m}$ the "linking flavor" of each monomial, and the exponents $i_1, i_2, \ldots i_m$ will capture a specific "weight" of a reference (an indeterminate) within each monomial.

By leaving open the concrete computational structure for a multivariate polynomial $(A,X,*,+)$ at the moment of the creation of a document element, we can express various forms of the linking within a single class attribute and postpone the decision of the concrete computational structure at a later stage, for example, at the point where a concrete query is formulated.

This constitutes a new form of abstract representation of dependency using an axiomatic type class, possibly reinforced by class invariants. Using parametric polymorphism, we construct an executable type for multivariate polynomial structures $(A,X,*,+)$:

$$('\nu\text{::}\textit{linorder},\ '\alpha\text{::}\textit{semiring})\ \textit{mpoly}$$

where $'\nu$ corresponds to the set of multivariates (assumed to be orderable for reasons of normal-form computations) and where $'\alpha$ corresponds to the coefficients. On a true *semiring* structure this type captures precisely the notion of how provenance and tracks how document elements relate to each other along the declaration of the instances. Note that this is a significantly more fine-grained approach than using invariants over instance relations captured by the product type as with the *resumee* attribute of the *conclusion* class constrained by the *is_form* invariant (cf. Figure 2.5).

The executable multivariate polynomials theory in the AFP is not directly usable as it does not define multivariate polynomials as a type, so they can not be constrained by an axiomatic type class. But we will use it indirectly to make our polynomials executable. For that we follow the data refinement principle [99]: an abstract type is replaced by a more concrete one in the generated code, to make abstract types efficiently executable. The *mpoly* type is specified as a subtype over the concrete *list* type using an invariant in a ***typedef*** command:

```
Isabelle
typedef (overloaded) ('v,'a) mpoly =
  {p. (mpoly_inv::(('v::linorder × nat) list × 'a::zero) list ⇒ bool) p}
  by (rule exI[of _ Nil], auto simp: mpoly_inv_def)
```

The *mpoly_inv* invariant is declared using the **definition** command:

```
Isabelle
definition mpoly_inv :: (('v::linorder × nat) list × 'a::zero) list ⇒ bool
  where mpoly_inv p ≡ (∀ c ∈ snd ‘ set p. c ≠ 0) ∧ distinct (map fst p)
```

It checks that the coefficients are not null and that each element in a monomial is unique.

The representation, i. e., the concrete type of the polynomial is similar to the representation of polynomials in the AFP theory. This way we can use the Lifting and Transfer packages [100] to automate the construction of the abstract type *mpoly*. “The Lifting package defines new constants on the abstract level, which is done by lifting terms from the concrete level to the abstract level” [99] and “the Transfer package helps to prove theorems on the abstract level (mainly properties of the lifted constants), which is done by transferring the goals on the abstract level to goals on the concrete level.” [99]

When making the abstract type *mpoly* an instance of the axiomatic type class *semiring*, we will be able to define new constant definitions for the abstract type, lift them to the concrete type *list* and prove properties by reusing definitions specified in the polynomials AFP theory.

First, the lifting infrastructure for *mpoly* is set up:

```
Isabelle
setup_lifting type_definition_mpoly
```

To facilitate the evaluation, we consider the abstraction function *var_mpoly* of type $(('\nu{::}linorder \times nat)\ list \times '\alpha{::}semiring\_0)\ list \Rightarrow ('\nu{::}linorder, '\alpha{::}semiring\_0)\ mpoly$ which converts a list of indeterminates indexed by their exponents and pondered by a coefficient into our type-constructor providing an efficient representation for multivariate polynomials $('\nu,'\alpha)\ mpoly$. The *var_mpoly* constant, considered a pseudo-constructor, is defined by lifting its definition using the **lift_definition** command:

```
Isabelle
lift_definition var_mpoly :: (('v::linorder × nat) list × 'a::zero) list ⇒ ('v, 'a)
mpoly
  is λ p. if mpoly_inv p then p else []
  by (auto simp: mpoly_inv_def)
```

This will define the new constant *var_mpoly* with the abstract type $(('v \times nat)\ list \times 'a)\ list \Rightarrow ('v, 'a)\ mpoly$ using a corresponding operation on the representation type specified by the term $\lambda p.\ if\ mpoly\_inv\ p\ then\ p\ else\ []$. *var_mpoly* will be used in pattern matching on the left-hand side of theorems.

Then the code generator is set up to view the constant as a constructor with the command:

```
code_datatype var_mpoly                                          Isabelle
```

Now we can instantiate the *mpoly* as a semiring:

```
instantiation mpoly :: (type, type) semiring                     Isabelle
```

We are asked to give specifications for the two operations of the semiring (+) and (∗). (+) uses a function *Polymorphic_Classes.poly_add* in its corresponding operation and is then lifted to the concrete type.

```
fun poly_add ::                                                  Isabelle
 (('v::linorder × nat) list × 'a:: semiring_0) list ⇒
  (('v::linorder × nat) list × 'a) list ⇒
  (('v::linorder × nat) list × 'a) list where
  poly_add [] q = q
| poly_add ((m,c) # p) q = (case List.extract (λ mc. fst mc = m) q of
    None ⇒ (m,c) # poly_add p q
  | Some (q1,(_,d),q2) ⇒ if (c+d = 0) then poly_add p (q1 @ q2) else (m,c+d)
# poly_add p (q1 @ q2))

lift_definition plus_mpoly:: ('v::linorder, 'u::semiring_0)mpoly ⇒
('v,'u)mpoly ⇒ ('v,'u)mpoly
  is λ p q. poly_add p q
```

For (+) we define a function *poly_times* which uses the *monom_mult_list* function defined in the AFP theory that multiply two monomials:

```
fun poly_times ::                                                Isabelle
 (('v::linorder × nat) list × 'a:: semiring_0) list ⇒
  (('v::linorder × nat) list × 'a) list ⇒
  (('v::linorder × nat) list × 'a) list where
  poly_times [] q = []
| poly_times p  [] = p
| poly_times ((m,c) # p) q =
    (let mon_list = q |> map (λ x. let (m1, c1) = x ;
                                   mon = monom_mult_list m m1 ;
                                   coef = c ∗ c1
                                in (mon, coef))
     in mon_list @ (poly_times p q))

lift_definition times_mpoly:: ('v::linorder, 'u::semiring_0)mpoly ⇒
('v,'u)mpoly ⇒ ('v,'u)mpoly
  is λ p q. poly_times p q
```

The new definitions of (+) and (∗) for *mpoly* as a type instance of a *semiring* axiomatic type class are then used to prove alternative code equations using pattern matching on our pseudo-constructor *var_mpoly*:

```
Isabelle
lemma plus_mpoly_code [code]:
    var_mpoly x + var_mpoly y = var_mpoly(poly_add x y)
lemma times_mpoly_code [code]:
    var_mpoly x ∗ var_mpoly y = var_mpoly(poly_times x y)
```

These lemmas will be used instead of the original definition of the operations when evaluating terms. For example, *var_mpoly x* + *var_mpoly y* will be translated to *var_mpoly* (*Polymorphic_Classes.poly_add x y*) when using diagnostic commands like `value∗`.

Now we zoom into our $paper^m$ example and present an alternative specification of the *definition* class:

```
Isabelle
doc_class 'α relation =
  rel :: 'α::semiring

doc_class ('α, 'β) text_element = 'β::semiring relation +
    authored_by :: 'α author set  <= {}
    level       :: int  option  <= None
    invariant authors_req :: authored_by σ ≠ {}
    and       level_req   :: (level σ) ≠ None ∧ the (level σ) > 1

doc_class ('α, 'β) technical = ('α, 'β::semiring) text_element +
  id :: nat
  formal_results  :: thm list

doc_class ('α, 'β) definition = ('α, 'β::semiring) technical +
    is_formal   :: bool
```

The *definition* class inherits the parametric polymorphic *rel* attribute from the *relation* class.

It is now possible to specify relations between *definition* instances and other document elements:

```
Isabelle
text*[def1::('α, (nat, int) mpoly) definition,
    rel =      var_mpoly [([((id @{theorem ‹safety›}), 1)], 1::int)]
             * var_mpoly [([((id @{theorem ‹security›}), 1)], 1::int)]
           +   var_mpoly [([((id @{result ‹proof1›}), 1)], 1::int)]
             * var_mpoly [([((id @{result ‹proof2›}), 1)], 1::int)]]‹... text ...›

value*‹rel @{definition ‹def1›}›
```

... text ...

```
term‹poly_of (PMult [PVar (2::int), PNum (1::int)])›
value‹Polynomials.poly_add (poly_of (PMult [PVar (2::int), PNum (1::int)])) (poly_of
(PMult [PVar (2::int), PNum (1::int)]))›
value*‹relation.rel @{definition ‹def444›}›
```

In our example, the indeterminates of the polynomial will be instances of theorem identifiers. We use *nat* for the identifiers and *int* for the coefficients, so the type of the *def1* instance we just declared is *('α, (nat, int) mpoly) definition*. *id @{theorem ''safety''}* refers to the *safety theorem* instance identifier. The evaluation of the *def1* instance *rel* attribute gives:

var_mpoly [([(123, 1), (456, 1)], 1), ([(789, 1), (987, 1)], 1)]

The value can be understood as follows: the annotated text will depend on the *Polymorphic_Classes.safety* instance *and* the *Polymorphic_Classes.security* instance *or* the *Polymorphic_Classes.proof1* instance *and* the *Polymorphic_Classes.proof2* instance, where *and* is expressed using the $(*)$ operator and *or* using the $(+)$ operator of the *semiring* when declaring the *rel* attribute-value of *def1*. Intuitively, the *or* is justified by the wish to offer consistently either a more abstract or a more concrete (proof-object based) representation of the dependence on other document elements.

As *Polymorphic_Classes.proof1* and *Polymorphic_Classes.proof2* are *result* class instances, they also inherit the *rel* attribute, and could also have specified relations that should be checked as well. For a certification purpose, this process could be automated and we are sure that the checking process will end due to the directed acyclic graph architecture of the document and the ordered declaration of the instances enforced by the parsing process. The relation can always be specified independently from being a formal or informal document element, where the checking of informal document elements has to be delegated to humans during the certification validation process.

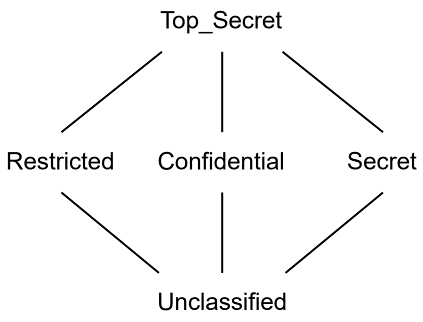


Figure 4.2: The Lattice of Security Labels.

4.4 An Access-Control Model for Integrated Isabelle/HOL Documents

One might object that the suggested *integrated* document model underlying our approach is incompatible with the reality of industrial projects, where their partners will need to enforce strategies to protect their intellectual property. However, a system having access to the integrated document is fundamental for mechanisms to ensure global consistency. A way out of this dilemma are fine-grained access control models allowing decisions on individual text elements for any user role. In this section, we show how such a fine-grained security model can be built with the existing mechanisms of ODL. [1]

Parametric polymorphism opens ways to implement security and integrity models based on lattices, such as Bell-LaPadula-like access control models. The key is to provide a *sec* attribute that is to define the basic access control status. This can also be useful when targeting certifications where roles and responsibilities of the involved entities (person, group, organization) are identified.

The *sec* attribute is implemented using a *lattice* class and declares the security label of the document element. The document elements *sec* attributes can be computed to express the minimum security level required to access a collection of document elements associated with each others, whether it is by the *is—a* relation of the class inheritance or by other means, like the how provenance just presented, or by looking at the instance attributes. First we declare the security labels using a datatype:

[1]Note that our current front-end Isabelle/jEdit does not support access-restriction functionality; however, we consider this as a current technical limitation.

```
Isabelle
datatype security = Unclassified | Restricted | Confidential
                  | Secret | Top_Secret
```

Then we make the *security* datatype a *lattice* class instance using *max_sec* and *min_sec* that define the order relation of the datatype constructors:

```
Isabelle
definition min_sec where
min_sec a b ≡ (if a = b then a
               else if a = Top_Secret then b
               else if b = Top_Secret then a
               else Unclassified)

definition max_sec where
max_sec a b ≡ (if a = b then a
               else if a = Unclassified then b
               else if b = Unclassified then a
               else Top_Secret)

instantiation security :: lattice
begin

definition inf_security where inf i j ≡ min_sec i j

definition sup_security where sup i j ≡ max_sec i j

definition less_eq_security where i ≤ j ⟷ max_sec i j = j

definition less_security where (i::security) < j ⟷ i ≤ j ∧ i ≠ j

instance
...
```

The *0::'a* of the semiring is defined as *Top_Secret* by making the security datatype an instance of the *0::'a* class:

```
Isabelle
instantiation security :: zero
begin

definition zero_security_def:
   0 = Top_Secret
...
```

and the *1::'a* is defined as *Unclassified*, reflecting the additive and multiplicative identities of the lattice.

The resulting lattice is shown in Figure 4.2 where security labels are ordered from less sensitive (*Unclassified*) to most sensitive (*Top_Secret*). Finally, we add syntactic definitions for the $(+)$ and $(*)$ operators.

```
Isabelle
instantiation security :: plus
begin
definition plus_security where (i::security) + j = min_sec i j
instance ..
end

instantiation security :: times
begin
definition times_security where (i::security) * j = max_sec i j
instance ..
end
```

The $(*)$ operator can be understood as read access where document users may read document elements only at or below their own security level, while the $(+)$ operator gives write access on document elements for users with a higher security level.

The new specification of the *result* ontological class, *sec_result*, now has a *sec* attribute inherited from *sec_relation*:

```
Isabelle
doc_class ('a, 'b) sec_relation =
  rel :: 'a::semiring
  sec :: 'b::lattice

doc_class ('α, 'β, 'γ) sec_text_element = ('β::semiring, 'γ::lattice) sec_relation
+
    authored_by :: 'α author set  <= {}
    level       :: int  option  <= None
    invariant authors_req :: authored_by σ ≠ {}
    and       level_req   :: (level σ) ≠ None ∧ the (level σ) > 1

doc_class   ('α,   'β,   'γ)   sec_technical   =   ('α,   'β::semiring,   'γ::lattice)
sec_text_element +
  id :: nat
  formal_results  :: thm list

doc_class ('α, 'β, 'γ) sec_result = ('α, 'β::semiring, 'γ::lattice) sec_technical +
  evidence :: sec_kind
  property :: ('α, 'β, 'γ) sec_theorem list <= []
  invariant has_property :: evidence σ = proof ⟷ property σ ≠ []
```

In this setting, we can specify access control as follows:

```
Isabelle
text*[proof3::('α, (nat, int) mpoly, security) sec_result,
      id = 789,
      evidence = proof,
      property=[@{sec_theorem ‹safety2›}, @{sec_theorem ‹security2›}],
      level = Some 2, authored_by = {@{author ‹church2›}},
      sec = Unclassified]‹... text›
```

The *proof3* instance *sec* attribute-value is *Unclassified*, and its *property* attribute is a list of *sec_theorems* that also have a security level. We can have a coarse grain policy where access control to *proof3* will use only its *sec* attribute-value. Then every user with a security level above *Unclassified* will be able to read all *proof3* information, including the information of every entry in its *property* attribute. With a more fine grain policy, the *sec* attribute-value of each *property* list element will also be checked. The *sec* attribute-value of the *security2* instance is *Confidential*, so a user with an *Unclassified* security level should be able to access *proof3* information but not *security2* information when querying *proof3 property* attribute. With the following query:

```
Isabelle
value*‹property @{sec_result ‹proof3›}
      |> filter (λx. statement x = statement @{sec_theorem ‹security2›})›
```

the access control checking mechanism should compute the involved instances security level using the (∗) operator for read access:

```
Isabelle

value*‹sec @{sec_result ‹proof3›} ∗ sec @{sec_theorem ‹security2›}›
```

which evaluates to *Confidential*. Whether the access control is coarse or fine grain, a user with an *Unclassified* security level will not be able to make this query because he can not access *security2* information.

4.5 Conclusion

Parametric polymorphic classes are used to extend the expressiveness in Isabelle/DOF. By separating the parsing and checking mechanisms, the declaration of default values in **doc_class** and **onto_class** commands of ODL can be integrated in the Isabelle/PIDE environment and allow the declaration of type constraints and type specifications in the usual manner. The Isabelle/DOF theory of meta-data extended with term anti-quotations is used to prove constraints, enforced through axiomatic type classes. These constraints apply to extensions of the linking semantics between document elements. Extensions are possible using axiomatic type classes only: the security model for the integrated document model relies only on the *lattice* class defined in Isabelle/HOL *HOL.Lattices* theory. To express the dependence between document elements, similar to the provenance concept, polymorphic attribute types are refined by algebraic structures which are constrained by other algebraic structures represented by classes. A fine-grained dependence can be expressed using polynomial semirings where weights can be given to each document element depending on others, and multiple linking flavors can be expressed in one single attribute-value. In both cases, parametric polymorphism lets the choice to the user: he can choose the security model he wishes by specifying another axiomatic type class, and he can change the meaning of the linking by choosing the refinement he desires. By polymorphism the linking is becoming parametric and thus more flexible: we do not need to specify concepts taking part in the linking using their internal structure, but only their relations. So the linking could serve as an element to differentiate ontology patterns: the difference will be specified by declaring properties on the linking itself. For example to specify a pattern that allows to define a link between two document elements, we could define morphism definitions and projection functions and obtain an equivalent to the *prod* product type in Isabelle/HOL.

Chapter 5

Deep Isabelle/DOF

5.1 Introduction

The straightforward way to reason about a logic is by specifying a metalogic. With an expressive logic, objects which are defined can be represented directly within this logic, hence giving a metalogic with the same logical language. Typically, with typed logics, logical objects will be represented using deep embedding and specification constructs to interact with the development environment. Depending on the goal of the metalogic and the typed logic itself, the metalogic might define the abstract objects of the logic like types and terms and it might propose meta specification constructs to offer a full meta-programming environment [81]. Nipkow et al. [13] formalize a metalogic for Isabelle/HOL in HOL, using deep embedding for types, terms, and proofs and shallow embedding for theories. The newly defined objects using deep embedding are parts of a theory: they exist as logical objects within a theory and can therefore become inhabitants of meta-level datatypes as specified by Isabelle/DOF and change the semantics of the meta-level anti-quotations. It is now possible to not only check these anti-quotations but also to elaborate a type, a term or a theorem to its representation in the logic using the formalized metalogic. For proof assistants, "each system comes with its own logical formalism, its own mathematical language, its own proof language and proof format, its own libraries."[101] So the same principles are used to translate abstract objects from a formalism to another, with the help of shallow embedding in addition to deep embedding [101]. The mixed ideas of the translation between formalisms, meta-level objects and the possibility to reference them, and metalogic within the logic lead to the consideration of proposing a methodology to help in the translation of theorems between theorems provers by using an ontology.

Theorems declared in proof assistants that implement HOL as the underlying logic generally do not need to store the proofs objects, following the LCF approach, as the only way to construct objects of the abstract type *thm* is by using primitive inference rules. [17] added proof objects generation to Isabelle/HOL by using an abstract ML type *proof*, providing explicit proof objects, represented using λ-terms in the spirit of the Curry-Howard isomorphism. The main drawback is that it leads to enormous proof objects for realistic applications consisting of primitive inferences rules linked by the reasoning mechanisms expressed in the *proof* inductive datatype. So [17] also comes with compression techniques where proofs objects of theorems used to prove a proposition can be discharged, using elaborated proof reconstruction mechanisms, and leading to proof objects of reasonable size. But proofs assistants also use tactics (methods in Isabelle/HOL lingua) to write proofs, and generally provide to the user a tactic language (for instance [102, 103] for Coq and [104] for Dedukti) to extend the body of tactics by defining his own tactics. For Isabelle/HOL, Eisbach [105] is such a language, "more pricely an infrastructure for defining new proof methods out of existing ones". [106]. Information on the tactics used in proof scripts (recall the difference between proof scripts and proof objects explained in section 2.2) is mostly lost when the proof object is reconstructed, at least with Isabelle/HOL, as it consists only of primitive inference rules. The specific structure of the *proof* inductive type and the specific structure of the proof object part corresponding to a specific method give hints to translate proof scripts from one formalism to another formalism, using pattern matching on the structure of the proof object. But even though, information on specific methods defined using the tactic language will be lost. Translating proof objects from one formalism to another allows to reuse already proven theorems as facts, but proof scripts reconstruction seems difficult to achieve with the loss of information. To facilitate sharing between proof assistants, Paulson et al. [14] advocates for a common language "offering portability of proofs exactly as today's programming languages offer portability between different machine architectures." But they also stress the difference of meaning between common existing instructions in the implementation of the Lean project [107] compared to the Isabelle Isar language. The keyword here is *meaning*. Even with a common language, its seems difficult to minimize explicit references to the underlying logical language and framework when proving theorems, if we compare for example subtype definitions in HOL where type are inhabited to a dependent type theory where type definitions can depend on terms. Proof systems offer declarations that might be specific to a system but accommodate well the proof development of this system. And what about the tactic language that also might use code specific to a proof assistant, using for example low level access to the underlying implementation language like ML? Which meaning should be given to user defined proof methods in Eisbach when translating them to Lean? Is this even possible? Ontologies might be an answer: a reference domain ontology could be defined which specify the semantics of the proof language instructions and the proof tactics. It will be formal, consensual and have the capability to be referenced, so that its formalized knowledge is shared and usable by each proof assistant. It also implies that each proof assistant should implement a thin layer that proposes the formalization of this knowledge in its logical language, i. e. an ontology language.

In this chapter, we first explain how proof objects are reconstructed from theorem abstract objects. This will help in understanding why and how we can reify proof objects, a process that is illustrated in the next section, where we use a metalogic of Isabelle/HOL to formalize Isabelle/DOF meta types and meta-level term anti-quotations in HOL: anti-quotations are now inhabited by reified objects allowing a definitional equality evaluation. Then we expose the methodology that we use to recreate the proof script structure: reified meta-data is used as an object to shape the structure of the proof object and meta-data as information is used to help in the proof object reconstruction by referring to a domain ontology. It might be relevant for import/export techniques of proofs scripts between theorem provers.

5.2 Proof Objects in Isabelle

Berghofer, in [17], added LCF style proof objects to Isabelle core, and an ML level API to manipulate and reconstruct proof objects from abstract ML type *thms*. The reconstruction consists in retrieving proof information of a theorem abstract object, implemented as an ML datatype. A theorem of type *thm* represents a proven proposition: a simplified version of the datatype is:

ML

```
datatype thm = Thm of
 deriv *                    (*derivation*)
 {cert: Context.certificate, (*background theory certificate*)
  ...
  prop: term}               (*conclusion*)
and deriv = Deriv of
 {...
  body: Proofterm.proof_body}
```

The second part of the datatype is composed of a proposition that is recorded as a term (the *prop* attribute of the record) and associated with an immutable certificate of the background theory, i. e. the logical context that is required for formulating statements and composing proofs. The certificate is used internally to generate or extract certified types and terms, that is abstract datatypes “that guarantee that their values have passed the full well-formedness (and well-typedness) checks, relative to the declarations of type constructors, constants etc. in the background theory." [21] This mechanism allows to use the same mechanism to generate new theorems reusing other theorems or new definitions. Indeed, we saw in section 2.4 that the Isabelle/HOL ***definition*** command defines a constant and its associated theorem. So a definition declaration can follow the conservative extension approach and derives its theorem from a bootstrapped axiomatized version. The mechanism to declare a new definition theorem is encoded as ML functions that are added as *operations* to a local theory abstract datatype. A local theory is merely an alias for a proof context: this is one of the reasons why we must deal with proof contexts when implementing the abstract over command pattern for Isabelle/DOF commands like ***definition****∗* (see subsection 2.9.1 and section 3.2), as we want to reuse the internal mechanisms of Isabelle/HOL. The operations happen outside of the inference kernel but use internally the theorem proving ML API to convert the proposition term into a theorem of type *thm*, by enforcing the term to pass all the checking mechanisms. Schematically, the theorem certificate is used to get the theory abstract datatype that represent the logical context, then the theory is switched to a local theory. When the local theory is initialized the operations are added as ML functions to the local theory datatype and then are used to define the constant and theorem definition.

The first part of the theorem datatype is the derivation where we can find the body attribute whose type is *proof_body*. We can see here how proof terms are intertwined with theorems. The *proof_body* is a datatype that is composed of the proof object of the theorem of type *proof* and an ordered list of the theorems involved in the proof with the following simplified version:

ML
```
datatype proof_body = PBody of
  {oracles: ((string * Position.T) * term option) Ord_List.T,
   thms: (serial * thm_node) Ord_List.T,
   proof: proof}
```

And the simplified version of the *proof* datatype is defined as:

ML
```
datatype proof =
   MinProof
 | PBound of int
 | Abst of string * typ option * proof
 | AbsP of string * term option * proof
 | op % of proof * term option
 | op %% of proof * proof
 | PAxm of string * term * typ list option
 | PThm of thm_header * thm_body
```

where *Abst* implements the abstraction over terms, % the application of propositions to terms, where *AbsP* implements the abstraction over proofs, and %% the application of propositions to propositions (see section 2.3). *PAxm* represents axioms as proof objects and *PThm* theorems. So schematically, a proof term object of a theorem is a *PThm* object that embeds a *proof_body*, using the *thm_body* datatype (not shown), with a list of theorems that are linked using the *proof* datatype constructors. And theorems within the list are representations of theorems of type *thm*, using the *thm_node* datatype (not shown), similar to the *thm* datatype. Once again the *prop* terms of the theorems pass all the checking mechanisms. The reconstruction mechanism of proof objects extracts theorems objects from the *proof_body* of a *thm* datatype and represents them within proof term objects as *thm_node*s.

This little dive into the implementation helps to understand why we can safely use the reconstruction mechanism to reify proof objects.

We can now use the ML API to reconstruct the proof object of a theorem encoded in ML. For example the proposition:

$[\![A;\ B]\!] \Longrightarrow B \wedge A$

means that if we have a proof for A and a proof for B (using the meta-level implication $(\Longrightarrow)$), then we have a proof for $B \wedge A$. A proof that uses the conjunction introduction lemma $[\![?P;\ ?Q]\!] \Longrightarrow\ ?P \wedge\ ?Q$ could yield the proof object:

$\lambda(H\colon ?A)\ Ha\colon ?B.\ conjI \cdot ?B \cdot ?A \bullet Ha \bullet H$

by reusing the notation of the section 2.3 for proof terms, the notation of the section 4.2 for schematic variables, and where the proof term *conjI* that denotes the proof of the conjunction introduction theorem is applied to the schematic terms variables *?A* and *?B* considered as propositions, and then to the proofs *Ha* and *H* of these terms, reflecting the $(\Longrightarrow)$ applications. A more realistic example would involves an Isabelle/HOL proof method, giving the proof script:

```
Isabelle
lemma rconjI: ‹A ⟹ B ⟹ B ∧ A›
  by simp
```

Here the *simp* method is used to call the simplifier. The proof object of *rconjI* is slightly more complex:

```
λ(H: ?A) Ha: ?B.
 equal_elim · True · ?B ∧ ?A •
  (symmetric · TYPE(prop) · ?B ∧ ?A · True •
    (combination · TYPE(bool) · TYPE(prop) · Trueprop · Trueprop · ?B ∧ ?A · True
• (reflexive · TYPE(bool ⇒ prop) · Trueprop) •
      (transitive · TYPE(bool) · ?B ∧ ?A · True ∧ True · True •
          (combination · TYPE(bool) · TYPE(bool) · (∧) ?B · (∧) True · ?A · True •
(combination · TYPE(bool) · TYPE(bool ⇒ bool) · (∧) · (∧) · ?B · True • (reflexive ·
TYPE(bool ⇒ bool ⇒ bool) · (∧)) • (Eq_TrueI · ?B • Ha)) •
          (Eq_TrueI · ?A • H)) •
              (eq_reflection · TYPE(bool) · True ∧ True · True • arity_type_bool •
(simp_thms_25 · True))))) •
   (equal_elim · True · (?B ⟹ True) • (symmetric · TYPE(prop) · (?B ⟹ True) ·
True • (implies_True_equals · ?B)) •
     (equal_elim · True · (?A ⟹ True) • (symmetric · TYPE(prop) · (?A ⟹ True) ·
True • (implies_True_equals · ?A)) • TrueI • H) •
     Ha)
```

It involves primitives rewriting rules like *symmetric* and *combination* that help in the construction of primitive rules but also proof terms of theorems like *simp_thms*(*25*) for the theorem $(?P \wedge ?P) = ?P$. It might not seem obvious looking at the proof object but the *simp* method is encoded in ML and as a program follows a pattern that yields in our case to a proof of the proposition $[\![A;\, B]\!] \Longrightarrow B \wedge A$.

We can make it more visible with the following lemma:

Example 8 *The cons_list theorem*

```
Isabelle
lemma cons_list : a#xs = [a]@xs
  apply (subst List.append.append_Cons)
  apply (subst List.append.append_Nil)
  apply (rule refl)
  done
```

The proof object reconstructed from the proof script is still small and legible and has roughly the following pattern:

$\lambda g\colon G\ A\ B\ .$
$\ (\lambda H\colon A\ .\ H) \bullet$
$\ ((\lambda g'\colon G\ B\ C\ .$
$\quad (\lambda H\colon B\ .\ H) \bullet$
$\quad\ (HOL.refl \cdot C \bullet g')) \bullet g)$

where G is composed of proof rewriting rules, A is the main goal $a\#xs = [a]@xs$, B is the subgoal $?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs$ after the application of the *subst List.append.append_Cons* rule, and C is the subgoal $?a\ \#\ ?xs = ?a\ \#\ ?xs$ after the application of the *subst List.append.append_Nil* rule (recall that • denotes the application of proof terms to proof terms). It gives the representation of the proof that is built step by step in a backward manner where any proof state is represented as a proof object of the form:

$\psi_1 \Longrightarrow \cdots \Longrightarrow \psi_n \Longrightarrow \varphi$

where φ is the proposition to be proved and $\psi_1, \ldots, \psi_n$ are the remaining subgoals. For the proof object above we have the following theorem:

$(C \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A$

So if we have a proof of the subgoal $?a\ \#\ ?xs = ?a\ \#\ ?xs$ and a proof of the subgoal $?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs$, then we have a proof of the main goal, that is the proposition $a\#xs = [a]@xs$, and we have a theorem represented by the term:

$(?a\ \#\ ?xs = ?a\ \#\ ?xs \Longrightarrow ?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs)$
$\quad \Longrightarrow (?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs)$
$\quad \Longrightarrow a\#xs = [a]@xs$

For the goal $a\#xs = [a]@xs$ and the subgoal $?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs$, the proof object has the form:

$equal\_elim \cdot (?a\ \#\ ?xs = [?a]\ @\ ?xs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs) \cdot (?a\ \#\ ?xs = ?a\ \#\ []\ @\ ?xs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs) \bullet$
$\ (combination \cdot TYPE(?'a\ list) \cdot TYPE(prop) \cdot (\lambda fooabs.\ (?a\ \#\ ?xs = fooabs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs)) \cdot (\lambda fooabs.\ (?a\ \#\ ?xs = fooabs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs)) \cdot [?a]\ @\ ?xs \cdot ?a\ \#\ []\ @\ ?xs \bullet$
$\quad (reflexive \cdot TYPE(?'a\ list \Rightarrow prop) \cdot (\lambda fooabs.\ (?a\ \#\ ?xs = fooabs \Longrightarrow ?a\ \#\ ?xs = [?a]\ @\ ?xs))) \bullet$

(*eq_reflection* · *TYPE*(*?'a list*) · [*?a*] @ *?xs* · *?a* # [] @ *?xs* • (*arity_type_list* · *TYPE*(*?'a*) • (*Pure.PClass type_class* · *TYPE*(*?'a*))) • (*append_Cons* · *TYPE*(*?'a*) · *?a* · [] · *?xs* • (*Pure.PClass type_class* · *TYPE*(*?'a*))))) •
(λ*H*: *?a* # *?xs* = [*?a*] @ *?xs*. *H*) •
PROOF

where *PROOF* is a proof object representing a proof for the subgoal *?a* # *?xs* = *?a* # [] @ *?xs*. The first part of the proof object amounts to the proof object of the first substitution *subst List.append.append_Cons* in the proof script. The application of the first substitution using the *subst* method yields the subgoal *?a* # *?xs* = *?a* # [] @ *?xs* that can be seen in the second term to which the *equal_elim* proof is applied. This subgoal is used to prove the main goal that appears right after as (λ*H*: *?a* # *?xs* = [*?a*] @ *?xs*. *H*). *PROOF* itself is composed of the subgoals *B* and *C* and has the same form:

equal_elim · (*?a* # *?xs* = *?a* # [] @ *?xs* ⟹ *?a* # *?xs* = *?a* # [] @ *?xs*) · (*?a* # *?xs* = *?a* # *?xs* ⟹ *?a* # *?xs* = *?a* # [] @ *?xs*) •
(*combination* · *TYPE*(*?'a list*) · *TYPE*(*prop*) · (λ*fooabs*. (*?a* # *?xs* = *?a* # *fooabs* ⟹ *?a* # *?xs* = *?a* # [] @ *?xs*)) · (λ*fooabs*. (*?a* # *?xs* = *?a* # *fooabs* ⟹ *?a* # *?xs* = *?a* # [] @ *?xs*)) · [] @ *?xs* · *?xs* •
(*reflexive* · *TYPE*(*?'a list* ⇒ *prop*) · (λ*fooabs*. (*?a* # *?xs* = *?a* # *fooabs* ⟹ *?a* # *?xs* = *?a* # [] @ *?xs*))) •
(*eq_reflection* · *TYPE*(*?'a list*) · [] @ *?xs* · *?xs* • (*arity_type_list* · *TYPE*(*?'a*) • (*Pure.PClass type_class* · *TYPE*(*?'a*))) • (*append_Nil* · *TYPE*(*?'a*) · *?xs* • (*Pure.PClass type_class* · *TYPE*(*?'a*))))) •
(λ*H*: *?a* # *?xs* = *?a* # [] @ *?xs*. *H*) •
PROOF'

where *PROOF'* is a proof of the subgoal *?a* # *?xs* = *?a* # *?xs*. The second substitution *subst List.append.append_Nil* in the proof script is applied to the subgoal *?a* # *?xs* = *?a* # [] @ *?xs*. In the *PROOF'* proof object, which appears right after the main goal (λ*H*: *?a* # *?xs* = [*?a*] @ *?xs*. *H*), we find the proof object which amounts to the second substitution *subst List.append.append_Nil* . Again, in the second term to which the *equal_elim* proof is applied, we can find the new subgoal *?a* # *?xs* = *?a* # *?xs*. If this subgoal is proven, it can be used to prove the old subgoal (λ*H*: *?a* # *?xs* = *?a* # [] @ *?xs*. *H*). Then the last application in the proof script *rule refl* conclude the proof and its proof object is:

HOL.refl · *TYPE*(*?'a list*) · *?a* # *?xs* • (*arity_type_list* · *TYPE*(*?'a*) • (*Pure.PClass type_class* · *TYPE*(*?'a*)))

As expected, the two substitutions follow exactly the same pattern. By looking at the proof object, two observations can be made. Firstly, the pattern of the proof object is the representation of the method semantics. To acknowledge this semantics, meta-data could be attached to the proof object using an ontology. For example, using a tag system described in an ontology, one can attach this tag to the proof object to help in the identification of patterns that represent methods. In fact, by using an ontology, any kind of meta-data can be attached to a proof object. Secondly, handling of proof objects at the ML level needs to take into account not only proofs, terms and types as we can see in the proof datatype but also sorts. The previous lemma proves a proposition on the algebraic structure of *list*s with elements of unspecified types. So its proof object includes schematic type variables for which a sort may be specified.

5.3 Meta Types Reification and Term Anti-Quotations

To represent proof terms in HOL, we need an environment that will give a formal representation of the *proof* datatype in HOL. In the chapter introduction, we mentionned that Nipkow et al. [13] have already formalized a metalogic for Isabelle/HOL in HOL. The implementation uses deep embedding allowing to represent types, terms, and proofs directly in HOL. The implementation in HOL of these abstract types share a lot of similiraties with the ML implementation. The abstract type *typ* implementation in ML is as follows:

ML
```
(*Indexnames can be quickly renamed
   by adding an offset to the integer part,
   for resolution.*)
type indexname = string * int;

(*Types are classified by sorts.*)
type class = string;
type sort = class list;
type arity = string * sort list * sort;

(*The sorts attached to TFrees and TVars specify
  the sort of that variable.*)
datatype typ = Type of string * typ list
             | TFree of string * sort
             | TVar of indexname * sort;
```

and its counterpart in HOL in the metalogic is:

```
Isabelle
type_synonym name = String.literal
type_synonym indexname = name × int

type_synonym class = String.literal

type_synonym sort = class set
abbreviation full_sort ≡ ({}::sort)


datatype variable = Free name | Var indexname

datatype typ =
  is_Ty: Ty name typ list |
  is_Tv: Tv variable sort
```

Type and type variables are implemented in the same way where schematic types variables are considered as names indexed by an integer, an *int* in ML for the ML implementation and the *int* from Isabelle/HOL library for the metalogic. The only noticeable difference comes from the implementation of sorts where sorts in ML use lists as in HOL they use the Isabelle/HOL implementation of *set*s. For example, the type *int* in Isabelle/HOL is encoded in ML as follows:

Type (*Int.int*, [])

It uses the *Type* constructor of the ML datatype and defines it using the string *Int.int* as name and then associates it with an empty list as it does not take any parameter. In the metalogic the type *int* is encoded in HOL and its representation is:

constT STR ''Int.int''

where *constT* is defined as an abbreviation:

```
Isabelle
abbreviation constT name ≡ Ty name []
```

and *STR ''Int.int''* encodes the HOL *string ''Int.int''* as a *String.literal* as required by the implementation. The Isabelle/HOL *'a list* data type has a parameter and then is encoded as:

Type (*List.list*, [*'a*])

It uses the associated type list to store the types of the parameters. In the same way in the metalogic:

Ty STR ''List.list''
[Tv (Free (String.Literal True True True False False True False STR ''a''))
(insert STR ''HOL.type'' full_sort)]

the associated *list* is used to add the parameter type. We can also see that the default class *type*, which is the default class added by the Isabelle kernel to any polymorphic term whose class is not specified, is added to the set of sorts of the polymorphic parameter, using the *insert* constant.

Terms in ML and in the metalogic are also pretty similar. In ML:

ML

```
datatype term =
    Const of string * typ
  | Free of string * typ
  | Var  of indexname * typ
  | Bound of int
  | Abs  of string*typ*term
  | op $ of term*term;
```

Bound variables are implemented as de Bruijn indices [108] using the *Bound* constructor and every other constructor parameter is associated with a string that will end up being the term itself. For example, the term $\lambda x.\ x$ whose term variable x is bound is represented in ML as follows:

Abs $(x,$ *'a*, *Bound 0*)

The bound variable is encoded with *Bound 0*. In the metalogic, the term datatype is quasi equivalently declared:

Isabelle

```
datatype term =
  is_Ct: Ct name typ |
  is_Fv: Fv variable typ |
  is_Bv: Bv nat |
  is_Abs: Abs typ term |
  is_App: App term term (infixl $ 100)
```

The *proof* type implementations are the most different ones. Indeed the metalogic does not need to cope with the implementation of theorems objects and can focus on the proof object, giving a much simpler implementation:

```
Isabelle
datatype proofterm = PAxm term tyinst list
  | PBound nat
  | Abst typ proofterm
  | AbsP term proofterm
  | Appt proofterm term
  | AppP proofterm proofterm
  | OfClass typ class
  | Hyp term
```

The main drawback is that the meta *proof* type does not store axioms and theorems names. This information is lost but it is not the purpose of the proof terms in the metalogic that aim at mirroring proof terms generated by Isabelle/HOL, and are not designed to record proofs in the inference system [13]. However this information could be attached as meta-data if we are using an ontology.

The translation of types and terms is straightforward thanks to the identical structure in ML and in the metalogic. Isabelle/DOF meta-level anti-quotations for types and terms (see section 2.7) can now be treated as fully inhabited elements. The new implementation of meta-level anti-quotations uses the reification mechanism and the refined process (see section 3.2) can now also elaborate types and terms to their representation in HOL using the metalogic. This opens the possibility to compute terms with types and terms in the logic that no more involve a referential equality but a definitional equality:

```
Isabelle
value*‹@{typ ‹int›} = @{typ ‹int›}›
```

The computation of the term @{*typ* ‹*int*›} = @{*typ* ‹*int*›} by the `value*` command returns *True* but this time the two types are evaluated. The new meta-level anti-quotations are fully integrated in Isabelle/DOF: ontologies can be defined to express knowledge about types and terms defined in HOL. Ontologies for formal domain like mathematics or engineering can represent this knowledge and queries on this elements can be made. This is an important aspect for advanced search in libraries: it allows to define queries and ontologies with an introspective purpose. For example, meta-data expressing knowledge about the defined types in a theory can be attached to a formal theory of multivariate polynomials. If we plan to use multivariate polynomials in a theory and we want to constrain them using axiomatic classes, we would like to know if multivariate polynomials are defined as a type (see section 4.3). By querying the ontology, we would find that polynomials are represented in two ways, as trees and as sums of monomials multiplied by some coefficient, and that the computation of monomials is only defined for the second form. But the second form is specified as a type synonym, not compatible with type instantiation mechanisms, so we must use the first representation for type instantiation. So we would find that there is no easy way to use the multivariate polynomial theory in the library for type instantiation of axiomatic classes like *semiring* for which specifications of the two operations $(+)$ and $(*)$ is required. Then, if the polynomials as sums of monomials are defined in another theory, we would be able to prove that we need to import the two theories in our theory to instantiate polynomials as *semiring* as we plan.

Another aspect is ontology definitions to express knowledge about the logic itself. For example, we can now express that a logical language like FOL must not involve distinguished types for Booleans and functions as these types are unsuitable.

The translation of proof terms is not as straightforward. In [13], the authors are interested in generating code for a proof checker, and they consider proved lemmas as axioms in the translation, but as we aim at expressing knowledge about proof scripts through their structure, we need to keep proof terms intact in the translation. A proof term of a theorem can not be translated to a proof term of an axiom as we will lose all information. It implies that we need to reify a proof term with its structure unspoiled. In [17], Berghofer distinguished proof objects and partial (or implicit) proof objects, where the latter have been compressed, giving proof objects with omitted information denoted by placeholders _, and provides reconstruction mechanisms. For example, the partial proof object of the *symmetric* meta-rewriting rule is represented as follows:

symmetric · TYPE(?'a) · _ · _

and the reconstructed proof object as:

symmetric · TYPE(?'a) · ?x · ?y

where the schematic variables have been instantiated. As we are interested in the proof structure and not so much in the arguments, partial proofs seems more suitable because they will have a less enormous size when extracting proofs from the realistic applications. But their representation, when compressed, erase valuable information that is tightly tied to their representation in the metalogic. If we look at the encoding of the previous proof object in ML, we can observe the differences with its partial version. The partial term is encoded as follows:

```
PAxm (Pure.symmetric,
        Const (Pure.imp, prop ⇒ prop ⇒ prop) $ (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop)
$ Var ((x, 0), ?'a) $ Var ((y, 0), ?'a)) $
          (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop) $ Var ((y, 0), ?'a) $ Var ((x, 0), ?'a)),
        NONE) %
    NONE % NONE
```

The *PAxm* type list value is NONE, meaning left unspecified. This type list is the list of the schematic type variable occurring in the axiom proposition, i. e. in the term:

```
Const (Pure.imp, prop ⇒ prop ⇒ prop) $ (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop) $ Var
((x, 0), ?'a) $ Var ((y, 0), ?'a)) $
        (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop) $ Var ((y, 0), ?'a) $ Var ((x, 0), ?'a))
```

As the compression algorithm keeps the proposition term intact, the type list can be replaced by *NONE* and can be reconstructed from the partial term. In the metalogic, the *PAxm* constructor also has a type list argument, but it is not simply a type list but a *tyinst list*, where *tyinst* is defined as a type synonym:

```
type_synonym tyinst = (variable × sort) × typ                                Isabelle
```

tyinst is used to encode type substitution for axioms in the metalogic as an association list instead of a function used for the rest of the metalogic implementation. Type substitution assigns a type to each type variable and sort pair. For a *PAxm*, the *tyinst* list is used internally to check the validity of a type against the theory and therefore that type instantiation is possible. Consequently, the *tyinst* list is mandatory in a *PAxm* of the metalogic, which means that if we want to translate axioms in the logic to axioms in the metalogic, the type list of *PAxms* in the logic need to be inhabited before the translation happens. Because of this limitation and to keep the translation simple, we choose to use full proof terms and not partial ones. The full proof term object of the previous example is:

```
 PAxm (Pure.symmetric,
       Const (Pure.imp, prop ⇒ prop ⇒ prop) $ (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop)
$ Var ((x, 0), ?'a) $ Var ((y, 0), ?'a)) $
        (Const (Pure.eq, ?'a ⇒ ?'a ⇒ prop) $ Var ((y, 0), ?'a) $ Var ((x, 0), ?'a)),
       SOME [?'a]) %
    SOME (Var ((x, 0), ?'a)) % SOME (Var ((y, 0), ?'a))
```

where we can see that the type list of the proposition is now inhabited with the schematic type variable occurring in the proposition term. We are now able to translate proof terms and keep their structure.

The @{*thm* ‹...›} term anti-quotation is updated to use the reification mechanism and now it can be used to reference a theorem or its proof term object by using depending on the context.

The new meta-type anti-quotations implementation with inhabited logical objects does not only extend the usage of domain ontologies as we explained above, it also give rise to a new consideration of the term anti-quotations themselves.

Indeed, now that meta-types are represented as terms, meta-type anti-quotations should be able to derive the object from embedded λ-terms, and therefore should support embedded term anti-quotations. For example, if we have a simple ontological class *ctag* that specifies the notion of proof tag:

Example 9 *A class with proof tagging information*

```
Isabelle
doc_class ctag =
proof_tag :: int <= 1
```

and the instance *inst_tag*:

Example 10 *A ctag class instance*

```
Isabelle
text*[inst_tag::ctag, proof_tag = 2]‹›
```

we can get the value of the *inst_tag* instance *proof_tag* attribute using the term anti-quotation @{*ctag ''inst_tag''*} and the **value*** command:

```
Isabelle
value*‹proof_tag @{ctag ‹inst_tag›}›
```

which returns the value *2*. But we would like to get the representation of the term in the logic to use it as information in an ontology using a @{*term* ‹...›} anti-quotation. For that, the @{*term* ‹...›} treatment is updated to support anti-quotation cascading. The following evaluation:

```
Isabelle
value*‹@{term ‹proof_tag @{ctag ‹inst_tag›}›}›
```

will returns the value *2* represented as a λ-term in the logic using the metalogic notation:

Ct STR ''Num.numeral_class.numeral'' (constT STR ''Num.num'' → constT STR ''Int.int'')
$ (Ct STR ''Num.num.Bit0'' (constT STR ''Num.num'' → constT STR ''Num.num'')
$ Ct STR ''Num.num.One'' (constT STR ''Num.num''))

This term is typed as *Core.term*, the type of terms in the metalogic, and the @{*term* ‹...›} anti-quotation refined process now elaborates the term. This comes in contradiction with the distinction between terms and values we made in section 3.2, where only values are elaborated. Similarly to the syntactic category of commands in Isabelle/HOL that are considered in Isabelle/DOF as abstract objects, meta-level term anti-quotations are references to objects of the metalogic represented in the HOL, and as such a @{*value* ‹...›} term anti-quotation is introduced with the same behaviour as the ***value*** command so that the @{*term* ‹...›} anti-quotation could mimic the ***term*** command. This gives rises to another questioning: should the evaluation with ***value*** of the terms @{*term* ‹*proof_tag* @{*ctag* ‹*inst_tag*›}›} and @{*value* ‹*proof_tag* @{*ctag* ‹*inst_tag*›}›} returns the same object? Evaluation is interpreted as an exhaustion process, where both the terms gives the same value, and the @{*value* ‹...›} anti-quotation is implemented identically to the ***value*** command, i. e. it is compatible with *eval* and *nbe* evaluation techniques.

In the same manner, another type anti-quotation should be added to allow type references using pattern matching. For example, let's suppose that we have a list of functions types $'a \Rightarrow\ 'b$ and we want to update this list of types to a list of product types $'a \times\ 'b$. We would like to specify a type constructor constant, using the ***definition*** command, that will be used to constructs types in the metalogic:

```
Isabelle
definition dest_funT
  where dest_funT f ≡ @{Type ‹fun A B => ‹(A, B)››}
```

The *dest_funT* definition should take a typing function as an argument and return the product type. But here we are limited by the implementation of term anti-quotations. Indeed, we saw in section 3.2 that term anti-quotation arguments are HOL *string*s, thus the type @{*Type* ‹*fun A B* => ‹(*A*, *B*)››} can not be understood as a function type whose arguments depends on the definition instantiation: the string "*fun A B* => ‹(*A*, *B*)›" supposes that the construction of the new type will depend on the value of its arguments. For the type *int* ⇒ *int* as argument, the const *dest_funT* will return the representation in the metalogic of the type *int* × *int*. But by using a string we can not abstract over the arguments and we loose the possibility to bind term variables using a lambda abstraction for the variables. To declare *dest_funT* we must redefine it as a function like the following:

```
Isabelle
fun dest_funT
  where dest_funT (Ty t [A, B]) = (let x = String.implode ''fun'' in case t of x
⇒ (A, B))
```

where the pattern matching is done directly at the logical level. Extending the type anti-quotation would require a major update in the design where types become dependent on terms, which is not supported for now by the implementation.

The support of cascading term anti-quotation for the @{*value* ‹...›} and @{*term* ‹...›} anti-quotation gives us sufficient expressiveness to reference terms in HOL extracted from an Isabelle/DOF ontology that can then be used as meta-data.

5.4 Proof Objects with Meta-Data

As lemmas are used to prove propositions on arbitrarily complex mathematical objects, advanced handling of proof scripts with attached meta-data can take several forms. Indeed , we might consider the final structure of a proof object or we might consider adding meta-data during the reconstruction process incrementally, or mixing both, depending on the meta-data we want to attach. We saw that proof script structure information is lost when reconstructing proof objects. To get back this structure, the idea is to use ontological meta-data to divide a proof object in several sub proof objects that represent each application of a proof method. We get the following pattern:

$(PROOF_1 \cdot DATA_1) \bullet \cdots \bullet (PROOF_n \cdot DATA_n)$

where meta-data of methods is attached to a subproof object representing the proof method step by applying subproof *proof* terms to meta-data terms represented as terms of the metalogic. This way, the structure of the proof is reconstructed, and by using pattern matching proof objects corresponding to proof methods can be used to reconstruct proof methods with the help of meta-data accessible directly in the proof object.

Recovering the structure of the proof script can use both approaches, pattern matching over the final structure of the proof object or incremental proof reconstruction. The latter implies an update of Isabelle/Isar commands, because in a theorem abstract object, the proof information contained in the proof body is stored as inferences rules, and the structure of the proof script is already lost. In fact, the incremental approach is quite complex: it would require modifications of code deeply integrated in Isabelle/Pure inference kernel in two fold. On the one hand, Isabelle/Isar commands need to be updated or new commands must be written and on the other hand methods code also need to be updated or new methods need to be written. It comes from the fact that Isabelle/Isar commands and methods are deeply intertwined when it comes to the generation of proof terms. The proof term generation happens when generating an abstract theorem datatype but this process uses already defined methods. So if we attach meta-data to methods, we need to store this information so it can later be used by the proof term generation process triggered by Isabelle/Isar commands. For example, the method name is stored directly after the parsing of an `apply` command in an ML structure representing a method. This name is then used to retrieve the method that is already declared in the logical context. To be sure to add meta-data information to the right method proof object, the reified ontological concepts need to be carried using the logical context through the code until the proof term is generated. We already told that the generation process happens when constructing the abstract theorem datatype. But the process also happens during the evaluation of the method. Therefore the meta-data information need to be conveyed to the core of Isabelle. This implementation is not yet done but we can draw an example for a specific method like *subst* with the **apply** command. The method *subst∗* is equivalent to the *subst* method but also accepts ontological concept declarations similarly to Isabelle/DOF commands. We also define a new *apply∗* command equivalent to **apply** but compatible with methods with attached meta-data. We can update the *cons_list* theorem in Example 8:

```
Isabelle
lemma cons_list : a#xs = [a]@xs
  apply (subst List.append.append_Cons)
  apply* (subst*[nil_info::ctag, proof_tag = 4] List.append.append_Nil)
  apply (rule refl)
  done
```

The *nil_info* class instance is added to the logical context when the substitution is applied with the *apply∗* command. More precisely, when applying the substitution, first the method is parsed and the *subst∗* method instantiation is updated with the ontological meta-data, then the method is retrieved and is applied. The application of the method code carries the meta-data and when the theorem is generated, the meta-data is reified and added to the subproof term representing the proof step. The proof object with ontological information will then have the form:

$$PROOF_1 \bullet (\lambda H\colon\ ?a \ \# \ ?xs = [?a] \ @ \ ?xs.\ H) \\ \bullet ((PROOF_2 \bullet (\lambda H\colon\ ?a \ \# \ ?xs = ?a \ \# \ [] \ @ \ ?xs.\ H) \bullet PROOF_3) \cdot DATA)$$

where $PROOF_1$ is the proof object for the subgoal $(\lambda H\colon\ ?a \ \# \ ?xs = [?a] \ @ \ ?xs.\ H)$, $PROOF_2$ the proof term for the subgoal $(\lambda H\colon\ ?a \ \# \ ?xs = ?a \ \# \ [] \ @ \ ?xs.\ H)$, $PROOF_3$ the proof object for the *refl* rule, and *DATA* the *nil_info* instance represented as a term in the metalogic. We can see that the full proof term $(PROOF_2 \bullet (\lambda H\colon\ ?a \ \# \ ?xs = ?a \ \# \ [] \ @ \ ?xs.\ H) \bullet PROOF_3)$ is applied to the *DATA* term to recreate the structure of the proof script.

The other approach we propose to recreate the proof script structure is pattern matching. Like in section 3.5, we will use a combination of term anti-quotations and Isabelle/HOL functional programming language capabilities. We show in the following through an example how this can be done using pattern matching over the structure of a proof object using the systematic pattern of subproof terms that represent proof script steps. We can use the @{*value* ‹...›} term anti-quotation that can reify terms and supports term anti-quotation cascading. We can define an HOL function using a *fun∗* command similar to the Isabelle/HOL **fun** command with the additional ability to process term anti-quotations. For example with the following definition:

```
Isabelle
fun∗ pattern where
  pattern (PAxm ((Ct a (Ty ba [constT bb, (Ty bc [constT bd, constT be])])) $ c $
d) [((Var e, f), constT g), ((Var h, i), j)]) =
                  (if be = STR ''prop''
                   then (Appt (PAxm ((Ct a (Ty ba [constT bb, (Ty bc [constT bd,
constT be])])) $ c $ d) [((Var e, f), constT g), ((Var h, i), j)])
                                   (@{value ‹proof_tag @{ctag ‹inst_tag›}›}))
                   else (PAxm ((Ct a (Ty ba [constT bb, (Ty bc [constT bd, constT
be])])) $ c $ d) [((Var e, f), constT g), ((Var h, i), j)]))
| pattern x = x
```

the *pattern* function is used to update proof objects using the @{*value* ‹*proof_tag* @{*ctag* ‹*inst_tag*›}›} term anti-quotation. The pattern matching will match the reflexive theorem *?x* ≡ *?x* whose representation in ML is as follows:

PAxm (*Pure.reflexive*, *Const* (*Pure.eq*, *?'a* ⇒ *?'a* ⇒ *prop*) $ *Var* ((*x*, *0*), *?'a*)
    $ *Var* ((*x*, *0*), *?'a*), *NONE*)
 % *NONE*

If the *Pure.eq* contant is indeed of type *prop*, then the proof object is updated by constructing a new proof object using the *Appt* constructor of the *proofterm* datatype of the metalogic to add the elaborated term anti-quotation term. Thus, we get a proof term with the form:

PROOF · *DATA*

where *PROOF* is the axiom and *DATA* is the reified value of the term anti-quotation @{*value* ‹*roof_tag* @{*ctag* ‹*inst_tag*›}›}.

We can then use the `value∗` command to evaluate a term that uses the *pattern* function:

```
Isabelle
value∗‹pattern
   (PAxm (mk_eq' propT Core.A Core.B ⟼ Core.A ⟼ Core.B)
              [((Var (String.Literal True True True False False True False STR ''a'',
0), full_sort), constT STR ''Int.int'')
                  , ((Var (String.Literal True True True False False True False STR
''b'', 0), full_sort), propT)])›
```

Here the argument of the *pattern* function matches and the `value∗` command returns the value:

Appt

(*PAxm* (*mk_eq′ propT Core.A Core.B* ⟼ *Core.A* ⟼ *Core.B*)
[((*Var* (*String.Literal True True True False False True False STR ′′a′′*, *0*), *full_sort*), *constT STR ′′Int.int′′*), ((*Var* (*String.Literal True True True False False True False STR ′′b′′*, *0*), *full_sort*), *propT*)])
(*Ct STR ′′Num.numeral_class.numeral′′* (*constT STR ′′Num.num′′* → *constT STR ′′Int.int′′*) $ (*Ct STR ′′Num.num.Bit0′′* (*constT STR ′′Num.num′′* → *constT STR ′′Num.num′′*) $ *Ct STR ′′Num.num.One′′* (*constT STR ′′Num.num′′*)))
:: *proofterm*

First the term anti-quotation is elaborated and returns the *proof_tag* attribute-value of the *inst_tag* instance declared in Example 10, i. e. the value *2* reified in the metalogic, and then the new term is constructed. We can see in the output that the value *2* represented in the metalogic by the term:

Ct STR ′′Num.numeral_class.numeral′′ (*constT STR ′′Num.num′′* → *constT STR ′′Int.int′′*)
$ (*Ct STR ′′Num.num.Bit0′′* (*constT STR ′′Num.num′′* → *constT STR ′′Num.num′′*)
$ *Ct STR ′′Num.num.One′′* (*constT STR ′′Num.num′′*))

is indeed the second term argument of the *Appt* constructor, and the resulting term is of type *proofterm*.

Proof terms with attached meta-data can then be passed to other proof assistants. By parsing the proof term, the structure of the proof script could be extracted using meta-data terms that may be tagged with specific identifier to distinguished them from classical terms of the proof object. Then the meta-data can be extracted from the proof object and the ontological knowledge can help in reconstructing the proof script step from the subproof object using a reference domain ontology.

5.5 Conclusion

The reification mechanism allows to reason over meta types in Isabelle/DOF. Using the metalogic tteerms, types and theorems are now objects formalized in HOL, and thus the semantics of meta-level term anti-quotations becomes pretty similar to Isabelle/DOF diagnostic commands like **value∗** and **term∗**. Both are used to embed terms and introduce a specific logical context for checking or evaluation. Meta-level term anti-quotations allow for ontology definitions that will represent knowledge with introspective components on the theories of Isabelle/HOL but also on the object logic itself. The meta-level anti-quotation for *thm*s can be used to extract and reconstruct proof objects, and we are sure the proof object is a correct representation of the theorem because we use the ML API that certify that the generated elements of the proof object have passed the full well-formedness checks. So we can define domain ontologies with types, terms, and theorems objects and then use them to add knowledge to proof objects by adding ontological elements to a proof object. We use this process to recreate the structure of the proof script. As a consequence, a proof object with attached meta-data can be transfered to another proof assistant: it embeds both the original structure of the proof script and the information to recreate the proof script by looking at the meta-data. Because the proof object is encoded in a logical language, meta-programming capabilities of another proof assistant could be used to reflect the meta-data and the proof object and thus reconstruct the proof script by referring to the domain ontology.

Chapter 6

Conclusion

6.1 Achievements

In this thesis we presented an extension of Isabelle/DOF, an ontology framework deeply integrating continuous-check/continuous-build functionality into the formal development process in HOL. The novel feature of term-contexts in Isabelle/DOF, which permits term anti-quotations elaborated in the parsing process, paves the way for the abstract specification of meta-data constraints as well as the possibility of advanced search in the meta-data of document elements. It reconsiders syntactic categories like commands in Isabelle/HOL: they become document elements with ontological meta-data using an abstraction pattern, and term anti-quotations are used to reference these documents. Term-contexts increase the possibility to reason on the document: properties on ontologies representing the knowledge of documents can be proved, reflecting document properties on their structure and their semantics. They also contribute to a better understanding of the notion of an integrated document: queries can be made to retrieve document elements and discriminate over the type of the elements, i. e. whether they are formal or informal, which logical objects they represent, etc. The abstracted pattern can be generalized to define ontologies that will represent knowledge but also meta-knowledge like properties on knowledge.

Isabelle/DOF ODL language is extended to support parametric polymorphic classes. It increases the expressiveness of Isabelle/DOF and permits a deeper integration into Isabelle/HOL. Thus, the generic caculus of axiomatic type classes is available and users can specify constraints for ontological class attributes using axiomatic class specifications from the Isabelle/HOL libraries of the AFP. Polymorphic classes also introduce a new possibility to describe the structure of a document: dependencies between document elements can be specified that redefine the notion of linking in Isabelle/DOF. The linking is not only defined between formal ontological concepts and informal text but between document elements using term-contexts. The linking can be specified to represent the structure of dependencies or to represent dependencies between specific document element instances. The modeling of "provenance" exemplifies the multiple notions that can be associated to the linking when using polymophism. The proposed security model is a first step in building access-control for integrated documents. Organizations that involve multiple decentralized stakeholders and develop safety or security-critical systems targeting certification should benefit from this model, as certification standards such as CENELEC 50128 [11] or Common Criteria [12] often define the concept of *role* that implies a security model.

The reification mechanism that uses a metalogic of Isabelle/HOL enables meta-level term anti-quotations as references to objects in HOL. Term anti-quotations semantics blends into Isabelle/DOF diagnostic commands semantics and a new meaning emerge where meta-level term anti-quotations are not just references to empty syntactic categories but to terms that are checked or evaluated and support cascading term anti-quotations. They also open ways to define ontologies with introspective support: an ontology can represent knowledge about formal elements of theories like theorems and prove properties on them, and ontologies can also represent knowledge about object logic to give information about this logic particularities. The reification mechanism also enables proof terms rewriting. We propose a methodoly that uses this possibility to embed ontological information into proof objects. The added meta-data should help in proof script reconstruction: a proof assistant using a different proof script language and a different tactic language could use a domain ontology as reference to understand how the embedded meta-data can be used to reconstruct a theorem using its own languages. And it could infer the structure of the theorem by looking at the structure of the proof object that recreates the structure of the theorem using meta-data as subproof objects.

6.2 Future Work

The ideas for future extensions fall into two categories: extensions of the Isabelle/DOF framework and interaction with external tools and environments.

6.2.1 Extension of the Isabelle/DOF Framework

An important aspect of Isabelle/DOF is the possibility to prove properties on ontologies. For now very few proofs were written to certify the behavior of Isabelle/DOF, and the examples are handcrafted without considering ontologies as meta objects to find general properties on ontologies and prove them. A major exploration should be done to have an eyesight of the full potential that the generation of a theory of meta-data offers. Also the new understanding of document elements as objects of ontologies should be investigated to better understand the implications. There should be potential properties on the new linking definition.

The support of parametric polymorphic allow to specify complex algebraic structure for the linking between document elements. This should be extended to support interaction with the IDE: then this new linking could help in integrating **rejects** and **accepts** clauses of monitors as simple class attributes. This way monitors could be redefined as simple ontological classes with an attribute that specify constraints on the structure of the document, and the support of structural constraints could be extended to support any type of computational model in addition to regular expression such as CSP [109].

Class invariants in Isabelle/DOF only support λ-terms as input source and derive a theorem from it. We are thinking of extending invariants to support theorems as input. This way, the generation of theorems asserting properties on class attributes could be simplified and delegated to the standard proof environment by using proof scripts. The implementation might be complex due to the incremental parsing process: a theorem can be referenced in a class if it is already written, so we might have to support theorem definitions when declaring a new class or maybe use a mechanism similar to the ***declare_reference**** command which will only generate the theorem from the proof script when the checking of invariants is triggered, that is when declaring a class instance.

We saw in section 5.3 that the implementation of term anti-quotations does not permit to generalize every meta-level term anti-quotations to support as input source a specific embedded ML declaration that depends on the value of its arguments. We need to have a better understanding of Isabelle/HOL parsing system to know if it is possible to redefine term anti-quotations as we wish.

Adding support for polymorphism entails a dual role for a record value that is referenced by a term anti-quotation (see section 4.2). We plan to work on a new design for polymorphic term anti-quotations where they will have the same behavior as definitions in Isabelle/HOL: they should be added to the logical context with schematic type variables and these schematic variables should be fixed when the term anti-quotation is elaborated.

The support of proof scripts with ontologized methods is for now just a proof of concept. It requires a lot of code duplication that needs to be imported from Isabelle implementation to be modified: we should investigate mechanisms that will avoid the code duplication too, but we are not optimistic because the proof object generation is deeply burried into Isabelle inference kernel. We have not found an API entry to have access to this code and modify it without using duplication. Maybe it would require an update of the Isabelle ML API to allow such modifications.

The main ontology languages like OWL support multiple inheritance. This might represent a challenge if we try to convert OWL-like ontologies to Isabelle/DOF which only supports single inheritance. The support of multiple inheritance in Isabelle/DOF would imply a major redesign of Isabelle/DOF. If wee keep with Isabelle/HOL records, we might offer a single inheritance but with support for class attributes with extended properties. We draw our inspiration from [110] that proposes classes whose attributes support values of two class instances with different types if the two instances inherit from the same super class at the same level in the class hierarchy. A true multiple inheritance support entails the extension of the record package. Another approach might be to look at the implementation of axiomatic classes in Isabelle/HOL that form an order sorted algebra with multiple inheritance support.

6.2.2 Tool Interactions

To allow an interaction between theorem provers targeting theorem import/export techniques, we mention a domain ontology of reference. To be consensual, its definition should involve stakeholders of several proof assistants to reach an agreement regarding the categories that should be defined and how they need to be defined. The main aspects to consider seem to be the logical language and and the specification constructs that should be natively used the proof script language and the tactic language. The ontology spares the burden of needing a fully formalized language for proof scripts and tactics but should use formalized concepts associated with each language to help in proving properties on these languages. The choice of the ontology language is also important. We argue that Isabelle/DOF ODL could be implemented by a proof assistant with little effort if the proof assistant uses a logical language as expressive as HOL. Meta level types will require a metalogic for the proof assistant. Furthermore, if the proof assistant benefits from advanced meta-programming features like Coq with METACoq [81], a mechanism similar to term anti-quotations could be implemented, with the drawback of using the metalogic specifications instead of the logic ones and thus requiring reflective mechanisms when evaluating terms.

The major interaction that should be developed is access to the AFP. AFP libraries are available for consultation online, so using Isabelle/DOF we should be able to query remote libraries. But the main usage of the online access is consultation, so a deeper integration into the AFP is desirable As a first step, Isabelle/DOF has been accepted in the AFP meanwhile as an ordinary component. We should deploy an testing environment that exposes semantic information in addition to document elements. Is can simply be a server that stores AFP theories such Isabelle/DOF itself and exposes semantic concepts. This deployed service will serve as an example to explain what can be done with Isabelle/DOF. We need to think about the design that will suit our needs: a thorough review of the Distributed Assertion Management Framework [111] should help us in the choice of technologies we plan to favor.

Bibliography

[1] B. Parsia, B. Motik, P. Patel-Schneider, OWL 2 web ontology language structural specification and functional-style syntax (second edition), W3C recommendation, W3C, https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/ (Dec. 2012).

[2] M. A. Musen, The protégé project: A look back and a look forward, AI Matters 1 (4) (2015) 4–12. `doi:10.1145/2757001.2757003`.
URL `https://doi.org/10.1145/2757001.2757003`

[3] A. D. Brucker, I. Ait-Sadoune, P. Crisafulli, B. Wolff, Using the isabelle ontology framework, in: F. Rabe, W. M. Farmer, G. O. Passmore, A. Youssef (Eds.), Intelligent Computer Mathematics, Springer International Publishing, Cham, 2018, pp. 23–38.

[4] A. D. Brucker, B. Wolff, Isabelle/dof: Design and implementation, in: P. C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2019, pp. 275–292.

[5] A. D. Brucker, B. Wolff, Using ontologies in formal developments targeting certification, in: W. Ahrendt, S. L. T. Tarifa (Eds.), Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings, Vol. 11918 of Lecture Notes in Computer Science, Springer, 2019, pp. 65–82. `doi:10.1007/978-3-030-34968-4_4`.
URL `https://doi.org/10.1007/978-3-030-34968-4_4`

[6] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, R. Saillard, Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory, CoRR abs/2311.07185 (2023). `arXiv:2311.07185`, `doi:10.48550/ARXIV.2311.07185`.
URL `https://doi.org/10.48550/arXiv.2311.07185`

[7] G. Hondet, F. Blanqui, The new rewriting engine of dedukti (system description), in: Z. M. Ariola (Ed.), 5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July

6, 2020, Paris, France (Virtual Conference), Vol. 167 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 35:1–35:16. doi:10.4230/LIPICS.FSCD.2020.35.
URL https://doi.org/10.4230/LIPIcs.FSCD.2020.35

[8] W. Crichton, S. Krishnamurthi, A core calculus for documents: Or, lambda: The ultimate document, Proc. ACM Program. Lang. 8 (POPL) (jan 2024). doi:10.1145/3632865.
URL https://doi.org/10.1145/3632865

[9] L. C. Paulson, Designing a theorem prover, CoRR cs.LO/9301110 (1993).
URL https://arxiv.org/abs/cs/9301110

[10] A. D. Brucker, I. Ait-Sadoune, P. Crisafulli, B. Wolff, Using the Isabelle ontology framework: Linking the formal with the informal, in: Conference on Intelligent Computer Mathematics (CICM), no. 11006 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2018. doi:10.1007/978-3-319-96812-4_3.
URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelle-ontologies-2018

[11] Bs en 50128:2011: Railway applications – communication, signalling and processing systems – software for railway control and protecting systems, Standard, Britisch Standards Institute (BSI) (Apr. 2014).

[12] Common criteria for information technology security evaluation (version 3.1, release 5), available at https://www.commoncriteriaportal.org/cc/. (2017).

[13] S. Roßkopf, T. Nipkow, A formalization and proof checker for isabelle's metalogic, J. Autom. Reason. 67 (1) (2023) 1. doi:10.1007/S10817-022-09648-W.
URL https://doi.org/10.1007/s10817-022-09648-w

[14] L. C. Paulson, T. Nipkow, M. Wenzel, From LCF to isabelle/hol, Formal Aspects Comput. 31 (6) (2019) 675–698. doi:10.1007/S00165-019-00492-1.
URL https://doi.org/10.1007/s00165-019-00492-1

[15] L. C. Paulson, Isabelle: The next 700 theorem provers, CoRR cs.LO/9301106 (1993).
URL https://arxiv.org/abs/cs/9301106

[16] L. C. Paulson, The foundation of a generic theorem prover, J. Autom. Reason. 5 (3) (1989) 363–397. doi:10.1007/BF00248324.
URL https://doi.org/10.1007/BF00248324

[17] S. Berghofer, Proofs, programs and executable specifications in higher order logic, Ph.D. thesis, Technical University Munich, Germany (2003).
URL https://mediatum.ub.tum.de/601727

[18] M. Wenzel, Isar - A generic interpretative approach to readable formal proof documents, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, L. Théry (Eds.), Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings, Vol. 1690 of Lecture Notes in Computer Science, Springer, 1999, pp. 167–184. doi:10.1007/3-540-48256-3_12.
URL https://doi.org/10.1007/3-540-48256-3_12

[19] M. Wenzel, Isabelle, isar - a versatile environment for human readable formal proof documents, Ph.D. thesis, Technical University Munich, Germany (2002).
URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf

[20] M. Wenzel, The Isabelle/Isar Reference Manual, part of the Isabelle distribution. (2020).

[21] M. Wenzel, The Isabelle/Isar Implementation, part of the Isabelle distribution. (2023).

[22] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL—A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002. doi:10.1007/3-540-45949-9.

[23] M. J. C. Gordon, T. F. Melham (Eds.), Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993.

[24] A. Church, A formulation of the simple theory of types, Journal of Symbolic Logic 5 (2) (1940) 56–68. doi:10.2307/2266170.

[25] F. Haftmann, Code generation from specifications in higher-order logic, Ph.D. thesis, Technical University Munich (2009).
URL http://mediatum2.ub.tum.de/node?id=886023

[26] T. Nipkow, Order-sorted polymorphism in Isabelle, in: G. Huet, G. Plotkin (Eds.), Workshop on Logical Environments, 1993, pp. 164–188.

[27] T. Nipkow, C. Prehofer, Type reconstruction for type classes, Journal of Functional Programming 5 (2) (1995) 201–224.

[28] A. D. Brucker, B. Wolff, Isabelle/DOF: Design and implementation, in: P. C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods (SEFM), no. 11724 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2019. doi:10.1007/978-3-030-30446-1_15.
URL https://www.brucker.ch/bibliography/abstract/brucker.ea-isabelledof-2019

[29] M. Horridge, S. Bechhofer, The OWL API: A java API for OWL ontologies, Semantic Web 2 (1) (2011) 11–21. doi:10.3233/SW-2011-0025.
URL https://doi.org/10.3233/SW-2011-0025

[30] M. Wenzel, Isabelle as document-oriented proof assistant, in: J. H. Davenport, W. M. Farmer, J. Urban, F. Rabe (Eds.), Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings, Vol. 6824 of Lecture Notes in Computer Science, Springer, 2011, pp. 244–259. doi:10.1007/978-3-642-22673-1_17.
URL https://doi.org/10.1007/978-3-642-22673-1_17

[31] M. Wenzel, READ-EVAL-PRINT in parallel and asynchronous proof-checking, in: C. Kaliszyk, C. Lüth (Eds.), Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012, Vol. 118 of EPTCS, 2012, pp. 57–71. doi:10.4204/EPTCS.118.4.
URL https://doi.org/10.4204/EPTCS.118.4

[32] M. Wenzel, Asynchronous user interaction and tool integration in isabelle/pide, in: G. Klein, R. Gamboa (Eds.), Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, Vol. 8558 of Lecture Notes in Computer Science, Springer, 2014, pp. 515–530. doi:10.1007/978-3-319-08970-6_33.
URL https://doi.org/10.1007/978-3-319-08970-6_33

[33] M. Wenzel, Interaction with formal mathematical documents in isabelle/pide, in: C. Kaliszyk, E. C. Brady, A. Kohlhase, C. S. Coen (Eds.), Intelligent Computer Mathematics - 12th International Conference, CICM

2019, Prague, Czech Republic, July 8-12, 2019, Proceedings, Vol. 11617 of Lecture Notes in Computer Science, Springer, 2019, pp. 1–15. doi:10.1007/978-3-030-23250-4_1.
URL https://doi.org/10.1007/978-3-030-23250-4_1

[34] M. Wenzel, Isabelle/jEdit, part of the Isabelle distribution. (2023).

[35] M. Wenzel, B. Wolff, Building formal method tools in the Isabelle/Isar framework, in: K. Schneider, J. Brandt (Eds.), TPHOLs 2007, no. 4732 in LNCS, Springer, 2007, pp. 352–367. doi:10.1007/978-3-540-74591-4_26.

[36] F. Haftmann, T. Nipkow, Code generation via higher-order rewrite systems, in: M. Blume, N. Kobayashi, G. Vidal (Eds.), Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings, Vol. 6009 of Lecture Notes in Computer Science, Springer, 2010, pp. 103–117. doi:10.1007/978-3-642-12251-4_9.
URL https://doi.org/10.1007/978-3-642-12251-4_9

[37] K. Aehlig, F. Haftmann, T. Nipkow, A compiled implementation of normalisation by evaluation, J. Funct. Program. 22 (1) (2012) 9–30. doi:10.1017/S0956796812000019.
URL https://doi.org/10.1017/S0956796812000019

[38] A. V. Hess, S. Mödersheim, A. D. Brucker, A. Schlichtkrull, Performing security proofs of stateful protocols, in: 34th IEEE Computer Security Foundations Symposium (CSF), Vol. 1, IEEE, 2021, pp. 143–158. doi:10.1109/CSF51468.2021.00006.
URL https://www.brucker.ch/bibliography/abstract/hess.ea-performing-2021

[39] M. R. Quillian, Word concepts: A theory and simulation of some basic semantic capabilities, Behavioral Science 12 (5) (1967) 410–430. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/bs.3830120511, doi:https://doi.org/10.1002/bs.3830120511.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/bs.3830120511

[40] M. Minsky, A framework for representing knowledge, Tech. rep., USA (1974).

[41] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider, The Description Logic Handbook: Theory, Implementation and Applications, 2nd Edition, Cambridge University Press. doi:10.1017/CBO9780511711787.

[42] R. Dapoigny, P. Barlatier, Modeling ontological structures with type classes in coq, in: H. D. Pfeiffer, D. I. Ignatov, J. Poelmans, N. Gadiraju (Eds.), Conceptual Structures for STEM Research and Education, 20th International Conference on Conceptual Structures, ICCS 2013, Mumbai, India, January 10-12, 2013. Proceedings, Vol. 7735 of Lecture Notes in Computer Science, Springer, 2013, pp. 135–152. doi:10.1007/978-3-642-35786-2_11.
URL https://doi.org/10.1007/978-3-642-35786-2_11

[43] G. Guizzardi, H. Herre, G. Wagner, On the general ontological foundations of conceptual modeling, in: S. Spaccapietra, S. T. March, Y. Kambayashi (Eds.), Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings, Vol. 2503 of Lecture Notes in Computer Science, Springer, 2002, pp. 65–78. doi:10.1007/3-540-45816-6_15.
URL https://doi.org/10.1007/3-540-45816-6_15

[44] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, J. ACM 42 (4) (1995) 741–843. doi:10.1145/210332.210335.
URL https://doi.org/10.1145/210332.210335

[45] G. Yang, M. Kifer, C. Zhao, Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web, in: R. Meersman, Z. Tari, D. C. Schmidt (Eds.), On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003, Vol. 2888 of Lecture Notes in Computer Science, Springer, 2003, pp. 671–688. doi:10.1007/978-3-540-39964-3_43.
URL https://doi.org/10.1007/978-3-540-39964-3_43

[46] S. Decker, M. Erdmann, D. Fensel, R. Studer, Ontobroker: Ontology based access to distributed and semi-structured information, in: R. Meersman, Z. Tari, S. M. Stevens (Eds.), Database Semantics - Semantic Issues in Multimedia Systems, IFIP TC2/WG2.6 Eighth Working Conference on Database Semantics (DS-8), Rotorua, New Zealand, January 4-8, 1999, Vol. 138 of IFIP Conference Proceedings, Kluwer, 1999, pp. 351–369.

[47] B. N. Grosof, M. Kifer, T. Swift, P. Fodor, J. Bloomfield, Ergo: A quest for declarativity in logic programming, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, Vol. 13900 of Lecture Notes in Computer Science, Springer, 2023, pp.

224–236. doi:10.1007/978-3-031-35254-6_18.
URL https://doi.org/10.1007/978-3-031-35254-6_18

[48] D. Brickley, R. Guha, RDF schema 1.1, W3C recommendation, W3C, https://www.w3.org/TR/2014/REC-rdf-schema-20140225/ (Feb. 2014).

[49] M. Krötzsch, P. Hitzler, B. Parsia, P. Patel-Schneider, S. Rudolph, OWL 2 web ontology language primer (second edition), W3C recommendation, W3C, https://www.w3.org/TR/2012/REC-owl2-primer-20121211/ (Dec. 2012).

[50] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, W3C recommendation, W3C, https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (Jan. 2008).

[51] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosofand, M. Dean, SWRL: A semantic web rule language combining OWL and RuleML, W3C Member Submission, last access on Dez 2008 at: http://www.w3.org/Submission/SWRL/ (May 2004).
URL http://www.w3.org/Submission/SWRL/

[52] M. J. O'Connor, A. K. Das, SQWRL: A query language for OWL, in: R. Hoekstra, P. F. Patel-Schneider (Eds.), Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009, Vol. 529 of CEUR Workshop Proceedings, CEUR-WS.org, 2009.
URL https://ceur-ws.org/Vol-529/owled2009_submission_42.pdf

[53] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, J. Web Semant. 5 (2) (2007) 51–53. doi:10.1016/J.WEBSEM.2007.03.004.
URL https://doi.org/10.1016/j.websem.2007.03.004

[54] D. Tsarkov, I. Horrocks, Fact++ description logic reasoner: System description, in: U. Furbach, N. Shankar (Eds.), Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, Vol. 4130 of Lecture Notes in Computer Science, Springer, 2006, pp. 292–297. doi:10.1007/11814771_26.
URL https://doi.org/10.1007/11814771_26

[55] N. F. Noy, M. A. Musen, The PROMPT suite: interactive tools for ontology merging and mapping, Int. J. Hum. Comput. Stud. 59 (6) (2003) 983–1024. doi:10.1016/J.IJHCS.2003.08.002.
URL https://doi.org/10.1016/j.ijhcs.2003.08.002

[56] J. Euzenat, P. Shvaiko, Ontology Matching, Second Edition., Springer, 2013. doi:10.1007/978-3-642-38721-0.

[57] Web service modeling language (wsml), Tech. rep., W3C, https://www.w3.org/submissions/WSML/ (Jun. 2005).

[58] B. N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: combining logic programs with description logic, in: G. Hencsey, B. White, Y. R. Chen, L. Kovács, S. Lawrence (Eds.), Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003, ACM, 2003, pp. 48–57. doi:10.1145/775152.775160.
URL https://doi.org/10.1145/775152.775160

[59] G. i. Stephan, H. i. Pascal, A. i. Andreas, Knowledge Representation and Ontologies, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 51–105. doi:10.1007/3-540-70894-4_3.
URL https://doi.org/10.1007/3-540-70894-4_3

[60] ISO Central Secretary, Industrial automation systems and integration - parts library - part 1: Overview and fundamental principles, Standard ISO 13584-1:2001, International Organization for Standardization, Geneva, CH (2001).
URL https://www.iso.org/standard/25103.html

[61] ISO Central Secretary, Industrial automation systems and integration - parts library - part 24: Logical resource: Logical model of supplier library, Standard ISO 13584-24:2003, International Organization for Standardization, Geneva, CH (2003).
URL https://www.iso.org/standard/34070.html

[62] G. Pierra, Context-explication in conceptual ontologies: the PLIB approach, in: R. Jardim-Gonçalves, J. Cha, A. Steiger-Garção (Eds.), Enhanced Interoperable Systems. Proceedings of the 10th ISPE International Conference on Concurrent Engineering (ISPE CE 2003), July 26-30, 2003, Madeira, Portugal, A. A. Balkema Publishers, 2003, pp. 243–253.

[63] ISO Central Secretary, Industrial automation systems and integration - product data representation and exchange - part 11: Description methods: The express language reference manual, Standard ISO 10303-11:2004, International Organization for Standardization, Geneva, CH (2003).
URL https://www.iso.org/standard/38047.html

[64] D. Schenck, P. Wilson, Information Modeling: The EXPRESS Way, Oxford University Press, 1994. doi:10.1093/oso/9780195087147.001.0001.
URL https://doi.org/10.1093/oso/9780195087147.001.0001

[65] Y. A. Ameur, F. Besnard, P. Girard, G. Pierra, J. Potier, Formal specification and metaprogramming in the EXPRESS language, in: SEKE'95, The 7th International Conference on Software Engineering and Knowledge Engineering, June 22-24, 1995, Rockville, Maryland, USA, Proceedings, Knowledge Systems Institute, 1995, pp. 181–188.

[66] S. Jean, G. Pierra, Y. A. Ameur, Domain ontologies: A database-oriented analysis, in: J. Filipe, J. Cordeiro, V. Pedrosa (Eds.), Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers, Vol. 1 of Lecture Notes in Business Information Processing, Springer, 2006, pp. 238–254. `doi:10.1007/978-3-540-74063-6\_19`.
URL `https://doi.org/10.1007/978-3-540-74063-6_19`

[67] Y. A. Ameur, D. Méry, Making explicit domain knowledge in formal system development, Sci. Comput. Program. 121 (2016) 100–127. `doi:10.1016/J.SCICO.2015.12.004`.
URL `https://doi.org/10.1016/j.scico.2015.12.004`

[68] J. Abrial, L. Mussat, Introducing dynamic constraints in B, in: D. Bert (Ed.), B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings, Vol. 1393 of Lecture Notes in Computer Science, Springer, 1998, pp. 83–128. `doi:10.1007/BFB0053357`.
URL `https://doi.org/10.1007/BFb0053357`

[69] K. Hacid, Y. A. Ameur, Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, Vol. 9952 of Lecture Notes in Computer Science, 2016, pp. 340–357. `doi:10.1007/978-3-319-47166-2\_24`.
URL `https://doi.org/10.1007/978-3-319-47166-2_24`

[70] L. Mohand-Oussaïd, I. Aït-Sadoune, Formal modelling of domain constraints in event-b, in: Y. Ouhammou, M. Ivanovic, A. Abelló, L. Bellatreche (Eds.), Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings, Vol. 10563 of Lecture Notes in Computer Science, Springer, 2017, pp. 153–166. `doi:10.1007/978-3-319-66854-3\_12`.
URL `https://doi.org/10.1007/978-3-319-66854-3_12`

[71] I. Aït-Sadoune, L. Mohand-Oussaïd, Building formal semantic domain model: An event-b based approach, in: K. Schewe, N. K. Singh (Eds.), Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse, France, October 28-31, 2019, Proceedings, Vol. 11815 of Lecture Notes in Computer Science, Springer, 2019, pp. 140–155. doi:10.1007/978-3-030-32065-2_10.
URL https://doi.org/10.1007/978-3-030-32065-2_10

[72] ISO Central Secretary, Industrial automation systems and integration - parts library - part 32: Implementation resources: Ontoml: Product ontology markup language, Standard ISO 13584-32:2010, International Organization for Standardization, Geneva, CH (2010).
URL https://www.iso.org/standard/50639.html

[73] I. Mendil, Y. Aït-Ameur, N. K. Singh, G. Dupont, D. Méry, P. A. Palanque, Formal domain-driven system development in event-b: Application to interactive critical systems, J. Syst. Archit. 135 (2023) 102798. doi:10.1016/J.SYSARC.2022.102798.
URL https://doi.org/10.1016/j.sysarc.2022.102798

[74] M. J. Butler, I. Maamria, Practical theory extension in event-b, in: Z. Liu, J. Woodcock, H. Zhu (Eds.), Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday, Vol. 8051 of Lecture Notes in Computer Science, Springer, 2013, pp. 67–81. doi:10.1007/978-3-642-39698-4_5.
URL https://doi.org/10.1007/978-3-642-39698-4_5

[75] I. Mendil, Y. Aït-Ameur, N. K. Singh, D. Méry, P. A. Palanque, Standard conformance-by-construction with event-b, in: Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS, Paris, France, Vol. 12863 of LNCS, Springer, 2021, pp. 126–146. doi:10.1007/978-3-030-85248-1_8.

[76] S. J. T. Fotso, M. Frappier, R. Laleau, A. Mammar, Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach, in: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ, Southampton, UK, Vol. 10817 of LLNCS, Springer, 2018, pp. 262–276. doi:10.1007/978-3-319-91271-4_18.

[77] S. Foster, Y. Nemouchi, M. Gleirscher, R. Wei, T. Kelly, Integration of formal proof into unified assurance cases with isabelle/sacm, Formal Aspects Comput. 33 (6) (2021) 855–884. doi:10.1007/s00165-021-00537-4.
URL https://doi.org/10.1007/s00165-021-00537-4

[78] D. P. Friedman, M. Wand, Reification: Reflection without metaphysics, in: R. S. Boyer, E. S. Schneider, G. L. S. Jr. (Eds.), Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984, ACM, 1984, pp. 348–355. doi:10.1145/800055.802051.
URL https://doi.org/10.1145/800055.802051

[79] P. Riviere, N. K. Singh, Y. A. Ameur, EB4EB: A framework for reflexive event-b, in: 26th International Conference on Engineering of Complex Computer Systems, ICECCS 2022, Hiroshima, Japan, March 26-30, 2022, IEEE, 2022, pp. 71–80. doi:10.1109/ICECCS54210.2022.00017.
URL https://doi.org/10.1109/ICECCS54210.2022.00017

[80] P. Rivière, N. K. Singh, Y. Aït-Ameur, Reflexive event-b: Semantics and correctness the eb4eb framework, IEEE Transactions on Reliability (2022) 1–16doi:10.1109/TR.2022.3219649.

[81] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, T. Winterhalter, The metacoq project, J. Autom. Reason. 64 (5) (2020) 947–999. doi:10.1007/S10817-019-09540-0.
URL https://doi.org/10.1007/s10817-019-09540-0

[82] D. Annenkov, J. B. Nielsen, B. Spitters, Concert: a smart contract certification framework in coq, in: J. Blanchette, C. Hritcu (Eds.), Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020, ACM, 2020, pp. 215–228. doi:10.1145/3372885.3373829.
URL https://doi.org/10.1145/3372885.3373829

[83] B. Venners, J. Gosling, Visualizing with JavaDoc, https://www.artima.com/articles/analyze-this#part3, [Online on artima.com; accessed 23-02-2023] (2003).

[84] O. Corp., The Java API Documentation Generator, https://docs.oracle.com/javase/1.5.0/docs/tool, [Online on artima.com; accessed 23-02-2023] (2011).

[85] I. N. de Recherche en Informatique et en Automatique, The OCaml Manual - Release 5, https://v2.ocaml.org/manual/ocamldoc.html, [Online on artima.com; accessed 23-02-2023] (2022).

[86] M.Eberl and G. Klein and A. Lochbihler and T. Nipkow and L. Paulson and R. Thiemann (eds), Archive of Formal Proofs, https://afp-isa.org, Accessed: 2022-03-15 (2022).

[87] M. Kohlhase, F. Rabe, Experiences from exporting major proof assistant libraries, J. Autom. Reason. 65 (8) (2021) 1265–1298. doi:10.1007/s10817-021-09604-0.
URL https://doi.org/10.1007/s10817-021-09604-0

[88] I. Horrocks, P. F. Patel-Schneider, A proposal for an owl rules language, in: S. I. Feldman, M. Uretsky, M. Najork, C. E. Wills (Eds.), Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, ACM, 2004, pp. 723–731. doi:10.1145/988672.988771.
URL https://doi.org/10.1145/988672.988771

[89] M. Krötzsch, S. Rudolph, P. Hitzler, Description logic rules, in: M. Ghallab, C. D. Spyropoulos, N. Fakotakis, N. M. Avouris (Eds.), ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, Vol. 178 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 80–84. doi:10.3233/978-1-58603-891-5-80.
URL https://doi.org/10.3233/978-1-58603-891-5-80

[90] T. Nipkow, Functional automata, Archive of Formal Proofshttps://isa-afp.org/entries/Functional-Automata.html, Formal proof development (March 2004).

[91] Eclipse Foundation, Atl - a model transformation technology, Accessed: 2022-03-15.
URL https://www.eclipse.org/atl/

[92] J. Breitner, B. Huffman, N. Mitchell, C. Sternagel, Certified hlints with isabelle/holcf-prelude, CoRR abs/1306.1340 (2013). arXiv:1306.1340.
URL http://arxiv.org/abs/1306.1340

[93] P. Buneman, S. Khanna, W. C. Tan, Why and where: A characterization of data provenance, in: J. V. den Bussche, V. Vianu (Eds.), Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings, Vol. 1973 of Lecture Notes in Computer Science, Springer, 2001, pp. 316–330. doi:10.1007/3-540-44503-X_20.
URL https://doi.org/10.1007/3-540-44503-X_20

[94] J. Cheney, L. Chiticariu, W. C. Tan, Provenance in databases: Why, how, and where, Found. Trends Databases 1 (4) (2009) 379–474. doi:10.1561/1900000006.
URL https://doi.org/10.1561/1900000006

[95] T. Imieliński, W. Lipski, Incomplete information in relational databases, J. ACM 31 (4) (1984) 761–791. doi:10.1145/1634.1886.
URL https://doi.org/10.1145/1634.1886

[96] Y. Amsterdamer, D. Deutch, V. Tannen, Provenance for aggregate queries, in: M. Lenzerini, T. Schwentick (Eds.), Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece, ACM, 2011, pp. 153–164. doi:10.1145/1989284.1989302.
URL https://doi.org/10.1145/1989284.1989302

[97] T. J. Green, G. Karvounarakis, V. Tannen, Provenance semirings, in: L. Libkin (Ed.), Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China, ACM, 2007, pp. 31–40. doi:10.1145/1265530.1265535.
URL https://doi.org/10.1145/1265530.1265535

[98] P. Senellart, Provenance and probabilities in relational databases, SIGMOD Rec. 46 (4) (2017) 5–15. doi:10.1145/3186549.3186551.
URL https://doi.org/10.1145/3186549.3186551

[99] F. Haftmann, A. Krauss, O. Kuncar, T. Nipkow, Data refinement in isabelle/hol, in: S. Blazy, C. Paulin-Mohring, D. Pichardie (Eds.), Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings, Vol. 7998 of Lecture Notes in Computer Science, Springer, 2013, pp. 100–115. doi:10.1007/978-3-642-39634-2_10.
URL https://doi.org/10.1007/978-3-642-39634-2_10

[100] B. Huffman, O. Kuncar, Lifting and transfer: A modular design for quotients in isabelle/hol, in: G. Gonthier, M. Norrish (Eds.), Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings, Vol. 8307 of Lecture Notes in Computer Science, Springer, 2013, pp. 131–146. doi:10.1007/978-3-319-03545-1_9.
URL https://doi.org/10.1007/978-3-319-03545-1_9

[101] C. Keller, B. Werner, Importing HOL light into coq, in: M. Kaufmann, L. C. Paulson (Eds.), Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings, Vol. 6172 of Lecture Notes in Computer Science, Springer, 2010, pp. 307–322. doi:10.1007/978-3-642-14052-5_22.
URL https://doi.org/10.1007/978-3-642-14052-5_22

[102] D. Delahaye, A tactic language for the system coq, in: M. Parigot, A. Voronkov (Eds.), Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings, Vol. 1955 of Lecture Notes in Computer Science, Springer, 2000, pp. 85–95. doi:10.1007/3-540-44404-1_7.
URL https://doi.org/10.1007/3-540-44404-1_7

[103] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, V. Vafeiadis, Mtac: a monad for typed tactic programming in coq, in: G. Morrisett, T. Uustalu (Eds.), ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013, ACM, 2013, pp. 87–100. doi:10.1145/2500365.2500579.
URL https://doi.org/10.1145/2500365.2500579

[104] R. Cauderlier, Tactics and certificates in meta dedukti, in: J. Avigad, A. Mahboubi (Eds.), Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings, Vol. 10895 of Lecture Notes in Computer Science, Springer, 2018, pp. 142–159. doi:10.1007/978-3-319-94821-8_9.
URL https://doi.org/10.1007/978-3-319-94821-8_9

[105] D. Matichuk, T. C. Murray, M. Wenzel, Eisbach: A proof method language for isabelle, J. Autom. Reason. 56 (3) (2016) 261–282. doi:10.1007/S10817-015-9360-2.
URL https://doi.org/10.1007/s10817-015-9360-2

[106] D. Matichuk, T. C. Murray, M. Wenzel, The Eisbach User Manual, part of the Isabelle distribution. (2023).

[107] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer, The lean theorem prover (system description), in: A. P. Felty, A. Middeldorp (Eds.), Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, Vol. 9195 of Lecture Notes in Computer Science, Springer, 2015, pp. 378–388. doi:10.1007/978-3-319-21401-6_26.
URL https://doi.org/10.1007/978-3-319-21401-6_26

[108] N. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem, Indagationes Mathematicae (Proceedings) 75 (5) (1972) 381–392. doi:https://doi.org/10.1016/1385-7258(72)90034-0.

URL https://www.sciencedirect.com/science/article/pii/1385725872900340

[109] S. Taha, L. Ye, B. Wolff, HOL-CSP Version 2.0, Archive of Formal Proofshttp://isa-afp.org/entries/HOL-CSP.html (Apr. 2019).

[110] A. D. Brucker, B. Wolff, An extensible encoding of object-oriented data models in hol, J. Autom. Reason. 41 (3-4) (2008) 219–249. doi:10.1007/S10817-008-9108-3.
URL https://doi.org/10.1007/s10817-008-9108-3

[111] F. A. Wardani, K. Chaudhuri, D. Miller, Formal reasoning using distributed assertions, in: U. Sattler, M. Suda (Eds.), Frontiers of Combining Systems - 14th International Symposium, FroCoS 2023, Prague, Czech Republic, September 20-22, 2023, Proceedings, Vol. 14279 of Lecture Notes in Computer Science, Springer, 2023, pp. 176–194. doi:10.1007/978-3-031-43369-6_10.
URL https://doi.org/10.1007/978-3-031-43369-6_10