Expressing predicate subtyping in computational logical frameworks

Gabriel Hondet

université
PARIS-SACLAY

Expressing predicate subtyping in computational logical frameworks

Expression du sous-typage par prédicats dans les cadres logiques calculatoires

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique, Référent : ENS Paris-Saclay

Thèse préparée dans l'unité de recherche Laboratoire Méthodes Formelles (Université Paris-Saclay, CNRS, ENS Paris-Saclay)
sous la direction de Frédéric Blanqui, chargé de recherche HDR
et le co-encadrement de Gilles Dowek, directeur de recherche HDR.

Thèse soutenue à Paris-Saclay, le 27 Septembre 2022, par

Gabriel Hondet

Composition du jury

Dale Miller Chargé de recherche, Inria Saclay	Président
Natarajan Shankar Distinguished senior scientist, SRI International	Rapporteur & examinateur
Aaron Stump Full professor, University of Iowa	Rapporteur & examinateur
Serenella Cerrito Professeure, Université d'Evry	Examinatrice
Assia Mahboubi Directrice de recherche HDR, Inria Nantes	Examinatrice
Enrico Tassi Chargé de recherche, Inria Sophia Antipolis	Examinateur
Frédéric Blanqui Chargé de recherche HDR, Inria Saclay	Directeur de thèse

THESE DE DOCTORAT

NNT : 2022UPASG070

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Expression du sous-typage par prédicats dans les cadres logiques calculatoires
Mots clés : interopérabilité, cadre logique, preuve formelle, réécriture, Dedukti, PVS

Résumé : Le typage permet d'apporter de la sûreté dans la programmation, et il est utilisé au cœur de la majorité des systèmes de preuve. Plus un système de types est expressif, plus il est aisé d'y encoder des invariants qui seront vérifiés mécaniquement lors du typage. Le sous-typage par prédicats est une extension des types simples dans laquelle les types peuvent dépendre de prédicats. Un sous-type {x : A | P(x)} est habité par les éléments t de type A pour lesquels P(t) est vrai. Cette extension fournit un système de type riche et intuitif mais indécidable.

Cet ouvrage est dédié à l'encodage du sous-typage par prédicats dans *Dedukti*, un cadre logique avec règles de calcul. On commence par encoder une version *explicite* du sous-typage par prédicats dans lequel on distingue les habitants des sous-types des habitants des types.

Le sous-typage par prédicat est souvent utilisé de manière implicite, sans différence syntaxique entre les habitants de A et ceux de {x : A | P(x)}. On enrichit le cadre logique avec un système de raffinement des termes afin d'ajouter ajouter ces marqueurs syntaxiques et expliciter le sous-typage au sein des termes. Cette transformation est appliquée à la bibliothèque standard de *PVS*, un assistant de preuve centré sur le sous-typage par prédicats.

Title : Expressing predicate subtyping in computational logical frameworks
Keywords : interoperability, logical framework, formal proof, rewriting, Dedukti, PVS

Abstract : Safe programming as well as most proof systems rely on typing. The more a type system is expressive, the more these types can be used to encode invariants which are therefore verified mechanically through type checking procedures. Predicate subtyping extends simple type theory by allowing terms to be defined by predicates. A predicate subtype {x : A | P(x)} is inhabited by terms t of type A for which P(t) holds. This extension provides a rich and intuitive but undecidable type system.

This work is dedicated to the encoding of predicate subtyping in *Dedukti* : a logical framework with computation rules. We begin by encoding *explicit* predicate subtyping for which terms of type A are syntactically different from terms of type {x : A | P(x)}.

Predicate subtyping, is often used implicitly, with no syntactic difference between terms of type A and terms of type {x : A | P(x)}. We enrich our logical framework with a term refiner which can add these syntactic markers in order to explicit subtyping in terms. This transformation is used to translate the standard library of *PVS*, a proof assistant using extensively predicate subtyping, to *Dedukti*.

Contents

Acknowledgements

En premier lieu, je remercie mes encadrants, Frédéric et Gilles. Il est clair que ce manuscrit n'aurait pas cette allure sans vous. J'ai beaucoup profité de votre capacité à aborder tout problème, ce qui m'a à plusieurs reprises permis de prendre des routes moins sinueuses pour parvenir à mes fins.

I am very grateful to Shankar and Aaron who reviewed this manuscript. I also thank warmly Dale, Assia, Enrico and Serenella for having accepted to be part of my jury.

I'd like to express my gratitude towards Stéphane Graham-Lengrand and Maria Paola Bonacina, who made my stay at SRI International both possible and delightful. I also thank Sam Owre for proofreading my Lisp code and providing some very valuable help with it.

Je remercie les permanent·es du laboratoire avec qui j'ai interagi, Chantal, Valentin, Bruno, Stéphane, Caroline, Serge ; aussi bien pour des considérations scientifiques que sociales.

Je remercie mes collègues doctorant·es et post-doctorant·es, avec qui j'ai partagé au quotidien mes peines et mes réjouissances, qu'elles soient scientifiques ou non. Parmi celles et ceux là, je remercie d'abord les aînés, François, Gaspard, Guillaume, Yacine et Franck, qui m'ont chaleureusement accueilli dans l'équipe, et qui, au cours des pauses café, se sont révélés être des enseignants inestimables. Je pense sincèrement que ce manuscrit et moi vous devons beaucoup. Je remercie aussi Aliaume, Emilie et Nathan, avec qui j'ai commencé mon stage et ma thèse, et avec qui j'ai partagé un bureau, des pauses café et des concerts, jusqu'à ce que le COVID nous repousse chez nous. Je remercie chaleureusement Giann-Karlo qui a partagé avec moi nos nouveaux bureaux à Gif-sur-Yvette ; et enfin Amélie, Fabricio, Louise, Luc, Pierre, Thiago et Yoan. Je remercie Monsieur Färber, avec qui j'ai partagé un COVID, pour sa bonne humeur contagieuse et semble-t-il insubmersible. Il me tarde de jouer à nouveau la Pavane avec vous.

Je remercie aussi les stagiaires et autres ni doctorant·es, ni post-doctorant·es ni permanent·es que j'ai pu voir, pour avoir amené de la fraîcheur et de la nouveauté dans l'équipe, je pense à Amélie, Corentin, Elliot, Émile, Houda, Loris, Quentin, Thomas ; et en particulier à Taïssir.

Parce qu'il n'y a pas que la science dans la vie, je remercie toutes les personnes extra-académiques qui m'ont accompagné pendant ces trois années, que ce soient mes parents qui ont assuré la logistique, mon frère ou Margaux. Je remercie aussi Hubert, pour m'avoir permis, chaque semaine de penser à autre choses que mes problèmes de typage, de terminaison ou autres ; bien que la construction des 12 gammes majeures reste une tâche assez calculatoire. Je remercie Choupine et Lilou d'avoir égayé mes journées de télétravail. Et enfin, je remercie Emilie d'avoir fait de cette période une expérience aussi stimulante intellectuellement qu'humainement.

If you feel that I should have thanked you, but I've not; it's highly likely that I've simply forgotten. My sincere apologies, and thank you.

Chapter 1

Introduction

Regardless of one's personal opinion of René Descartes's philosophy, it may not be exaggerated to say that his *Discours de la méthode*[1] (Descartes 1637) has shaped scientific methodology of all western cultures. In the second part, Descartes establishes four principles by which one should abide to establish unquestionable facts. The first and second principles read as follows

> Le premier [principe] était [...] de ne comprendre rien de plus en mes jugements, que ce qui se présenterait si clairement et si distinctement à mon esprit que je n'eusse aucune occasion de le mettre en doute.
>
> Le second, de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourroit, et qu'il seroit requis pour les mieux résoudre.

and in English

> The first [law] was [...] to comprise nothing more in my judgement than what was presented to my mind so clearly and distinctly as to exclude all ground of doubt.
>
> The second, to divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution.

[1] *Discourse on the Method of Rightly Conducting One's Reason and of Seeking Truth in the Sciences*, translations are by John Veitch

Descartes acknowledges that these ideas are not novel and come from his understanding of geometry proofs, but he extends this procedure to a general method to establish truth regardless of the discipline. This methodology has been called reductionism, as opposed to holism.

Coupled with a formal language, reductionism became *de facto* prevalent to provide foundations for mathematics. Mathematical foundations are formulated by a limited number of unquestionable *axioms*. For instance, during the 19th century, Giuseppe Peano proposed an axiomatic formulation of arithmetic with natural numbers (Peano 1889). In his (emblematic) theory, the proposition '$2 + 2 = 4$' can be proved using the axiom of transitivity 'if $x = y$ and $y = z$, then $x = z$' and the definition of addition '$x + (y + 1) = (x + y) + 1$' and '$x + 0 = x$'.

However, a theorem as simple as '$2 + 2 = 4$', already consumes no less than five inference steps. Considering the 'complexity' and the relatively low intellectual impact of the proposition, it seems almost impossible to formalise substantial mathematical developments as they would quickly become overwhelming and human beings could not possibly review or write them without making a mistake.

Fortunately, machines able to repeat simple but tedious logical operations *ad nauseam* have since been invented. When fed with proofs written in some language they understand, these machines can be programmed to mechanically check that these proofs are nothing more than successive applications of axioms on which programmers and mathematicians agreed. In the late 60s, de Bruijn designed the formal system *Automath* (H. Barendregt and Rezus 1983) around an automatic proof checker that verifies the correctness of specifications written in a formal language.

While automatic proof checkers completely transferred the verification to computers, writing proofs—especially formal proofs—remains pedestrian. To sort this problem out, Milner 1972 designed the interactive system *LCF*. Its proof checker can be used by human beings to generate formal proofs, not only to check them once finished. The automated reasoning system *Nqthm* by Boyer and Moore 1979[2] inherited the interactive aspect of *LCF* and has been used to check over 16,000 theorems.

While formalising mathematics, one establishes propositions and theorems to describe the behaviour of the notions he or she introduces. Proofs are given afterward to assert that propositions hold. Therefore, we claim that mathematics initially consider propositions as the primary building block. General purpose

[2]Also called the 'Boyer-Moore Theorem Prover'.

programming can be seen as the opposite approach: one describes procedures and calculations, and then attributes meaning to these calculations. Indeed, it is easy to write syntactically correct but semantically incorrect expressions, such as '$3 \times$ `true`' in general purpose programming languages. To reject nonsensical computations, *types* that represent meaning are attributed to values. In the previous example, 3 is an integer, which may be denoted '$3 :$ `int`', `true` is a boolean, denoted '`true` : `bool`' and $\times$ is a function whose type may be denoted '$\times :$ `int` $\rightarrow$ `int` $\rightarrow$ `int`'. Typing a program ensures it has a meaning according to a *type system* (Pierce 2002). Verifying that a program has a given meaning is the same as checking that a proof proves a given proposition[3].

1.1 Predicate subtyping

1.1.1 Subtyping for more understandable developments

Type systems must find a balance between rigidity, which allows to define precisely the meaning of expressions, and flexibility, which eases the task of writing expressions that are understood by the type system. It is difficult to translate one's ideas into type system that is too rigid: the translation of one's thoughts into a language understood by a computer is likely to be difficult, as well as the debugging. On the contrary, it is easy to write expressions that make sense to humans in system that is too flexible, but in turn less meaning is conveyed to the computer and its ability to spot erroneous statements becomes limited.

Subtyping (Cardelli 1984) makes type systems more flexible. It allows to structure types of expressions hierarchically: different meanings can be attributed to the same expression, especially when some are generalisations of others. A magpie is a bird as well as an animal, a geranium is a flowering plant as well as a vegetable &c. Hierarchical organisation may be particularly well suited for the expression of human-made concepts: our own memory can be modelled by semantic networks whose nodes are hierarchically organised (Collins and Quillian 1969).

The success (in the sense of adoption) of object oriented programming may support our claim. Object oriented languages (Abadi and Cardelli 1995) allow a relatively straightforward (in comparison to other formalisms) encoding of the concepts industries deal with. Any concept can be modelled by a class, and its relationships with other concepts can be modelled by subtyping. In order

[3]This equivalence between proofs and programs and between types and propositions has been called the 'propositions-as-type principle', or 'Curry-Howard correspondence'.

to encode systems people deal with, object oriented systems tend to have very liberal subtyping. Liberal to the point that one may lose the intuition behind subtyping (AbdelGawad 2014; Cargill 1991).

Predicate subtyping (J. M. Rushby, Owre and Shankar 1998) allows one to classify expressions by the properties they validate. It can be straightforwardly interpreted as set comprehension: the set of roots of unity is classically defined by $\{z : \mathbf{C} \mid \exists n, z^n = 1\}$, which is a predicate subtype whose support is $\mathbf{C}$ (the set of complex numbers) and whose predicate is $\exists n, z^n = 1$. When we manipulate a root of unity z, we can conjugate z to $\bar{z}$ because z is *also* a complex number. This ability to consider z as a root of unity and a complex number is precisely what subtyping allows. Therefore, just like object oriented languages allow to model concepts easily, predicate subtyping is an intuitive encoding of mathematics as we learned them. It is also valuable when writing program specifications: guards, preconditions and postconditions are easily expressed as predicate subtypes. For instance, a function that pops an element from a stack must take as argument a non-empty stack; the domain of that function may thus be $\{s : \texttt{stack} \mid \neg\texttt{empty?}(s)\}$.

Because predicate subtyping entangles proof checking with type checking, it provides a very rich type system where subtle invariants may be expressed in order to reject a large number of pathological expressions (such as $\frac{1}{1-1}$). This expressiveness comes at the cost of undecidable type checking. Indeed, for a term t to be judged of type $\{x : A \mid P\}$, the predicate $\{t/x\}\,P$ must be proved; which has no reason *a priori* to be decidable (F. Gilbert 2018).

1.1.2 The Prototype Verification System: an implementation of predicate subtyping

To our knowledge, no proof assistant has chosen predicate subtyping as its paramount feature, except the Prototype Verification System (Owre, J. Rushby et al. 1998; Owre, J. M. Rushby and Shankar 1992), or *PVS* for short. Even though proof assistants whose types can depend on values allow to express types of the form $\{x : A \mid P\}$, most of them do not consider A as a subtype of the former. *PVS* is a specification development environment developed at SRI International based on simple type theory and predicate subtyping. It is made of a specification language parser, a type checker and a theorem prover.

In *PVS*, specifications are split across 'theories'. The validation of a theory is performed in two steps. The first rejects ill-typed expressions not taking subtyping into account: terms of the form `TRUE OR 1` are rejected, but not `1 / 0`. The second phase collects all proof obligations due to predicate subtyping to

generate 'type correctness conditions' (shortened TCC in *PVS* jargon). For instance, type checking the expression $\frac{1}{x}$ generates the type correctness condition $x \neq 0$. This type correctness condition can be solved in an interactive proof mode. To avoid overwhelming users with either redundant or trivial type correctness conditions, a substantial effort has been put in automating theorem proving.

There are mainly two stratagems to automate theorem proving in *PVS*. The first, and most common, is to provide elaborated decision procedures such as binary decision diagrams or satisfiability modulo theory solvers (such as *Yices* (Dutertre 2014) also developed at SRI). The other capitalises on the entanglement of theorem proving and type checking, and is implemented by the system of 'judgements' (Owre, Shankar et al. 2020, page 25). Broadly, judgements extract typing judgements that may be relevant to the theorem prover, such as '$\exp\left(\frac{2i\pi}{3}\right)$ is a root of unity'. Such a judgement will raise a type correctness condition that requires to prove the claim, so that similar type correctness conditions appearing later on may be automatically solved by instantiation of the judgement. Judgements can be a lot more general than that, in order to catch as many redundant type correctness conditions as possible.

The proof theory *PVS* is based on classical sequent calculus (Owre and Shankar 1997b). Proofs are solved by providing tactics operating on a proof state (Shankar et al. 2021). Because type checking and theorem proving are entangled, well-typedness may depend on the provability of some formula which itself may depend on some logical context. For instance, consider the expression

$$x \neq 1 \Rightarrow \frac{1}{1-x} = 1 + x + x^2 + \cdots + x^n + O(x^{n+1})$$

The right-hand side of the implication may be well-typed only if $1 - x \neq 0$ is provable. This can be proved only if the hypothesis $x \neq 1$ is added to the context in which the right-hand side is type checked. Similarly, in the ternary `IF THEN ELSE` construction, the condition must be added to the context to type check the 'then' branch while its negation must be added to the context to type check the 'else' branch.

Thanks to predicate subtyping, *PVS* features a clear distinction between the specification phase which consists in designing objects and concepts as well as their desired properties, from the proving phase where one proves the type correctness conditions raised while typing the specification. This separation is also physical: proofs are saved as tactic scripts in auxiliary files, away from the specification.

1.1.3 A minimal and formalised version of *PVS*

Because *PVS* is a complex system with many features, F. Gilbert 2018 has extracted a minimal and essential core from *PVS* in order to study predicate subtyping. He defined two languages: the vernacular language in which specifications are written is called *PVS-Core*. Type checking is undecidable in *PVS-Core*, it is a minimal version of the specification language of *PVS*. The second language named *PVS-Cert* is a language for certificates of *PVS-Core*. The type system of *PVS-Cert* enjoys many properties, such as strong normalisation and decidability of type checking. Typing derivations of *PVS-Core* can be translated to judgements of *PVS-Cert*. In order to obtain decidable type checking, proofs of type correctness conditions are included in the terms of *PVS-Cert*. Therefore, in *PVS-Cert*, a root of unity is not just a complex, but a complex with a proof that it is a root of unity. We therefore lose subtyping, or at least its implicit aspect. Subtyping is made explicit in the sense that any root of unity can be transformed into a complex number, by forgetting the proof it is a root of unity.

1.2 Logical frameworks for interoperability

In the beginning of the 20th century, diverse formal systems and axiomatic theories have been designed to found mathematics. Some of them were made to palliate defects or paradoxes of their predecessors (see Russell's paradox (Russell 1903) and his subsequent 'type theory' (Whitehead and Russell 1997)) while others were experimental. One may wonder whether two such formal systems agree with each other, whether they have the same truth. One could—or rather should—wonder first whether this question makes sense. Just like two strangers that do not talk the same language cannot exchange ideas (and hence cannot agree, at least consciously); propositions expressed in one formal system may not make sense in the other. Even assuming that two formal systems have the same language, what should we think of propositions that are provable in one system but not in the other? Are there, after all, propositions that are provable in both systems? Metatheoretically, it is very unlikely that two formal systems with approximately the same scope share no proof: what good is a system in which we cannot prove '$2 + 2 = 4$' for mathematics? Even if there seems to be no such thing as 'universal truth', it must nevertheless be the case that formal systems used to express mathematics share some elementary notions. But how can they be related?

For years, the validity of proofs—whether a proof effectively proves what its

authors claim it proves—depended exclusively on whether contemporary experts were convinced the proof holds or not. With axiomatic theories, experts founded their reasoning and the validity of proofs became objective, but still dependent on the attention and perseverance of experts to correctly unfold the axioms (not taking into account that formal proofs were not likely to be written nor unfolded by hand). The mechanisation of proof checking finally allowed the validity of proofs not to depend on the reviewer. On the other hand, we still need reviewers to check that formalised statements make sense. Furthermore, the validity of proofs depends strongly on a new parameter, namely the implementation of the proof checker.

We are faced with two questions:

- Can we avoid relying on a particular and potentially flawed implementation of some formal system to verify our mathematics?
- When a proposition has been proved to hold in some system, can we assume the proposition holds in an other system? Furthermore, how similar are the proofs?

Avizienis 1985 proposes multiversion programming to answer the first question. If we denote by p the probability of introducing an error into the implementation of a formal system, several implementations can be independently developed. When n such implementations agree, the probability of a false positive (or false negative) may be brought down to p^n. However, the efficiency of multiversion programming has been contested by Knight and Leveson 1986 and has become controversial (see Knight and Leveson 1990). For the second question, we have to study and compare logical systems themselves, so we are not using formal systems to prove propositions in them, but we are comparing them and proving propositions *about* them. But in which formal system should we formalise formal systems? If only we could design a formal system expressive enough for propositions and proofs of other systems to be expressed, we could compare propositions themselves, and even proofs. We could check proofs from different formal systems, as long as their axioms are translated into the more expressive logic. The 'predicate logic' of Frege 1879 fulfills these requirements. For instance, set theories have been expressed in it. Such formal systems are called 'logical frameworks' and are designed to define other logics. They strive to remain as weak as possible, in the sense that they provide as few native constructions as possible. Other logical frameworks have been conceived since predicate logic, such as 'Edinburgh's logical framework' (Harper, Honsell and Plotkin 1993) (sometimes written *LF*, *ELF*, $\lambda\Pi$, or $\lambda\mathcal{P}$).

1.3 Contributions

This thesis shows how predicate subtyping can be expressed into a logical framework based on dependent types and equational theories: the λΠ-calculus modulo equations (hereafter λΠme). The study is both theoretic and applied to the expression of *PVS* into *Dedukti* (Deducteam 2022a), an implementation of λΠme. The thesis can be separated into three parts.

1.3.1 Encoding explicit predicate subtyping

Chapter 2 provides a new interpretation of *PVS-Cert* into λΠme, a logical framework where types are identified up to arbitrary congruences (while *LF* identifies types up to reduction of functions). The novelty lies in the encoding of the conversion relation of *PVS-Cert*. It implements a weak form of proof irrelevance by erasing proof terms from the language. This reduction is encoded as a set of equations. These equations may put in relation well-typed and ill-typed terms which is a nuisance when one wants to reason on well-typed terms only. We show that we can get rid of intermediate ill-typed terms when considering an equality between two well-typed terms.

We establish typing preservation for *PVS-Cert*: any well-typed judgement of *PVS-Cert* can be encoded as a well-typed judgement of λΠme.

We then discuss the implementation of a decidable relation in order to obtain decidable type checking. Rewriting is used to provide a confluent relation, but its termination is left as a conjecture, with potential tracks to prove it. The section ends on some preliminary results for the conservativity of the encoding: if two terms are equivalent with respect to the rewriting relation, then they are equivalent with respect to the equations. This property requires a proof because proof irrelevance is implemented in the rewrite relation using new symbols that are not used in the equations. We therefore have to prove that these symbols do not alter proofs of equivalence.

1.3.2 Handling implicit predicate subtyping

Chapters 3 and 4 are dedicated to the management of implicit subtyping. Implicit subtyping may be seen as a coercion insertion problem, motivating the implementation and study of term refiners. We model a coercion system on top of this refiner that is able, in particular, to transform non functional values into functional ones.

We study the implementation of a coercion system using rewrite rules. The rewrite rules are used to implement a function computing the coercion of a term between two types. The rewrite rules required to encode such a function quickly become difficult to study and are known to yield non convergent rewrite systems on untyped terms. Nonetheless we show that such a coercion system allows us to implement sophisticated coercions in our framework. We also need to handle proof obligations, which are modelled with existential variables in the framework. The rewrite relation is extended to allow the generation of such existential variables.

In Chapter 4, we provide a coercion system suitable to type check judgements of λΠme that use implicit predicate subtyping. This coercion system is able to interpret terms of *PVS-Core* expressed in λΠme into terms of *PVS-Cert* expressed into λΠme. In particular, while F. Gilbert 2018 provides an interpretation of complete derivations of *PVS-Core* to *PVS-Cert* (just like the 'Penn-translation' by Tannen et al. 1991), the mechanisms established in this thesis allows us to generate, inside λΠme, well-typed certificates (*i.e.* judgements of *PVS-Cert*) from well-typed judgements of *PVS-Core*. We show a type preservation theorem: well-typed judgements of *PVS-Core* can be refined to well-typed judgements of λΠme. In particular, this theorem requires a proof that the refiner preserves substitution in the sense that substituting before or after refinement should be the same (up to convertibility).

1.3.3 Working with *PVS*

Chapters 5 and 6 present an implementation of the translation from *PVS-Core* to *PVS-Cert* encoded in *Dedukti* and use it to translate the standard library of *PVS* (called 'Prelude'). The Prelude uses more features available in *PVS* than are described in the language encoded so far, such as polymorphism, tuples, overloading &c. We present and discuss additional encodings for these features.

PVS stores proofs as sequences of tactics. Its *Lisp* image is able to rerun these tactic sequences to obtain proof trees. We sketch procedures to generate complete proof terms from these proof trees.

1.4 Related works

Subtyping on sorts has been added to pure type systems in order to obtain an infinite hierarchy of universes that is easier to manipulate. These systems are called cumulative type systems (Barras 1999; Luo 1990; François Thiré 2020).

Predicate subtyping as considered in this work is based on the ideas developed in *PVS* (J. M. Rushby, Owre and Shankar 1998). It has been theoretically studied by Owre and Shankar 1997b, where they formalise a substantial part of *PVS* (including pairs, theories, polymorphism, logical context) and give a set-theoretical interpretation of types and expressions. Although both ibid. and F. Gilbert 2018 formalise a fragment of *PVS*, they do not handle subtyping similarly. Ibid. keeps as much information as possible in the terms of the certificate language, and subtyping is performed in small steps: from a type to its supertype or one of its subtypes. The typing rules given by Owre and Shankar 1997b stipulate that upon application $(f\,a)$, term a must validate all proof obligations from the topmost supertype of the domain of f to the actual domain of f; we can think of a as being coerced from its current type to its topmost type, and then coerced back to the domain of f. Stump 2003 provides another approach to predicate subtyping, closer to the one of Owre and Shankar 1997b than of F. Gilbert 2018. It defines a system $\mathrm{PF}_{\mathrm{sub}}$ which handles partial functions and with an emphasis on the distinction between the type system and the proof system. In particular, there may be expressions that are not typable in *PVS* because of unsolvable type correctness conditions, such as $\frac{1}{i} > 0 \Rightarrow i \neq 0$ (where i is a real number). This expression is typable in $\mathrm{PF}_{\mathrm{sub}}$, and even valid.

Barras and Bernardo 2008 provide a language with implicit constructions whose type checking is not decidable (like *PVS-Core*), and an extraction from the implicit language to an annotated one whose type checking is decidable (like *PVS-Cert*). Just like in *PVS-Cert* the conversion in the decidable type system operates on terms that are stripped of their implicit subterms. An emulation of *PVS* is also provided where predicate subtypes are encoded by dependent pairs whose second component is implicit, which allows to perform proof irrelevance since these proofs are erased—because implicit—by the conversion.

Emulation of predicate subtyping *à la PVS* in other systems has already been attempted several times. Hurd 2001 emulates predicate subtyping in *HOL* using predicates instead of predicate subtypes. This approach suits well the higher order nature of *HOL* but forces subtyping judgements to be stated as theorems rather than typing judgements: while *PVS* can quantify over predicate subtypes, *HOL* cannot quantify over values that validate a predicate. It brings complications to keep the logical context in which terms are type checked.

Predicate subtyping has been encoded in the calculus of constructions by Sozeau 2006. The encoding is similar to *PVS-Cert*, but the conversion relation is much richer. In particular, it includes η-equivalence $f = \lambda x, f\,x$ (when $x \notin f$). The counterpart of *PVS-Core* is the language *Russell*. *Russell* handles predicate subtyping implicitly. Type correctness conditions are replaced

with existential variables which are handled as unknown terms whose type is a proposition. Expressions are written in *Russell* and then translated to the calculus of constructions by an *ad-hoc* coercion insertion algorithm. More technical comparisons will be carried out later on, in particular in Section 3.2.6.

Kaufmann and Moore 1997 provide in *ACL2* a system of 'guards' into its logic based on *Common Lisp*. These guards can mimic the expression of preconditions with predicate subtyping. In *ACL2*, arguments of functions can be guarded by predicates. Functions are said 'gold' when the functions used in their body have their guards validated assuming the guards of the current function. This mechanism replicates logical context we find in *PVS*: a ternary 'if-then-else' expression is gold whenever its 'then' branch is gold assuming the branching condition and its 'else' branch is gold assuming the negation of the branching condition. We can say that a symbol is gold when its type correctness conditions have been solved.

Salvesen and Smith 1988 have studied the introduction of predicate subtyping into Martin-Löf type theory under the name 'subset types'. The subset formation and subset introduction rules are identical to the ones of *PVS-Core*. Subset elimination has been studied in two different formalisms: *intensional* type theory where the equality is definitional and *extensional* type theory where the equality is extensional and undecidable. One of the aims of the authors is to avoid proving the same lemmas several times. For instance, proposition $\forall x \in \{y : A \mid P\}, \{x/y\}\, P$ should not require any lemma: proving $\{x/y\}\, P$ should take advantage of x being in $\{y : A \mid P\}$. Martin-Löf type theory with extensional equality (and subset types) has been implemented in *Nuprl* (Constable et al. 1986).

Cauderlier and Dubois 2014 have expressed object oriented type systems with subtyping in λΠme. Unlike *PVS*, type checking in this type system is decidable, it only performs structural subtyping on record types: a record type A is a subtype of a record type B if the set of projections of A is a superset of the projections of B. That way, any record of type A can be seen as a record of type B. Subtyping is expressed through an explicit coercion function, like in *PVS-Cert*, but the coercion function is more general: it takes two encoded types as arguments, a proof of subtyping, and coerces an element of the former encoded type to the latter one. In the encoded calculus, expressions of type B are bundles containing an encoded type A, an object of type A and a proof that A is a subtype of B. The article proposes an alternative encoding which is more shallow but non terminating.

Refinement types by Lovas and Pfenning 2010 enrich type systems while keeping type checking decidable. Refinement types act as a layer on top of the

base type system that allows to provide more invariants on functions: refinement 'sorts' are not native types. Refinement types can be seen as an intensional version of predicate subtyping: atomic refinements are axiomatised, such as $pos \sqsubset nat$, and the type checker is able to attribute refinement sorts to expressions in canonical forms using these declarations. Interestingly, refinement types can be interpreted by predicates into λΠ with proof irrelevance.

Ferreira and Pientka 2014 show how elaboration can be used in logical frameworks to separate a user-level syntax ('the language of programs') from a kernel-level syntax ('the language of types and terms'). The set of terms is richer than ours as it contains recursive functions and pattern matching. Thanks to elaboration, types and arguments may be omitted from the language, but the elaboration cannot be parametrised.

Pfenning 2001 has studied the interactions between proof irrelevance and intensional and extensional type theories. The type theory presented is an extension of λΠ with different notions of truth: the judgement $\vdash M \div A$ states A is provable, but the proof is hidden (the judgement $\vdash M : A$ provides a proof that is taken into account). Similarly, the function type $\Pi x \div A, B$ has an irrelevant argument. When a function takes a proof irrelevant argument, the application is itself proof irrelevant, allowing the definitional equality to ignore the argument.

Back to Martin-Löf type theory, Abel, Coquand and Pagano 2011 distinguish propositions A from 'proof-irrelevant propositions $\mathsf{Prf}(A)$'. While A can be inhabited by several normal forms, $\mathsf{Prf}(A)$ is inhabited by a single normal form.

Werner 2008 embeds proof irrelevance into the *extended calculus of constructions* of Luo 1990 using the dependent pairs of the calculus. Dependent pair types and right (or second) projections come in two flavours, proof irrelevant or vanilla. The reduction at the heart of the congruence reduces terms tagged irrelevant to a canonical proof ϵ. In this work, it is shown how such proof irrelevance can be used to form a language with explicit coercions closely related to *PVS*.

Proof irrelevance is native in *Lean* (Moura et al. 2015), and *Matita* supports it as well (Asperti, Ricciotti and Coen 2014, Section 9.3). Proof assistants *Coq* and *Agda* (G. Gilbert et al. 2019) have both a specific sort for proof irrelevant propositions (`SProp` for *Coq* and `Prop` for *Agda*).

Aspinall and Compagnoni 2001 have studied subtyping in λΠ, but a translation from λΠ with subtyping to λΠ was left as future work. Such a translation is given in (Tannen et al. 1991) and will be discussed in Chapter 3.

Some earlier forms of predicate subtyping can also be seen in the *OBJ* lan-

guages which implement sub-sorting. Futatsugi et al. 1985 describe how sub-sorting is performed; in particular, sub-sorting declarations such as 'non empty lists is a subtype of lists' are interpreted as coercion operators. Bouhoula, J. Jouannaud and Meseguer 2000 use membership declarations to provide more information to the type checker in order to infer more precise types (or sorts). Membership declarations are similar to the 'judgements' mechanism of *PVS*.

1.5 Notations and definitions

Lower case letters generally stand for objects, and uppercase letters for types, x, y, z, v generally stand for variables, s for sorts, Γ, Δ, Ξ for contexts.

For any set $\mathcal{E}$, any natural number k, we denote $\mathcal{E}^k = \mathcal{E} \times \mathcal{E}^{k-1}$ when $k > 1$ and $\mathcal{E}^1 = \mathcal{E}$; and $\mathcal{E}^* = \bigcup_{k \geq 0} \mathcal{E}^k$.

Hoare triples Given a procedure e that may diverge or fail, the notation (from Hoare 1969) $\{P\}\ e\ \{Q\}$ states that whenever precondition P holds and e terminates without failure then postcondition Q holds as well. Any Hoare triple $\{P\}\ f\ \{Q\}$ can be seen as a *specification* for procedure f, and we say that an implementation of f obeys its specification when it validates the Hoare triple.

Terms λ terms are defined inductively on a countably infinite set of variables $\mathcal{X}$ by

$$t ::= x \in \mathcal{X} \mid t\,t \mid \lambda x : t, t \mid \Pi x : t, t. \tag{1.1}$$

The set of λ-terms is denoted $\mathcal{T}(\mathcal{X})$ (the set of variables will be omitted in general). A dependent product $\Pi x : t, u$ may be written $t \to u$ if u does not contain variable x.

The application of a substitution σ to a term t is denoted σt and $\{u/x\}$ denotes the substitution of term t for variable x.

Vectors (which are considered to be the same as finite sequences) are denoted in bold $\boldsymbol{x}$ or in parentheses $(x_i)_i$ where i is the index. Vectors may be used in substitution $\{\boldsymbol{u}/\boldsymbol{x}\}\,t$ or binders '$\boldsymbol{\lambda x} : \boldsymbol{t}, u$', '$\boldsymbol{\Pi x} : \boldsymbol{t}, u$'. Concatenation and adding are written with the comma ','. Both '$x, \boldsymbol{x}$' and '$\boldsymbol{x}, \boldsymbol{y}$' are correct sequences (and the latter is not a sequence of sequences). Substitution is extended to sequences by $\sigma\boldsymbol{x} = (\sigma x_i)_i$. Contexts can be seen as sequences of pairs of variables and types, hence adding a binding $(x : T)$ to a context Γ is written '$\Gamma, (x : T)$'. The empty context is denoted $\emptyset$.

For any judgement J, we write $\mathcal{D} :: J$ if $\mathcal{D}$ is a derivation tree whose conclusion is judgement J.

Subterms and positions A *position* is a string of natural numbers, ϵ is the empty string, $\mathcal{P}$ is the set of positions, and for any term t in $\mathcal{T}(\mathcal{X})$, $\mathcal{P}(t)$ is the set of positions of t defined as follows:

$$\begin{aligned}
&\mathcal{P}(f\,t_1\ ...\ t_n) = \{\epsilon\} \bigcup_{i=1}^{n} \{i, p \mid p \in \mathcal{P}(t_i)\} \\
&\mathcal{P}(\lambda x : t, u) = \{\epsilon\} \cup \{1, p \mid p \in \mathcal{P}(t)\} \cup \{2, p \mid p \in \mathcal{P}(u)\} \\
&\mathcal{P}(\Pi x : t, u) = \{\epsilon\} \cup \{1, p \mid p \in \mathcal{P}(t)\} \cup \{2, p \mid p \in \mathcal{P}(u)\} \\
&\mathcal{P}(x) = \{\epsilon\} \text{ if } x \in \mathcal{X}.
\end{aligned}$$

For any term t the subterm of t at position $p \in \mathcal{P}(t)$ is defined by

$$\begin{aligned}
&t|_\epsilon = t; && (f\,t_1\ ...\ t_n)|_{i,p} = t_i|_p \\
&(\lambda x : t_1, t_2)|_{i,p} = t_i|_p; && (\Pi x : t_1, t_2)|_{i,p} = t_i|_p.
\end{aligned}$$

For any terms s and t, the *replacement* of the subterm of t at position $p \in \mathcal{P}(t)$ by term s is denoted $\{s/p\}\,t$ and defined as

$$\begin{aligned}
&\{s/\epsilon\}\,t = s \\
&\{s/i, p\}\,(f\,t_1\ ...\ t_n) = (f\,t_1\ ...\ (\{s/p\}\,t_i)\ ...\ t_n) \\
&\{s/1, p\}\,(\lambda x : t, u) = \lambda x : \{s/p\}\,t, u \\
&\{s/2, p\}\,(\lambda x : t, u) = \lambda x : t, \{s/p\}\,u \\
&\{s/1, p\}\,(\Pi x : t, u) = \Pi x : \{s/p\}\,t, u \\
&\{s/2, p\}\,(\Pi x : t, u) = \Pi x : t, \{s/p\}\,u
\end{aligned}$$

The *prefix order* $\leq$ is defined as $p \leq q$ if there is p' such that $p, p' = q$. A position p is *below* position q if $q \leq p$ and *strictly below* q if $q \leq p$ and $q \neq p$ (*above* is defined similarly). Two positions are *disjoint* or *parallel* if they are not comparable with respect to $\leq$.

A *context* is a term with at most one occurrence of a variable $\Box$. If γ is a context, $\gamma[t]$ denotes the substitution of term t for variable $\Box$. When not mentioned, sets of variables for the language do not contain $\Box$ to avoid confusion.

Relations and rewriting For any relation R we note $R^=$ its reflexive closure, R^+ its transitive closure, R^{-1} its inverse and R^* its reflexive and transitive closure. For any relations R and S, we write $RS = \{(t,u) \mid \exists v, t\ R\ v \wedge v\ S\ u\}$ the composition of R and S. For any relation R, two terms s and t are *joinable* denoted $s \downarrow_R t$ whenever there is a term u such that $s\ R^*\ u\ (R^{-1})^*\ t$. The *congruence* of a relation R is the smallest equivalence relation containing R that is closed by context and substitution; it is denoted $\simeq_R$.

An equation is a pair of terms (t,u) denoted $t = u$. A rewrite rule is an equation (t,u) denoted $t \hookrightarrow u$ when t is not a variable and all free variables of u are in t. When R is a rewrite system (a set of rewrite rules), $\hookrightarrow_R$ is the closure by context and substitution of the rewrite rules of R. If $\hookrightarrow$ is a rewrite relation, $\hookrightarrow^=$ denotes its reflexive closure, $\hookleftarrow$ its inverse, $\leftrightarrow$ its symmetric closure, $\hookrightarrow^+$ its transitive closure and $\hookrightarrow^*$ its reflexive and transitive closure.

Definition 1. A rewrite relation $\hookrightarrow$ is called

- *Church-Rosser* when $t \leftrightarrow^* u$ implies $t \downarrow u$.
- *confluent* when $s \hookleftarrow^* t \hookrightarrow^* u$ implies $s \downarrow u$.

Proposition 1. *A relation has the Church-Rosser property if and only if it is confluent.*

Proof. See Baader and Nipkow 1998, Theorem 2.1.5. □

A rewrite relation $\hookrightarrow$ is *terminating* whenever there is no infinite reduction chain $t_0 \hookrightarrow t_1 \hookrightarrow ...$ and it is *convergent* when it is both terminating and confluent.

Given a relation $\leftrightarrow$ that is symmetric, stable by context and stable by substitution (but not transitive), if $\simeq\ = \leftrightarrow^*$ (*i.e.* $\simeq$ is the least congruence containing $\leftrightarrow$), then a proof of congruence $t \simeq u$ is a sequence of terms $(s_i)_i$ such that $A \leftrightarrow s_1 \leftrightarrow s_2 \leftrightarrow \cdots \leftrightarrow B$.

1.6 Expressing systems in computational logical frameworks

We present informally some usual techniques to express systems in the computational logical framework λΠme.

Given a theory $\mathcal{S}$ made of a language and typing rules, expressing it in λΠme consists in providing a set of functions and equations in order to translate valid

judgements of $\mathcal{S}$ into λΠme. The set of functions and equations of λΠme to embed $\mathcal{S}$ is denoted $\lambda\Pi[\mathcal{S}]$.

In order to get acquainted with general techniques used to embed systems in logical frameworks, we show how to express simply typed λ-calculus (Church 1940) and minimal predicate logic (Frege 1879) in λΠme.

Example 1 (λ-calculus). To embed the simply typed λ-calculus with one sort, we declare a type $\mathtt{I}$ to represent this unique sort, a function $\mathtt{app} : \mathtt{I} \to \mathtt{I} \to \mathtt{I}$ that stands for the application and a function $\mathtt{abs} : (\mathtt{I} \to \mathtt{I}) \to \mathtt{I}$ that stands for the abstraction.

Using these functions, the term $\lambda x, \lambda y, (y\, x)$ is represented by

$$(\mathtt{abs}\, (\lambda x : I, (\mathtt{abs}\, (\lambda y : I, (\mathtt{app}\, y\, x))))) .$$

This encoding may seem confusing, as we could simply use the identity for embedding, since λΠme contains both the abstraction and the application. However, we use the functions $\mathtt{app}$ and $\mathtt{abs}$ to separate the abstractions of the framework that are used as tools to bind variables from the actual abstraction of the source logic, here the λ-calculus, which we embed using $\mathtt{abs}$.

The embedding is not yet finished, as β reductions of the λ-calculus are not reflected in our embedding: whereas, for any free variable x, $((\lambda y, y)\, x) = x$, we do not have such equality in the embedding: $(\mathtt{app}\, (\mathtt{abs}\, (\lambda y, y))\ x)$ is not equal to x. Therefore, we add the following equation to the framework

$$(\mathtt{app}\ (\mathtt{abs}\, b)\ e) = (b\, e) .$$

The embedding consists finally of the following set of functions and equations (where $\star$ is the type of types in λΠme),

$$\mathtt{I} : \star$$
$$\mathtt{app} : \mathtt{I} \to \mathtt{I} \to \mathtt{I}$$
$$\mathtt{abs} : (\mathtt{I} \to \mathtt{I}) \to \mathtt{I}$$
$$(\mathtt{app}\ (\mathtt{abs}\, b)\ e) = (b\, e)$$

We can finally prove that the embedding preserves typing, *i.e.* that any valid typing judgements of the source system is embedded as a valid typing judgement of λΠme; and we can prove that it is complete: embedded types can be inhabited in λΠme only if they can be inhabited in the λ-calculus.

Example 2 (Minimal predicate logic). Predicate logic with a single sort, in its minimal form, is made of the universal quantification $\forall$ and the implication $\Rightarrow$,

and a sort I. Just like in the previous example, the sort I is embedded by a type $\texttt{I} : \star$. We add the type of propositions denoted $\texttt{Prop} : \star$. The implication $\Rightarrow$ is embedded by a function $\texttt{imply} : \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop}$. The quantifier $\forall$ binds a variable, therefore, we use the absraction of the framework as a binding facility: for any proposition P, denoting P' the embedding of P, '$\forall x.P$' is embedded as $(\texttt{all}\,(\lambda x : \texttt{I}, P'))$, where $\texttt{all} : (\texttt{I} \to \texttt{Prop}) \to \texttt{Prop}$.

Our current embedding allows us to write well-formed propositions, but we cannot express proofs yet. For this, we introduce a dependent type $\texttt{Prf} : \texttt{Prop} \to \star$ such that, for any proposition P, $(\texttt{Prf}\,P)$ can be interpreted as the type of proofs of P. For any propositions P and Q embedded as P' and Q', proofs of $P \Rightarrow Q$ cannot be terms of type $(\texttt{imply}\,P'\,Q')$, because the latter is an object, it is not a type. However, it can be typed by $(\texttt{Prf}\ (\texttt{imply}\,P'\,Q'))$ which is now a type.

Finally, the Brouwer-Heyting-Kolmogorov interpretation of proofs says that a proof of $P \Rightarrow Q$ should be a function mapping proofs of P to proofs of Q. This interpretation is not reflected in the embedding: $(\texttt{Prf}\ (\texttt{imply}\,P'\,Q'))$ is not a function type. But thanks to the computational capabilities of λΠme, we can add the following equation

$$(\texttt{Prf}\ (\texttt{imply}\,P\,Q)) = (\texttt{Prf}\,P) \to (\texttt{Prf}\,Q)$$

so that the type of proofs of $P \Rightarrow Q$ is identified with the type of functions from proofs of P to proofs of Q.

More techniques to express theories in λΠme can be seen in (Blanqui, Dowek et al. 2021; Guillaume Burel et al. 2016).

Chapter 2

Encoding explicit predicate subtyping

This chapter is based on (Hondet and Blanqui 2021). The modifications brought in this manuscript are listed in Table 2.1.

2.1 *PVS-Cert*: A minimal system with predicate subtyping

Because of its size, encoding the whole of *PVS* cannot be achieved in one step. Consequently, F. Gilbert 2018 extracted, formalised and studied a subsystem

Table 2.1: Main differences with (Hondet and Blanqui 2021)

Expanded formalism in Section 2.1
New section on conservativity of computations (Section 2.5)
New section on bidirectional type checkers (Section 2.4.2)
Replaced encoded `Prop` by `El o`
Removed encoded `type` and `Kind` which were the encoding of the sorts of *PVS-Cert*, only `Type` remains as encoded sort (and `o`)

of *PVS* which captures the essence of predicate subtyping named *PVS-Cert*. Unlike *PVS*, *PVS-Cert* contains proof terms, thus type checking is decidable in *PVS-Cert* while it is not in *PVS*. Hence *PVS-Cert* is a suitable logical system in which to encode *PVS* specifications and proofs so that they may be cross checked.

In comparison with (F. Gilbert 2018), we use equations rather than reduction rules and slightly change the syntax of terms. The system *PVS-Cert* remains a two layer system composed of predicate subtyping on top of simple type theory (Church 1940).

2.1.1 Type system modulo theory

To describe the various type systems used in this work, we will use type systems modulo introduced by Blanqui 2001. Type systems modulo are an extension of pure type systems (H. Barendregt and Hemerik 1990) with symbols of fixed arity declared in a typing signature and an arbitrary congruence. Pure type systems use the reflexive transitive symmetric closure of the β reduction as equivalence.

Definition 2 (Syntax of type systems modulo)**.** The terms of type systems modulo are parametrised by a set of sorts $\mathcal{S}$, a set of variables $\mathcal{X}$ and a set of symbols $\mathcal{F}$. The set of terms is denoted $\mathcal{T}(\mathcal{X}, \mathcal{S}, \mathcal{F})$ and is described by the following grammar

$$t ::= s \in \mathcal{S} \mid x \in \mathcal{X} \mid f \in \mathcal{F} \mid t\, t \mid \lambda x : t, t \mid \Pi x : t, t \tag{2.1}$$

A contexts is a subsets of $(\mathcal{X} \times \mathcal{T}(\mathcal{X}, \mathcal{S}, \mathcal{F}))^*$ where each variable is bound at most once.

We will often abuse notations and omit arguments of $\mathcal{T}(-)$ whenever they can be unambiguously inferred from the context. The set of variables can almost always be omitted, since we only work with set $\mathcal{X}$.

Definition 3 (Signature)**.** For any set of symbols $\mathcal{F}$, any set of sorts $\mathcal{S}$, such that $\mathcal{F}$, $\mathcal{S}$ and $\mathcal{X}$ are pairwise disjoint, denoting $\mathcal{T}$ as an abbreviation for $\mathcal{T}(\mathcal{X}, \mathcal{S}, \mathcal{F})$, a typing signature is a partial function $\Sigma : \mathcal{F} \rightharpoonup (\mathcal{X} \times \mathcal{T})^* \times \mathcal{T} \times \mathcal{S}$. Furthermore, for any triple $((\boldsymbol{x}, \boldsymbol{A}), B, s)$ in the image of Σ, variables $\boldsymbol{x}$ are pairwise distinct.

A mapping from a symbol f to a triple $((\boldsymbol{x}, \boldsymbol{t}), T, s)$ is denoted $f[\boldsymbol{x} : \boldsymbol{t}] : T : s$ To say that f is mapped to $((\boldsymbol{x}, \boldsymbol{A}), B, s)$ by Σ, we either write $\Sigma(f) = ((\boldsymbol{x}, \boldsymbol{A}), B, s)$ or $f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma$. For any signature Σ, the *domain* of Σ is denoted $\mathrm{dom}(\Sigma)$.

$$\textsc{empty}\ \frac{}{\vdash \emptyset} \qquad \textsc{decl}\ \frac{x \notin \Gamma \qquad \Gamma \vdash A : s}{\vdash \Gamma, x : A} \qquad \textsc{var}\ \frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{conv}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash t : B} \qquad \textsc{sort}\ \frac{\vdash \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}$$

$$\textsc{prod}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{P}}{\Gamma \vdash \Pi x : A, B : s_3}$$

$$\textsc{abst}\ \frac{\Gamma \vdash \Pi x : A, B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B} \qquad \textsc{appl}\ \frac{\Gamma \vdash t : \Pi x : A, B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\, u) : \{u/x\}\, B}$$

$$\textsc{sign}\ \frac{f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma \qquad \boldsymbol{x} : \boldsymbol{A} \vdash B : s \qquad \left(\Gamma \vdash t_i : \left\{t_j/x_j\right\}_{j<i} A_i\right)_i}{\Gamma \vdash (f\, \boldsymbol{t}) : \{\boldsymbol{t}/\boldsymbol{x}\}\, B}$$

Figure 2.1: Typing rules of a type system modulo. They are parametrised by axioms $\mathcal{A}$, product rules $\mathcal{P}$, congruence $\simeq$ and signature Σ.

The notation $[\boldsymbol{x} : \boldsymbol{t}]$ is reminiscent of de Bruijn's *telescopes* (de Bruijn 1991) where each term t may depend on the variables bound earlier in the context.

Remark 1. Signatures provide an *arity* to each symbol in their domain. For any signature Σ, for any $(f[(x_i : A_i)_{i \leq \ell}] : B : s) \in \Sigma$, $(f\, \boldsymbol{t})$ is well-typed only when f is applied to ℓ arguments (see rule SIGN of Fig. 2.1), therefore ℓ can be seen as the arity of f.

Definition 4 (Type system modulo specification)**.** A type system modulo *specification* is a 6-uple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{F}, \simeq, \Sigma)$ where

- $\mathcal{S}$ is a finite set of constants called *sorts*,
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a relation called *axioms*,

- $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a relation called *product rules*,
- $\mathcal{F}$ is a set of function symbols,
- $\simeq$ is a congruence on $\mathcal{T}(\mathcal{S}, \mathcal{F})$ and
- Σ is a typing signature.

When omitted, the set of symbols $\mathcal{F}$ defaults to the domain of Σ.

Definition 5 (Type system modulo)**.** For any specification $\mathfrak{T} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{F}, \simeq, \Sigma)$, abbreviating $\mathcal{T}(\mathcal{S}, \mathcal{F})$ by $\mathcal{T}$, a *type system modulo* is a ternary relation (also called 'typing relation') in $(\mathcal{X} \times \mathcal{T})^* \times \mathcal{T} \times \mathcal{T}$ denoted '$\Gamma \vdash t : u$' where Γ is a context and t and u are terms. A triple (Γ, t, u) is in the relation if and only if $\Gamma \vdash t : u$ can be derived can be derived using the inference rules of Fig. 2.1 parametrised by $\mathfrak{T}$.

For any specification $\mathfrak{T}$, the type system modulo parametrised by $\mathfrak{T}$ is denoted '$\cdot \vdash_{\mathfrak{T}} \cdot : \cdot$' (where '$\cdot$' denotes the position of arguments). The annotation $\mathfrak{T}$ can be omitted when the specification can be unambiguously inferred. In that case, we simply write '$\cdot \vdash \cdot : \cdot$'. Because a specification identifies uniquely a type system modulo, we may quantify over type systems modulo instead of specification, so that the sentence 'for any type system modulo $\mathfrak{T}$' should be understood as 'for any type system modulo specification $\mathfrak{T}$'. For any specification, for any triple (Γ, t, A), the notation $\Gamma \vdash t : A$ is also called a 'judgement' in the sense that it judges t to be of type A in context Γ.

Definition 6 (Well-formed signature)**.** A signature Σ is well-formed in a type system modulo $\mathfrak{T}$, written $\vdash_{\mathfrak{T}} \Sigma$, if for any judgement $f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma$, we have $\boldsymbol{x} : \boldsymbol{A} \vdash_{\mathfrak{T}} B : s$.

Remark 2. We do not pay attention to how signatures and equivalences are formed in general. The formation rules of signatures may be defined in the type system (Gaspard Férey 2021; Guillaume Burel et al. 2016; Saillard 2015). These presentations are suitable for formalising the meta theory of logical frameworks, when the process of creating a signature is discussed, and some properties must be kept through extension of the signature. In this thesis, we consider a restricted number of signatures which are designed so that they have desirable properties.

Definition 7. Let $\mathfrak{T}$ be a type system modulo parametrised by a set of sorts $\mathcal{S}$. We say that a context Γ is *well-formed* in $\mathfrak{T}$ whenever $\vdash_{\mathfrak{T}} \Gamma$ is derivable. We say

that a type T is *well-sorted* in context Γ in $\mathfrak{T}$ whenever there is a sort $s \in \mathcal{S}$ such that $\Gamma \vdash_{\mathfrak{T}} T : s$. We say that a term t is *well-typed* in a context Γ in $\mathfrak{T}$ whenever there is a type T well-sorted in context Γ (in system $\mathfrak{T}$) such that $\Gamma \vdash_{\mathfrak{T}} t : T$ is derivable.

Definition 8 (λΠme)**.** For any set of variables $\mathcal{X}$, for any set of symbols $\mathcal{F}$, for any set of equations $\mathcal{E}$ and for any typing signature Σ, a λΠme system is a type system modulo parametrised by $(\{\star, \Box\}, \{(\star, \Box)\}, \mathcal{P}^{\lambda\Pi}, \mathcal{F}, \simeq, \Sigma)$ where $\mathcal{P}^{\lambda\Pi} = \{(\star, \star, \star), (\star, \Box, \Box)\}$ and $\simeq$ is the smallest congruence containing equations of $\mathcal{E}$ and Eq. (β) Page 31. Any λΠme system is specified by the triple $(\mathcal{F}, \mathcal{E}, \Sigma)$.

Any instance of λΠme with an empty set of equations is an instance of the Edinburgh's logical framework. The family of λΠme type systems modulo can be seen as a logical framework extended with an equational theory.

2.1.2 Simple type theory

PVS and *PVS-Cert* are both based on simple type theory (Church 1940), which can be represented by the following type system modulo.

Definition 9 (Simple type theory, λHOL)**.** Simple type theory, or λHOL, is the type system modulo defined by the following parameters

- $\mathcal{S} = \{\texttt{Prop}, \texttt{Type}, \texttt{Kind}\}$,
- $\mathcal{A} = \{(\texttt{Prop}, \texttt{Type}), (\texttt{Type}, \texttt{Kind})\}$,
- $\mathcal{P} = \{(\texttt{Prop}, \texttt{Prop}, \texttt{Prop}), (\texttt{Type}, \texttt{Type}, \texttt{Type}), (\texttt{Type}, \texttt{Prop}, \texttt{Prop})\}$,
- $\mathcal{F} = \emptyset$,
- $\Sigma = \emptyset$,
- $\simeq$ is the congruence of the β-reduction defined by Eq. (β)

$$((\lambda x, t)\ u) \hookrightarrow \{u/x\}\, t \qquad (\beta)$$

λHOL is also a pure type system (H. Barendregt and Hemerik 1990) since the signature is empty and the equivalence is the closure of the β-reduction.

We denote by $\simeq_\beta$ the congruence of rule Eq. (β).

$$\texttt{psub}\,[T : \texttt{Type}; P : T \to \texttt{Prop}] : \texttt{Type} : \texttt{Kind}$$
$$\texttt{pair}\,[T : \texttt{Type}; P : T \to \texttt{Prop}; x : T; h : Px] : \texttt{psub}(T, P) : \texttt{Type}$$
$$\pi_\ell\,[T : \texttt{Type}; P : T \to \texttt{Prop}; x : \texttt{psub}(T, P)] : T : \texttt{Type}$$
$$\pi_r\,[T : \texttt{Type}, P : T \to \texttt{Prop}, x : \texttt{psub}(T, P)] : P(\pi_\ell(T, P, x)) : \texttt{Type}$$

Figure 2.2: Signature Σ_{Pe} of *PVS-Cert*.

2.1.3 Predicate subtyping

Predicate subtyping introduces four new symbols in simple type theory: the type construction for predicate subtypes `psub`, an introduction of predicate subtypes `pair` and two eliminators π_ℓ and π_r. A predicate subtype $\texttt{psub}(A, P)$ is defined from a supertype A and predicate P over A. Terms inhabiting a predicate subtype $\texttt{psub}(A, P)$ are built with $\texttt{pair}(A, P, t, h)$ made of a term t that stands for an actual value of type A and a proof h that t validates P. Eliminators allow either to retrieve a value out of u using $\pi_\ell(A, P, u)$, or to retrieve a proof that u validates P with $\pi_r(A, P, u)$. The constructors and eliminators can be seen as coercions from types to predicate subtypes and vice versa: they allow either to attach some logical content to a value, or to retrieve the actual value to perform some computation. Unlike *PVS-Cert*, *PVS* does not use coercions `pair`, π_ℓ and π_r. In *PVS*, subtyping is implicit: terms do not have a unique type, and its choice is left to the type checker.

These symbols are declared in the signature in Fig. 2.2. In addition to simple type theory (Definition 9) and these symbols, *PVS-Cert* uses a congruence $\simeq_{\mathrm{Pe}}$ that identifies more terms than $\simeq_\beta$ which will be defined in Definition 10.

Remark 3. Unlike the original presentation of *PVS-Cert* by F. Gilbert 2018, projections and pairs are annotated with the type of their argument to prove more easily that the translation of *PVS-Cert* terms is well-defined (Proposition 2).

Proof irrelevance So far, no real difference has been evinced between *PVS-Cert* and dependent pairs: predicate subtype $\texttt{psub}(A, P)$ is just a restricted version of dependent pairs $(\Sigma x : A, Px)$ (see ibid., Definition 4.2.3). The difference lies in the equivalence relations and the fact that *PVS-Cert* implements *proof irrelevance* in pairs.

Proofs contained in terms are essential for typing purposes. On the other hand, these proofs are a burden regarding the equivalence of terms. Were

these proofs taken into account (as $\simeq_\beta$ does), too many terms would be distinguished. For example, consider two terms $t = \texttt{pair}(\texttt{nat}, \texttt{evenp}, 2, h)$ and $t' = \texttt{pair}(\texttt{nat}, \texttt{evenp}, 2, h')$ that stand for the number 2 that has been proved even. Without proof irrelevance, t and t' are not considered equal because they do not have the same proof (h and h') that 2 is even. We end up with one even number 2 per proof that 2 is even.

As stated by de Bruijn 1994, most mathematicians seek convertibility of t and t' and care more about what h and h' prove than the proofs themselves. In that regard, *PVS-Cert* has proof irrelevant pairs: proofs attached to terms are not taken into account when checking the equivalence of two pairs. This property is embedded in the equivalence relation used in the conversion rule of *PVS-Cert* which does not attach any importance to the proofs of pairs. Consequently, the eliminator π_r provides *a* proof, but we do not know which one *a priori*. We only know what it proves.

Definition 10. The equivalence of *PVS-Cert* is noted $\simeq_{\text{Pe}}$ and is the smallest congruence containing Eqs. (2.2), (2.3) and (β)

$$\texttt{pair}(t, u, m, h_0) = \texttt{pair}(t, u, m, h_1) \tag{2.2}$$

$$\pi_\ell(t_0, u_0, \texttt{pair}(t_1, u_1, m, h)) = m \tag{2.3}$$

Equation (2.3) allows the projection to compute. The right projection does not compute[1] to avoid implementing full proof irrelevance: the addition of such a reduction rule causes all proofs (i.e. terms of type `Prop`) to be equivalent (since $h = \pi_r(a, p, \texttt{pair}(a, p, x, h)) = \pi_r(a, p, \texttt{pair}(a, p, x, h')) = h'$), which may imply other axioms such as the uniqueness of identity proofs (G. Gilbert et al. 2019).

Definition 11 (*PVS-Cert*)**.** *PVS-Cert* is the same system as λHOL but with $\mathcal{F}_{\text{Pe}} = \{\texttt{psub}, \pi_\ell, \texttt{pair}, \pi_r\}$, the signature Σ_{Pe} defined in Fig. 2.2 and congruence $\simeq_{\text{Pe}}$ (Definition 10). Typing judgements of *PVS-Cert* may be written '$\vdash_{\text{Pe}}$:' to avoid confusion.

The conversion relation used by F. Gilbert 2018 contains only β and the following reductions that erase coercions:

$$\pi_\ell(T, P, X) \hookrightarrow X$$
$$\texttt{pair}(T, P, X, H) \hookrightarrow X.$$

These reduction rules cannot be included in congruence $\simeq_{\text{Pe}}$ because they do not preserve typing: the left-hand side and the right-hand side of both rules cannot

[1] In contrast to the reduction of Sozeau 2006, which contains both projections.

have the same type. On the other hand, this congruence contains surjective pairing $\texttt{pair}(t, p, \pi_\ell(t, p, e), \pi_r(t, p, e)) \hookrightarrow^* e$ whereas $\simeq_{\mathrm{Pe}}$ does not. Equations of λΠme must preserve typing for the type checker to behave well, which prevents from using the above reduction rules as equations.

Proofs of $T \simeq_\beta U$ or $T \simeq_{\mathrm{Pe}} U$ can use untyped intermediate terms, which can be problematic when proving properties hold on typed terms only.

Example 3. While β-reduction preserves typing in *PVS-Cert*, its symmetric, β-expansion, does not. For instance, assume that signature Σ declares

$$\texttt{Nat} : \texttt{Type}; \qquad \texttt{zero} : \texttt{Nat}; \qquad \texttt{String} : \texttt{Type}.$$

Then `zero` (which is well-typed) β-expands to $((\lambda x : \texttt{String}, x)\,\texttt{zero})$ which is ill-typed (because the domain of the abstraction, `String`, is not convertible with the type of the argument, `Nat`).

Proof irrelevance does not preserve well-typedness. Assume we use the previous signature Σ, and we add the declarations

$$\texttt{Even} : \texttt{Nat} \to \texttt{Prop} \qquad \text{and} \qquad \texttt{zE} : (\texttt{Even}\,\texttt{zero})$$

where `Even` is a predicate, and `zE` is a proof that `zero` validates `Even`. In the new signature, the following judgements hold by rule SIGN

$$\vdash \overbrace{\texttt{pair}(\texttt{Nat}, \texttt{Even}, \texttt{zero}, \texttt{zE})}^{\alpha} : \texttt{psub}(\texttt{Nat}, \texttt{Even})$$

However, α is convertible, *via* Eq. (2.2), with $\texttt{pair}(\texttt{Nat}, \texttt{Even}, \texttt{zero}, \texttt{zero})$ where the latter term is not typeable because the fourth argument (`zero`) is not a proof of $(\texttt{Even}\,\texttt{zero})$, hence rule SIGN does not apply.

We therefore show that if T and U are well-typed, there is a proof of $T \simeq_{\mathrm{Pe}} U$ that uses only well-typed terms. In the case of $\simeq_\beta$, the problem is solved by confluence of $\hookrightarrow_\beta$. We now prove a similar property for $\simeq_{\mathrm{Pe}}$:

Lemma 1. *Let* $\hookrightarrow_{\beta,\pi_\ell} = \hookrightarrow_\beta \cup \hookrightarrow_{\pi_\ell}$ *where* $\hookrightarrow_{\pi_\ell}$ *is the closure by substitution and context of Eq.* (2.3) *oriented from left to right, and let* $\leftrightarrow_{pi}$ *be the smallest congruence containing Eq.* (2.2)*. Then*

- $\simeq_{\mathrm{Pe}} \subseteq \hookrightarrow^*_{\beta,\pi_\ell} \leftrightarrow^*_{pi} \hookleftarrow^*_{\beta,\pi_\ell}$
- *if* t *and* u *are well typed and* $t \simeq_{\mathrm{Pe}} u$*, then they have the same type modulo* $\simeq_{\mathrm{Pe}}$.

Proof. We prove that $\leftrightarrow_{\mathrm{pi}}$ steps can be postponed: $\leftrightarrow_{\mathrm{pi}} \hookrightarrow_{\beta,\pi_\ell} \subseteq \hookrightarrow^{=}_{\beta,\pi_\ell} \leftrightarrow^{*}_{\mathrm{pi}}$. Assume that the $\leftrightarrow_{\mathrm{pi}}$ step is at position p and the $\hookrightarrow_{\beta,\pi_\ell}$ step is at position q. If p and q are disjoint, this is immediate. If p is above q, then

$$\mathtt{pair}(a, b, m, h_1) \leftrightarrow_{\mathrm{pi}} \mathtt{pair}(a, b, m, h_2) \hookrightarrow_{\beta,\pi_\ell} \mathtt{pair}(a', b', m', h_2').$$

If the $\hookrightarrow_{\beta,\pi_\ell}$ step is not applied in a subterm of h_2, then $h_2 = h_2'$ and the rewrite sequence can be transformed into

$$\mathtt{pair}(a, b, m, h_1) \hookrightarrow_{\beta,\pi_\ell} \mathtt{pair}(a', b', m', h_1) \leftrightarrow_{\mathrm{pi}} \mathtt{pair}(a', b', m', h_2).$$

If $\hookrightarrow_{\beta,\pi_\ell}$ is applied on a subterm of h_2, then $(a, b, m) = (a', b', m')$ and we can directly apply Eq. (2.2):

$$\mathtt{pair}(a, b, m, h_1) \leftrightarrow_{\mathrm{pi}} \mathtt{pair}(a, b, m, h_2').$$

If q is above p, we have either

- $(\lambda x : a, t)\, u \leftrightarrow_{\mathrm{pi}} (\lambda x : a', t')\, u' \hookrightarrow_{\beta,\pi_\ell} \{u'/x\}\, t'$ and $(\lambda x : a, t)\, u \hookrightarrow_{\beta,\pi_\ell} \{u/x\}\, t \leftrightarrow_{\mathrm{pi}} \{u'/x\}\, t'$, or
- the $\leftrightarrow_{\mathrm{pi}}$ step is applied to a subterm erased by $\hookrightarrow_{\pi_\ell}$, in which case $\leftrightarrow_{\mathrm{pi}} \hookrightarrow_{\pi_\ell} \subseteq \hookrightarrow_{\pi_\ell}$, or
- the $\leftrightarrow_{\mathrm{pi}}$ step is applied to a subterm that is not erased by $\hookrightarrow_{\pi_\ell}$ in which case $\leftrightarrow_{\mathrm{pi}} \hookrightarrow_{\pi_\ell} \subseteq \hookrightarrow_{\pi_\ell} \leftrightarrow_{\mathrm{pi}}$.

The relation $\hookrightarrow_{\beta,\pi_\ell}$ is confluent because Eq. (β) and oriented Eq. (2.3) form an orthogonal combinatory reduction system (*i.e.* whose rules are left-linear and non-overlapping) (Klop, Oostrom and Raamsdonk 1993). We show 1. $\simeq_{\mathrm{Pe}} \subseteq \hookrightarrow^{*}_{\beta,\pi_\ell} \leftrightarrow^{*}_{\mathrm{pi}} \hookleftarrow^{*}_{\beta,\pi_\ell}$ by confluence of $\hookrightarrow_{\beta,\pi_\ell}$ and postponement of equational steps and by induction on the number of $\leftrightarrow_{\mathrm{pi}}$ steps.

We now prove that 2. $\hookrightarrow_{\beta}$ preserves typing. For this, it is enough to prove that, if $(\Pi x : a, b)$ and $(\Pi x : a', b')$ are typeable, and $(\Pi x : a, b) \simeq_{\mathrm{Pe}} (\Pi x : a', b')$, then $a \simeq_{\mathrm{Pe}} a'$ and $b \simeq_{\mathrm{Pe}} b'$ (for more details, see Blanqui 2005), which follows from Item 1. We now prove that 3. $\hookrightarrow_{\pi_\ell}$ preserves typing. Assume that $\pi_\ell(a_0, p_0, (\mathtt{pair}(a_1, p_1, m, h)))$ is of type C. By inversion of typing rules, the type of $\mathtt{pair}(a_1, p_1, m, h)$ is convertible with $\mathtt{psub}(a_0, p_0)$ and $a_0 \simeq_{\mathrm{Pe}} C$. By inversion again, the type of m is convertible with a_1 and $\mathtt{psub}(a_0, p_0) \simeq_{\mathrm{Pe}} \mathtt{psub}(a_1, p_1)$. By Item 1, $a_0 \simeq_{\mathrm{Pe}} a_1$ and $p_0 \simeq_{\mathrm{Pe}} p_1$. Therefore, m is of type C.

For the following sub-proof, note that 4. $\leftrightarrow_{\text{pi}}^* = \Leftrightarrow_{\text{pi}}$ where $\Leftrightarrow_{\text{pi}}$ consists in applying several $\leftrightarrow_{\text{pi}}$ steps such that if there is a $\leftrightarrow_{\text{pi}}$ rewriting at position p, there is no rewriting below position $4.p$ (no rewriting below the proof irrelevant argument). Indeed, if

$$\overbrace{\texttt{pair}(a,p,m,h_1)}^{t} \leftrightarrow_{\text{pi}} \texttt{pair}(a,p,m,(\ldots(\texttt{pair}(a',p',m',h'_1))\ldots)) \leftrightarrow_{\text{pi}} \underbrace{\texttt{pair}(a,p,m,(\ldots(\texttt{pair}(a',p',m',h'_2))\ldots))}_{u}$$

then $t \leftrightarrow_{\text{pi}} u$ as well. Now that $\hookrightarrow_{\beta,\pi_\ell}$ preserves typing, we show 5. : 'if $t \leftrightarrow_{\text{pi}}^* u$ and t and u are well typed, then they have the same type (modulo $\simeq_{\text{Pe}}$)' by induction on the number of positions rewritten in $t \Leftrightarrow_{\text{pi}} u$. If no position is rewritten, then t is syntactically equal to u, and hence they have the same type. Otherwise, assume Item 5 for $n > 0$ positions, $t \Leftrightarrow_{\text{pi}} u$ with $n+1$ equational steps and let $p \in \mathcal{P}(t)$ such that a $\leftrightarrow_{\text{pi}}$ equational step is applied at p, and no more than n equational steps are applied below p. Such a position exists since there is at least one equational step, and if there are more than n equational steps below p, we may take the first position below p where an equational step occurs. Then by hypothesis, $t|_p \Leftrightarrow_{\text{pi}} u|_p$ with no more than n equational steps, and both $t|_p$ and $u|_p$ are typeable since t and u are typeable. Thus by induction hypothesis, $t|_p$ and $u|_p$ have the same type. Now by induction hypothesis, $\left\{u|_p/p\right\}t \Leftrightarrow_{\text{pi}} u$ with no more than n rewrite steps, and both terms are typeable, so they have the same type. Since we have $t \Leftrightarrow_{\text{pi}} \left\{u|_p/p\right\}t$ and both terms typeable, we have also that t and $\left\{u|_p/p\right\}t$ have the same type.

We can now conclude: if $t \simeq_{\text{Pe}} u$ and both t and u are typeable, then by Item 1 we have t' and u' such that $t \hookrightarrow_{\beta,\pi_\ell}^* t' \leftrightarrow_{\text{pi}}^* u' \hookleftarrow_{\beta,\pi_\ell}^* u$. By Items 2 and 3, we have that t' has the same type as t and u' has the same type as u. By Item 5, t' and u' have the same type. Finally, by transitivity of $\simeq_{\text{Pe}}$, t and u have the same type (modulo $\simeq_{[\text{Pe}]}$). □

2.2 Encoding *PVS-Cert* in λΠme

Encoding *PVS-Cert* into a logical framework such as λΠme allows to express terms of the former into the latter. Because logical frameworks strive to remain minimal, constructions such as $\texttt{pair}(A,P,m,h)$ or $\texttt{psub}(A,P)$ are not built-in:

$$\Sigma_{[\mathrm{stt}]}\begin{cases}\mathtt{Type} : \star : \square & (2.4)\\ \mathtt{o} : \mathtt{Type} : \star & (2.5)\\ \mathtt{El}\,[t : \mathtt{Type}] : \star : \square & (2.6)\\ \mathtt{Prf}\,[p : (\mathtt{El}\,\mathtt{o})] : \star : \square & (2.7)\\ \forall\,[t : \mathtt{Type}; p : (\mathtt{El}\,(t \rightsquigarrow \mathtt{o}))] : (\mathtt{El}\,\mathtt{o}) : \square & (2.8)\\ \Rightarrow [p : (\mathtt{El}\,\mathtt{o})\,, q : (\mathtt{Prf}\,p) \to (\mathtt{El}\,\mathtt{o})] : (\mathtt{El}\,\mathtt{o}) : \square & (2.9)\\ \rightsquigarrow [t : \mathtt{Type}, u : (\mathtt{El}\,t) \to \mathtt{Type}] : \mathtt{Type} : \square & (2.10)\end{cases}$$

$$\mathcal{E}_{[\mathrm{stt}]}\begin{cases}(\mathtt{Prf}\ (\forall\, t\, p)) = \Pi x : \mathtt{El}, t, (\mathtt{Prf}\ (p\, x)) & (2.11)\\ (\mathtt{Prf}\,(p \Rightarrow q)) = \Pi h : \mathtt{Prf} p, (\mathtt{Prf}\ (q\, h)) & (2.12)\\ (\mathtt{El}\,(t \rightsquigarrow u)) = \Pi x : \mathtt{El}, t, \mathtt{El}, (u\, x) & (2.13)\end{cases}$$

Figure 2.3: Signature $\Sigma_{[\mathrm{stt}]}$ and equations $\mathcal{E}_{[\mathrm{stt}]}$ to encode simple type theory into λΠme.

they must be expressed into the language of the logical framework through an encoding. We hence define the symbols allowing to emulate predicate subtyping using the terms of λΠme.

2.2.1 Encoding simple type theory in λΠme

Following the stratification of λΠme (Harper, Honsell and Plotkin 1993), we say that term t typeable by T is an *object* when T is typable by $\star$ and a *type* when T is typeable by $\square$.

The encoding of λHOL given in Fig. 2.3 Page 37 follows the method settled by Blanqui, Dowek et al. 2021; Cousineau and Dowek 2007. Symbols that constitute the encoding in λΠme are written in blue.

The sort `Type` is encoded by the type `Type`, and the sort `Prop` by the object `o` in declarations (2.4) and (2.5). One can already note that the declaration `o` : `Type` encodes the axiom `Prop` : `Type` of *PVS-Cert*.

For objects and types of λHOL, the idea is to manipulate them as objects of λΠme. We call *type codes* the types of λHOL encoded as objects of λΠme. When a type from λHOL is needed, for instance to encode $\lambda x : \mathtt{nat}, x$, we use families of types of λΠme indexed by type codes to lift λΠme objects to λΠme types.

Terms of type `Type` are encoded as type codes of type `Type`. These type codes

can be interpreted as λΠme types with the family of types `El` (2.6). Because `o` (2.5) is the type code of propositions, the latter are encoded as objects of type $\texttt{El}, \texttt{o}$. The type of proofs is given by the family `Prf` (2.7) indexed by propositions. For instance, given a λHOL type ι and a λHOL proposition P both encoded as λΠme objects, the abstractions $\lambda x : \texttt{El}, \iota, x$ and $\lambda h : \texttt{Prf}P, h$ are valid λΠme terms.

Constants $\forall$ and $\Rightarrow$ represent respectively the universal quantification, which can also be seen as a dependent product binding values (of type codes) into propositions; and the (dependent) implication which binds proofs into propositions, and can also be seen as a dependent product at the level of propositions. Constant $\Rightarrow$ represents the dependent version of the functional arrow, at the level of type codes.

Equations (2.11) to (2.13) are used to map encoded products to λΠme products.

Remark 4. Symbols $\rightsquigarrow$ and $\Rightarrow$ are written infix for readability. Furthermore, these two operators have a binder as second argument to express the dependency of the second argument on the first one, like with dependent product. We may abuse notations and write $(x \rightsquigarrow y)$ when $y : \texttt{Type}$ does not depend on x, instead of $(x \rightsquigarrow \lambda v : \texttt{Type}, y)$.

Definition 12 (λΠ[stt])**.** The encoding of simple type theory in λΠme denoted λΠ[stt] is the λΠme type system parametrised by signature $\Sigma_{[\text{stt}]}$ and equations $\mathcal{E}_{[\text{stt}]}$ both defined in Fig. 2.3.

2.2.2 Encoding explicit predicate subtyping in λΠme

Predicate subtypes are defined in declaration (2.14) as encoded types (*i.e.* terms of type `Type`) built from encoded type t and predicate defined on t. Pairs are encoded in Eq. (2.15), where the second argument is the predicate that defines the type of the pair. The two projections are encoded in declarations (2.16) and (2.17).

Definition 13 (λΠ[Pe])**.** Let $\Sigma_{[\text{Pe}]}$ be the union of signature $\Sigma_{[\text{stt}]}$ (defined in Fig. 2.3) and typing declarations of Fig. 2.4. Let $\mathcal{E}_{[\text{Pe}]}$ the set of equations containing $\mathcal{E}_{[\text{stt}]}$ and Eqs. (2.18) and (2.19) defined in Fig. 2.4. We denote λΠ[Pe] the encoding of *PVS-Cert* in λΠme. It is the λΠme type system parametrised by signature $\Sigma_{[\text{Pe}]}$ and equations $\mathcal{E}_{[\text{Pe}]}$. The congruence of λΠ[Pe] is denoted $\simeq_{[\text{Pe}]}$.

$$\texttt{psub}\,[t : \texttt{Type}; p : (\texttt{El}\,(t \rightsquigarrow \texttt{o}))] : \texttt{Type} : \star \tag{2.14}$$

$$\texttt{pair}\,[t : \texttt{Type}; p : (\texttt{El}\,(t \rightsquigarrow \texttt{o}))\,; m : (\texttt{El}\,t)\,, h : (\texttt{Prf}\ (p\,m))] : (\texttt{El}\ (\texttt{psub}\,t\,p)) : \star \tag{2.15}$$

$$\texttt{fst}\,[t : \texttt{Type}; p : (\texttt{El}\,(t \rightsquigarrow \texttt{o}))\,; m : (\texttt{El}\ (\texttt{psub}\,t\,p))] : \texttt{El}, t : \star \tag{2.16}$$

$$\texttt{snd}\,[t : \texttt{Type}; p : (\texttt{El}\,(t \rightsquigarrow \texttt{o}))\,; m : (\texttt{El}\ (\texttt{psub}\,t\,p))] : (\texttt{Prf}\ (p\ (\texttt{fst}\,t\,p\,m))) : \star. \tag{2.17}$$

$$(\texttt{pair}\,t\,p\,m\,h) = (\texttt{pair}\,t\,p\,m\,h') \tag{2.18}$$

$$(\texttt{fst}\,t\,p\ (\texttt{pair}\,t'\,p'\,m\,h)) = m \tag{2.19}$$

Figure 2.4: Signature and equations to encode predicate subtyping into λΠme.

2.2.3 Translation of *PVS-Cert* terms into λΠ[Pe]

Definition 14 (Translation)**.** Let Γ be a well-formed context.

- The *term translation* of term M typeable in Γ, noted $[M]_\Gamma$, is defined in Figs. 2.5 and 2.6.
- The *type translation* of `Kind` and the terms M typeable by a sort in Γ, noted $[\![M]\!]_\Gamma$, is defined in Fig. 2.7.
- The *context translation* $[\![\Gamma]\!]$ is defined by induction on Γ as

$$[\![\emptyset]\!] = \emptyset; \quad [\![\Gamma, x : T]\!] = [\![\Gamma]\!]\,, x : [\![T]\!]_\Gamma$$

Proposition 2. *The translation function* $[-]_\_$ *that maps a context and a* PVS-Cert *term typeable in this context to a* λΠ*me term is well-defined.*

Proof. After Lemma 1 and Blanqui 2001, Lemma 41, the types of a term are unique up to equivalence. Moreover, the arguments of the translation function are decreasing with respect to the (strict) subterm relation. □

For now, we have no guarantee on the behaviour of our encoding, and thus successfully type checking an encoded theory into λΠme brings no useful information on the source theory. Section 2.3 contains theorems to recover some guarantees on the encoding.

$$
\begin{aligned}
[x]_\Gamma &= x \\
[\texttt{Prop}]_\Gamma &= \texttt{o} \\
[\texttt{Type}]_\Gamma &= \texttt{Type} \\
[M\,N]_\Gamma &= [M]_\Gamma\ [N]_\Gamma \\
[\lambda x : T, M]_\Gamma &= \lambda x : \big(\texttt{El}\ [T]_\Gamma\big), [M]_{\Gamma, x:T} \\
[\Pi x : T, U]_\Gamma &= [T]_\Gamma \rightsquigarrow \Big(\lambda x : [\![T]\!]_\Gamma, [U]_{\Gamma, x:T}\Big) \\
&\qquad \text{when } \Gamma \vdash_{\overline{\text{Pe}}} T : \texttt{Type} \text{ and } \Gamma, x : T \vdash_{\overline{\text{Pe}}} U : \texttt{Type} \\
[\Pi x : T, P]_\Gamma &= \forall\, [T]_\Gamma\ \Big(\lambda x : [\![T]\!]_\Gamma, [P]_{\Gamma, x:T}\Big) \\
&\qquad \text{when } \Gamma \vdash_{\overline{\text{Pe}}} T : \texttt{Type} \text{ and } \Gamma, x : T \vdash_{\overline{\text{Pe}}} P : \texttt{Prop} \\
[\Pi h : P, Q]_\Gamma &= [P]_\Gamma \Rightarrow \Big(\lambda h : [\![P]\!]_\Gamma, [Q]_{\Gamma, h:P}\Big) \\
&\qquad \text{when } \Gamma \vdash_{\overline{\text{Pe}}} P : \texttt{Prop} \text{ and } \Gamma, h : P \vdash_{\overline{\text{Pe}}} Q : \texttt{Prop}
\end{aligned}
$$

Figure 2.5: Translation from λHOL to λΠ[stt].

$$
\begin{aligned}
[\texttt{psub}(T, P)]_\Gamma &= \big(\texttt{psub}\ [T]_\Gamma\ [P]_\Gamma\big) \\
[\texttt{pair}(T, P, M, N)]_\Gamma &= \big(\texttt{pair}\ [T]_\Gamma\ [P]_\Gamma\ [M]_\Gamma\ [N]_\Gamma\big) \\
[\pi_\ell(T, P, M)]_\Gamma &= \big(\texttt{fst}\ [T]_\Gamma\ [P]_\Gamma\ [M]_\Gamma\big) \\
[\pi_r(T, P, M)]_\Gamma &= \big(\texttt{snd}\ [T]_\Gamma\ [P]_\Gamma\ [M]_\Gamma\big)
\end{aligned}
$$

Figure 2.6: Translation from *PVS-Cert* to λΠ[Pe].

$$\llbracket T \rrbracket_\Gamma = \left(\texttt{El}\ \llbracket T \rrbracket_\Gamma\right) \qquad \text{when } \Gamma \vdash_{\overline{\text{Pe}}} T : \texttt{Type}$$
$$\llbracket T \rrbracket_\Gamma = \left(\texttt{Prf}\ \llbracket T \rrbracket_\Gamma\right) \qquad \text{when } \Gamma \vdash_{\overline{\text{Pe}}} T : \texttt{Prop}$$
$$\llbracket \texttt{Kind} \rrbracket = \star$$
$$\llbracket \texttt{Type} \rrbracket = \texttt{Type}$$

Figure 2.7: Translation of types from *PVS-Cert* to λΠ[Pe].

2.2.4 Examples of encoded theories

We provide here some examples that take advantage of proof irrelevance or predicate subtyping. While these examples could have been presented in *PVS-Cert*, we write them into λΠ[Pe]. For any context Γ, name f, type A and term t, the notation '$f\,[\Gamma] : A := t$' is syntactic sugar for a signature declaration '$f\,[\Gamma] : A : s$' (we drop the sort in the declaration for brevity) and an equation '$(f\,\boldsymbol{x}) = t$' where $\boldsymbol{x}$ are the variables defined in Γ. We will generally omit the first two arguments of `fst`, `pair` and `snd` so the term $(\texttt{fst}\,m)$ stands for $(\texttt{fst}\,a\,p\,m)$ for some terms a, p. These examples show that the encoding is relatively lightweight and thus suitable for human-made developments. Furthermore, interpretation functions `El` and `Prf` could be omitted using coercion facilities later described in Chapter 3 Page 61, as shown in Example 12 Page 74.

Because equality is one of the most common mathematical predicates, we start by defining a signature with equality and inequality in Fig. 2.8. The signature defines a polymorphic equality predicate =, a constructor `refl` and an eliminator `eqind` stating that terms may be substituted by equals. This signature is assumed to be prepended to the signatures we define in the remaining of this section. Therefore, in the following signatures, all functions declared or defined in Fig. 2.8 are available.

Example 4 (Stacks with predicate subtypes). This example comes from the language reference manual of *PVS* (Owre, Shankar et al. 2020) and illustrates the use of predicate subtyping and the generation of type correctness conditions through a specification of stacks in Fig. 2.9.

Predicate subtyping is used to define the type of nonempty stacks, which allows the function `pop` to be total. In the definition of the theorem `pop2push2`, term α is a proof that the first argument of the pair is not empty. Such term is the encoding's counterpart of *PVS*' type correctness conditions. We can thus see

$$
\begin{aligned}
&\texttt{=}\,[a : \texttt{Type}] : (\texttt{El}\ (a \rightsquigarrow a \rightsquigarrow \texttt{o}))\\
&\texttt{refl}\,[a : \texttt{Type}] : (\texttt{Prf}\ \ (\forall\, a\, (\lambda x, (\texttt{=}\, a\, x\, x))))\\
&\texttt{eqind}\,[a : \texttt{Type}] :\\
&\quad (\texttt{Prf}\ \ (\forall\, (a \rightsquigarrow \texttt{o})\, (\lambda p, (\forall\, a\, \lambda x, (\forall\, a\, \lambda y, (p\, x) \Rightarrow (\texttt{=}\, x\, y) \Rightarrow (p\, y))))))\\
&\texttt{false} : (\texttt{El}\ \texttt{o}) := (\forall\, \texttt{o}\, (\lambda x, x))\\
&\texttt{not} : (\texttt{El}\ (\texttt{o} \rightsquigarrow \texttt{o})) := \lambda p, p \Rightarrow \texttt{false}\\
&\texttt{!=}\,[a : \texttt{Type}] : (\texttt{El}\ (a \rightsquigarrow a \rightsquigarrow \texttt{o})) := \lambda x, \lambda y, (\texttt{not}\ \ (\texttt{=}\, a\, x\, y))
\end{aligned}
$$

Figure 2.8: Signature for a polymorphic equality and inequality. `refl` is the constructor of the equality type and `eqind` is the eliminator. Falsity and negation are encoded using standard higher order techniques.

that type correctness conditions of *PVS* have a clear and explicit representation in the encoding, allowing its benefits to be transported to λΠme.

Example 5 (Sorted lists and proof irrelevance). This example is inspired by sorted lists in the *Agda* manual (The Agda Team 2021, section 'Irrelevance'). Because we have not encoded dependent types, we use the native product of the framework to encode them. The specification is given in Fig. 2.10.

This example illustrates the conciseness of predicate subtyping: the proof obligation $(h \leq b)$ is encoded into the type of h rather than passed as a standalone argument in Agda, shortening the type of `scons`. In Fig. 2.11, we declare two (non-convertible) axioms p_1 and p_2 to be proofs of $(\texttt{zero} \leq \texttt{suc}\,\texttt{zero})$ and two lists containing `zero` and proved to be bounded by $(\texttt{suc}\,\texttt{zero})$ using p_1 for ℓ_1 and p_2 for ℓ_2. Without proof irrelevance, equality $(=(\texttt{slist}\,(\texttt{suc}\,\texttt{zero}))\,\ell_1\,\ell_2)$ is not provable using `refl` because it requires $p_1 \simeq p_2$. With proof irrelevance, $(\texttt{refl}\,\ell_1)$ is an acceptable proof.

2.3 Preservation of typing by the encoding

In this section, we prove that the encoding preserves typing: if a *PVS-Cert* type is inhabited then its translation is inhabited too. This property is sometimes called correctness or soundness, and is a requirement for adequacy as defined by Harper and Licata 2007. Typing preservation increases the trust in the source system because in case the translation does not type check in λΠme, then the

$$
\begin{aligned}
&\texttt{stack} : \texttt{Type}\\
&\texttt{empty} : (\texttt{El}\ \texttt{stack})\\
&\texttt{nat} : \texttt{Type}\\
&\texttt{nonempty\_stack?} : (\texttt{El}\ (\texttt{stack} \rightsquigarrow \texttt{o})) := \lambda s, s \neq \texttt{empty}\\
&\texttt{nonempty\_stack} : (\texttt{psub}\ \texttt{nonempty\_stack?})\\
&\texttt{push} : (\texttt{El}\ (\texttt{stack} \rightsquigarrow \texttt{nat} \rightsquigarrow \texttt{nonempty\_stack}))\\
&\texttt{pop} : (\texttt{El}\ (\texttt{nonempty\_stack} \rightsquigarrow \texttt{stack}))\\
&\texttt{pop\_push} :\\
&\quad (\texttt{Prf}\ (\forall\, \lambda x : (\texttt{El}\ \texttt{nat}), (\forall\, \lambda s : (\texttt{El}\ \texttt{stack}), (\texttt{pop}\ (\texttt{push}\ x\ s)) = s)))\\
&\texttt{pop2push2}\,[x\ y : (\texttt{El}\ \texttt{nat})\,, s : (\texttt{El}\ \texttt{stack})] :\\
&\quad (\texttt{Prf}\ (\texttt{pop}\ (\texttt{pair}\ (\texttt{pop}\ (\texttt{push}\ x\ (\texttt{fst}\ (\texttt{push}\ y\ s))))\ \alpha) = s)) :=\\
&\quad \ldots
\end{aligned}
$$

Figure 2.9: Specification of stacks.

$$
\begin{aligned}
&\texttt{zero} : (\texttt{El}\ \texttt{nat})\\
&\texttt{suc} : (\texttt{El}\ (\texttt{nat} \rightsquigarrow \texttt{nat}))\\
&\leq : (\texttt{El}\ (\texttt{nat} \rightsquigarrow \texttt{nat} \rightsquigarrow \texttt{o}))\\
&\texttt{slist} : \Pi n : (\texttt{El}\ \texttt{nat}), \texttt{Type}\\
&\texttt{snil} : (\texttt{El}\ (\texttt{nat} \rightsquigarrow (\lambda n, (\texttt{slist}\ n))))\\
&\texttt{bounded} : \Pi n : (\texttt{El}\ \texttt{nat}), \texttt{Type} := \lambda b, (\texttt{psub}\ (\lambda n, n \leq b))\\
&\texttt{scons} :\\
&\quad (\texttt{El}\ (\texttt{nat} \rightsquigarrow \lambda b, (\texttt{bounded}\ b) \rightsquigarrow \lambda h, (\texttt{slist}\ h) \rightsquigarrow (\texttt{slist}\ b)))
\end{aligned}
$$

Figure 2.10: Specification of sorted lists.

$$p_1 : (\texttt{Prf}\,(\texttt{zero} \leq (\texttt{suc}\,\texttt{zero})))$$
$$p_2 : (\texttt{Prf}\,(\texttt{zero} \leq (\texttt{suc}\,\texttt{zero})))$$
$$\ell_1 : \ldots := (\texttt{scons}\ (\texttt{suc}\,\texttt{zero})\ (\texttt{pair}\,\texttt{zero}\,p_1)\ \texttt{snil})$$
$$\ell_2 : \ldots := (\texttt{scons}\ (\texttt{suc}\,\texttt{zero})\ (\texttt{pair}\,\texttt{zero}\,p_2)\ \texttt{snil})$$

Figure 2.11: Definition of two sorted lists with different proofs.

initial judgement must not type check in *PVS-Cert*.

Typing preservation does not guarantee that if a (translated) proof is inhabited in λΠme, then it is also inhabited in *PVS-Cert*. The translation that maps all types to $\Pi x : \texttt{Type}, (\texttt{El}\,(x \rightsquigarrow x))$ trivially preserves typing because all type translations are inhabited by $\lambda x : \texttt{Type}, \lambda y : (\texttt{El}\,x), x$. But we can provide a proof for the translation of falsity.

Conservativity (also called completeness) prevents such an anomaly by stating that any type in the image of the translation must be inhabited in the source system. The latter pathological translation violates this property.

It is more difficult in general to prove that encodings are conservative than type-preserving. Cousineau and Dowek 2007 prove that terminating pure type systems are conservative: a type in the image of the translation is inhabited in the source system if it is inhabited by a normal form in the framework. In our case, the termination of system $\mathcal{R}_{[\mathrm{Pe}]}$ with Eq. (β) can only be conjectured. More recently, Assaf 2015 has proved completeness for pure type systems that may be non terminating. But pure type systems use the congruence of β which is included in the congruence used by *PVS-Cert*. Finally, Felicissimo 2022 favour deep embeddings over shallow ones to prove more easily conservativity. Deep embeddings mark the distinction between administrative β redexes from meaningful redexes of the embedded system that are materialised by embedded and annotated abstractions ($\texttt{abs}\,A\,(\lambda x, B)\,(\lambda x, e)$) and annotated applications ($\texttt{app}\,A\,(\lambda x, B)\,e_1\,e_2$).

Lemma 2 (Preservation of substitution)**.** *If* $\Gamma, x : U, \Delta \vdash_{\overline{\mathrm{P}}} M : T$ *and* $\Gamma \vdash_{\overline{\mathrm{P}}} N : U$*, then* $[\{N/x\}\,M]_{\Gamma,\{N/x\}\Delta} = \left\{[N]_\Gamma/x\right\}[M]_{\Gamma,x:U,\Delta}$.

Proof. By structural induction on M. The proof is straightforward because applications are translated as applications and abstractions as abstractions. For the product case, note that by stratification of *PVS-Cert* (F. Gilbert 2018,

Proposition 5.4.1), the sort of a term is stable by substitution, so if $[\Pi x : M_1, M_2] = (\forall\ [\![M_1]\!]\ (\lambda x, [M_2]))$ (for instance), then $[\{N/x\}\,\Pi z : M_1, M_2]$ is still translated with a $\forall$ because $\{N/x\}\,M_1$ has the same sort as M_1 (and same for M_2). □

Lemma 3 (Preservation of equivalence)**.** *Let M and N be two well typed terms in Γ. If $M \simeq_{\mathrm{Pe}} N$, then $[M]_\Gamma \simeq_{[\mathrm{Pe}]} [N]_\Gamma$.*

Proof. We first show that if $M \simeq_{\mathrm{Pe}} N$ in a single step, then $[M] \simeq_{[\mathrm{Pe}]} [N]$. Using the notations of Lemma 1 Page 34, we show that

1. computational steps of $\hookrightarrow_{\beta,\pi_\ell}$ restricted to typeable terms are preserved,
2. equational steps of $\leftrightarrow_{pi}$ restricted to well typed terms are preserved.

These two properties are shown by induction on a context γ such that $M = \gamma[\hat{M}]\ R\ \gamma[\hat{N}] = N$ where R is any of the two relations applied at the head of $\hat{M}$ and $\hat{N}$. We will only detail the base cases of inductions, the other cases being straightforward.

Preservation of computation There are two possible cases: When $M = ((\lambda x, t)\,u) \hookrightarrow_\beta \{u/x\}\,t$, we have,

$$[(\lambda x : U, t)\ u]_\Gamma = \Big((\lambda x : [\![U]\!]_\Gamma, [t]_{\Gamma,x:U})\ [u]_\Gamma\Big) = \Big\{[u]_\Gamma/x\Big\}\,[t]_\Gamma \simeq_{[\mathrm{Pe}]} [\{u/x\}\,t]_\Gamma$$

where the equivalence is given by Lemma 2. When

$$M = \pi_\ell(T_1, P_1, \mathtt{pair}(T_0, P_0, t, h)) \hookrightarrow_{\pi_\ell} t$$

we have the following equalities

$$\begin{aligned}
&[\pi_\ell(T_1, P_1, (\mathtt{pair}(T_0, P_0, t, h)))]_\Gamma \\
&= \Big(\mathtt{fst}\ [T_1]_\Gamma\ [P_1]_\Gamma\ [\mathtt{pair}(T_0, P_0, t, h)]_\Gamma\Big) \\
&= \Big(\mathtt{fst}\ [T_1]_\Gamma\ [P_1]_\Gamma\ \Big(\mathtt{pair}\ [T_0]_\Gamma\ [P_0]_\Gamma\ [t]_\Gamma\ [h]_\Gamma\Big)\Big) \simeq_{[\mathrm{Pe}]} [t]_\Gamma
\end{aligned}$$

with the last equivalence provided by Eq. (2.3) Page 33.

Preservation of Proof Irrelevance Assume that $M = \mathtt{pair}(T, P, t, h) \leftrightarrow_{\mathrm{pi}} \mathtt{pair}(T, P, t, h')$

$$[\mathtt{pair}(T, P, t, h)]_\Gamma = \mathtt{pair}\,[T]_\Gamma\ [P]_\Gamma\ [t]_\Gamma\ [h]_\Gamma \simeq_{[\mathrm{Pe}]} \mathtt{pair}\,[T]_\Gamma\ [P]_\Gamma\ [t]_\Gamma\ [h']_\Gamma = [\mathtt{pair}(T, P, t, h')]_\Gamma$$

where the equivalence is given by Eq. (2.2) Page 33.

We now prove the main proposition. By Lemma 1 Page 34, we know that there are H_0 and H_1 such that $M \hookrightarrow^*_{\beta,\pi_\ell} H_0 \leftrightarrow^*_{\mathrm{pi}} H_1 \hookleftarrow^*_{\beta,\pi_\ell} N$ and that M, H_0, H_1 and N are typeable. For $R \in \{\leftrightarrow_{\mathrm{pi}}, \hookrightarrow_{\beta,\pi_\ell}\}$, we have $t\ R^*\ u \Rightarrow [t] \simeq_{[\mathrm{Pe}]} [u]$ by induction on the number of R steps, using the lemma just proved before for the base case. Therefore, $[M]_\Gamma \simeq_{[\mathrm{Pe}]} [H_0]_\Gamma \simeq_{[\mathrm{Pe}]} [H_1]_\Gamma \simeq_{[\mathrm{Pe}]} [N]_\Gamma$, which gives, by transitivity of $\simeq_{[\mathrm{Pe}]}$, $[M]_\Gamma \simeq_{[\mathrm{Pe}]} [N]_\Gamma$. □

Theorem 1 (Typing preservation)**.** *If* $\Gamma \vdash_{\mathrm{Pe}} M : T$*, then* $\Gamma \vdash_{[\mathrm{Pe}]} [M]_\Gamma : [\![T]\!]_\Gamma$*. For all* Γ*, if* $\vdash_{[\mathrm{Pe}]} \Gamma$*, then* $\vdash_{[\mathrm{Pe}]} [\![\Gamma]\!]$*.*

Proof. The two propositions are shown simultaneously by induction on the typing derivation of $\Gamma \vdash_{\mathrm{Pe}} M : T$.

EMPTY $\dfrac{}{\vdash_{\mathrm{Pe}} \emptyset}$

We have $[\![\emptyset]\!] = \emptyset$ and $\vdash_{[\mathrm{Pe}]} \emptyset$.

DECL $\dfrac{v \notin \Gamma \qquad \Gamma \vdash_{\mathrm{Pe}} T : s}{\vdash_{\mathrm{Pe}} \Gamma, v : T}$

We have $[\![\Gamma, v : T]\!] = [\![\Gamma]\!], v : [\![T]\!]_\Gamma$. By induction hypothesis, we have $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [T]_\Gamma : [\![s]\!]_\Gamma$, for $s \in \mathcal{S}$ and hence $[\![s]\!]_\Gamma$ is either `Elo`, `Type` or $\star$. If s is `Kind`, then T is `Type` (because `Type` is the only inhabitant of `Kind`). The induction hypothesis becomes by definition of the translation $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \mathtt{Type} : \star$ and we can derive $\vdash_{[\mathrm{Pe}]} [\![\Gamma]\!], v : \mathtt{Type}$ with DECL

If s is `Type`, since $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \mathtt{Type} : \star$ by declaration (2.4) Page 37, we can derive with DECL $\vdash_{[\mathrm{Pe}]} [\![\Gamma, v : T]\!]$ because $[\![\mathtt{Type}]\!] = \mathtt{Type}$. Otherwise, s is `Type` or `Prop` and $[\![T]\!] = \xi\ [T]_\Gamma$ where ξ is `El` or `Prf`. Because both `El` and `Prf` have $\star$ as domain (declarations (2.6) and (2.7)), $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [\![T]\!]_\Gamma : \star$ and finally, $\vdash_{[\mathrm{Pe}]} [\![\Gamma, v : T]\!]$ by application DECL.

VAR $\dfrac{v : T \in \Gamma \qquad \vdash_{\mathrm{Pe}} \Gamma}{\Gamma \vdash_{\mathrm{Pe}} v : T}$

By definition, $[v] = v$ and by induction hypothesis, $\vdash_{[\mathrm{Pe}]} [\![\Gamma]\!]$. Since $v : T \in \Gamma$, by definition, there is $\Delta \subsetneq \Gamma$ where $\vdash_{\mathrm{Pe}} \Delta$ such that, $v : [\![T]\!]_{\Delta} \in [\![\Gamma]\!]$. Hence $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} v : [\![T]\!]_{\Delta}$ and finally $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} v : [\![T]\!]_{\Gamma}$ because contexts are well formed.

SORT $\dfrac{\vdash_{\mathrm{Pe}} \Gamma \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash_{\mathrm{Pe}} s_1 : s_2}$

Sort s_1 is either `Prop` or `Type`.

- If $s_1 = \texttt{Prop}$, then $s_2 = \texttt{Type}$, and the judgement is translated as $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \texttt{o} : \texttt{Type}$. By induction hypothesis $\vdash_{[\mathrm{Pe}]} [\![\Gamma]\!]$ holds. Using rule SIGN with Eq. (2.5), the judgement $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \texttt{o} : \texttt{Type}$ is derivable.
- If $s_1 = \texttt{Type}$, then $s_2 = \texttt{Kind}$. In that case, the judgement is translated as $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \texttt{Type} : \star$. It is derivable using SIGN with declaration (2.4) and the induction hypothesis to have $\vdash_{[\mathrm{Pe}]} [\![\Gamma]\!]$.

PROD $\dfrac{\Gamma \vdash_{\mathrm{Pe}} T : s_1 \qquad \Gamma, x : T \vdash_{\mathrm{Pe}} U : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{P}}{\Gamma \vdash_{\mathrm{Pe}} \Pi x : T, U : s_3}$

We only detail for the product $(s_1, s_2, s_3) = (\texttt{Type}, \texttt{Prop}, \texttt{Prop})$, others being processed similarly. Because

$$[\Pi x : T, U]_{\Gamma} = \forall\, [T]_{\Gamma}\ \left(\lambda x : [\![T]\!]_{\Gamma}, [U]_{\Gamma, x:T}\right)$$

we want to show that

$$[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \forall\, [T]_{\Gamma}\ \left(\lambda x : [\![T]\!]_{\Gamma}, [U]_{\Gamma, x:T}\right) : [\![\texttt{Prop}]\!]\,.$$

By induction hypothesis we have $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [T] : [\![\texttt{Type}]\!]$ and $[\![\Gamma, x : T]\!] \vdash_{[\mathrm{Pe}]} [U] : [\![\texttt{Prop}]\!]$. By definition of the translation, we obtain $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [T] : \texttt{Type}$ and $[\![\Gamma]\!], x : [\![T]\!]_{\Gamma} \vdash_{[\mathrm{Pe}]} [U] : \texttt{Elo}$. Each judgement of the following sequence implies the derivability of the successive one,

$$\begin{aligned}
&[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \lambda x : [\![T]\!]_{\Gamma}, [U]_{\Gamma, x:T} : [\![T]\!]_{\Gamma} \to \texttt{Elo} && \text{derivable in } \lambda\Pi\text{me}\\
&[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \lambda x : [\![T]\!]_{\Gamma}, [U]_{\Gamma, x:T} : \texttt{El}\, [T]_{\Gamma} \to \texttt{Elo} && \text{by definition of } [\![-]\!]\\
&[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \lambda x : [\![T]\!]_{\Gamma}, [U]_{\Gamma, x:T} : \texttt{El}([T]_{\Gamma} \rightsquigarrow \texttt{o}) && \text{by CONV and Eq. (2.13)}\\
&[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \forall\, [T]_{\Gamma}\ \left(\lambda x, [\![T]\!]_{\Gamma} [U]_{\Gamma, x:T}\right) : \texttt{Elo} && \text{by typing of } \forall \text{ in } \Sigma_{[\mathrm{Pe}]}\\
&[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \forall\, [T]_{\Gamma}\ \left(\lambda x, [\![T]\!]_{\Gamma} [U]_{\Gamma, x:T}\right) : [\![\texttt{Prop}]\!] && \text{because } [\![\texttt{Prop}]\!] = \texttt{Elo}
\end{aligned}$$

$$\textbf{ABST}\ \frac{\Gamma, v : T \vdash_{\mathrm{Pe}} M : U \qquad \Gamma \vdash_{\mathrm{Pe}} \Pi v : T, U : s}{\Gamma \vdash_{\mathrm{Pe}} \lambda v : T, M : \Pi v : T, U}$$

We have $[\lambda v : T, M]_\Gamma = \lambda v : [\![T]\!]_\Gamma, [M]_\Gamma$ and the following sequence of implications

$$\begin{array}{ll}
[\![\Gamma, v : T]\!] \vdash_{[\mathrm{Pe}]} [M]_{\Gamma, v:T} : [\![U]\!]_{\Gamma, v:T} & \text{by induction hypothesis} \\
[\![\Gamma]\!], v : [\![T]\!]_\Gamma \vdash_{[\mathrm{Pe}]} [M]_{\Gamma, v:T} : [\![U]\!]_{\Gamma, v:T} & \text{by definition of } [\![-]\!] \\
[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \lambda v : [\![T]\!]_\Gamma, [M]_{\Gamma, v:T} : \Pi v : [\![T]\!]_\Gamma, [\![U]\!]_{\Gamma, v:T} & \text{by ABST}
\end{array}$$

In the last step, the product is well typed in λΠme since $[\![U]\!]$ and $[\![T]\!]$ are both of type $\star$ and thus the product is of type $\star$ as well.

Finally, we proceed by case distinction on sorts s_T and s_U such that $\Gamma \vdash_{\mathrm{Pe}} T : s_T$ and $\Gamma \vdash_{\mathrm{Pe}} U : s_U$. We will detail the case $(s_T, s_U) = (\texttt{Type}, \texttt{Prop})$. We have

$$\Pi v : [\![T]\!]_\Gamma, [\![U]\!]_{\Gamma, v:T} \simeq_{[\mathrm{Pe}]} \texttt{Prf}(\forall\, [T]_\Gamma\ (\lambda x : [\![T]\!]_\Gamma, [U]_{\Gamma, v:T})) = [\![\Pi v : T, U]\!]_\Gamma$$

which allows to conclude.

$$\textbf{APPL}\ \frac{\Gamma \vdash_{\mathrm{Pe}} M : \Pi v : T, U \qquad \Gamma \vdash_{\mathrm{Pe}} N : T}{\Gamma \vdash_{\mathrm{Pe}} (M\,N) : \{N/v\}\,U}$$

By induction hypothesis and conversion, we have $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [M]_\Gamma : \Pi v : [\![T]\!]_\Gamma$, $[\![U]\!]_{\Gamma, v:T}$ (shown by case distinction on the sorts of T and U) and $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [N]_\Gamma : [\![T]\!]_\Gamma$. Since $[M\,N]_\Gamma = [M]\ [N]$, we obtain using APPL $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [M\,N] : \left\{[N]_\Gamma/v\right\} [\![U]\!]_{\Gamma, v:T}$ and by Lemma 2 Page 44, we obtain $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [M\,N] : [\![\{N/v\}\,U]\!]_\Gamma$.

$$\textbf{CONV}\ \frac{\Gamma \vdash_{\mathrm{Pe}} M : U \qquad \Gamma \vdash_{\mathrm{Pe}} T : s \qquad T \simeq_{\mathrm{Pe}} U}{\Gamma \vdash_{\mathrm{Pe}} M : T}$$

By induction hypothesis, $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [M]_\Gamma : [\![U]\!]_\Gamma$.

We now prove that if $T \simeq_{\mathrm{Pe}} U$, then $[\![T]\!]_\Gamma \simeq_{[\mathrm{Pe}]} [\![U]\!]_\Gamma$ and $\Gamma \vdash_{[\mathrm{Pe}]} [\![T]\!] : \star$ which will allow us to conclude using CONV in λΠme.

By Lemma 1 Page 34 and because T and U are typeable, $\Gamma \vdash_{\mathrm{Pe}} U : s$. By Lemma 3 Page 45, $[T]_\Gamma \simeq_{[\mathrm{Pe}]} [U]_\Gamma$.

If $s = \texttt{Prop}$, then $[\![T]\!]_\Gamma = \texttt{Prf}\,[T]_\Gamma \simeq_{[\mathrm{Pe}]} \texttt{Prf}\,[U]_\Gamma = [\![U]\!]_\Gamma$. Moreover we have $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [\![T]\!]_\Gamma : \star$ because, by induction hypothesis, $[T]_\Gamma : [\![\texttt{Prop}]\!] =$

$\texttt{El}\,[\texttt{Prop}] = (\texttt{El}\,\texttt{o})$, and declaration (2.7) Page 37. If $s = \texttt{Type}$, $[\![T]\!]_\Gamma = \texttt{El}\,[T]_\Gamma \simeq_{[\mathrm{Pe}]} \texttt{El}\,[U]_\Gamma = [\![U]\!]_\Gamma$. By induction hypothesis, $[T]_\Gamma : [\![\texttt{Type}]\!]_\Gamma = \texttt{Type}$. If $s = \texttt{Kind}$, then $T = U = \texttt{Type}$ (Type is the only inhabitant of Kind). Finally, $[\![\texttt{Type}]\!] = \texttt{Type} : \star$.

$$\textsc{sign}\ \frac{f[\boldsymbol{x} : \boldsymbol{T}] : U : s \in \Sigma_{\mathrm{Pe}} \qquad \boldsymbol{x} : \boldsymbol{T} \vdash U : s \qquad \left(\Gamma \vdash t_i : \left\{t_j/x_j\right\}_{j<i} T_i\right)_i}{\Gamma \vdash (f\,\boldsymbol{t}) : \{\boldsymbol{t}/\boldsymbol{x}\}\,U}$$

We first observe from Fig. 2.4 Page 39 that for each $f \in \Sigma_{\mathrm{Pe}}$, we have a counterpart symbol $\hat{f} \in \Sigma_{[\mathrm{Pe}]}$ such that if $f[\boldsymbol{x} : \boldsymbol{T}] : U : s \in \Sigma_{\mathrm{Pe}}$ then $\hat{f}[\boldsymbol{x} : [\![\boldsymbol{T}]\!]] : [\![U]\!]_{\boldsymbol{x}:\boldsymbol{T}} : \star \in \Sigma_{[\mathrm{Pe}]}$.

We verify that for each declaration in the signature, the second premise $\boldsymbol{x} : \boldsymbol{T} \vdash U : s$ holds.

By induction hypothesis, for each i, we have

$$[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [t_i]_\Gamma : \left[\!\!\left[\left\{t_j/x_j\right\}_{j<i} T_i\right]\!\!\right]_\Gamma$$

which we can write as, thanks to Lemma 2,

$$[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} [t_i]_\Gamma : \left\{\left[t_j\right]_\Gamma/x_j\right\}_{j<i} T_i.$$

Using rule SIGN, we are able to conclude $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \hat{f}\ [\boldsymbol{t}]_\Gamma : \{[\boldsymbol{T}]/\boldsymbol{x}\}\,[\![U]\!]$. By Lemma 2, we obtain $[\![\Gamma]\!] \vdash_{[\mathrm{Pe}]} \hat{f}\ [\boldsymbol{t}]_\Gamma : [\![\{\boldsymbol{t}/\boldsymbol{x}\}\,U]\!]$. Moreover, we have taken care to define the translation in Fig. 2.6 Page 40 such that $[f(\boldsymbol{t})] = \hat{f}\,[\boldsymbol{t}]$. □

2.4 Mechanising type checking

2.4.1 Deciding equivalence

The encoding of *PVS-Cert* into λΠme can be used to proof check terms of *PVS-Cert* using a type checker for λΠme. But because of the CONV rule, type checking cannot be decidable if $\simeq_{[\mathrm{Pe}]}$ is not. To implement decidable equivalences, one may resort to rewriting (Baader and Nipkow 1998): given a *convergent* (*i.e.* confluent and terminating) rewrite relation $\hookrightarrow_R$, if $\simeq_R$ is the smallest congruence containing $\hookrightarrow_R$, then $s \simeq_R t$ can be decided by computing and comparing

the normal forms of s and t with respect to $\hookrightarrow_R$. Consequently, while type checkers cannot be provided for λΠme in general, they can when $\simeq$ is the smallest congruence containing a convergent rewrite relation (Guillaume Burel et al. 2016). Such type systems are named 'λΠ-calculus modulo rewriting' shortened λΠmr.

Definition 15 (λΠmr)**.** A λΠmr type system is a λΠme type system whose set of equations is replaced by a rewrite system. A λΠmr type system is parametrised by a triple $\mathfrak{R} = (\mathcal{F}, \mathcal{R}, \Sigma)$ where $\mathcal{R}$ is a rewrite system. It has the same typing rules as λΠme but its congruence $\simeq$ is the joinability relation $\downarrow_{\beta,\mathcal{R}}$ defined by the rule (β) (defined Page 31) and the rules of $\mathcal{R}$.

Definition 16 (Subject reduction)**.** We say that reduction $\hookrightarrow$ has the subject reduction property, or that it preserves typing if whenever $\Gamma \vdash t : A$ and $t \hookrightarrow u$, then $\Gamma \vdash u : A$.

Definition 17 (Type preserving rewrite rule)**.** Let Σ be a signature and $\mathcal{R}$ be a rewrite system. Let $\mathfrak{T}$ be the λΠmr type system parametrised by Σ and $\mathcal{R}$. A rewrite rule $\ell \hookrightarrow r$ *preserves typing* in $\mathfrak{T}$ if for any substitution σ, for any context Γ well-formed in $\mathfrak{T}$, for any term A well-sorted in $\mathfrak{T}$, if $\Gamma \vdash_{\mathfrak{T}} \sigma\ell : A$, then $\Gamma \vdash_{\mathfrak{T}} \sigma r : T$.

Definition 18 (Well-formed λΠmr system)**.** Let Σ be a signature and $\mathcal{R}$ be a rewrite system whose terms are in $\mathcal{T}(\{\star, \Box\}, \mathrm{dom}(\Sigma))$. We say that the λΠmr type system $\mathfrak{T}$ parametrised by Σ and $\mathcal{R}$ is *well-formed* when Σ is well-formed in $\mathfrak{T}$ and the rewrite system $\mathcal{R}$ is convergent, type-preserving in $\mathfrak{T}$, and, for any rule $\ell \hookrightarrow r$ in $\mathcal{R}$, ℓ and r are neither the sort $\star$ nor products of the form $\boldsymbol{\Pi x : U}, \star$.

In the latter definition, the last condition on the rewrite system allows to recover some of the benefits of the stratification of λΠmr (see Saillard 2015, Figure 2.1), the most useful one being the following corollary. A similar definition can be found in (Blanqui 2005).

Proposition 3. *For any well-formed* λΠ*mr type system, for any typeable term* t*, if* $t \simeq \Box$ *then* $t = \Box$*, if* $t \simeq \star$*, then* $t = \star$*.*

Proof. By well-formedness of the type system, there is no rule that rewrite to $\star$ by definition, and because $\Box$ is not typeable, there is no rule that rewrites to $\Box$. □

$$(\texttt{pair}\, t\, p\, m\, h) \hookrightarrow (\texttt{pair}^{\dagger} t\, p\, m) \tag{2.22}$$
$$(\texttt{fst}\, t_0\, p_0\ (\texttt{pair}^{\dagger}\, t_1\, p_1\, m)) \hookrightarrow m \tag{2.23}$$
$$(\texttt{Prf}\ (\forall\, t\, p)) \hookrightarrow \Pi x : (\texttt{El}\, t), (\texttt{Prf}\ (p\, x)) \tag{2.24}$$
$$(\texttt{El}\, (t \rightsquigarrow u)) \hookrightarrow \Pi x : (\texttt{El} t), (\texttt{El}\, (u\, x)) \tag{2.25}$$
$$(\texttt{Prf}\, (p \Rightarrow q)) \hookrightarrow \Pi h : (\texttt{Prf} p), (\texttt{Prf}\ (q\, h)) \tag{2.26}$$

Figure 2.12: Rewrite system $\mathcal{R}_{[\mathrm{Pe}]}$ obtained from the completion of equation of *PVS-Cert*.

Given an equational theory, a convergent rewrite system whose joinability is the same as the equational theory can be obtained through *completion procedures* (Baader and Nipkow 1998). However, completion procedures rely on well-founded orders that cannot be provided in the case of *PVS-Cert* because of Eq. (2.2) Page 33 which cannot be oriented since each side of the equation has a free variable which is not in the other side.

As noted by Knuth and Bendix 1983, the addition of a symbol to the signature can circumvent the issue. Hence, we add a symbol for proof irrelevant pairs, and make it equal to pairs

$$\texttt{pair}^{\dagger}\, [t : \texttt{Type}; p : (\texttt{El}\, (t \rightsquigarrow \texttt{o}))\, ; x : (\texttt{El}\, t)] : (\texttt{El}\ (\texttt{psub}\, t\, p)) : \star \tag{2.20}$$
$$(\texttt{pair}\, t\, p\, m\, h) = (\texttt{pair}^{\dagger}\, t\, p\, m) \tag{2.21}$$

thus $(\texttt{pair}\, t\, p\, m\, h) = (\texttt{pair}^{\dagger}\, t\, p\, m) = (\texttt{pair}\, t\, p\, m\, h')$. The new set of identities given by Eqs. (2.3), (2.11) to (2.13) and (2.21) Pages 33, 37 and 51 can be completed into a rewrite system $\mathcal{R}_{[\mathrm{Pe}]}$ which is equivalent to the equations.

Proposition 4 (Confluence)**.** *The rewrite relation* $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} = \hookrightarrow_{\beta} \cup \hookrightarrow_{\mathcal{R}_{[\mathrm{Pe}]}}$ *is confluent.*

Proof. The rewrite system in Fig. 2.12 is orthogonal, hence confluent (Klop, Oostrom and Raamsdonk 1993). □

Proposition 5. *Let* $\simeq_{\mathcal{R}_{[\mathrm{Pe}]}}$ *be the smallest congruence containing* $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$ *where* $\mathcal{R}_{[\mathrm{Pe}]}$ *is defined in Fig. 2.12 and* $\simeq_{[\mathrm{Pe}]}$ *the congruence of* $\lambda\Pi[\mathrm{Pe}]$ *(defined in Definition 13 Page 38). Then* $\simeq_{[\mathrm{Pe}]} \subseteq \simeq_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$.

Proof. It is enough to prove that every equation of *PVS-Cert* is included in $\simeq_{\mathcal{R}_{[\mathrm{Pe}]},\beta}$. This is immediate for Eqs. (2.11) to (2.13) since they are equal to the rules (2.24) to (2.26). For Eq. (2.2), we have

$$(\mathtt{pair}\, t\, p\, m\, h_0) \hookrightarrow_{\mathcal{R}_{[\mathrm{Pe}]}} (\mathtt{pair}^\dagger\, t\, p\, m) \hookleftarrow_{\mathcal{R}_{[\mathrm{Pe}]}} (\mathtt{pair}\, t\, p\, m\, h_1)\,.$$

Finally, for Eq. (2.3),

$$(\mathtt{fst}\, t_0\, p_0\ (\mathtt{pair}\, t_1\, p_1\, m\, h)) \hookrightarrow_{\mathcal{R}_{[\mathrm{Pe}]}} (\mathtt{fst}\, t_0\, p_0\ (\mathtt{pair}^\dagger\, t_1\, p_1\, m)) \hookrightarrow_{\mathcal{R}_{[\mathrm{Pe}]}} m.\square$$

Conjecture 1. *Rewrite relation* $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$ *is terminating.*

One possible solution to prove that conjecture is to extend the proof of termination for the encoding of simple type theory presented by Dowek 2017 to the rewrite relation $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$.

Assuming conjecture 1, relation $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$ is convergent, and therefore a type checker can be provided for λΠ[Pe].

Remark 5. The rewrite relation generated by the rewrite system $\mathcal{R}_{[\mathrm{Pe}]}$ Fig. 2.12 preserves typing (see Blanqui, Dowek et al. 2021, Theorem 9).

2.4.2 Bidirectional type checkers

Type systems presented so far use an undirected ternary relation $\Gamma \vdash t : A$. In such relations, just like in *Prolog* clauses, there is no notion of input or output. Undirected type systems are more succinct to formalise, but they are not suited for functional implementations because it requires to guess types. Guessing can be avoided by specifying carefully in inference rules what should be considered as inputs or outputs: bidirectional type systems (Benthem Jutting, McKinna and Pollack 1993; Dunfield and Krishnaswami 2019; Lennon-Bertrand 2021; Pierce and Turner 2000) use two relations, synthesis and checking. As devised by McBride 2018, we distinguish inputs that are assumed well formed from subjects that may not be well formed and from outputs which exist and are well formed if the judgement holds. Synthesis—or type inference—is denoted $\Gamma \vdash t \Rightarrow A$ where Γ is an input, t is the subject and A is an output. Checking $\Gamma \vdash t \Leftarrow A$ asserts that t is typeable by A where Γ and A are inputs and t is the subject.

Bidirectional type checkers are not only a matter of implementations and algorithms: they constrain more the shape of typing derivations than undirected type checkers.

Figure 2.13 Page 54 provides the inference rules of a bidirectional type checker for λΠmr, based on the work of Lennon-Bertrand 2021. Following (ibid.), we introduce constrained inference whenever some specific shape for terms is expected. In rules B-PROD-C and B-SORT-C, we must ensure that reduction $\hookrightarrow_{\beta,\mathcal{R}}$ preserves the postconditions of synthesis: if $\Gamma \vdash t : A$ and $A \hookrightarrow_{\beta,\mathcal{R}} B$, then $\Gamma \vdash t : B$. The first premise of rule B-SORT-C does not involve rewriting because of Proposition 3.

There is no need to check the well formedness of signatures in the bidirectional type checker (for the same reason as for contexts). Well formedness of contexts and signatures is assumed as preconditions for the type checker to behave well.

Proposition 6 (Correctness of checking and inference)**.** *Let $\mathfrak{T}$ be a well-formed λΠmr type system parametrised by a (well-formed) signature Σ and a (convergent and type preserving) rewrite system $\mathcal{R}$. Inference and checking satisfy the following properties*

$$\{\vdash_{\mathfrak{T}} \Gamma\}\ \Gamma \vdash_{\mathfrak{T}} t \Rightarrow A\ \{\Gamma \vdash_{\mathfrak{T}} t : A\}$$
$$\{\vdash_{\mathfrak{T}} \Gamma \wedge \Gamma \vdash_{\mathfrak{T}} A : s \wedge s \in \{\star, \Box\}\}\ \Gamma \vdash_{\mathfrak{T}} t \Leftarrow A\ \{\Gamma \vdash_{\mathfrak{T}} t : A\}$$

Proof. The proof is close to the one of (ibid., Theorem 2), we remind the key points. By mutual induction on the typing derivation. Rules of the bidirectional system (including the new rule B-SIGN) are replaced by rules of the undirected system where B-CHECK, B-PROD-C and B-SORT-C are replaced by CONV. Rule B-ABST uses an additional PROD rule, we detail the derivation in the undirected system as an example (we dropped the type system annotation):

$$\dfrac{\dfrac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : s_B}{\Gamma \vdash \Pi x : A, B : s}\ \textsc{prod} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}\ \textsc{abst}$$

where we omit the premise $(\star, s_B, s) \in \mathcal{P}^{\lambda\Pi}$ for spacing issues. All leaves of the tree are obtained by induction hypothesis, with $\Gamma, x : A \vdash B : s_B$ obtained by validity (sometimes called 'correctness of types') of pure type systems (H. P. Barendregt, Dekkers and Statman 2013; Blanqui 2001; Coquand and Huet 1988). In rule B-APPL, because the reduction $\hookrightarrow_{\beta,\mathcal{R}}$ preserves typing (by hypothesis on $\mathcal{R}$), A_1 is well sorted and hence the induction hypothesis can be applied on the second premise. □

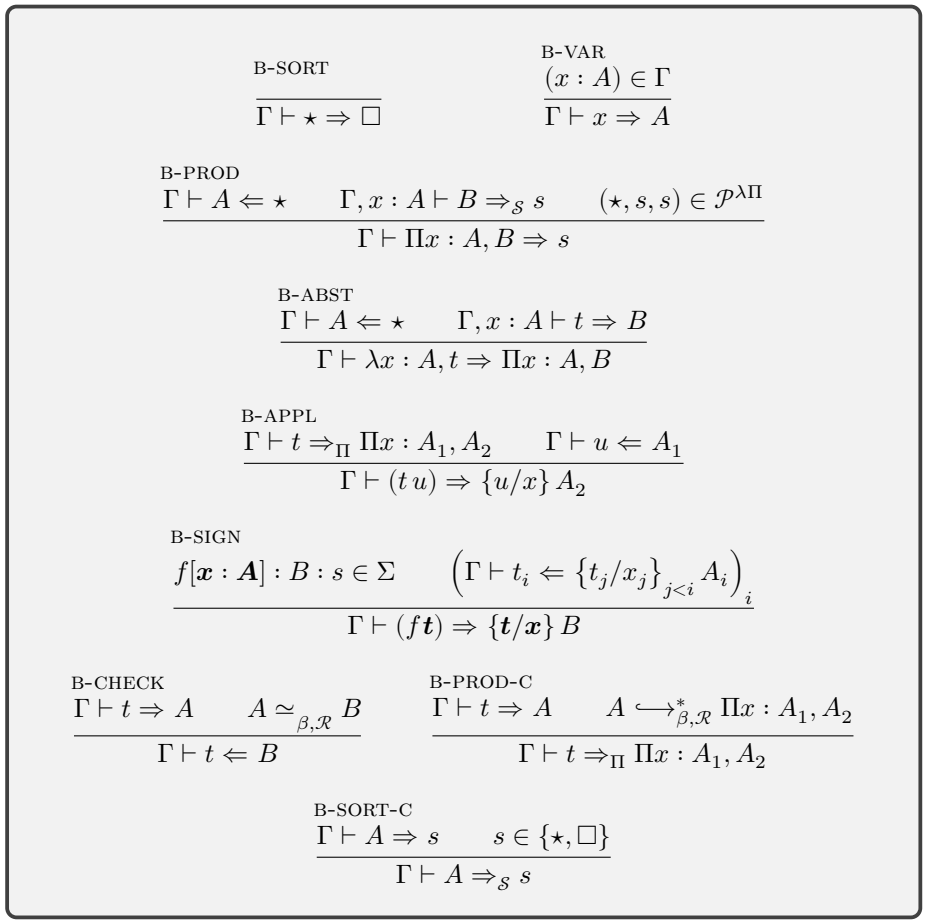

Figure 2.13: Bidirectional type checker for λΠmr.

Proposition 7 (Completeness of inference)**.** *For any well-formed* λΠ*mr type system* $\mathfrak{T}$ *whose congruence is denoted* $\simeq$*, If* $\Gamma \vdash_{\mathfrak{T}} t : A$ *then there is* A' *such that* $\Gamma \vdash_{\mathfrak{T}} t \Rightarrow A'$ *and* $A \simeq A'$*.*

Proof. By induction on the undirected typing derivation.

For rule ABST, the induction hypothesis gives $\Gamma \vdash \Pi x : A, B \Rightarrow R$, and inversion of typing rules provides $\Gamma \vdash A \Leftarrow \star$. By induction hypothesis we have $\Gamma, x : A \vdash t \Rightarrow B'$ where $B' \simeq B$, and hence we conclude $\Gamma \vdash \lambda x : A, t \Rightarrow \Pi x : A, B'$ where $\Pi x : A, B \simeq \Pi x : A, B'$.

For rule PROD, we have by induction hypothesis that $\Gamma \vdash A \Rightarrow R$ where $R \simeq \star$ and $\Gamma, x : A \vdash B \Rightarrow S$ where $S \simeq s \in \{\star, \Box\}$. By confluence of $(\beta \cup \mathcal{R})$, if $S \simeq s$, then $S \hookrightarrow^*_{\beta,\mathcal{R}} s$ (because $s \in \{\star, \Box\}$ is in normal form), hence $\Gamma, x : A \vdash B \Rightarrow_{\mathcal{S}} s$.

For rule APPL, induction hypothesis gives $\Gamma \vdash t \Rightarrow C$ with $C \simeq \Pi x : A, B$. By confluence of $\beta \cup \mathcal{R}$ and product compatibility (see Saillard 2015, Theorem 2.6.11), $C \hookrightarrow_{\beta,\mathcal{R}} \Pi x : C_1, C_2$ where $C_1 \simeq A$ and $C_2 \simeq B$. Hence $\Gamma \vdash t \Rightarrow_{\Pi} \Pi x : C_1, C_2$. Induction hypothesis also gives $\Gamma \vdash u \Rightarrow A'$ with $A' \simeq A$. By transitivity of $\simeq$, we can apply B-CHECK to derive $\Gamma \vdash u \Leftarrow C_1$ and then apply rule B-APPL.

Rule SIGN can be replaced by B-SIGN: induction hypotheses give $\Gamma \vdash t_i \Rightarrow \{t_j/x_j\}_{j<i} A'_i$ with $\{t_j/x_j\} A'_i \simeq \{t_j/x_j\} A_i$, with which B-CHECK can be applied sequentially. Note that Σ (or at least the declaration used) is well formed by hypothesis, since $\boldsymbol{x} : \boldsymbol{A} \vdash B : s$ is a premise.

For rule CONV, the induction hypothesis directly provide the premises required to apply B-CHECK. □

We prove an additional lemma that will be used afterwards,

Lemma 4. *For any well-formed* λΠ*mr type system, if* $\Gamma \vdash t \Rightarrow A$*, then* $\Gamma \vdash A \Rightarrow_{\mathcal{S}} s$*.*

Proof. If $\Gamma \vdash t \Rightarrow A$, by correctness, $\Gamma \vdash t : A$. By validity, $\exists s \in \{\star, \Box\}, \Gamma \vdash A : s$. By completeness $\exists u, \Gamma \vdash A \Rightarrow u$ and $u \simeq s$. Denoting $\mathcal{R}$ the rewrite system parametrising the (well-formed) λΠmr type system, by confluence of $\mathcal{R}$ and because s is a normal form, $u \hookrightarrow^*_{\beta,\mathcal{R}} s$. □

2.5 Conservativity of computations

We declared in Section 2.3 Page 42 that an encoding is conservative whenever the inhabitation of an encoded type implies the inhabitation of the original type

in the original system. To prove such property, the conservativity of computations is needed: whenever encoded term $[t]$ computes to encoded term $[u]$ in the framework, then t computes to u in the original system. In our case, we must prove that the joinability defined by β and the rewrite rules of $\mathcal{R}_{[\mathrm{Pe}]}$ (*i.e.* the congruence for the encoding of *PVS-Cert* in λΠmr) does not identify more terms than the original equivalence $\simeq_{\mathrm{Pe}}$ (Definition 10) (see Felicissimo 2022, Proposition 37). In the presence of proof irrelevance and mechanised typing, the introduction of terms $\mathtt{pair}^\dagger$ may prevent the former property to hold. We prove in Proposition 8 Page 59 that symbols added to encodings to implement proof irrelevance do not allow the conversion to identify more terms.

We denote

- for any rewrite relation $\hookrightarrow$, $s \overset{p}{\hookrightarrow} t$ if $s \hookrightarrow t$ where the rewrite rule is applied on position p (so $s = \{\sigma\ell/p\}\, s \hookrightarrow \{\sigma r/p\}\, s = t$ where $\ell \hookrightarrow r$)
- $\hookrightarrow_\dagger$ the context and substitution closure of rule (2.22) Page 51,
- $\hookrightarrow_{\mathtt{fst}^\dagger}$ the context and substitution closure of rewrite rule (2.23),
- $\hookrightarrow_{\mathtt{fst}}$ the context and substitution closure of Eq. (2.19) oriented from left to right.
- $\mathcal{R}^+$ the rewrite system containing rules (2.24) to (2.26) and identity (2.19) oriented from left to right (therefore, $\mathcal{R}^+$ is $\mathcal{R}_{[\mathrm{Pe}]}$ without the rules that involve $\mathtt{pair}^\dagger$, which are replaced by another rule).

We intend to show that any $\hookrightarrow_\dagger$ rewrite step can either be grouped with a $\hookrightarrow_{\mathtt{fst}^\dagger}$ rewrite step, resulting in a $\hookrightarrow_{\mathtt{fst}}$ rewrite step, or it can be postponed. Said differently, any $\hookrightarrow_\dagger$ rewrite step is either involved in a pair projection, or it is used to discard a proof, in which case the proof can be discarded at the end of the rewrite sequence. Therefore, any rewrite sequence can be reordered into a sequence s_+ that involve $\hookrightarrow_{\mathtt{fst}}$ rewrite steps but no $\hookrightarrow_\dagger$ step, followed by a sequence of $\hookrightarrow_\dagger$ steps. Rewrite steps of s_+ are straightforwardly conservative with respect to $\simeq_{[\mathrm{Pe}]}$, and then, remarking that for any position p, $\overset{p}{\hookrightarrow}_\dagger \overset{p}{\hookleftarrow}_\dagger$ is precisely the equation for proof irrelevance, it is contained in $\simeq_{[\mathrm{Pe}]}$. We thus show that any sequence $\hookrightarrow_\dagger^* \hookleftarrow_\dagger^*$ can be reordered into a sequence $\left(\overset{p_i}{\hookrightarrow}_\dagger \overset{p_i}{\hookleftarrow}_\dagger\right)_i^*$.

Lemma 5. *Let p and q be two positions and s, t, u such that $s \overset{p}{\hookleftarrow}_{\dagger} t \overset{q}{\hookrightarrow}_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} u$. If $t \overset{q}{\hookrightarrow}_{\mathtt{fst}^{\dagger}} u$ and $p = 3, q$ then $s \hookrightarrow_{\mathtt{fst}} u$. Otherwise, there is t' such that $s \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} t' \hookrightarrow_{\dagger}^{=} u$.*

Proof. By inspection of the concerned rewrite rules. □

We show here that $\hookrightarrow_{\dagger}$ rewrite steps are either part of the computation of a projection, in which case they can be grouped with a $\hookrightarrow_{\mathtt{fst}^{\dagger}}$ into a $\hookrightarrow_{\mathtt{fst}}$ rewrite rule (included in $\mathcal{R}^{+}$), or the rewrite step can be postponed.

Lemma 6. *Let t and u be two terms of $\mathcal{T}\left(\mathcal{F}_{[\mathrm{Pe}]}\right)$ such that $t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u$. Then $t \hookrightarrow_{\beta,\mathcal{R}^{+}}^{*} \hookrightarrow_{\dagger}^{*} u$.*

Proof. First, note that for any positions p, q, any terms s, t, u, if $s \overset{p}{\hookleftarrow}_{\dagger} t \overset{q}{\hookrightarrow}_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} u$ and if $t \overset{q}{\hookrightarrow}_{\mathtt{fst}^{\dagger}} u$ and $p = 3, q$ then $s \hookrightarrow_{\mathtt{fst}} u$. Otherwise, there is t' such that $s \hookrightarrow_{\beta,\mathcal{R}^{+}} t' \hookrightarrow_{\dagger}^{=} u$. This can be shown by inspection of the concerned rewrite rules.

We proceed by induction on the number of $\hookrightarrow_{\mathtt{fst}\dagger}$ reduction. If there is no $\hookrightarrow_{\mathtt{fst}\dagger}$ rewrite step, then all $\hookrightarrow_{\dagger}$ rewrite steps can be postponed (using the former lemma). Now assume $t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u$ with $n \hookrightarrow_{\mathtt{fst}\dagger}$ rewrite steps, where $n \geq 1$. The rewrite sequence is of the form $t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} t_0 \hookrightarrow_{\mathtt{fst}\dagger} \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u$ where there is no $\hookrightarrow_{\mathtt{fst}\dagger}$ reduction between t and t_0. There must be a $\hookrightarrow_{\dagger}$ reduction to form the redex in t_0 because t does not contain any $\mathtt{pair}^{\dagger}$ symbol since it is in $\mathcal{T}\left(\mathcal{F}_{[\mathrm{Pe}]}\right)$. Therefore the reduction is of the form

$$t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} t_1 \hookrightarrow_{\dagger} t_2 \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} t_0 \hookrightarrow_{\mathtt{fst}\dagger} \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u$$

Since there is no $\hookrightarrow_{\mathtt{fst}\dagger}$ reduction between t_2 and t_0, the $\hookrightarrow_{\dagger}$ rewrite step can be postponed:

$$t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} t_3 \overset{3,p}{\hookleftarrow}_{\dagger} t_0 \overset{p}{\hookrightarrow}_{\mathtt{fst}\dagger} \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u.$$

The rewrite sequence can be transformed into

$$t \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} t_3 \hookrightarrow_{\mathtt{fst}} \hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}^{*} u$$

which contains $n-1$ $\hookrightarrow_{\mathtt{fst}\dagger}$ rewrite steps. We conclude by induction hypothesis. □

Note that t cannot contain symbol $\texttt{pair}^\dagger$ when $t \in \mathcal{T}\left(\mathcal{F}_{[\mathrm{Pe}]}\right)$. The second item of the following proposition is the converse of confluence. It will be used to group $\hookrightarrow_\dagger$ rewrite steps by pairs operating at the same position because for any position p, ${}^{p}\!\hookrightarrow_\dagger {}^{p}\!\hookleftarrow_\dagger \subseteq \simeq_{[\mathrm{Pe}]}$.

Lemma 7. *For any positions p and q,*

- ${}^{p}\!\hookrightarrow_\dagger {}^{q}\!\hookrightarrow_\dagger \subseteq {}^{q}\!\hookrightarrow_\dagger {}^{p}\!\hookrightarrow_\dagger^{=}$
- *If $p \neq q$,* ${}^{p}\!\hookrightarrow_\dagger {}^{q}\!\hookleftarrow_\dagger \subseteq {}^{q}\!\hookleftarrow_\dagger {}^{p}\!\hookrightarrow_\dagger$

Proof. By case analysis: either the first rewrite step is applied on a subterm erased by the second rewrite step (below the fourth argument of $\texttt{pair}$), in which case it can be discarded; or the two rewrite steps do not interfere with each other.

Assume there are e and e' such that $e\ {}^{p}\!\hookrightarrow_\dagger\ c\ {}^{q}\!\hookleftarrow_\dagger\ e'$. First we know that we have redexes at positions p and q in e and e' respectively, $e|_p = (\texttt{pair}\, e_0\, e_1\, e_2\, e_3)$ and $e'|_q = (\texttt{pair}\, e'_0\, e'_1\, e'_2\, e'_3)$. But we also know that $e|_q = c|_q$ and $e'|_p = c|_p$, so $e|_q = (\texttt{pair}^\dagger\, e'_0\, e'_1\, e'_2)$ and $e'|_p = (\texttt{pair}^\dagger\, e_0\, e_1\, e_2)$. Thus we can build $c' = \{\texttt{pair}\, e'_0\, e'_1\, e'_2\, e'_3/q\}\, e = \{\texttt{pair}\, e_0\, e_1\, e_2\, e_3/p\}\, e'$ such that $e\ {}^{q}\!\hookleftarrow_\dagger\ c'\ {}^{p}\!\hookrightarrow_\dagger\ e'$. □

We show next that if t and u do not contain any $\texttt{pair}^\dagger$ and they both reduce to a same term by discarding their proofs, then they are convertible with respect to proof irrelevance.

Lemma 8. *If $t, u \in \mathcal{T}\left(\mathcal{F}_{[\mathrm{Pe}]}\right)$ and $t \hookrightarrow_\dagger^* u \hookleftarrow_\dagger^* s$, then $t \simeq_{[\mathrm{Pe}]} s$.*

Proof. Using Lemma 7, we can commute rewrite steps of a sequence $x \hookrightarrow_\dagger^* y$ in order to obtain a sequence of minimal length (removing reflexive steps) where each rewrite step is applied on a position p such that $u|_p = \texttt{pair}^\dagger\ldots$. Because t and u are in $\mathcal{T}(\Sigma_{[\mathrm{Pe}]})$, there is no symbol $\texttt{pair}^\dagger$ in t or s, and thus there is exactly one rewrite step for each $\texttt{pair}^\dagger$ symbol in u (can be shown by induction on the number of $\texttt{pair}^\dagger$ symbols where the induction hypothesis is applied on subterms of t). Therefore, the two sequences have the same length, and if there is a step ${}^{p}\!\hookrightarrow_\dagger$ in one, then there is the same step in the other.

Last, with Lemma 7, the valley $t \hookrightarrow_\dagger^{=} u \hookleftarrow_\dagger^* s$ can be rearranged into $t\ ({}^{p}\!\hookrightarrow {}^{p}\!\hookleftarrow)^*\ s$ and because ${}^{p}\!\hookrightarrow {}^{p}\!\hookleftarrow \subseteq \simeq_{[\mathrm{Pe}]}$, we obtain $t \simeq_{[\mathrm{Pe}]} s$. □

Proposition 8. *Let* $M, N \in \mathcal{T}\left(\mathcal{F}_{[\mathrm{Pe}]}\right)$ *such that* $M \simeq_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} N$. *Then* $M \simeq_{[\mathrm{Pe}]} N$.

Proof. Because $\hookrightarrow_{\beta,\mathcal{R}_{[\mathrm{Pe}]}}$ is confluent (Proposition 4 Page 51), there is α such that $M \hookrightarrow^*_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} \alpha \hookleftarrow^*_{\beta,\mathcal{R}_{[\mathrm{Pe}]}} N$. By Lemma 6, there are M' and N' such that

$$M \hookrightarrow^*_{\beta,\mathcal{R}^+} M' \hookrightarrow^*_{\dagger} \alpha \hookleftarrow^*_{\dagger} N' \hookleftarrow^*_{\beta,\mathcal{R}^+} N.$$

By Lemma 8, $M' \simeq_{[\mathrm{Pe}]} N'$. Since $\hookrightarrow_{\beta,\mathcal{R}^+} \subseteq \simeq_{[\mathrm{Pe}]}$ (all rules of $\mathcal{R}^+$ and Eq. (β) are contained in $\simeq_{[\mathrm{Pe}]}$), we have $M \simeq_{[\mathrm{Pe}]} M'$, $N \simeq_{[\mathrm{Pe}]} N'$, and finally by transitivity of $\simeq_{[\mathrm{Pe}]}$, $M \simeq_{[\mathrm{Pe}]} N$. □

In implementations Regarding *Dedukti*, the existence of $\mathtt{pair}^\dagger$ threatens the conservativity of any development. Indeed, for any predicate P, the function '$\lambda e, (\mathtt{snd}\, T\, P\, (\mathtt{pair}^\dagger\, e))$' maps any term e (of type $(\mathtt{El}\, T)$) to a proof of (Pe) regardless of the provability of (Pe) in *PVS-Cert*.

But if we remember that the symbol $\mathtt{pair}^\dagger$ has been created only to implement a proof irrelevant conversion, Dedukti developers may verify that their development does not contain the symbol $\mathtt{pair}^\dagger$: each time an inhabitant of a subtype is needed, the symbol `pair` must be used. If proof irrelevance is needed, the conversion will take care of erasing proofs.

To help developers enforce such an invariant, Dedukti relies on the notion of scope (in the usual sense for programming languages) and modules (see F. Thiré and G. Férey 2019). We say that a symbol is *protected*[2] if it is used to implement proof irrelevance. A module $\mathcal{M}$ is a pair made of a list of imported signatures and a signature, where a signature is simply a list of declarations. See the work of Chrzaszcz 2003; Courant 1997; Norell 2007 for studies on modules in pure type systems.

Let $\mathcal{M}$ be a module whose signature $\Sigma^\dagger$ contains protected symbols declarations, and let Σ be $\Sigma^\dagger$ without protected symbol declarations. Because Σ must declare rewrite rules that reduce to terms containing $\mathtt{pair}^\dagger$, there is no restriction on the usage of $\mathtt{pair}^\dagger$ in Σ. When working on a module $\mathcal{M}'$ that imports $\mathcal{M}$, only Σ can be used to build terms (types, definitions and rewrite rules). Nonetheless, this restriction can be softened in the case of rewrite rules left-hand sides. If $\mathcal{M}'$ declares a rewrite rule $\ell \hookrightarrow r$ with `pair` in ℓ, then a critical pair is formed: taking the case of *PVS-Cert*, for any context γ,

[2] Protected symbols are called *private* by F. Thiré and G. Férey 2019.

$\gamma\,[\ell[\texttt{pair}^\dagger\,x\,y\,z]\sigma] \longleftarrow \gamma[\ell\sigma] \hookrightarrow \gamma[r\sigma]$. This critical pair can be avoided by using $\texttt{pair}^\dagger$ instead of $\texttt{pair}$ in ℓ. Therefore, protected symbols are allowed in left-hand sides, regardless of the module, since in rewrite rules, only terms of the right-hand side may be created. François Thiré 2020 has used such mechanisms to encode proof irrelevance for cumulative type systems. The mechanism described here is similar to private types studied by Blanqui, Hardin and Weis 2007 that have been implemented in *OCaml* (Leroy et al. 2022, Section 10.3).

2.6 Conclusion

In Section 2.1, we defined the family of *type systems modulo*. This family contains λΠme, the extension of Edinburgh's logical framework with equations, simple type theory as well as simple type theory with explicit predicate subtyping and *proof irrelevance* which has been named *PVS-Cert*.

Section 2.2 presents an λΠme signature suitable to embed terms from *PVS-Cert*. The embedding function transforming *PVS-Cert* typing judgements into λΠme typing judgements is also given, and we give examples of theories expressed in *PVS-Cert*, but encoded in λΠme. These examples feature predicate subtyping and proof irrelevance.

In Section 2.3, the former encoding is proved to *preserve typing*: whenever a judgement holds in *PVS-Cert*, its embedding holds in λΠme. The converse, called *completeness* or *conservativity* is left open.

The penultimate Section 2.4 revolves around decidability of type checking. The family of type systems λΠmr is introduced. An λΠmr type system is like an λΠme type system whose equational theory is defined by a convergent rewrite system, in order to obtain a decidable equational theory. The equational theory for *PVS-Cert* is translated into a confluent rewrite system whose termination is left open. In particular, symbols have to be added to λΠmr to handle proof irrelevance. To ensure decidability of type checking, *bidirectional type checking* is introduced.

The last section anticipates the proof of conservativity of the encoding by showing that the rewrite system used to encode *PVS-Cert* in λΠmr is conservative: whenever two embedded terms are equivalent in λΠmr, then both terms are equivalent in *PVS-Cert*. The section closes on some implementation-related directions to avoid soundness issues raised by the symbols added to embed proof irrelevance.

Chapter 3

Coercions in computational logical frameworks

The previous chapter laid the foundations of a language with predicate subtyping along with its encoding in Edinburgh's Logical Framework modulo equations (λΠme). Type checking in that encoding is decidable because subtyping is explicit and types are equivalent up to a decidable convertibility relation implemented by a convergent rewrite system (convergence, and hence decidability are rather a conjecture).

Predicate subtyping is generally used implicitly: the type checker guesses where subtyping occurs, and there is no syntactic construction to mark the type of terms. Our next objective is to synthesise these subtyping information. That synthesis allows first to cross check proofs of systems that use implicit predicate subtyping (such as *PVS*); and second to write much more concise expressions in the framework. That information can either be generated with *ad-hoc* type checking algorithms (F. Gilbert 2018; Sozeau 2006); or with generic coercion insertion algorithms.

Unlike F. Gilbert 2018; Luo, Soloviev and Xue 2013; Tannen et al. 1991, we work with typing judgements rather than typing derivations. In these works, translation functions are defined on typing derivations (with subtyping rules) and return terms with coercions. For instance, for *PVS-Cert* defined in Fig. 2.2 Page 32, we would have the following definition where $[\![-]\!]$ denotes the transla-

tion

$$\left[\!\!\left[\dfrac{\dfrac{\rho}{\Gamma \vdash t : \mathtt{psub}(A,P)}}{\Gamma \vdash t : A}\right]\!\!\right] := \pi_\ell\left(A, P, \left[\!\!\left[\dfrac{\rho}{\Gamma \vdash t : \mathtt{psub}(A,P)}\right]\!\!\right]\right)$$

where the translation function insert the coercion π_ℓ to transform the implicit subtyping rule

$$\dfrac{\Gamma \vdash t : \mathtt{psub}(A,P)}{\Gamma \vdash t : A}$$

into a SIGN rule (defined in Fig. 2.1 Page 29). Our goal is to define a translation function which has the same output but that is defined on typing judgements rather than typing derivations, so that we would have $[\![\Gamma \vdash t : A]\!] = \pi_\ell(A, P, \dots)$.

In the rest of this document, because we focus on one type system that we extend, we will not define each new feature as a new system, but will add these features on a system named $\mathfrak{S}$. For now, $\mathfrak{S}$ is λΠmr, its set of terms is $\mathcal{T}(\mathcal{X}, \{\star, \Box\}, \mathcal{F})$ for some countable $\mathcal{X}$, some finite $\mathcal{F}$, $\mathcal{X}$, $\mathcal{F}$ and $\{\star, \Box\}$ pairwise disjoints and its typing relation is defined by the inference rules of Fig. 2.13 Page 54 parameterised by a rewrite system $\mathcal{R}$ and a signature Σ.

The translation function $[\![-]\!]$ relies on a term refiner that will be presented formally. That refiner is able to transform a term by the means of coercions. It is possible to compute coercions with a rewrite system: some examples as well as some limitations will be provided. We will need to encode yet unknown terms into our framework. For this we introduce *holes* as lightweight existential variables (Muñoz 1997).

3.1 Term refiner

Refiners—also called elaborators—are type checkers which accept a larger class of terms than the type checkers exposed so far. As a ternary relation, they are not designed to be correct with respect to type checkers: there may be terms t such that $\Gamma \vdash t \Rightarrow A$ holds with a refiner but not without. Refiners return the input term being type checked as an output of the type checking (and inference) relations which are now quaternary relations: synthesis is written $\Gamma \vdash t \rightsquigarrow u : A$ and reads 'term t refines to u of type A (in context Γ)' while checking is written $\Gamma \vdash t : A \rightsquigarrow u$ and reads 'term t refines to u when checked against type A (in context Γ)'. In both judgements, terms (and contexts) on the left of the arrow

'$\rightsquigarrow$' are either subjects or inputs of the judgement and terms that are on its right are outputs.

The output term serves two purposes. The first is to avoid performing twice term transformations during type checking by passing refined term around judgements. It also serves as a justification, indicating how the refiner transformed the term to accept it.

3.1.1 Definitions

Figure 3.1 defines a refiner parametrised by a cast relation for λΠmr. Most inference rules are the same as the type checker's one in Fig. 2.13 Page 54 with a new output. The main difference lies in the check rule B-CHECK: the judgement $\Gamma \vdash t \Leftarrow A$ may hold even when the type inferred from t is not convertible with A. In that case a coercion may be used to transform t (of type T) into a term u of type A. Denoting T the type inferred from t, this transformation is represented by the judgement $t : T <\!: A \rightsquigarrow u$. Type T may be called the source type and A the target type.

Definition 19 (Cast relation)**.** A *cast* relation is a subset of $\mathcal{T}^4$ where $\mathcal{T}$ abbreviates the set of terms of λΠmr. The cast relation is denoted $t : A <\!: B \rightsquigarrow u$.

For any well-formed λΠmr type system, a cast relation is *valid* if it satisfies the following property

$$\{\vdash \Gamma \wedge \Gamma \vdash A \Rightarrow_{\mathcal{S}} s \wedge \Gamma \vdash t \Leftarrow A \wedge \Gamma \vdash B \Rightarrow_{\mathcal{S}} s\}$$
$$t : A <\!: B \rightsquigarrow t'$$
$$\{\Gamma \vdash t' \Leftarrow B\}$$

Definition 20 (Coercion, coercion system)**.** We call *coercion system* any set of inference rules or axioms that can be used to derive a cast $t : T <\!: U \rightsquigarrow u$. Axioms of such coercion systems can be called *coercions*.

Lemma 9 (Correctness of R-CAST)**.** *Rule* R-CAST *(defined in Fig. 3.1) is a valid cast relation.*

Proof. If the rule applies, then $t' = t$ and $A \simeq_{\beta,\mathcal{R}} B$. We can conclude using B-CHECK. □

The following lemma provides inversion rules for the refiner.

Lemma 10 (Inversion of typing rules Fig. 3.1)**.**

- *If* $\Gamma \vdash x \rightsquigarrow t : R$ *then* $x : R \in \Gamma$ *and* $t = r$.

$$\textsc{R-SORT}\ \frac{}{\Gamma \vdash \star \rightsquigarrow \star : \square} \qquad \textsc{R-VAR}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : A}$$

$$\textsc{R-ABST}\ \frac{\Gamma \vdash A : \star \rightsquigarrow A' \qquad \Gamma, x : A' \vdash t \rightsquigarrow t' : B}{\Gamma \vdash \lambda x : A, t \rightsquigarrow \lambda x : A', t' : \Pi x : A', B}$$

$$\textsc{R-PROD}\ \frac{\Gamma \vdash A : \star \rightsquigarrow A' \qquad \Gamma, x : A' \vdash B \rightsquigarrow B' :_{\mathcal{S}} s \qquad (\star, s, s) \in \mathcal{P}^{\lambda\Pi}}{\Gamma \vdash \Pi x : A, B \rightsquigarrow \Pi x : A', B' : s}$$

$$\textsc{R-APPL}\ \frac{\Gamma \vdash t \rightsquigarrow t' :_{\Pi} \Pi x : A_1, A_2 \qquad \Gamma \vdash u : A_1 \rightsquigarrow u'}{\Gamma \vdash (t\, u) \rightsquigarrow (t'\, u') : \{u'/x\}\, A_2}$$

$$\textsc{R-SIGN}\ \frac{f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma \qquad \left(\Gamma \vdash t_i : \left\{t'_j/x_j\right\}_{j<i} A_i \rightsquigarrow t'_i\right)_i}{\Gamma \vdash (f\, \boldsymbol{t}) \rightsquigarrow (f\, \boldsymbol{t'}) : \{\boldsymbol{t'}/\boldsymbol{x}\}\, B}$$

$$\textsc{R-PROD-C}\ \frac{\Gamma \vdash t \rightsquigarrow t' : A \qquad A \hookrightarrow^*_{\beta,\mathcal{R}} \Pi x : A_1, A_2}{\Gamma \vdash t \rightsquigarrow t' :_{\Pi} \Pi x : A_1, A_2}$$

$$\textsc{R-SORT-C}\ \frac{\Gamma \vdash t \rightsquigarrow t' : s \qquad s \in \{\star, \square\}}{\Gamma \vdash t \rightsquigarrow t' :_{\mathcal{S}} s} \qquad \textsc{R-CHECK}\ \frac{\Gamma \vdash t \rightsquigarrow t' : A \qquad t' : A <: B \rightsquigarrow t''}{\Gamma \vdash t : B \rightsquigarrow t''}$$

$$\textsc{R-CAST}\ \frac{A \simeq_{\beta,\mathcal{R}} B}{t : A <: B \rightsquigarrow t}$$

Figure 3.1: Bidirectional type inference and type checking with refinement for λΠmr.

- *If* $\Gamma \vdash \lambda x : T_1, t_2 \rightsquigarrow t' : R$ *then* $R = \Pi x : T_1', R_2$ *for some* T_1' *and* R_2 *with* $(\Gamma \vdash T_1 : \star \rightsquigarrow T_1')$, $(\Gamma, x : T_1' \vdash t_2 \rightsquigarrow t_2' : R_2)$ *and* $t' = \lambda x : T_1', t_2'$.
- *If* $\Gamma \vdash (t_1\, t_2) \rightsquigarrow t' : R$, *then there are some types* T_{11}, R_1 *such that* $\Gamma \vdash t_1 \rightsquigarrow t_1' : \Pi x : T_{11}, R_1$, $\Gamma \vdash t_2 : T_{11} \rightsquigarrow t_2'$, *and* $R = \{t_2'/x\}\, R_1$, $t' = (t_1'\, t_2')$.
- *If* $\Gamma \vdash \Pi x : T_1, T_2 \rightsquigarrow t' : R$ *then there are sorts* s_1, s_2 *and* s_3 *and types* T_1', T_2' *with* $R = s_1$, $t' = \Pi x : T_1', T_2'$, $(\Gamma \vdash T_1 : \star \rightsquigarrow T_1')$, $(\Gamma, x : T_1' \vdash T_2 \Rightarrow_{\mathcal{S}} s_2)$ *and* $(s_1, s_2, s_3) \in \mathcal{P}^{\lambda\Pi}$.
- *If* $\Gamma \vdash (f\,\boldsymbol{t}) \rightsquigarrow t' : R$ *where* $f \in \Sigma$, $R = \{\boldsymbol{t}'/\boldsymbol{x}\}\, B$ *for some* B *with* $f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma$, *for all* i, $\Gamma \vdash t_i : \left\{t_j'/x_j\right\}_{j<i} A_i \rightsquigarrow t_i'$ *and* $t' = \boldsymbol{t}'$.
- *If* $\Gamma \vdash t \rightsquigarrow t' :_{\Pi} R$ *then* $R = \Pi x : T_1, T_2$ *for some* T_1, T_2 *with* $\Gamma \vdash t \rightsquigarrow t' : T$ *and* $T \hookrightarrow^*_{\mathcal{R}} \Pi x : T_1, T_2$.
- *If* $\Gamma \vdash t \rightsquigarrow t' :_{\mathcal{S}} R$ *then* $R \in \{\star, \Box\}$ *with* $\Gamma \vdash t \rightsquigarrow t' : s'$ *and* $s' \hookrightarrow^*_{\mathcal{R}} R$.
- *If* $\Gamma \vdash t : R \rightsquigarrow t'$ *then* $t'' : T \lessdot R \rightsquigarrow t'$ *for some* t'', T *with* $\Gamma \vdash t \rightsquigarrow t'' : T$.

Proof. By inspection of the typing rules Fig. 3.1. □

Definition 21. We say that B is *reachable* from A if there are terms t and u such that $t : A \lessdot B \rightsquigarrow u$, and we write $A \lessdot B$. We write $t : A \lessdot B$ to state that there a term u such that $t : A \lessdot B \rightsquigarrow u$.

Potential additional checking rules Some refiners have more than one introduction rule for the checking relation. In particular, Asperti, Ricciotti, Coen and Tassi 2018; Norell 2007; Pierce and Turner 2000 provide the following rule

R-CHECK-ABST

$$\frac{\Gamma \vdash A : \star \rightsquigarrow A' \qquad B \hookrightarrow^*_{\beta,\mathcal{R}} \Pi x : B_1, B_2 \qquad A' \simeq_{\beta,\mathcal{R}} B_1 \qquad \Gamma, x : A' \vdash t : B_2 \rightsquigarrow t'}{\Gamma \vdash \lambda x : A, t : B \rightsquigarrow \lambda x : A', t'}.$$

It pushes typing information up to the leaves of terms, traversing abstractions. This allows deriving more precise coercion problems, but it is not required for

completeness with respect to the bidirectional type checker. Given an algorithm to generate coercions, it extends the set of admissible terms, as shown in Example 6.

Example 6 (Checking abstractions with a dedicated rule). Let Γ be the context containing the declarations

$$\Gamma = nat : \texttt{Type}, even? : (\texttt{El}\ (nat \rightsquigarrow \texttt{o})), e : (\texttt{El}\ (\texttt{psub}\ nat\ even?))\,.$$

Checking the function $\lambda x : (\texttt{El}\ nat), e$ against $(\texttt{El}\ nat) \to (\texttt{El}\ nat)$ yields the following derivation tree (using rules of Fig. 3.1),

$$\frac{\begin{array}{c}\vdash \lambda x, e \rightsquigarrow \lambda x, e : (\texttt{El}\ nat \to (\texttt{El}\ (\texttt{psub}\ evenp))) \\ \lambda x, e : (\texttt{El}\ nat \to (\texttt{El}\ (\texttt{psub}\ evenp))) <: (\texttt{El}\ nat) \to (\texttt{El}\ nat) \rightsquigarrow \ldots\end{array}}{\vdash \lambda x, e : (\texttt{El}\ nat) \to (\texttt{El}\ nat) \rightsquigarrow \ldots}.$$

The second premise shows that the coercion system must be able to traverse products. However, if rule R-CHECK-ABST is added to the type checker, the coercion problem becomes $e : (\texttt{El}\ (\texttt{psub}\ evenp)) <: (\texttt{El}\ nat) \rightsquigarrow (\texttt{fst}\ e)$ and does not need the former specific coercion rule.

Such an additional checking rule pushes typing information inside the abstraction and transfers the task of inspecting terms from the coercion algorithm to the type checker. However, such a rule is generally used when the inference is not defined on abstractions, so that there may be only one rule to derive $\Gamma \vdash \lambda x : E, e \Leftarrow A$. In that case, only terms in β-normal form may be type checked, (in objects, inference is required only for heads of applications, so we miss only terms that have an abstraction as the head of an application, that is, a β-redex). Because we intend to refine terms that are not in β-normal form, we keep the inference of abstractions.

3.1.2 Refiner specification

We give formal specifications for the three judgements $\Gamma \vdash t \rightsquigarrow t : t$, $\Gamma \vdash t : t \rightsquigarrow t$ and $t : t <: t \rightsquigarrow t$.

The soundness of the refiner is established with respect to the typing relations defined by the rules of Fig. 2.13 Page 54 parametrised by the same signature and rewrite system.

Proposition 9 (Validity of R-CAST)**.** *The cast relation implemented by the rule* R-CAST *defined in Fig. 3.1 is valid.*

Proof. For any well-formed λΠmr type system parametrised by a rewrite system $\mathcal{R}$, if the rule R-CAST applies, then $t' = t$ and $A \simeq_{\beta,\mathcal{R}} B$. We can conclude using rule B-CHECK. □

Proposition 10 (Correctness of refiner)**.** *For any well-formed* λΠ*mr type system, any valid cast relation, the inference and checking relations defined in Fig. 3.1 validate the following property*

$$\{\vdash \Gamma\}\ \Gamma \vdash t \rightsquigarrow t' : A\ \{\Gamma \vdash A \Rightarrow_{\mathcal{S}} s \wedge \Gamma \vdash t' \Leftarrow A\}$$
$$\{\vdash \Gamma \wedge \Gamma \vdash A \Rightarrow_{\mathcal{S}} s\}\ \Gamma \vdash t : A \rightsquigarrow t'\ \{\Gamma \vdash t' \Leftarrow A\}$$

Proof. By mutual induction on refiner typing derivations. For rules R-VAR, R-SORT, R-ABST, R-PROD, R-APPL, it is enough to replace judgements of the form $\Gamma \vdash t \rightsquigarrow t' : A$ by $\Gamma \vdash t' \Rightarrow A$ and $\Gamma \vdash t : A \rightsquigarrow t'$ by $\Gamma \vdash t' \Leftarrow A$ to obtain correct derivations in the bidirectional system. For rule R-SIGN, the same operation holds, noting that because Σ is well-formed, preconditions hold for the sequence of premises that type $(t_i)_i$. The procedure allows to derive the first precondition using Lemma 4 Page 55 and the second because if $\Gamma \vdash t \Rightarrow A$, then $\Gamma \vdash t \Leftarrow A$.

For the rule R-CHECK, induction hypothesis ensures that the preconditions required by the cast relation hold, and correctness of the cast relation (Lemma 9) gives $\Gamma \vdash t'' \Leftarrow A$. □

Proposition 11 (Partial completeness of refiner)**.** *For any well-formed* λΠ*mr type system,*

- *if* $\Gamma \vdash t \Rightarrow A$*, then* $\Gamma \vdash t \rightsquigarrow t : A$*;*
- *for any type* A *well-sorted in* Γ*, if* $\Gamma \vdash t \Leftarrow A$*, then* $\Gamma \vdash t : A \rightsquigarrow t$*.*

Proof. The two propositions are proved simultaneously by induction on the typing derivation. The proof is straightforward, bidirectional premises can be replaced by refiner-style premises. The only exception is the rule B-CHECK: $\Gamma \vdash t \Rightarrow A$ can be replaced by $\Gamma \vdash t \rightsquigarrow t : A$ by induction hypothesis. Since $A \simeq B$ (where $\simeq$ is the congruence of the λΠmr type system), we can deduce $t : A \lessdot B \rightsquigarrow t$, and finally we can conclude with R-CHECK. □

3.1.3 Properties of coercion systems

Coercions are often used to specify semantics of type systems with implicit subtyping (Luo, Soloviev and Xue 2013; Tannen et al. 1991). Coercion operators transform implicit subtyping inference rules into application rules. In that

context, coercion insertion functions work on typing derivations of the source language (with implicit subtyping). Such a function is *coherent* if derivations with the same conclusion are translated to behaviourally equivalent terms (i.e. convertible terms).

Example 7. In the context of programming languages, it is common to coerce from numbers to booleans. Assume we have several coercions from integers to booleans $\texttt{IntToBool} : \texttt{Int} \to \texttt{Bool}$, from integers to float (less common) $\texttt{IntToFloat} : \texttt{Int} \to \texttt{Float}$ and from floats to booleans $\texttt{FloatToBool} : \texttt{Float} \to \texttt{Bool}$. There are at least two derivations with conclusion $\texttt{1} : \texttt{Bool}$ whose translations are $(\texttt{IntToBool}\,\texttt{1})$ (where $\texttt{1}$ is coerced from $\texttt{Int}$ to $\texttt{Bool}$) and $(\texttt{FloatToBool}\,(\texttt{IntToFloat}\,\texttt{1}))$ (where $\texttt{1}$ is coerced from $\texttt{Int}$ to $\texttt{Float}$ to $\texttt{Bool}$).

Definition 22 (Coherence)**.** Let $\simeq$ be a congruence. A cast relation is *coherent* with respect to $\simeq$ if and only if all elements of the set $\{u \mid t : A \lessdot B \rightsquigarrow u\}$ are convertible with respect to $\simeq$.

Note that in our case, because coercion insertion (which can be seen as interpretation) is performed together with the elaboration of the derivation, non-coherence would result from the non determinism of the elaboration of the derivation.

Example 8. In Example 7, there are two ways to coerce an integer to a boolean: either directly with $\texttt{IntToBool}$ or by composing $\texttt{IntToFloat}$ with $\texttt{FloatToBool}$. The coercion system is coherent if whatever the coercion system used, the result is the same, *i.e.*, denoting the function composition with $\circ$, $\texttt{IntToBool} = \texttt{FloatToBool} \circ \texttt{IntToFloat}$.

Subtyping rules may often be used more than once in a row. For instance, assuming the derivation

$$\dfrac{\dfrac{\Gamma \vdash 2 : \texttt{Even}}{\Gamma \vdash 2 : \texttt{Int}}}{\Gamma \vdash 2 : \texttt{Bool}}$$

its translation, noted $[\![-]\!]$ is

$$\left[\!\!\left[\dfrac{\dfrac{\Gamma \vdash 2 : \texttt{Even}}{\Gamma \vdash 2 : \texttt{Int}}}{\Gamma \vdash 2 : \texttt{Bool}}\right]\!\!\right] = \left(\texttt{IntToBool}\ \left[\!\!\left[\dfrac{\Gamma \vdash 2 : \texttt{Even}}{\Gamma \vdash 2 : \texttt{Int}}\right]\!\!\right]\right) = (\texttt{IntToBool}\,(\texttt{EvenToInt}\,2))\,.$$

In our case, because we do not start from the typing derivation, but from the conclusion of that derivation $\Gamma \vdash 2 : \texttt{Bool}$, our algorithmic coercion system must be able to synthesise a sequence of coercions.

Definition 23 (Transitivity)**.** A cast relation is *transitive* whenever the following rule is admissible

$$\frac{s : S <\!: T \rightsquigarrow t \qquad t : T <\!: U \rightsquigarrow u}{s : S <\!: U \rightsquigarrow u'}.$$

The output of the conclusion for transitivity is u' and not u: there is no reason for u and u' to be convertible *a priori*.

Example 9. Considering the coercion system composed of the two rules

$$\frac{}{e : \texttt{Float} <\!: \texttt{Bool} \rightsquigarrow (\texttt{FloatToBool}\, e)}$$

$$\frac{}{e : \texttt{Int} <\!: \texttt{Float} \rightsquigarrow (\texttt{IntToFloat}\, e)}$$

a transitivity rule is required to derive $e : \texttt{Int} <\!: \texttt{Bool} \rightsquigarrow e'$.

Definition 24 (Stability)**.** A cast relations is *stable by substitution* (or just stable) if for any term u, the following rule is admissible

$$\frac{t : A <\!: B \rightsquigarrow t'}{\{u/x\}\, t : \{u/x\}\, A <\!: \{u/x\}\, B \rightsquigarrow \{u/x\}\, t'}.$$

3.1.4 Standard coercions for functions

Since products and abstractions are native objects of the language, we can define higher order coercion rules to handle products. It is common to have covariance on the codomain: a function type $\Pi x : A, B$ is a subtype of $\Pi x : A, C$ if B is a subtype of C. For instance, with coercions defined in Example 9, the function type $\texttt{Bool} \to \texttt{Int}$ is a subtype of the function type $\texttt{Bool} \to \texttt{Float}$. We may also have contravariance on the domain which states that $B \to A$ is a subtype of $C \to A$ if C is a subtype of A (the order of subtyping is reversed on domains) (Pierce 2002, pp. 184-185). For instance, the function type $\texttt{Float} \to \texttt{Bool}$ is a subtype of $\texttt{Int} \to \texttt{Bool}$.

The following rule may be used to implement covariance (on the codomain) and contravariance (on the domain),

$$\frac{x : B_1 <\!: A_1 \rightsquigarrow e_x \qquad (f\, e_x) : A_2 <\!: B_2 \rightsquigarrow e}{f : \Pi x : A_1, A_2 <\!: \Pi x : B_1, B_2 \rightsquigarrow \lambda x : B_1, e}$$

or if only covariance on the codomain is desired, we may use the following one:

$$\frac{A_1 \simeq_{\beta,\mathcal{R}} B_1 \qquad (f\,x) : A_2 \lessdot B_2 \rightsquigarrow e_f}{f : \Pi x : A_1, A_2 \lessdot \Pi x : B_1, B_2 \rightsquigarrow \lambda x : B_1, e_f}.$$

These two rules η-expand their arguments. To avoid such modification of the term, and because λΠmr does not have η-equivalence, we can replace the latter rule by a restricted form

$$\frac{A_1 \simeq_{\beta,\mathcal{R}} B_1 \qquad e \hookrightarrow^*_{\mathcal{R},\beta} \lambda x : E_0, e_0 \qquad e_0 : A_2 \lessdot B_2 \rightsquigarrow e_1}{e : \Pi x : A_1, A_2 \lessdot \Pi x : B_1, B_2 \rightsquigarrow \lambda x : B_1, e_1}.$$

Note that as soon as the precondition $\Gamma \vdash e \Leftarrow \Pi x : A_1, A_2$ is validated, whenever $e \hookrightarrow^*_{\mathcal{R}} \lambda x : E_0, e_0$, we have $E_0 \simeq_{\beta,\mathcal{R}} A_1$ by inversion of rules B-CHECK and B-ABST (Page 54).

3.1.5 Coercing to functions

When inferring the type of an application $(f\,t)$, a product type must be found for the head of the application f. In R-PROD Page 64, products are only searched among reducts. Were the type checker able to coerce the head to some functions, more terms would be accepted:

Example 10. Let Σ be $\Sigma_{[\mathrm{Pe}]}$ (defined Fig. 2.4 Page 39) extended with the declarations

$$\begin{aligned}
&\mathtt{nat} : \mathtt{Type}\\
&0 : (\mathtt{El}\,\mathtt{nat})\\
&\mathtt{cont?} : (\mathtt{El}\,((\mathtt{nat} \rightsquigarrow \mathtt{nat}) \rightsquigarrow \mathtt{o}))\\
&\mathtt{h} : (\mathtt{Prf}\ (\mathtt{cont?}\,(\lambda x : (\mathtt{El}\,\mathtt{nat}), x)))\,.
\end{aligned}$$

The application of $(\mathtt{pair}\,(\lambda x : (\mathtt{El}\,\mathtt{nat}), x)\,\mathtt{h})$ to 0 does not type check in Σ because the head of the application is a pair, hence its type is of the form $(\mathtt{El}\ (\mathtt{psub}\ ...))$ which is not convertible with a product.

On the other hand, the application of $(\mathtt{fst}\ (\mathtt{pair}\,(\lambda x : (\mathtt{El}\,\mathtt{nat}), x)\,h))$ to 0 type checks (and reduces to $((\lambda x, x)\,0)$).

The following rule may be used to coerce heads of applications to functions

$$\frac{\Gamma \vdash f \rightsquigarrow f' : A \qquad \Gamma \vdash \Pi x : A_1, A_2 \Rightarrow_{\mathcal{S}} s \qquad f' : A \lessdot \Pi x : A_1, A_2 \rightsquigarrow f''}{\Gamma \vdash f \rightsquigarrow f'' :_{\Pi} \Pi x : A_1, A_2}$$

but it requires to guess types A_1 and A_2. Although asking for a terminating procedure enumerating all well-formed reachable types is certainly too much, we may be able to define a terminating relation that computes some subtypes of a given type.

Definition 25 (Subtype projection)**.** For any well-formed λΠmr type system, a relation $\prec$ on types is a valid *subtype projection* if

$$\{\vdash \Gamma \wedge \Gamma \vdash A \Rightarrow_{\mathcal{S}} s\}\ A \prec B\ \{\Gamma \vdash B \Rightarrow_{\mathcal{S}} s \wedge A <: B\}\,.$$

Example 11. In the encoding of *PVS-Cert* Fig. 2.4 Page 39, ($\texttt{nat} \rightsquigarrow \texttt{nat}$) is a well-formed type code that can be extracted from ($\texttt{psub}\,(\texttt{nat} \rightsquigarrow \texttt{nat})\,\texttt{cont?}$) and that can be mapped to a product through $\texttt{El}$. The subtyping relation thus ought to contain (where $\prec^*$ is the transitive and reflexive closure of $\prec$)

$$(\texttt{El}\ (\texttt{psub}\,(\texttt{nat} \rightsquigarrow \texttt{nat})\,\texttt{cont?})) \prec^* (\texttt{El}\,\texttt{nat}) \to (\texttt{El}\,\texttt{nat})\,.$$

We extend (and replace) rule R-PROD-C with

$$\text{R-PROD-C}\quad \frac{\Gamma \vdash t \rightsquigarrow t' : A \qquad A \prec^* \Pi x : A_1, A_2 \qquad t' : A <: \Pi x : A_1, A_2 \rightsquigarrow t''}{\Gamma \vdash t \rightsquigarrow t'' :_\Pi \Pi x : A_1, A_2}$$

Definition 26. System $\mathfrak{S}$ is extended with rule R-PROD-C parametrised by a subtype projection $\prec$.

Lemma 11 (Correctness of R-PROD-C with respect to inference)**.** *Rule* R-PROD-C *is correct with respect to Proposition 10 Page 67.*

Proof. Induction hypothesis and post-conditions of the first and second premises provide the required preconditions for the third premise to hold. Correctness of the coercion relation allows to conclude. □

Remark 6. Rule R-PROD-C could also be formulated

$$\frac{\Gamma \vdash t \rightsquigarrow t' : A \qquad A \prec^* \Pi x : A_1, A_2 \qquad \Gamma \vdash t' : \Pi x : A_1, A_2 \rightsquigarrow t''}{\Gamma \vdash t \rightsquigarrow t'' :_\Pi \Pi x : A_1, A_2}$$

but this formulation infers twice the type of t: once in the first premise, and the second as the premise of $\Gamma \vdash t : \Pi x : A_1, A_2 \rightsquigarrow t'$.

3.2 Computing coercions

For now, the only way to introduce $t : A <: B \rightsquigarrow u$ is through rule R-CAST page 64. We have also seen other potential introduction rules for functions. This section investigates how derivations for the cast relations can be built using a rewrite system. The goal here is to provide an implementation or a decidable procedure to derive coercion judgements given a finite set of coercions.

3.2.1 Initial observations

The set of admissible cast judgements denoted $t : T <: U \rightsquigarrow u$ is defined by sets of inference rules whose conclusion (or premises) are schemes of cast judgements $\underline{t} : \underline{T} <: \underline{U} \rightsquigarrow \underline{u}$, where terms of the form $\underline{t}$ denote schemes, that is, terms that may contain metavariables. Given a set of inference rules, procedures to derive admissible relation instances can be implemented using λ*Prolog*-like inference systems (Miller and Nadathur 2019; Tassi and Coen 2022). Such algorithms are based on unification: to build a derivation tree for $t : A <: B \rightsquigarrow u$, terms t, A and B (the inputs of the problem) are unified with the conclusions of the available rules, which may contain metavariables (like the rules described in Section 3.1.4 Page 69).

However, full-fledged unification is not needed in the case of coercions. Inputs of coercion judgements[1] are λ terms and do not contain metavariables. When judging whether $t : A <: B \rightsquigarrow u$ holds, t, A and B are λ terms. It suffices to look for coercion rules whose conclusion $\underline{p} : \underline{X} <: \underline{Y} \rightsquigarrow \underline{y}$ is such that $\underline{p}$ filters t, $\underline{X}$ filters A and $\underline{Y}$ filters B (we do not care about $\underline{u}$ because it is an output of the judgement, so it will always be a metavariable). If such a rule is found, noting σ the substitution such that $\sigma\underline{p} = t$, $\sigma\underline{X} = A$ and $\sigma\underline{Y} = B$, for each premise $\underline{p'} : \underline{X'} <: \underline{Y'} \rightsquigarrow \underline{y'}$ of the coercion rule, we are left with new coercion judgements $\sigma\underline{p'} : \sigma\underline{X'} <: \sigma\underline{Y'} \rightsquigarrow \sigma\underline{y'}$.

Since only filtering is needed, it may be possible to implement the inference algorithm using a rewrite system defining an operator κ: for each coercion rule of the form

$$\frac{\left(\underline{p_i} : \underline{X_i} <: \underline{Y_i} \rightsquigarrow \underline{e_i}\right)_i}{\underline{p} : \underline{X} <: \underline{Y} \rightsquigarrow \underline{e}\left[\underline{e_i}\right]_i}$$

[1] In the sense given in Section 2.4, opposed to outputs

where $\underline{e}[\underline{e}_i]_i$ denotes that terms $\underline{e}_i$ may occur in $\underline{e}$, declare a rewrite rule

$$\kappa\left(\underline{p} : \underline{X} \lessdot \underline{Y}\right) \hookrightarrow \underline{e}\left[\kappa\left(\underline{p}_i : \underline{X}_i \lessdot \underline{Y}_i\right)_i\right].$$

3.2.2 Computing coercions with a rewrite system

Coercing a term t from a type A to another type B amounts to triggering some computation on term t depending on A and B (and sometimes t itself). Since our system $\mathfrak{S}$ already handles computation through rewriting, we detail how to compute coercions with a rewrite system. Computational content for coercions is added on a symbol κ which extends the syntax of $\mathfrak{S}$. In this context, $t : A \lessdot B \rightsquigarrow u$ holds whenever $(\kappa\, A\, B\, t)$ reduces to term u.

Definition 27. Let $\mathcal{F}$ be a set of symbols, and κ a symbol not in $\mathcal{F}$. We call *$\mathcal{F$-coercion rewrite system*} any rewrite system whose terms are in $\mathcal{T}(\mathcal{F} \cup \{\kappa\})$. We may omit the set of symbols and call it a *coercion rewrite system* when the set of symbols can be unambiguously inferred.

Definition 28. Consider system $\mathfrak{S}$ parametrised by a signature Σ, let $\mathcal{F}$ be the domain of Σ and $\kappa \notin \mathcal{F}$. We extend the system by adding the rule

$$\text{R-COERCE}\quad \frac{(\kappa\, A\, B\, t) \hookrightarrow^{+}_{\beta,\mathcal{C},\mathcal{R}} t' \qquad \kappa \notin t'}{t : A \lessdot B \rightsquigarrow t'}$$

parametrised by a $\mathcal{F}$-coercion system $\mathcal{C}$ to the rules of Fig. 3.1 Page 64.

Adding coercions to the system amounts to declare rewrite rules $(\kappa\, x\, y\, z) \hookrightarrow r$ on system $\mathcal{C}$. This rewrite system is used in R-COERCE to transform the input term t. The second premise of the former rule ensures that the symbol κ is erased from t', meaning that all coercions have been found. It is equivalent to saying that $t' \in \mathcal{T}(\{\star, \square\}, \mathcal{F})$.

Remark 7. The relation in the first premise of R-COERCE is not reflexive, because if no rewrite step is performed, there is no way to erase the symbol κ. The symmetric closure $\hookrightarrow^{*}_{\beta,\mathcal{C},\mathcal{R}}$ could be used instead: if no rewrite step occur, the second premise fails.

Example 12 (El coercion). Using rule R-COERCE and signature $\Sigma_{[\mathrm{Pe}]}$, the type family El can be used as a coercion when the rewrite rule $(\kappa\,\mathtt{Type}\,\star\,x) \hookrightarrow \mathtt{El}x$ is in $\mathcal{C}$. This coercion allows terms to not use the function El at all, allowing for instance the term $(\lambda nat : \mathtt{Type}, \lambda n : nat, n)$ to be typeable.

We only show parts of the typing derivation that are relevant regarding the coercion

$$\dfrac{\dfrac{}{nat : \mathtt{Type} \vdash nat \rightsquigarrow nat : \mathtt{Type}}\ \text{R-VAR} \qquad \dfrac{(\kappa\,\mathtt{Type}\,\star\,nat) \hookrightarrow (\mathtt{El}\,nat)}{nat : \mathtt{Type} \lessdot : \star \rightsquigarrow (\mathtt{El}\,nat)}\ \text{R-COERCE}}{nat : \mathtt{Type} \vdash nat : \star \rightsquigarrow (\mathtt{El}\,nat)}\ \text{R-CHECK}$$

where the conclusion is used to prove the judgement $nat : \mathtt{Type} \vdash \lambda n : nat, n \rightsquigarrow \lambda n : (\mathtt{El}\,nat), n : \Pi n : (\mathtt{El}\,nat), (\mathtt{El}\,nat)$ using rule R-ABST.

Remark 8. To keep type checking decidable, we need $\hookrightarrow_{\beta,\mathcal{R},\mathcal{C}}$ to be convergent, at least using a fixed strategy (Pol 2001).

Proposition 12. *Let $\mathfrak{T}$ be a well-formed λΠmr type system parametrised by a (well-formed) signature Σ and a (convergent and type preserving) rewrite system $\mathcal{R}$. Consider also $\mathfrak{S}$ parametrised by Σ, $\mathcal{R}$ and the empty coercion system, whose judgements are noted '$\cdot \Vdash_{\emptyset} \cdot : \cdot$'.*

- *If $\Gamma \Vdash_{\emptyset} t \rightsquigarrow t' : A$, then $\Gamma \Vdash_{\mathfrak{T}} t \Rightarrow A$;*
- *for any type A such that there is $s \in \{\star, \Box\}$, $\Gamma \Vdash_{\emptyset} A :_{\mathcal{S}} s$, if $\Gamma \Vdash_{\emptyset} t : A \rightsquigarrow t'$ then $\Gamma \Vdash_{\mathfrak{T}} t \Leftarrow A$.*

Proof. By mutual induction over the typing derivation. All cases are handled by induction hypothesis except R-COERCE. If the input coercion system is empty, rule R-COERCE cannot be used and only rule R-CAST can be used. Hence t and t' are syntactically equal, and by Proposition 10 Page 67, $\Gamma \vdash t' \Rightarrow A$. □

In order to maintain the soundness of the relation $t : A \lessdot : B \rightsquigarrow t'$, the rewrite system $\mathcal{C}$ must enforce some invariants.

Definition 29. Let $\mathfrak{T}$ be a well-formed λΠmr type system parametrised by a signature Σ and a rewrite system $\mathcal{R}$. Let $\mathfrak{K}$ be the (well-formed) λΠmr type system parametrised by the signature $\Sigma \cup \{\kappa[A : \star, B : \star, t : A] : B : \star\}$ and $\mathcal{R}$. A coercion rewrite system $\mathcal{C}$ is *type preserving* if for any rewrite rule $\ell \hookrightarrow r \in \mathcal{C}$, for any context Γ well-formed in $\mathfrak{T}$, for any type A such that $\Gamma \Vdash_{\mathfrak{T}} A \Leftarrow_{\mathcal{S}} s$, for any substitution σ if $\Gamma \Vdash_{\mathfrak{K}} \sigma\ell \Leftarrow \sigma A$, then $\Gamma \Vdash_{\mathfrak{K}} \sigma r \Leftarrow \sigma A$.

Example 13. In this example, we use the typing relation λΠ[sPe]. The following coercion

$$(\kappa\,\texttt{Type}\,\star\,x) \hookrightarrow (\texttt{El}\,x)$$

preserves typing. For any substitution σ, the left-hand side is well-typed if σx is typeable by `Type`. In that case, the right-hand side $(\texttt{El}\,(\sigma x))$ is well-typed and has the same type as the left-hand side, this type is $\star$.

In the former definition, the extended declaration is well-formed although the type of symbol κ cannot be typed into λΠmr because it is polymorphic (its type would be $\Pi A : \star, \Pi B : \star, A \to B$). Using the signature, terms with κ can be typed when it is fully applied.

Lemma 12. *Let $\mathfrak{T}$ be a well-formed λΠmr type system parametrised by a signature Σ and a rewrite system $\mathcal{R}$. Let $\mathfrak{R}$ be the λΠmr type system parametrised by the signature $\Sigma \cup \{\kappa[A : \star, B : \star, x : A] : B : \star\}$ and $\mathcal{R}$. Then for any context Γ well-formed in $\mathfrak{T}$, for any terms t and A of $\mathcal{T}(\{\star, \Box\}, \mathrm{dom}(\Sigma))$,*

- *if $\Gamma \vdash_{\mathfrak{R}} t \Rightarrow A$ then $\Gamma \vdash_{\mathfrak{T}} t \Rightarrow A$;*
- *for any sort s such that $\Gamma \vdash_{\mathfrak{T}} A \Rightarrow_{\mathcal{S}} s$, if $\Gamma \vdash_{\mathfrak{R}} t \Leftarrow A$, then $\Gamma \vdash_{\mathfrak{T}} t \Leftarrow A$.*

Proof. By simultaneous induction over $\Gamma \vdash_{\mathfrak{R}} t \Rightarrow A$ and $\Gamma \vdash_{\mathfrak{R}} t \Leftarrow A$. We use the typing rules of Fig. 2.13 Page 54.

Rules B-PROD, B-ABST, B-APPL are handled by induction hypothesis.

For rule B-SIGN, because Σ is in $\mathcal{T}(\{\star, \Box\}, \mathcal{F})$, κ cannot occur in any term of any judgement of Σ (in particular in types of symbol declarations), hence induction hypothesis is enough.

For rule B-PROD-C, by induction hypothesis, $\Gamma \vdash_{\mathfrak{T}} t \Rightarrow A$ holds, and $\Pi x : A_1$, A_2 does not contain κ because $\mathcal{R}$ has terms in $\mathcal{T}(\mathcal{F})$. The case of B-SORT-C is similar.

For rule B-CHECK, by induction hypothesis, $\Gamma \vdash_{\mathfrak{T}} t \Rightarrow A$. By hypothesis, B does not contain κ. By confluence of $\mathcal{R}$, there is C such that $A \hookrightarrow^*_{\mathcal{R}} C$ and $B \hookrightarrow^*_{\mathcal{R}} C$. Because $\mathcal{R}$ has its terms in $\mathcal{T}(\mathcal{F})$, A and B are both in $\mathcal{T}(\mathcal{F})$, we have $C \in \mathcal{T}(\mathcal{F})$. □

Proposition 13 (Validity of R-COERCE)**.** *The cast relation implemented by the rule* R-COERCE *page 73 is a valid.*

Proof. Let $\Lambda = \Sigma \cup \{\kappa[A : \star, B : \star, x : A] : B : \star\}$, $(\vdash_\Lambda :)$ the type checking relation using signature Λ. By induction on the length of the reduction, we prove that the following property holds

$$\{\Gamma \vdash t \Leftarrow A \wedge \Gamma \vdash B \Rightarrow_{\mathcal{S}} s\}\ (\kappa\, A\, B\, t) \hookrightarrow^*_{\beta,\mathcal{R},\mathcal{C}} u\ \{\Gamma \vdash_\Lambda u \Leftarrow B\}$$

The base case is proved by induction on a context C such that $C[\sigma\ell] \hookrightarrow C[\sigma r]$ where $\ell \hookrightarrow r \in \mathcal{C}$. When the context C is empty, either $(\kappa\, a\, b\, t) \hookrightarrow_{\beta,\mathcal{R}} (\kappa\, a'\, b'\, t')$ where we can conclude because $\{\beta\} \cup \mathcal{R}$ has the subject reduction property. Or $(\kappa\, a\, b\, t) \hookrightarrow_{\mathcal{C}} u$ and we can conclude by type preservation of $\mathcal{C}$. By structural induction on context C, we can conclude for the base case.

We can conclude the proof by induction on the length of the reduction, using the base case to prove heredity as well.

Using Lemma 12, if $(\kappa\, A\, B\, t) \hookrightarrow_{\beta,\mathcal{R},\mathcal{C}} u$ and $\kappa \notin u$, then $\Gamma \vdash u \Leftarrow B$. □

As a corollary, we have that a refiner with a type preserving coercion system is still correct by Proposition 10 Page 67.

Using a rewrite system also provides stability by substitution for free because rewriting is stable by substitution by definition.

Corollary 1. *Any coercion system implemented by rule* R-COERCE, R-CAST *and a coercion rewrite system $\mathcal{C}$ is stable by substitution (Definition 24 Page 69).*

Proof. By inversion of inference rules that allow to derive $t : A \lessdot: B \rightsquigarrow u$. □

3.2.3 Standard coercions

The coercions defined in Section 3.1.3 Page 67 can be implemented by a higher-order rewrite system. Covariance and contravariance may be implemented by

$$(\kappa\, (\Pi x : A_1, A_2)\, (\Pi x : B_1, B_2)\, f) \hookrightarrow \lambda x : B_1, (\kappa\, A_2\, B_2\ (f\ (\kappa\, B_1\, A_1\, x)))\,.$$

We can avoid η-expansion if we filter only abstractions

$$(\kappa\, (\Pi x : A_1, A_2)\, (\Pi x : B_1, B_2)\, (\lambda x, X[x])) \hookrightarrow \lambda x : B_1, (\kappa\, A_2\, B_2\, X[\kappa\, B_1\, A_1\, x])\,.$$

Further details on the shape of patterns and higher order matching can be found in the works of Hondet and Blanqui 2020; Klop, Oostrom and Raamsdonk 1993; Miller 1991. In the latter rule, we consider that the bound variable x in the

right-hand side captures the variable that was bound in X. If we are only interested in covariance in the codomain, we can use the non-linear rule

$$(\kappa\,(\Pi x : A, B)\,(\Pi x : A, C)\,(\lambda x, X[x])) \hookrightarrow \lambda x : A, (\kappa\, B\, C\, X[x])\,. \qquad (\mathcal{C}\text{-}\Pi)$$

With Eq. ($\mathcal{C}$-Π) and a rule $(\kappa\,\texttt{Int}\,\texttt{Float}\,N) \hookrightarrow (\texttt{IntToFloat}\,N)$, the coercion of $\lambda b, n$ from $\texttt{Bool} \to \texttt{Int}$ to $\texttt{Bool} \to \texttt{Float}$ is computed by

$$(\kappa\,(\texttt{Bool} \to \texttt{Int})\,(\texttt{Bool} \to \texttt{Float})\,(\lambda b, n)) \hookrightarrow_{\mathcal{C}} (\lambda b : \texttt{Bool}, (\kappa\,\texttt{Int}\,\texttt{Float})) \hookrightarrow_{\mathcal{C}} (\lambda b : \texttt{Bool}, (\texttt{IntToFloat}\,n)).$$

In presence of recursive coercions, such as

$$(\kappa\ (\texttt{El}\ (\texttt{psub}\,T\,P))\ U\,X) \hookrightarrow (\kappa\,T\,U\ (\texttt{fst}\,X))$$

an explicit elimination rule may be required to be able to obtain terms without κ. Such elimination rule would be non recursive variants like

$$(\kappa\ (\texttt{El}\ (\texttt{psub}\,T\,P))\ T\,X) \hookrightarrow (\texttt{fst}\,X)\,.$$

However, this approach requires each recursive coercion rule to have a non-recursive (and non-linear) counterpart. We can reduce the number of non-linear rules using a generic eliminator that encodes the identity coercion,

$$(\kappa\,T\,T\,X) \hookrightarrow X. \qquad (\mathcal{C}\text{-Id})$$

Note that this rule subsumes rule R-CAST page 64: by confluence of $\mathcal{R}$, if $A \simeq_{\mathcal{R}} B$, there is C such that $A \hookrightarrow^*_{\mathcal{R}} C$ and $B \hookrightarrow^*_{\mathcal{R}} C$, hence $(\kappa\,A\,B\,t) \hookrightarrow^*_{\mathcal{R}} (\kappa\,C\,C\,t) \hookrightarrow_{\mathcal{C}} t$.

3.2.4 Non-linearity threatens convergence

Termination

Abel and Coquand 2020 showed that Eq. ($\mathcal{C}$-Id) may harm termination in presence of polymorphism (*à la* System F by Girard 1971) or with a proof irrelevant propositional equality and impredicativity because of non-linear filtering on types (Harper and Mitchell 1999). The example of non-termination given in (Abel and Coquand 2020) has been encoded into *PVS-Cert* in Appendix B Page 169.

In our case, we may take advantage of the encoding, and prove at least that terms in the image of the encoding are weakly normalising. In that case, we may use normalisation strategies (Pol 2001) to reach normal forms.

$$\Pi x : (\texttt{El}\, T), (\texttt{El}\, U) \hookrightarrow (\texttt{El}\, (T \rightsquigarrow U)) \tag{3.1}$$
$$(\pi^{-1}\ (\texttt{El}\, X)) \hookrightarrow X \tag{3.2}$$
$$(\kappa\, (A \rightsquigarrow B)\, (A \rightsquigarrow C)\, (\lambda x, E[x])) \hookrightarrow \lambda x, (\kappa'\, B\, C\, E[x]) \tag{3.3}$$
$$(\kappa\, A\, A\, X) \hookrightarrow X. \tag{3.4}$$

Figure 3.2: Rewrite system $\mathcal{R}_{\text{inv}}$ to retrieve type codes from types.

Confluence

Non linearity breaks confluence over untyped terms (Klop 1980). It may be useful to restrict non-linearity to type codes rather than the framework's types to be able to use layering methods provided by Gaspard Férey 2021; Gaspard Férey and J.-P. Jouannaud 2021. For this, we may type κ by $\kappa : \Pi a : \texttt{Type}$, $\Pi b : \texttt{Type}, (\texttt{El}\, (a \rightsquigarrow b))$ and we replace rule R-COERCE with

$$\frac{(\kappa\, a\, b\, t) \hookrightarrow^* u \qquad \kappa \notin u}{t : (\texttt{El}\, a) <: (\texttt{El}\, b) \rightsquigarrow u}.$$

However, there is no reason that inference returns only terms of the form $(\texttt{El}\, X)$, in particular when inferring the type of an abstraction: the normal form of the type of an encoded function is a product of the framework.

It is possible to invert rewrite rules: Fig. 3.2 Page 78 inverts products in the image of El to retrieve a type code. Other rules define coercion elimination and coercion of functions, filtering on type codes rather than types. For this, function κ is given type $\Pi a : \texttt{Type}, \Pi b : \texttt{Type}, (\texttt{El}\, (a \rightsquigarrow b))$ to operate on type codes a and b. Rule R-COERCE page 73 can be finally be replaced with

$$\frac{(\kappa\ (\pi^{-1}\, T)\ (\pi^{-1}\, U)\ e) \hookrightarrow^*_{\text{inv}} e' \qquad \kappa \notin e'}{e : T <: U \rightsquigarrow e'}$$

In this work, we stick to the basic implementation with non-linear rewriting, although it may be either non terminating or non confluent. We believe that there is a large, non trivial class of terms on which the system is normalising and confluent.

3.2.5 Examples of coercions

Luo, Soloviev and Xue 2013 provide a nomenclature of some different coercions. We analyse how these coercions are translated in our framework.

Plain coercions are simply rewrite rules whose source and targets are ground (without variables) terms

$$(\kappa\,\texttt{Child}\,\texttt{Human}\,t) \hookrightarrow (\texttt{c}\,t)\,.$$

Note that we cannot η-reduce the rewrite rule because κ has to be fully applied to be typeable.

Dependent coercions are rewrite rules where the target type depends on the coercion. For instance, given a theory of lists and vectors, a function that transforms lists to vectors may be defined in the main rewrite system $\mathcal{R}$,

$$(\texttt{lv}\,\texttt{nil}) \hookrightarrow \texttt{vnil}$$
$$(\texttt{lv}\,(\texttt{cons}\,x\,\ell)) \hookrightarrow (\texttt{vcons}\,(\texttt{len}\,\ell)\,x\,(\texttt{lv}\,\ell))$$

where `nil` is the empty list, `cons` the consing operator on lists, `vnil` and `vcons` are their vector counterparts and $(\texttt{len}\,\ell)$ computes the length of ℓ. We can define the coercion

$$(\kappa\,\texttt{List}\;(\texttt{Vec}\,n)\;\ell) \hookrightarrow (\texttt{lv}\,\ell)$$

where the target type is $(\texttt{Vec}\,(\texttt{len}\,\ell))$.

Coercions that have a dependent source type are (somewhat confusingly) named 'parametrised' coercions. If we define a transformation from vectors to lists,

$$(\texttt{vl}\,\texttt{vnil}) \hookrightarrow \texttt{nil}$$
$$(\texttt{vl}\;(\texttt{vcons}\,n\,x\,v)) \hookrightarrow (\texttt{cons}\;(\texttt{vl}\,v))$$

the coercion from vectors to list can be defined with

$$(\kappa\;(\texttt{Vec}\,n)\;\texttt{List}\,v) \hookrightarrow (\texttt{vl}\,n\,v)$$

where the source type of v, $(\texttt{Vec}\,n)$, depends on a parameter n.

Coercion rules, or parametric coercions, are coercions that may depend on other coercions. For instance, the coercion from $(\texttt{List}\,a)$ to $(\texttt{List}\,b)$ may be defined as

$$(\kappa\;(\texttt{List}\,a)\;(\texttt{List}\,b)\;\ell) \hookrightarrow (\texttt{map}\,(\lambda x : \texttt{El}a, (\kappa\;(\texttt{El}\,a)\;(\texttt{El}\,b)\;x))\,\ell)$$

Table 3.1: Feature comparison of different coercion systems. Each column stand for a feature, and for each system, there is a bullet if the feature is supported by the system. Abbreviation 'dep. tgt.' stands for 'dependent target', 'dep. src.' for 'dependent source' for 'dependent source', 'param.' for 'parametrised' and 'nonunif.' for 'non uniform'.

	plain	dep. tgt.	dep. src.	param.	nonunif.
Coq	•	•	•		
Matita	•	•	•		•
Plastic	•	•	•	•	
$\mathfrak{S}$	•	•	•	•	•

where we use the function `El` defined in Fig. 2.3 Page 37 and use usual encoding techniques given in Section 2.2.1 Page 37. In our framework, parametric coercions are coercions where the right hand side issues recursive calls to the coercion operator. Parametric coercions can also be used to inline the definition of `lv`,

$$(\kappa\,\texttt{List}\ (\texttt{Vec}\,n)\ \texttt{nil}) \hookrightarrow \texttt{vnil}$$
$$(\kappa\,\texttt{List}\,(\texttt{Vec}\,(\texttt{succ}\,n))\,(\texttt{cons}\,x\,\ell)) \hookrightarrow (\texttt{vcons}\,(\texttt{len}\,\ell)\,x\ (\kappa\,\texttt{List}\ (\texttt{Vec}\,n)\ \ell))$$

Coen and Tassi 2009 introduce *nonuniform* coercions, that is, coercions that may depend on the value being coerced. They give for example the promotion from support to semi-groups: one may promote `Z` to $(\texttt{Z},+)$ but $(\texttt{List}\,\texttt{Z})$ to $((\texttt{List}\,\texttt{Z})\,,\texttt{append})$. Our coercion system allows such coercions because it is able to match on the coerced term. Assuming the definitions of the type of semigroups `SemiGroup`, a constructor for semigroups `semig`,

$$(\kappa\,\texttt{Type}\,\texttt{SemiGroup}\,\texttt{Z}) \hookrightarrow (\texttt{semig}\,\texttt{Z}\,+)$$
$$(\kappa\,\texttt{Type}\,\texttt{SemiGroup}\ (\texttt{List}\,X)) \hookrightarrow (\texttt{semig}\ (\texttt{List}\,X)\ (\texttt{append}\,X))$$

Table 3.1 compares the features of the different coercion systems that have been reviewed.

We can already define the eliminator for predicate subtyping `fst` as a coercion

$$((\kappa\ (\texttt{El}\ (\texttt{psub}\,a\,p))\ (\texttt{El}\,b)\ e)) \hookrightarrow (\kappa\ (\texttt{El}a)\ (\texttt{El}\,b)\ (\texttt{fst}\,a\,p\,e))$$

to remove spurious `pair` constructions automatically. Furthermore, an object of type $(\texttt{psub}\,(a \rightsquigarrow b)\,p)$ can be coerced to a function of type $(\texttt{El}(a \rightsquigarrow b))$. This

feature appears to be a direct benefit of the logical framework, even though coercions only coerce from encoded objects to encoded objects, the shallowness of the encoding reflects the coercions in the encoding into the logical framework: objects coerced to encoded functions are also coerced to functions of the framework.

However, the converse rule adding `pair` constructions cannot be encoded yet: we have no way to populate the proof obligation of pairs. In *Matita* and *Russell* by Sozeau 2006, existential variables are used for such a task.

3.2.6 Related work on coercions

Coercions are used extensively in programming languages such as *C* (see ISO 2018, section 6.3), often to avoid having a numerical operator for each number type. For instance, the *OCaml* language does not use coercions and consequently has an addition for floats `(+.)` and one for integers `(+)` while the *C* language has one `+` for floats, integers, unsigned integers &c.

Coercions for programming languages are generally built into the language, one cannot add a new coercion for a new datatype, say, in *C*, to coerce between structures. Some interactive proof assistants provide such facilities and coercions may be declared like functions or theorems (using special syntax). We review here some of these interactive proof assistants and how coercions are defined, but more importantly, what kind of coercions they can define.

In *Coq* (Saïbi 1999), coercions are declared between 'classes' that behave like approximations of types: functions and sorts are grouped into their own metaclass and type families are classes. Sources of coercions must be classes, they cannot be variables that may be instantiated to coercible classes upon application. Coercions are compiled into an 'inheritance' graph. Graph structures allow efficient lookup for coercions: finding a coercion between two classes amounts to find a path in the graph. However, there may be issues if two paths have the same source and target: this could probably lead to two different, unequal coercions. To ensure coherence—the property that two coercions are behaviourally equivalent if they have the same source and same target—the order of declaration of coercions is kept significant.

The coercion system of *Matita* (Asperti, Ricciotti, Coen and Tassi 2018) is somewhat similar to the previous one, but it uses existential variables and unification. As a consequence, the framework handles naturally coercion from non functional to functional objects. Furthermore, coercions may create proof obligations that are represented with existential variables. Like in *Coq*, the types of the sources and targets of coercions is a type approximation, it does

not contain variables, existential variables nor higher order terms. Like in (Saïbi 1999), the order of declaration of coercions is significant, the most recent path prevails.

The logical framework *Plastic* by Luo, Soloviev and Xue 2013 defines coercion using inference rules in which the premises specify coercions to be found so that the coercion of the conclusion may be derived. Therefore *Plastic* allows parametric[2] coercions such as

$$\frac{\Gamma \vdash A <_c B : \mathit{Type}}{\Gamma \vdash \mathit{List}(A) <_{\mathit{map}(c)} \mathit{List}(B) : \mathit{Type}}$$

where coercion c called a 'prerequisite' coerces objects of type A to type B, and $\mathit{map}(c)$ is the derived coercion from $\mathit{List}(A)$ to $\mathit{List}(B)$. The coherence condition is stricter than the former ones: the identity coercion must not be derivable using the coercion rules and if two coercions have the same source and target, then they must be equal. The order of declaration is not taken into account.

Sozeau 2006 designs a system to derive parametric coercions for predicate subtyping. Subtyping is contravariant on the domain and covariant on the codomain of functional objects. Contravariance on the domain requires the coercion insertion algorithm to η expand its argument which in turn requires the congruence of the target system to contain η equivalence. Functional objects wrapped in predicate subtypes can be coerced to functions using a map $\mu_\bullet$ which performs approximately the same computations as our relation $\prec$. For instance, $\mu_\bullet(\{f : A \to B \mid f(x) = f(y) \Rightarrow x = y\}) = A \to B$. The coercion system notably provides coercions for dependent sum types. Unicity of coercions, which is necessary to prove that the system is conservative, is ensured by the equational theory of the system which furthermore includes surjective pairing for dependent pairs and for elements of predicate subtypes.

3.3 Holes

Holes (also called placeholders by Asperti, Ricciotti, Coen and Tassi 2012) stand for yet unknown terms. In proof assistants, they allow to reduce the size of terms of the concrete syntax: terms may contain holes that are refined into *existential variables* (Muñoz 1997) which are instantiated using typing constraints and

[2]Sacerdoti Coen and Tassi 2011 say that a coercion is *parametric* if it depends on other coercions.

unification. We used such a facility in Chapter 2 Page 27 to write $(\texttt{fst}\, x)$ instead of $(\texttt{fst}\, a\, p\, x)$. In proof assistants such as *Coq* (The Coq Development Team 2022), $(\texttt{fst}\, x)$ is first transformed into $(\texttt{fst}\, \Diamond\, \Diamond\, x)$ then into $(\texttt{fst}\, ?a\, ?p\, x)$ by the type checker, where $?a$ and $?p$ are existential variables. Then a unification algorithm tries to instantiate these existential variables to make the whole term well typed.

In our case, we are only interested in marking places where proofs have to be provided. These proofs cannot be automatically generated by unification algorithms. Therefore, we do not consider full-fledged existential variables, but only holes, that is, places in terms where a proof has to be provided. Holes are not typed but can be easily replaced by existential variables by traversal of the term.

Definition 30. We extend system $\mathfrak{S}$ with *holes*. Given a set of symbols $\mathcal{F}$, (typed in a signature Σ), the terms of $\mathfrak{S}$ are $\mathcal{T}(\mathcal{F} \cup \{\Diamond, \kappa\})$.

Holes are not typeable. However, because we intend to extend the coercion judgement to allow the creation of holes, we may use the rule

$$\textsc{b-hole}\quad \frac{}{\Gamma \vdash \Diamond \Leftarrow A}$$

in order to state the correctness of $t : A \lessdot B \rightsquigarrow u$.

Definition 31 (Type preserving coercion system)**.** Let $\mathfrak{A}$ be a well-formed $\lambda\Pi$mr type system parametrised by a signature Σ and a rewrite system $\mathcal{R}$. Let $\mathfrak{H}$ be the $\lambda\Pi$mr type system parametrised by symbols $\mathrm{dom}(\Sigma) \cup \{\kappa, \Diamond\}$, signature $\Sigma \cup \{\kappa[A : \star, B : \star, t : A] : B : \star\}$, rewrite system $\mathcal{R}$ and extended with the rule B-HOLE. A coercion rewrite rule $(\kappa\, A\, B\, e) \longhookrightarrow r$ preserves typing if for any context Γ, substitution σ, if $\Gamma \vdash_{\mathfrak{A}} \sigma e \Leftarrow \sigma A$, then $\Gamma \vdash_{\mathfrak{H}} \sigma r \Leftarrow \sigma B$.

Note that as long as holes are only generated by coercions, there is no need to refine holes. Rule B-HOLE is added to the type checker to keep the correctness of the refiner, that is, whenever $\Gamma \vdash t \rightsquigarrow t' : T$, then $\Gamma \vdash t' \Leftarrow T$.

Remark 9 (From holes to proof obligations). Given a term with holes, it is easy to transform these holes into proper existential variables. For that, assume existential variables are elements of a countable set $\mathcal{Y}$ noted $?x$, and define proof problems as set of judgements of the form $\Gamma \vdash ?x : T$ which states that a term of type T in context Γ has to be found. We define type checking judgements

$\Gamma \vdash t \rightsquigarrow t : t \vDash \mathcal{P}$ and $\Gamma \vdash t : t \rightsquigarrow t \vDash \mathcal{P}$ which produce proof problems. This relation is defined by the rules given in Fig. 3.1 Page 64 where for each rule, each premise output a proof problem $\mathcal{P}_i$ and the conclusion outputs the proof problem $\bigcup_i \mathcal{P}_i$ (or the empty set if the rule does not have any premise). For instance, the abstraction R-ABST is transformed into

$$\frac{\Gamma \vdash A : \star \rightsquigarrow A' \vDash \mathcal{P}_1 \qquad \Gamma, x : A' \vdash t \rightsquigarrow t' : B \vDash \mathcal{P}_2}{\Gamma \vdash \lambda x : A, t \rightsquigarrow \lambda x : A', t' : \Pi x : A', B \vDash \mathcal{P}_1 \cup \mathcal{P}_2}.$$

To these rules, we add the following one

$$\frac{?x \in \mathcal{Y}}{\Gamma \vdash \Diamond : T \rightsquigarrow ?x \vDash \{\Gamma \vdash ?x : T\}}$$

which transforms holes into existential variables.

3.4 Implementation

The refiner described in this section has been implemented in *Lambdapi* (Deducteam 2022b, version 2.1.0). Furthermore, *Lambdapi* features existential variables, which are used to implement holes. Ferreira and Pientka 2014 use existential variables to provide implicit arguments. *Lambdapi* contains additional type checking rules, for instance to infer properly a type when an existential variable appears at the head of an application.

Refinement slows down type checking substantially. We conjecture this slowdown is caused by the necessity to destructure and then restructure not only types, but also terms during type checking. Indeed, sparing term reconstruction speeds up the process. For instance, upon the inference of an abstraction $\lambda x : T$, t, we made the refiner return, along with the refined body t, whether the term t was modified, and the same for the domain T. If the two terms are not modified, then we may as well return the term given as input rather than build a new abstraction with body t (from the recursive call to the refiner) and domain T. In particular, the cost of construction of abstraction may vary depending on how binders are implemented. In *Lambdapi*, they are represented using the *Bindlib* library by Lepigre 2022. It provides a safe automated programmable interface and features higher order abstract syntax, which provides efficient substitution at the cost of an expensive binder construction (in comparison with de Bruijn 1972 indices).

3.5 Conclusion

We have defined a new family of type systems $\mathfrak{S}$ based on λΠmr (Definition 15 Page 50). It features term refinement—also called elaboration—which allows to turn incomplete terms into well-typed terms, hence separating a user-level syntax of incomplete terms and a kernel-level syntax. Refinement is parameterised by a *cast* relation and a *subtype projection*. The cast relation can be implemented by a rewrite system. Terms of an $\mathfrak{S}$ type system can contain *holes* which stand for yet unknown proofs. An $\mathfrak{S}$ type system is finally parameterised by

- a rewrite system $\mathcal{R}$ to implement a decidable congruence;
- a signature Σ to declare and type symbols;
- a coercion rewrite system $\mathcal{C}$ that implements a cast relation;
- a subtype projection $\prec$ to be able to refine the head of application;

its type checking rules are summarised in Appendix A Page 167.

Chapter 4

Implicit predicate subtyping

We have seen in Chapter 2 Page 27 that predicate subtyping can be encoded into the logical framework λΠmr. For this, we designed a translation function from the source system (*PVS-Cert*) to its encoding in λΠmr, and we showed that this translation preserves typing: whenever an object inhabits a type in the source system, then its translation inhabits the translation of its type in the framework.

In Chapter 3 Page 61, we provided a term refiner that can type check incomplete terms inserting coercions to make them well typed. Coercions implement some form of implicit subtyping: if the domain of the coercion is the subtype, and its range the supertype, then any term can be typed by one of its supertypes by the insertion of a coercion.

We now show how a system with implicit predicate subtyping like *PVS* can be encoded into our system $\mathfrak{S}$ (which is λΠmr with a refiner and existential variables). We begin by a description of the system to be encoded, then we provide an encoding along with a translation function. Finally we prove that the encoding preserves typeability.

$$\textsc{subtype-elim}\ \frac{\Gamma \vdash t : \mathtt{psub}(A, P)}{\Gamma \vdash t : A} \qquad \textsc{subtype-intro}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \mathtt{psub}(A, P) : \mathtt{Type} \qquad \Gamma \vdash (Pt)}{\Gamma \vdash t : \mathtt{psub}(A, P)}$$

Figure 4.1: Inference rules for implicit predicate subtyping.

4.1 *PVS-Core*: A system with implicit predicate subtyping

4.1.1 Definition

F. Gilbert 2018 defines the system *PVS-Core* as an idealisation of the core of *PVS*. It is made of simple type theory (like *PVS-Cert*), with the `psub` construction and additional typing rules for implicit predicate subtyping. The semantics of the subtyping of *PVS-Core* are given by a translation from *PVS-Core* derivations to *PVS-Cert* judgements.

Definition 32 (*PVS-Core*)**.** *PVS-Core* is the extension of the type system modulo parametrised by

$$\mathfrak{M} = (\mathcal{S}_{\mathrm{Po}} = \{\mathtt{Prop}, \mathtt{Type}, \mathtt{Kind}\}, \mathcal{A}_{\mathrm{Po}} = \{(\mathtt{Prop}, \mathtt{Type}), (\mathtt{Type}, \mathtt{Kind})\}, \mathcal{P}_{\mathrm{Po}} = \{(\mathtt{Type}, \mathtt{Type}, \mathtt{Type})\}, \mathcal{F}_{\mathrm{Po}}, \Sigma_{\mathrm{Po}}, \simeq_\beta)$$

$$\Sigma_{\mathrm{Po}} \begin{cases} \forall[A : \mathtt{Type}, P : (A \to \mathtt{Prop})] : \mathtt{Prop} : \mathtt{Type} \\ \Rightarrow[P : \mathtt{Prop}, Q : \mathtt{Prop}] : \mathtt{Prop} : \mathtt{Type} \\ \mathtt{psub}[A : \mathtt{Type}, P : (A \to \mathtt{Prop})] : \mathtt{Type} : \mathtt{Kind} \end{cases}$$

and where $\mathcal{F}_{\mathrm{Po}} = \{\forall, \Rightarrow, \mathtt{psub}\}$. The ternary typing relation defining *PVS-Core* depends on a binary relation $\Gamma \vdash P$ whose derivation rules are given in (ibid.). Judgements of the form $\Gamma \vdash t : A$ are derived with the type system in Fig. 2.1 (Page 29) with two extra rules for subtyping given in Fig. 4.1. The congruence of *PVS-Core*, denoted $\simeq_\beta$ is the smallest one containing β-reduction (Eq. (β) Page 31).

The third premise of the rule SUBTYPE-INTRO corresponds to *PVS*' type correctness conditions. We are not interested in their derivation because our translation axiomatises them by placing holes where proofs for type correctness conditions are required.

$$
\begin{aligned}
&[X] = X \\
&[\texttt{Prop}] = \texttt{o} \\
&[\Pi x : A, B] = ([A] \rightsquigarrow (\lambda x : \texttt{El}\,[A], [B])) \\
&[\texttt{psub}(A, P)] = (\texttt{psub}\ [A]\ [P]) \\
&[x] = x \\
&[\forall x : A, P] = (\forall\ [A]\ (\lambda x : (\texttt{El}\ [A]), [P])) \\
&[P \Rightarrow Q] = ([P] \Rightarrow [Q]) \\
&[\lambda x : A, t] = \lambda x : (\texttt{El}\ [A]), [t] \\
&[\![\texttt{Type}]\!] = \texttt{Type} \\
&[\![A]\!] = (\texttt{El}\ [A])\ \text{ otherwise}
\end{aligned}
$$

Figure 4.2: Translation from *PVS-Core* terms to (a subset of) λΠ[Pe] terms.

There is a straightforward translation from *PVS-Core* terms to *PVS-Cert* terms (see ibid., Definition 9.3.1). The main differences that we see between the two systems are that

- `pair`, π_ℓ and π_r are not part of the syntax of *PVS-Core*, subtyping is implicit in *PVS-Core*;
- deduction and typing are separated: there is no proof term in *PVS-Core*.

Lemma 13. *Functions* $[-]$ *and* $[\![-]\!]$ *defined in Fig. 4.2 are well-defined and terminate on terms of* PVS-Core. *They both preserve substitution:* $[\{u/x\}\, t] = \{[u]/x\}\, [t]$ *(and similarly replacing* $[-]$ *by* $[\![-]\!]$*).*

Proof. The function $[-]$ is called recursively on strict subterms of its argument, thus it terminates. Preservation of substitution for $[-]$ is shown by structural induction on t. For $[\![-]\!]$, either the argument is `Type`, in which case termination is immediate, or it is a direct consequence of the termination of $[-]$. We can say the same for preservation of substitution. □

4.1.2 Encoding *PVS-Core* in λΠmr

Since terms of *PVS-Core* can be translated to the encoding of *PVS-Cert* in λΠmr, namely λΠ[Pe], we do not need to define a new encoding for *PVS-Core*.

However, because coercions $\mathtt{pair}$ and $\mathtt{fst}$ are inserted into encoded terms in order to type check them, we need to take care that inserting these coercions do not break computation: whenever two terms are convertible in *PVS-Core*, their translation and refinement should also be convertible in λΠ[Pe].

Example 14. Let $e = ((\lambda x : (\mathtt{El}\, A), (f\, x))\, u)$ be a term in the image of the translation defined in Fig. 4.2. Note that $e \hookrightarrow_\beta (f\, u)$. Let $\Gamma = A : \mathtt{Type}, P : (\mathtt{El}\, (\mathtt{Type} \rightsquigarrow \mathtt{o})), f : (\mathtt{El}\, ((\mathtt{psub}\, A\, P) \rightsquigarrow A)), u : (\mathtt{El}\ (\mathtt{psub}\, A\, P))$.

In order to have a well-typed term e, it is refined

$$\Gamma \vdash e \rightsquigarrow ((\lambda x, (f\ (\mathtt{pair}\, x\, \Diamond)))\ (\mathtt{fst}\, u)) : ...$$

and the refinement β-reduces to $(f\ (\mathtt{pair}\ (\mathtt{fst}\, u)\ \Diamond))$.

The refinement of the reduct of e, namely $(f\, u)$ can itself be refined into $(f\, u)$. Therefore, to keep convertibility through translation and refinement, we need $(f\, u) \simeq (f\ (\mathtt{pair}\ (\mathtt{fst}\, u)\ \Diamond))$.

The latter example shows that to preserve convertibility, the encoded congruence must contain some sort of surjective pairing, namely $(\mathtt{pair}\ (\mathtt{fst}\, e)\ \Diamond) \simeq e$. Therefore, we add this identity to the set of equations of the encoding of *PVS-Cert* in λΠme:

$$\begin{aligned}
&(\mathtt{Prf}\ (\forall\, t\, p)) = \Pi x : (\mathtt{El}\, t), (\mathtt{Prf}\ (p\, x))\\
&(\mathtt{Prf}\ (p \Rightarrow q)) = \Pi h : (\mathtt{Prf}\, p), (\mathtt{Prf}\, q)\\
&(\mathtt{El}\, (t \rightsquigarrow u)) = \Pi x : (\mathtt{El}\, t), (\mathtt{El}\ (u\, x))\\
&(\mathtt{pair}\, t\, p\, e\, h) = (\mathtt{pair}\, t\, p\, e\, h')\\
&(\mathtt{fst}\, t_0\, p_0\ (\mathtt{pair}\, t_1\, p_1\, e\, h)) = e\\
&(\mathtt{pair}\, t_0\, p_0\ (\mathtt{fst}\, t_0\, p_0\, e)\ h) = e.
\end{aligned}$$

We can complete this set of equations into the rewrite system $\mathcal{R}_{[\mathrm{sPe}]}$.

Definition 33 ($\mathcal{R}_{[\mathrm{sPe}]}$)**.** We define the rewrite system $\mathcal{R}_{[\mathrm{sPe}]}$ as the union of the system $\mathcal{R}_{[\mathrm{Pe}]}$ (defined in Fig. 2.12 Page 51) and the rule

$$(\mathtt{pair}^\dagger\, T\, P\, (\mathtt{fst}\, U\, P\, M)) \hookrightarrow M. \qquad (\mathrm{SP}^\dagger)$$

Equation ($\mathrm{SP}^\dagger$) threatens the confluence of the system because it is non linear. Non linearity can sometimes be avoided when only well-typed terms are considered, as in Eq. (2.3) Page 33, but not in the case of surjective pairing. Assume that $\Gamma \vdash \sigma\ell \Leftarrow A$ where ℓ is the left-hand side of Eq. ($\mathrm{SP}^\dagger$) where the

second occurrence of P (when reading from left to right) is replaced with Q so that ℓ is algebraic, Γ is a well-formed context, A is a well-formed type in Γ, and σ is a substitution. Then by inversion, A is of the form $A = (\texttt{El}\ (\texttt{psub}\,(\sigma A)\,(\sigma P)))$. By inversion, σX is of type $(\texttt{El}\ (\texttt{psub}\,(\sigma B)\,(\sigma Q)))$, and $(\texttt{El}\,(\sigma U)) \simeq (\texttt{El}\,(\sigma T))$. The right-hand side σX has the same type as the left-hand side only if $P \simeq Q$.

Non linear rule are famous to break desirable properties in untyped λ calculus such as confluence (Klop 1980). Fortunately, Pottinger 1981 showed that simply typed λ-calculus with surjective pairing remains confluent using the weak normalisation of the calculus. Curien and Cosmo 1996 have provided a confluent rewrite system for the typed λ-calculus with η-equivalence, surjective pairing and terminal objects.

Conjecture 2. *Rewrite relation $\hookrightarrow_{\beta,\mathcal{R}_{[\text{sPe}]}}$ defined in Definition 33 is confluent on well-typed terms.*

Definition 34 ($\lambda\Pi$[sPe])**.** We denote $\lambda\Pi$[sPe] the $\lambda\Pi$mr type system defined by the signature $\Sigma_{[\text{Pe}]}$ of $\lambda\Pi$[Pe] (defined in Definition 13 Page 38) and the rewrite system $\mathcal{R}_{[\text{sPe}]}$ (defined in Definition 33). Its congruence is denoted $\simeq_{[\text{sPe}]}$.

4.2 Tuning the refiner for *PVS-Core*

The refiner of the system $\mathfrak{S}$ is parametrised by a coercion system and a subtype projection $\prec$. This section provides these two components, and establishes some of their properties.

4.2.1 Abstract coercion rules

Coercions to implement implicit predicate subtyping in $\mathfrak{S}$ parametrised by $\Sigma_{[\text{Pe}]}$ and $\mathcal{R}_{[\text{sPe}]}$ are given in Fig. 4.3. Rules SUB-ELIM and SUB-INTRO follow the intuitions given in Section 2.1.3 Page 32. Rule SUB-FUN implements covariance on the domain of abstractions. Note that there is no need to constrain T_0 to be the same as T_1, since preconditions ensure that $\Gamma \vdash t \Leftarrow \Pi x : T_1, T_2$, and hence $T_0 \simeq_{[\text{sPe}],\beta} T_1$. Rule SUB-RED allows to reduce terms in order to apply coercion rules. For instance, the coercion problem

$$\lambda x, e_x : (\texttt{El}\,(X \rightsquigarrow Y)) \lessdot (\texttt{El}\,X) \to (\texttt{El}\,Z) \rightsquigarrow \lambda x, e_z$$

requires term $(\texttt{El}\,(X \rightsquigarrow Y))$ to be reduced to a product in order to apply rule SUB-FUN.

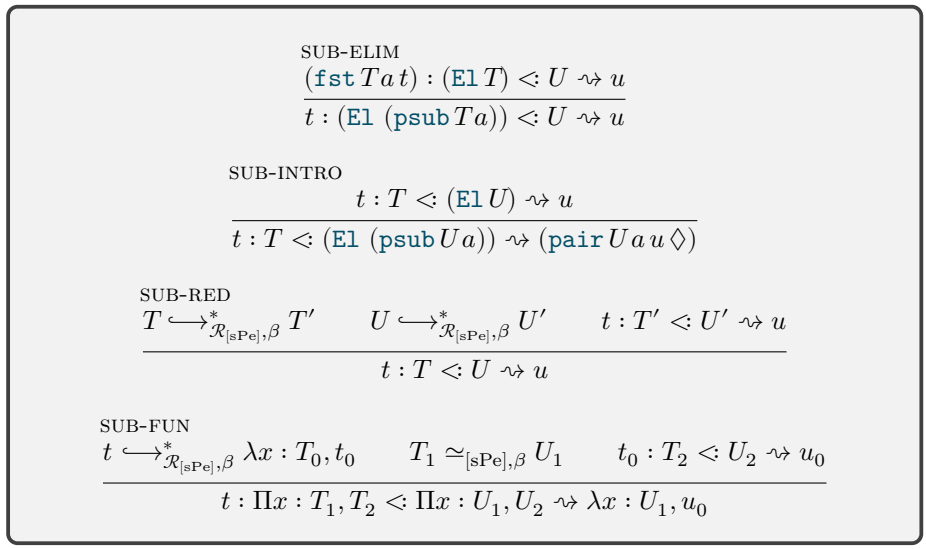

Figure 4.3: Coercion rules for predicate subtyping in *PVS-Cert*.

Remark 10. Rule SUB-RED is very liberal and allows any reduction to be performed into the source and target of coercions. Many of these computations are irrelevant regarding coercions: most rules of $\mathcal{R}_{[\mathrm{sPe}]}$ cannot transform a term into a pattern of a conclusion of a coercion rule (*i.e.* introduce `El` symbols on top of terms, or transform types to products).

It may be interesting to only allow relevant computations to take place while coercing terms, either to have more control over the shape of returned terms, or to avoid performing unnecessary (and costly) computations. The only relevant computations are β-reduction (to transform types which would be β-redexes) and Eq. (2.13) Page 37 to transform encoded arrow types into the framework's product type. The latter transformation could be allowed with the following

coercion rules

$$\textsc{sub-p-elim}\quad \frac{t : \Pi x : (\texttt{El}\, T_1), (\texttt{El}\ (T_2\, x)) \lessdot U \rightsquigarrow u}{t : (\texttt{El}\, (T_1 \rightsquigarrow T_2)) \lessdot U \rightsquigarrow u}$$

$$\textsc{sub-p-intro}\quad \frac{t : T \lessdot \Pi x : (\texttt{El}\, U_1), (\texttt{El}\ (U_2\, x)) \rightsquigarrow u}{t : T \lessdot (\texttt{El}\, (U_1 \rightsquigarrow U_2)) \rightsquigarrow u}$$

and β-reduction could be allowed with rule SUB-RED where $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$ is replaced with $\hookrightarrow_{\beta}$.

The following lemmas state some properties that can be obtained by inverting the coercion rules.

Lemma 14. *Assume* $\vdash_{\lambda\Pi[\mathrm{sPe}]} \Gamma$, T *and* $\Pi x : C_1, C_2$ *are well-sorted in* Γ *using* $\lambda\Pi[\mathrm{sPe}]$ *and* $\Gamma \vdash_{\lambda\Pi[\mathrm{sPe}]} \lambda x : T_0, t \Leftarrow T$. *If* $(\lambda x : T_0, t) : T \lessdot \Pi x : C_1, C_2 \rightsquigarrow u$, *then '*$u = \lambda x : U_0, u_0$*',* '$T_0 \simeq_{[\mathrm{sPe}],\beta} U_0$', '$T \simeq_{[\mathrm{sPe}],\beta} \Pi x : T_0, T_1$', '$C_1 \simeq_{[\mathrm{sPe}],\beta} T_0$' *and* '$t : T_1 \lessdot C_2 \rightsquigarrow u_0$'.

Proof. By inversion of typing, T must be a product. By definition of $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$, products are not convertible with terms of the form (El (psub ...)), so only rule SUB-FUN can be applied. □

Lemma 15. *Assume* $\vdash_{\lambda\Pi[\mathrm{sPe}]} \Gamma$, $\Pi x : T_1, T_2$ *and* $\Pi x : U_1, U_2$ *are well-sorted in* Γ *using* $\lambda\Pi[\mathrm{sPe}]$ *and* $\Gamma \vdash_{\lambda\Pi[\mathrm{sPe}]} f \Leftarrow \Pi x : T_1, T_2$. *If* $f : \Pi x : T_1, T_2 \lessdot \Pi x : U_1, U_2 \rightsquigarrow g$ *and* f *is not an abstraction, then* $\Pi x : T_1, T_2 \simeq_{[\mathrm{sPe}],\beta} \Pi x : U_1, U_2$.

Proof. Rules SUB-ELIM and SUB-INTRO cannot be applied. Because f is not an abstraction, rule SUB-FUN cannot be used either. We are left with rule R-CAST which demands that $\Pi x : T_1, T_2 \simeq_{[\mathrm{sPe}],\beta} \Pi x : U_1, U_2$ and gives $g = f$. □

Lemma 16. *The coercion system defined in Fig. 4.3 is stable by* $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$*: if* $t : T \lessdot U \rightsquigarrow u$, $T \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} T'$ *and* $U \hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}} U'$, *then* $t : T' \lessdot U' \rightsquigarrow u$.

Proof. Terms of the form (El (psub T P)) or $\Pi x : T_1, T_2$ are in head normal form: if (El (psub T P)) $\hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} U$, then U is of the form (El (psub U_0 Q)). The same property holds for products. □

Lemma 17. *The coercion system defined in Fig. 4.3 is transitive.*

Proof. Assume that the inference rule of Definition 23 Page 69 can be used. We show that any derivation using this rule can be transformed into a derivation not using this rule by induction on the size of the derivation.

If the last inference rule is not the transitivity rule, we conclude by induction hypothesis.

If the last inference is the transitivity rule, we operate by case distinction on the two premises $s : S \lessdot: T \rightsquigarrow t$ and $t : T \lessdot: U \rightsquigarrow u$.

Case SUB-ELIM/any $S = (\texttt{El}\ (\texttt{psub}\, S_0\, P))$
$(\texttt{fst}\, S_0\, Ps) : (\texttt{El}\, S_0) \lessdot: T \rightsquigarrow t$
and by induction we can assume the right-hand derivation to be transitivity free, as well as the derivation of $(\texttt{El}\, S_0) \lessdot: T$. We use the transitivity rule to build a derivation of $(\texttt{fst}\, S_0\, Ps) : S_0 \lessdot: U \rightsquigarrow u'$. This derivation is strictly smaller than the original, so the induction hypothesis can be used to derive a transitivity-free derivation. We conclude using rule SUB-ELIM.

Case any/SUB-INTRO $U = (\texttt{El}\ (\texttt{psub}\, U_0\, P))\,; T \lessdot: U_0$
Transitivity rule can be used to build a derivation of $s : S \lessdot: U_0 \rightsquigarrow u_0$. This derivation is smaller than the original one, therefore by induction hypothesis it is transitivity-free. We can conclude using rule SUB-INTRO.

Case R-CAST/any $S \simeq_{[\mathrm{sPe}],\beta} T$
By confluence of $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$, there is a common reduct V of S and T. Therefore, we also have $t : V \lessdot: U \rightsquigarrow u'$ by Lemma 16. By induction hypothesis this derivation is transitivity-free. We conclude using SUB-RED.

Case any/R-CAST Idem.

Case SUB-INTRO/SUB-ELIM $T = (\texttt{El}\ (\texttt{psub}\, T_0\, P))$
$S \lessdot: T_0; \quad T_0 \lessdot: U$
We can use the transitivity rule to build a derivation of $s : S \lessdot: U \rightsquigarrow u'$. Since this derivation is strictly smaller than the original one, by induction hypothesis it is transitivity-free.

Case SUB-FUN/SUB-FUN $S = \Pi x : S_1, S_2; \quad T = \Pi x : T_1, T_2$
$U = \Pi x : U_1, U_2; \quad S_2 \lessdot: T_2; \quad T_2 \lessdot: U_2$
$S_1 \simeq_{[\mathrm{sPe}],\beta} T_1; \quad T_1 \simeq_{[\mathrm{sPe}],\beta} U_1$
Induction hypothesis allows to build a transitivity-free derivation of $S_2 \lessdot: U_2$. By transitivity of $\simeq_{[\mathrm{sPe}],\beta}$, we have $S_1 \simeq_{[\mathrm{sPe}],\beta} U_1$. We apply rule SUB-FUN to conclude.

Case SUB-RED/any $S \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} S_0; \quad T \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} T_0$
$s : S_0 \lessdot: T_0 \rightsquigarrow t$
Lemma 16 gives $t : T_0 \lessdot: U \rightsquigarrow u$. Therefore we can build by induction hypothesis a transitivity-free derivation of $s : S_0 \lessdot: U \rightsquigarrow u$. We conclude using rule SUB-RED.

Case any/SUB-RED Idem. □

Lemma 18. *The following properties hold for the coercion system defined by Fig. 4.3.*

1. *It is stable by substitution.*
2. *It is symmetric: if $T <\!: U$, then $U <\!: T$.*
3. *For any sorts s and s', if $t : s <\!: s' \rightsquigarrow t'$, then $s = s'$ and $t = t'$.*

Proof. 1. By induction on the coercion derivation.

2. By induction on the coercion derivation.

3. The only rule that can be applied is R-CAST page 64 (sorts are constant with respect to $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$). □

Lemma 19. *For any terms t_0 and t_1, for any types A and B, if there is u_0, $t_0 : A <\!: B \rightsquigarrow u_0$ and $t_0 \simeq_{[\mathrm{sPe}],\beta} t_1$, then there is u_1, $t_1 : A <\!: B \rightsquigarrow u_1$ and $u_0 \simeq_{[\mathrm{sPe}],\beta} u_1$.*

Proof. By induction on $t_0 : A <\!: B \rightsquigarrow u_0$.

Case SUB-ELIM $A = (\mathtt{El}\ (\mathtt{psub}\, A_0\, P))\,;(\mathtt{fst}\, t_0) : (\mathtt{El}\, A_0) <\!: B \rightsquigarrow u$
Induction hypothesis gives $(\mathtt{fst}\, t_1) : (\mathtt{El}\, A_0) <\!: B \rightsquigarrow u_1$ with $u_0 \simeq_{[\mathrm{sPe}],\beta} u_1$ since $(\mathtt{fst}\, t_0) \simeq_{[\mathrm{sPe}],\beta} (\mathtt{fst}\, t_1)$. We conclude using rule SUB-ELIM.

Case SUB-INTRO $B = (\mathtt{El}\ (\mathtt{psub}\, B_0\, P))\,; u_0 = (\mathtt{pair}\, u_{00}\, \Diamond)$
$t_0 : A <\!: (\mathtt{El}\, B_0) \rightsquigarrow u_{00}$
Induction hypothesis gives $t_1 : A <\!: (\mathtt{El}\, B_0) \rightsquigarrow u_{10}$ with $u_{10} \simeq_{[\mathrm{sPe}],\beta} u_{00}$. We can apply rule SUB-INTRO to obtain $t_1 : A <\!: (\mathtt{El}\ (\mathtt{psub}\, B_0\, P)) \rightsquigarrow (\mathtt{pair}\, u_{10}\, \Diamond)$. We obtain

$$(\mathtt{pair}\, u_{10}\, \Diamond) \hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}} (\mathtt{pair}^{\dagger}\, u_{10}) \simeq_{[\mathrm{sPe}],\beta} (\mathtt{pair}^{\dagger}\, u_{00}) \hookleftarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}} (\mathtt{pair}^{\dagger}\, u_{00})\,.$$

Case SUB-RED $A \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} A'; \quad B \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} B'$
$t_0 : A' <\!: B' \rightsquigarrow u_0$
We conclude by induction hypothesis and rule SUB-RED.

Case SUB-FUN $A = \Pi x : A_1, A_2; \quad B = \Pi x : B_1, B_2$
$t_0 \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} \lambda x : T_0, t_{00}$
$t_{00} : A_2 <\!: B_2 \rightsquigarrow u_{00}$
$u_0 = \lambda x : T_0, u_{00}$
By (conjectured) confluence of $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$, we have $t_1 \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} \lambda x : T_1, t_{10}$, and

$\lambda x : T_1, t_{10} \simeq_{[\mathrm{sPe}],\beta} \lambda x : T_0, t_{00}$. By induction hypothesis, $t_{10} : A_2 <: B_2 \rightsquigarrow u_{10}$ and $u_{10} \simeq_{[\mathrm{sPe}],\beta} u_{00}$. We conclude using rule SUB-FUN.

Case R-CAST $A \simeq_{[\mathrm{sPe}],\beta} B; t_0 = u_0$
We also get $u_1 = t_1$, hence $u_1 \simeq_{[\mathrm{sPe}],\beta} u_0$. □

We can obtain a stronger property than transitivity: not only the transtivity rule is admissible, but there is no difference (up to $\simeq_{[\mathrm{sPe}],\beta}$) between coercing in two steps or coercing at once. This property is required to show that the refiner preserves substitution.

Lemma 20. *If* $s : S <: T \rightsquigarrow t$, $t' : T <: U \rightsquigarrow u$, $t \simeq_{[\mathrm{sPe}],\beta} t'$ *and* $s : S <: U \rightsquigarrow u'$, *then* $u \simeq_{[\mathrm{sPe}],\beta} u'$.

Proof. Note that by transitivity (Lemma 17), judgement $s : S <: U \rightsquigarrow u'$ can always be derived whenever $s : S <: T \rightsquigarrow t$ and $t' : T <: U \rightsquigarrow u$. By induction on $s : S <: U \rightsquigarrow u$.

Case R-CAST $S \simeq_{[\mathrm{sPe}],\beta} U; \quad s = u'$
We proceed by another induction on $s : S <: T \rightsquigarrow t$. If the last rule is SUB-ELIM, then $S = (\mathtt{El}\ (\mathtt{psub}\, S_0\, P))$ and we have

$$\frac{(\mathtt{fst}\, s) : (\mathtt{El}\, S_0) <: T \rightsquigarrow t}{s : (\mathtt{El}\ (\mathtt{psub}\, S_0\, P)) <: T \rightsquigarrow t'}; \frac{t' : T <: (\mathtt{El}\, S_0) \rightsquigarrow u_0'}{t : T <: (\mathtt{El}\ (\mathtt{psub}\, S_0\, P)) \rightsquigarrow (\mathtt{pair}\, u_0'\, \Diamond)}$$

If U is not of the form $(\mathtt{El}\ (\mathtt{psub}\, S_0\, P))$, by confluence of $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$, there is a common reduct of S and U of the form $(\mathtt{El}\ (\mathtt{psub}\, R\, Q))$ (because $(\mathtt{El}\ (\mathtt{psub}\ ...))$ is a head normal form). Therefore without loss of generality, we consider U to be $(\mathtt{El}\ (\mathtt{psub}\, S_0\, P))$. By induction hypothesis, we obtain $(\mathtt{fst}\, s) \simeq_{[\mathrm{sPe}],\beta} u_0'$. Therefore, the conclusion holds if $s \simeq_{[\mathrm{sPe}],\beta} (\mathtt{pair}\, u_0'\, \Diamond)$, *i.e.* $s \simeq_{[\mathrm{sPe}],\beta} (\mathtt{pair}\ (\mathtt{fst}\, s)\, \Diamond)$. It holds by Eq. (SP†) Page 90.
If the last rule is SUB-INTRO,

$$\begin{array}{l} T = (\mathtt{El}\ (\mathtt{psub}\, T_0\, P)) \\ s : S <: (\mathtt{El}\, T_0) \rightsquigarrow t_0; \quad t = (\mathtt{pair}\, t_0\, \Diamond) \\ (\mathtt{fst}\, t') : (\mathtt{El}\, T_0) <: S \rightsquigarrow u \end{array}$$

Noting that $(\mathtt{fst}\, t') \simeq_{[\mathrm{sPe}],\beta} (\mathtt{fst}\ (\mathtt{pair}\, t_0\, \Diamond)) \hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}} t_0$, induction hypothesis gives $u' \simeq_{[\mathrm{sPe}],\beta} u$.
If the last rule is SUB-RED,

$$S \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} S'; \quad T \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}],\beta}} T'; \quad s : S' <: T' \rightsquigarrow t$$

By application of rule SUB-RED, $t' : T' \lessdot: U \rightsquigarrow u$ holds. We conclude by induction hypothesis.
If the last rule is SUB-FUN,

$$S = \Pi x : S_1, S_2; \quad T = \Pi x : T_1, T_2;$$
$$s \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \lambda x : S_0, s_0; t = \lambda x : T_1, t_0$$
$$s_0 : S_2 \lessdot: T_2 \rightsquigarrow t_0$$

Since $U \simeq_{[\text{sPe}],\beta} S$, $U \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \Pi x : U_1, U_2$. Similarly, we get $t' \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \lambda x, t'_0$. Using rule SUB-RED, we can obtain $t' : \Pi x : T_1, T_2 \lessdot: \Pi x : U_1, U_2 \rightsquigarrow u$. Applying SUB-FUN, we obtain $t'_0 : T_2 \lessdot: U_2 \rightsquigarrow u_0$. Induction hypothesis gives that $t_0 : S_2 \lessdot: U_2 \rightsquigarrow u'_0$ and $u_0 \simeq_{[\text{sPe}],\beta} u'_0$. We conclude using rule SUB-FUN.

Case SUB-ELIM $S = (\texttt{El}\ (\texttt{psub}\, S_0\, P))$
$(\texttt{fst}\, s) : (\texttt{El}\, S_0) \lessdot: U \rightsquigarrow u'$
By induction hypothesis, if $(\texttt{fst}\, s) : (\texttt{El}\, S_0) \lessdot: T \rightsquigarrow t$ then $(\texttt{fst}\, s) : (\texttt{El}\, S_0) \lessdot: U \rightsquigarrow u'$ with $u \simeq_{[\text{sPe}],\beta} u'$. Applying rule SUB-ELIM, we obtain $s : S \lessdot: U \rightsquigarrow u'$.

Case SUB-INTRO $U = (\texttt{El}\ (\texttt{psub}\, U_0\, P))\,; \quad s : S \lessdot: (\texttt{El}\, U_0) \rightsquigarrow u_0$
$u' = (\texttt{pair}\, u'_0\, \Diamond)$
Induction hypothesis gives $u'_0 \simeq_{[\text{sPe}],\beta} u_0$ where $t' : T \lessdot: (\texttt{El}\, U_0) \rightsquigarrow u_0$. Using rule SUB-INTRO, we get $t' : T \lessdot: U \rightsquigarrow (\texttt{pair}\, u_0\, \Diamond)$ and $(\texttt{pair}\, u_0\, \Diamond) \simeq_{[\text{sPe}],\beta} (\texttt{pair}\, u'_0\, \Diamond)$.

Case SUB-RED $S \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} S'; U \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} U'$
$s : S' \lessdot: U' \rightsquigarrow u'$
By rule SUB-RED, both $s : S' \lessdot: T \rightsquigarrow t$ and $t' : T \lessdot: U' \rightsquigarrow u$ hold. The result follows from induction hypothesis.

Case SUB-FUN $S = \Pi x : S_1, S_2; \quad U = \Pi x : U_1, U_2; \quad S_1 \simeq_{[\text{sPe}],\beta} U_1$
$s \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \lambda x : S_0, s_0; \quad u' = \lambda x : U_1, u'_0$
$s_0 : S_2 \lessdot: U_2 \rightsquigarrow u'_0$
By induction on $s : S \lessdot: T \rightsquigarrow t$. If T is a product $T = \Pi x : T_1, T_2$ (and rule SUB-FUN as well), the result follows from induction hypothesis.
If T is of the form $T = (\texttt{El}\ (\texttt{psub}\, T_0\, a))$, then

$$\frac{\lambda x : S_0, s_0 : \Pi x : S_1, S_2 \lessdot: (\texttt{El}\, T_0) \rightsquigarrow t_0}{\lambda x : S_0, s_0 : \Pi x : S_1, S_2 \lessdot: (\texttt{El}\ (\texttt{psub}\, T_0\, a)) \rightsquigarrow (\texttt{pair}\, t_0\, \Diamond)}$$
$$\frac{(\texttt{fst}\ (\texttt{pair}\, t_0\, \Diamond)) : (\texttt{El}\, T_0) \lessdot: \Pi x : U_1, U_2 \rightsquigarrow u}{(\texttt{pair}\, t_0\, \Diamond) : (\texttt{El}\ (\texttt{psub}\, T_0\, a)) \lessdot: \Pi x : U_1, U_2 \rightsquigarrow u}$$

We can apply induction hypothesis since $(\texttt{fst}\ (\texttt{pair}\, t_0\, \Diamond)) \hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}} t_0$, we obtain $u \simeq_{[\text{sPe}],\beta} u'$. □

$$(\kappa\ (\texttt{El}\ (\texttt{psub}\ A\ P))\ (\texttt{El}\ B)\ X) \hookrightarrow (\kappa\ (\texttt{El}\ A)\ (\texttt{El}\ B)\ (\texttt{fst}\ A\ P\ X)) \tag{4.1}$$

$$(\kappa\ (\texttt{El}\ A)\ (\texttt{El}\ (\texttt{psub}\ B\ P))\ X) \hookrightarrow (\texttt{pair}\ B\ P\ (\kappa\ (\texttt{El}\ A)\ (\texttt{El}\ B)\ X)\ \Diamond) \tag{4.2}$$

$$(\kappa\ (\Pi x : A, B)\ (\Pi x : A, C)\ (\lambda x, X)) \hookrightarrow \lambda x : A, (\kappa\ B\ C\ X) \tag{$\mathcal{C}$-Π}$$

$$(\kappa\ X\ X\ Y) \hookrightarrow Y \tag{$\mathcal{C}$-Id}$$

Figure 4.4: Coercion rewrite rules $\mathcal{C}_{[\mathrm{Pe}]}$ for implicit predicate subtyping. Rules ($\mathcal{C}$-Π) and ($\mathcal{C}$-Id) are discussed in Section 3.2.3 Page 76.

4.2.2 Coercions by rewriting

Next, just like we proposed to implement rewriting as an implementation of coercions in Section 3.2.2 Page 73, we provide the adequate rewrite rules to implement the coercion rules of Fig. 4.3 Page 92 in Fig. 4.4. Equations (4.1) and (4.2) are the rewriting counterparts of rules SUB-ELIM and SUB-INTRO. The two other rules have been discussed in Section 3.2.3 Page 76.

Lemma 21. *The coercion system defined by inference rule* R-COERCE *and the rewrite system defined by Fig. 4.4 is correct with respect to rules of Fig. 4.3 and rule* R-CAST *page 64: If* $(\kappa\ A\ B\ t) \hookrightarrow^*_{\beta,\mathcal{R}_{[\mathrm{sPe}]},\mathcal{C}_{[\mathrm{Pe}]}} u$ *and* $\kappa \notin u$*, then there is* u' *such that* $t : A <: B \rightsquigarrow u'$ *and* $u \simeq_{[\mathrm{sPe}],\beta} u'$*.*

Proof. By induction on the number of $\hookrightarrow_{\mathcal{C}_{[\mathrm{Pe}]}}$ rewrite steps.

There is at least one rewrite rule to eliminate symbol κ: Eq. ($\mathcal{C}$-Id). In that case, we have

$$(\kappa\ A\ B\ t) \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}]},\beta} (\kappa\ C\ C\ t) \hookrightarrow_{\mathcal{C}_{[\mathrm{Pe}]}} t \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}]},\beta} u.$$

Since A and B can be rewritten to C, we have $A \simeq_{[\mathrm{sPe}],\beta} B$. Hence we can derive $t : A <: B \rightsquigarrow t$ with rule R-CAST, with $t \simeq_{[\mathrm{sPe}],\beta} u$.

For the recursive case, we are in the following situation

$$(\kappa\ R\ S\ e_0) \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}]},\beta} (\kappa\ R'\ S'\ e_1) \hookrightarrow_{\mathcal{C}_{[\mathrm{Pe}]}} e_2 \hookrightarrow^*_{\mathcal{C}_{[\mathrm{Pe}]},\mathcal{R}_{[\mathrm{sPe}]},\beta} e_3.$$

Where the second rewrite step is either Eq. ($\mathcal{C}$-Π), Eq. (4.1) or Eq. (4.2). If it is Eq. (4.1), then $R' = (\texttt{El}\ (\texttt{psub}\ R'_0\ a))$, $e_2 = (\kappa\ (\texttt{El}\ R'_0)\ S'\ (\texttt{fst}\ e_1))$. By

induction hypothesis, there is a derivation tree τ whose conclusion is $(\texttt{fst}\, e_1) : (\texttt{El}\, R'_0) <: S' \rightsquigarrow e_3$. We can use rule SUB-ELIM to thus deduce

$$\dfrac{\dfrac{\tau :: (\texttt{fst}\, e'_2) : (\texttt{El}\, R'_0) <: S' \rightsquigarrow e_3}{e_1 : (\texttt{El}\ (\texttt{psub}\, R'_0\, a)) <: S' \rightsquigarrow e_3}\ \text{SUB-ELIM}}{e_0 : R <: S \rightsquigarrow e_3}\ \text{SUB-RED}.$$

When rewrite rule Eq. (4.2) is used, then $S' = (\texttt{El}\ (\texttt{psub}\, S'_0\, a))$ and $e_2 = (\texttt{pair}\ (\kappa\, R'\ (\texttt{El}\, S'_0)\ e_1)\ \Diamond)$. Because $\texttt{pair}$ is constant for $\hookrightarrow_{\mathcal{R}_{[\text{sPe}]},\mathcal{C}_{[\text{Pe}]}}$, we have $e_3 = (\texttt{pair}\, e'_3\, \Diamond)$. By induction hypothesis, there is a derivation tree τ whose conclusion is $e_1 : R' <: (\texttt{El}\, S'_0) \rightsquigarrow e'_3$. Like before, we can use rule SUB-INTRO,

$$\dfrac{\dfrac{\tau :: e_1 : R' <: (\texttt{El}\, S'_0) \rightsquigarrow e'_3}{e_1 : R' <: (\texttt{El}\ (\texttt{psub}\, S'_0\, a)) \rightsquigarrow (\texttt{pair}\, e'_3\, \Diamond)}\ \text{SUB-INTRO}}{e_0 : R <: S \rightsquigarrow e_3}\ \text{SUB-RED}.$$

If Eq. ($\mathcal{C}$-Π) is used, then $R' = \Pi x : R'_1, R'_2$, $S' = \Pi x : R'_1, S'_2$ and $e_1 = \lambda x : T, e_{11}$. We also have $e_2 = \lambda x : R'_1, (\kappa\, R'_2\, S'_2\, e_{11})$ and $e_3 = \lambda x : E_1, e_{30}$ because abstractions on top of terms cannot be rewritten. Therefore we have $(\kappa\, R'_2\, S'_2\, e_{11}) \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta},\mathcal{C}_{[\text{Pe}]}} e_{30}$. By induction hypothesis, there is a derivation tree $\tau :: e_{11} : R'_2 <: S'_2 \rightsquigarrow e_{30}$ (where we have $\Gamma, x : T$ with $e_1 = \lambda x : T, e_{11}$ well-typed in context Γ by type-preservation of $\hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}}$, and e_0 well-typed in Γ by precondition of coercion). We can therefore apply rule SUB-FUN to derive $\lambda x, e_{11} : \Pi x : R'_1, R'_2 <: \Pi x : R'_1, S'_2 \rightsquigarrow \lambda x : R'_1, e_{30}$. We still have to prove that $\lambda x : R'_1, e_{30} \simeq_{[\text{sPe}],\beta} e_3 = \lambda x : E_1, e_{30}$, so we just have to show $R'_1 \simeq_{[\text{sPe}],\beta} E_1$. This proposition holds because we know that there is no coercion κ in either of them, and that $R'_1 \hookrightarrow_{\beta,\mathcal{R}_{[\text{sPe}],}\mathcal{C}_{[\text{Pe}]}} E_1$. Therefore $\hookrightarrow_{\mathcal{C}_{[\text{Pe}]}}$ rewrite steps cannot be used in the convertibility proof. □

Lemma 22. *The coercion system defined by rewrite system Fig. 4.4 is complete with respect to rules of Fig. 4.3 Page 92 and rule* R-CAST *page 64: If $t : A <: B \rightsquigarrow u$, then $(\kappa\, A\, B\, t) \hookrightarrow^*_{\mathcal{C}_{[\text{Pe}]},\mathcal{R}_{[\text{sPe}],\beta}} u$ and $\kappa \notin u$.*

Proof. By induction on the derivation of $t : T <: U \rightsquigarrow u$. For rule R-CAST, we have $T \simeq_{[\text{sPe}],\beta} U$ and because $\hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}}$ is confluent, there is V such that $T \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} V \hookleftarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} U$. Thus we have $(\kappa\, T\, U\, t) \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} (\kappa\, V\, V\, t) \hookrightarrow_{\mathcal{C}_{[\text{Pe}]}} t$ where the last rewrite step is obtained with Eq. ($\mathcal{C}$-Id).

For rule SUB-RED, we invoke confluence of $\simeq_{[\mathrm{sPe}],\beta}$.

For rule SUB-FUN, induction hypothesis gives $(\kappa\, B\, C\, b) \hookrightarrow_{\mathcal{C}_{[\mathrm{Pe}]},\mathcal{R}_{[\mathrm{sPe}],\beta}} c$, and we can apply Eq. ($\mathcal{C}$-$\Pi$).

For rules SUB-INTRO and SUB-ELIM, we conclude using induction hypothesis, either rule Eq. (4.2) or Eq. (4.1) and transitivity of the rewrite relation. □

Example 15. Surjective pairing is required for coherence: the identity coercion for type $(\mathtt{El}\ (\mathtt{psub}\, T\, a))$ may be derived with

$$\dfrac{\dfrac{\dfrac{T \simeq_{[\mathrm{sPe}],\beta} T}{(\mathtt{fst}\, t) : T \lessdot T \rightsquigarrow (\mathtt{fst}\, t)}\ \text{R-CAST}}{(\mathtt{fst}\, t) : T \lessdot (\mathtt{El}\ (\mathtt{psub}\, T\, a)) \rightsquigarrow (\mathtt{pair}\ (\mathtt{fst}\, t)\ \Diamond)}\ \text{SUB-INTRO}}{t : (\mathtt{El}\ (\mathtt{psub}\, T\, a)) \lessdot (\mathtt{El}\ (\mathtt{psub}\, T\, a)) \rightsquigarrow (\mathtt{pair}\ (\mathtt{fst}\, t)\ \Diamond)}\ \text{SUB-ELIM}$$

where $(\mathtt{pair}\ (\mathtt{fst}\, t)\ \Diamond)$ is convertible to t only if surjective pairing (Eq. ($\mathrm{SP}^\dagger$) Page 90) is included in the convertibility relation.

4.2.3 Coercing to functions

Predicate subtypes can be organised as trees, where nodes are types (and subtypes), and there is an edge between types A and B if $B = \mathtt{psub}(A, P)$. Such a graph is indeed a tree: any type has at most one supertype which is the support of the predicate, and any number of subtypes. Roots of such trees are called 'maximal types' by Owre and Shankar 1997b. The maximal type of any type is computed by the function μ defined in (ibid.),

$$\mu(\mathtt{psub}(T, a)) = \mu(T)$$
$$\mu(T \to U) = T \to \mu(U)$$
$$\mu(s) = s \text{ for any primitive sort } s.$$

We may mimic this procedure to look for product types among supertypes of a type: given a type T, the procedure iterates through the path from T to its maximal type, and stops as soon as a product is found. To this end we define a subtype projection $\prec$ that computes less than the μ operator. This subtype projection essentially contains the equation $\mathtt{psub}(T, a) \prec T$. However, we cannot simply take the context closure of that relation, because subtyping is invariant on the domain. Therefore we do not take the contextual closure but we reduce modulo $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}],\beta}}$.

Definition 35. Let $\mathcal{T}$ be the set of terms of λΠ[sPe]. We define the relation $\prec \subseteq \mathcal{T} \times \mathcal{T}$ as the smallest relation such that

$$(\texttt{El}\ (\texttt{psub}\, T\, a)) \prec (\texttt{El}\, T) \tag{4.3}$$

$$\hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}} \subseteq \prec. \tag{4.4}$$

Proposition 14. *Relation $\prec$ is a subtype projection (Definition 25 Page 71).*

Proof. For each rule, by inversion, when the left-hand side is well-typed, the right-hand side is well-typed as well.

For Eq. (4.3), the right-hand side can be coerced to the right-hand side by rule SUB-ELIM. For Eq. (4.4), we can conclude with R-CAST. □

Lemma 23. *For any terms t, T, U_1 and U_2 such that $\Gamma \vdash_{\lambda\Pi[\text{sPe}]} t \Leftarrow T$, there is s, $\Gamma \vdash_{\lambda\Pi[\text{sPe}]} \Pi x : U_1, U_2 \Rightarrow_S s$, if $t : T \lessdot \Pi x : U_1, U_2 \rightsquigarrow u$, then there are S_1 and S_2 such that $T \prec^* \Pi x : S_1, S_2$ and $\Pi x : S_1, S_2 \lessdot \Pi x : U_1, U_2$.*

Proof. If T is a product $T = \Pi x : T_1, T_2$, then $T \prec^= \Pi x : T_1, T_2$ and $\Pi x : T_1, T_2 \lessdot \Pi x : U_1, U_2$ by hypothesis.

Otherwise, T must be of the form $(\texttt{El}\, T_0)$. By inspection of the coercion rules Fig. 4.3 Page 92, the only rules that can be used are SUB-RED and SUB-ELIM.

By induction on the derivation $t : T \lessdot \Pi x : U_1, U_2 \rightsquigarrow u$.

Case SUB-ELIM $T_0 = (\texttt{psub}\, T_{00}\, a)$
$(\texttt{fst}\, t) : (\texttt{El}\, T_{00}) \lessdot \Pi x : U_1, U_2 \rightsquigarrow u$

By induction hypothesis, $(\texttt{El}\, T_{00}) \prec^* \Pi x : T_1, T_2$ and $\Pi x : T_1, T_2 \lessdot \Pi x : U_1, U_2$. By definition of $\prec$, $T = (\texttt{El}\ (\texttt{psub}\, T_{00}\, a)) \prec (\texttt{El}\, T_{00})$.

Case SUB-RED $T \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} T'$; $\Pi x : U_1, U_2 \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \Pi x : U_1', U_2'$
$t : T' \lessdot \Pi x : U_1', U_2' \rightsquigarrow u$

When T_0 is of the form $(\texttt{psub}\, T_{00}\, a)$, then T' must be of the form $T' = (\texttt{El}\ (\texttt{psub}\, T_0'\, a'))$ where $T_{00} \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} T_0'$ and $a \hookrightarrow^+_{\mathcal{R}_{[\text{sPe}],\beta}} a'$. Induction hypothesis gives $T' \prec^* \Pi x : T_1, T_2$ and $\Pi x : U_1', U_2' \lessdot \Pi x : T_1, T_2$. Equation (4.4) gives $T \prec \Pi x : T_1, T_2$. When T_0 is of the form $T_{01} \rightsquigarrow T_{02}$, then $(\texttt{El}\, T_0) \hookrightarrow^*_{\mathcal{R}_{[\text{sPe}],\beta}} \Pi x : (\texttt{El}\, T_{01}), (\texttt{El}\, T_{02})$. We conclude by induction hypothesis and inclusion of $\hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}}$ into $\prec^*$.

Case R-CAST $T \simeq_{[\text{sPe}],\beta} \Pi x : U_1, U_2$

By confluence of $\hookrightarrow_{\mathcal{R}_{[\text{sPe}],\beta}}$ (conjecture 2 Page 91), there is a common reduct V. By Eq. (4.4), $T \prec^* V$. We conclude with rule SUB-RED. □

Lemma 24. *If $t : T \lessdot: U \nrightarrow u$ and $t : T \lessdot: V \nrightarrow v$, then u is convertible with an abstraction (modulo $\simeq_{[\mathrm{sPe}],\beta}$) if and only if v is.*

Proof. If u is convertible with an abstraction, by correctness of the cast relation, U is of the form $U = \Pi x : U_1, U_2$. □

With a rewrite system

Definition 36. We define the rewrite system $\mathcal{R}_\nu$ on symbols $\mathcal{F}_{[\mathrm{Pe}]} \cup \{\nu\}$.

$$(\nu\ (\mathtt{El}\ (\mathtt{psub}\ A\ P))) \hookrightarrow (\nu\ (\mathtt{El}\ A)) \tag{ν.1}$$
$$(\nu\,(\Pi x : A, B)) \hookrightarrow \Pi x : A, B \tag{ν.2}$$

Lemma 25. *Let $\mathfrak{N}$ be the $\lambda\Pi mr$ type system parametrised by the signature $\Sigma_{[\mathrm{Pe}]} \cup \{\nu[A : \star] : \star : \Box\}$. Then $\mathfrak{N}$ is well-formed and relation $\hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}]},\beta,\mathcal{R}_\nu}$ preserves typing in $\mathfrak{N}$: for any well-formed context Γ (in $\mathfrak{N}$), any substitution σ, if $\Gamma \vdash_{\mathfrak{N}} \sigma A \Leftarrow \star$ and $A \hookrightarrow^*_{\mathcal{R}_{[\mathrm{sPe}]},\beta,\mathcal{R}_\nu} \Pi x : A_1, A_2$, then $\Gamma \vdash_{\lambda\Pi[\mathrm{sPe}]} \sigma(\Pi x : A_1, A_2) \Leftarrow \star$.*

Proof. System $\mathfrak{N}$ is well-formed because $\Sigma_{[\mathrm{Pe}]}$ is well-formed and the declaration $\nu[A : \star] : \star : \Box$ is also well-formed.

First we prove that if $(\nu t) \hookrightarrow_{\mathcal{R}_\nu} \Pi x : t_1, t_2$, then $\nu \notin \Pi x : t_1, t_2$ by induction on the number of rewrite steps. If $(\nu t) \hookrightarrow_{\mathcal{R}_\nu} \Pi x : t_1, t_2$, then Eq. ($\nu$.2) must be used. By hypothesis, there is no ν in t, and the rewrite rule erases the leading ν. Now assume there is a reduction chain $(\nu t) \hookrightarrow^*_{\mathcal{R}_\nu} \Pi x : t_1, t_2$. Then $(\nu t) \hookrightarrow_{\mathcal{R}_\nu} u \hookrightarrow^*_{\mathcal{R}_\nu} \Pi x : t_1, t_2$. The first rewrite step cannot use Eq. (ν.2), otherwise u would be a normal form. Hence it uses Eq. (ν.1), and u is of the form $(\nu u')$, and induction hypothesis allows to conclude.

Then we show that $\hookrightarrow_{\mathcal{R}_\nu}$ preserves typing in signature $\Sigma_{[\mathrm{Pe}]} \cup \{\nu[A : \star] : \star : \Box\}$. For type preservation, we prove that each rewrite rule preserves typing. For Eq. (ν.1), by several inversions of B-SIGN page 54, we have the left-hand side well typed if $\Gamma \vdash (\sigma A) \Leftarrow \mathtt{Type}$. Using several B-SIGN, we can derive $\Gamma \vdash (\nu\ (\mathtt{El}\ A)) \Leftarrow \star$. For the second rule, the left-hand side is well typed if $\Pi x : (\sigma A), (\sigma B)$ has type $\star$.

Finally, we show that for any term t, if $\nu \notin t$, then t is typable in $\Sigma_{[\mathrm{Pe}]}$. We show that by simultaneous inductions on derivations $\Gamma \vdash_{\mathfrak{N}} t \Rightarrow A$ and for any A such that $\Gamma \vdash_{\lambda\Pi[\mathrm{Pe}]} A \Rightarrow_{\mathcal{S}} s$, $\Gamma \vdash_{\mathfrak{N}} t \Leftarrow A$. For rule B-SIGN, since ν is not in t, and because ν is not in $\Sigma_{[\mathrm{Pe}]}$, all typing declarations of $\Sigma_{[\mathrm{Pe}]}$ are typeable

in λΠ[Pe] by induction hypothesis. For rule B-CHECK, by induction hypothesis, $\Gamma \vdash_{\lambda\Pi[\mathrm{Pe}]} t \Rightarrow A'$. Then because there is no rule with symbol ν in $\mathcal{R}_{[\mathrm{Pe}]}$ and because ν does not appear in A' nor A, we have $A' \simeq_{[\mathrm{sPe}]} A$. Other rules are easily handled using the induction hypotheses. □

Proposition 15. *Relation $\prec_\nu$ defined by $T \prec_\nu \Pi x : U_1, U_2$ if and only if $(\nu T) \hookrightarrow^*_{\mathcal{R}_\nu} \Pi x : U_1, U_2$ is sound and complete with respect to $\prec$ from Definition 35: $\prec_\nu = \prec^*$.*

Proof. We first show that $\prec_\nu \subseteq \prec^*$ by induction on the number of $\hookrightarrow_{\mathcal{R}_\nu}$ rewrite steps. Assume $T \prec_\nu \Pi x : U_1, U_2$. There is at least one rewrite step to eliminate ν, which may be $(\nu\ (\mathtt{El}\ (T_0 \rightsquigarrow T_1))) \hookrightarrow_{\mathcal{R}_\nu} \Pi x : (\mathtt{El}\ T_0), (\mathtt{El}\ (T_1\ x))$. In that case, we conclude with Eq. (4.4). If $(\nu(\Pi x : T_0, T_1)) \hookrightarrow_{\mathcal{R}_\nu} \Pi x : T_0, T_1$, we conclude by reflexivity of $\prec^=$.

Assume there are n rewrite steps. The reduction must be of the form $(\nu\ (\mathtt{El}\ (\mathtt{psub}\, T_0\, a))) \hookrightarrow_{\mathcal{R}_\nu} (\nu\ (\mathtt{El}\, T_0)) \hookrightarrow^+_{\mathcal{R}_\nu} \Pi x : U_1, U_2$. By induction hypothesis, $(\mathtt{El}\, T_0) \prec^* \Pi x : U_1, U_2$. By Eq. (4.3), $(\mathtt{El}\ (\mathtt{psub}\, T_0\, a)) \prec (\mathtt{El}\, T_0)$. We conclude by transitivity of $\prec^*$.

We show the converse: $\prec^* \subseteq \prec_\nu$ by induction on the number of $\prec$ steps. Assume $T \prec^* \Pi x : U_1, U_2$. If $T = \Pi x : U_1, U_2$, then we conclude with Eq. (ν.2). If $T \prec S \prec^* \Pi x : U_1, U_2$, then $T = (\mathtt{El}\ (\mathtt{psub}\, T_0\, a))$ and $S = (\mathtt{El}\, T_0)$. We conclude by Eq. (ν.1), induction hypothesis and transitivity of $\hookrightarrow^*_{\mathcal{R}_\nu}$. □

Example 16. Let $\Sigma = \Sigma_{[\mathrm{Pe}]} \cup \{\mathtt{nat} : \mathtt{Type} : \star, \mathtt{inj?}[n : (\mathtt{El}\ \mathtt{nat})] : (\mathtt{El}\ \mathtt{o}) : \star\}$. Then the judgement

$$f : (\mathtt{El}\ (\mathtt{nat} \rightsquigarrow \mathtt{nat})), \rho : (\mathtt{Prf}\ (\mathtt{inj?}\ f)) \vdash ((\mathtt{pair}\ f\ \rho)\ 0) : (\mathtt{El}\ \mathtt{nat}) \rightsquigarrow t'$$

cannot be derived without ν and rule R-PROD-C page 71. Indeed, the psub hides the encoded arrow in $(\mathtt{psub}\, (\mathtt{nat} \rightsquigarrow \mathtt{nat})\, \mathtt{inj?})$, so it does not rewrite to a product. However we have the following reduction chain

$$(\nu\ (\mathtt{El}\ (\mathtt{psub}\, (\mathtt{nat} \rightsquigarrow \mathtt{nat})\, \mathtt{inj?}))) \hookrightarrow_{\mathcal{R}_\nu} (\mathtt{El}\, (\mathtt{nat} \rightsquigarrow \mathtt{nat})) \hookrightarrow_{\mathcal{R}_{[\mathrm{sPe}]}} (\mathtt{El}\, \mathtt{nat}) \to (\mathtt{El}\, \mathtt{nat})\,.$$

Definition 37 (𝔖+[sPe])**.** We denote 𝔖+[sPe] the type system 𝔖 (Section 3.5 Page 85) parametrised by signature $\Sigma_{[\mathrm{Pe}]}$ (Fig. 2.4 Page 39), by the rewrite relation $\mathcal{R}_{[\mathrm{sPe}]}$ (Definition 33 Page 90), by the coercion system $\mathcal{C}_{[\mathrm{Pe}]}$ (Fig. 4.4 Page 98) and by the subtype projection $\prec$ (Definition 35 Page 101).

4.2.4 Preservation of substitution by refinement

We now prove a property similar to a substitution lemma (Barras 1999, Lemme 4.28), but generalised to refined terms: if the subject and inputs are substituted, then the output is also substituted. Such a property is essential to prove that valid judgements of *PVS-Core* can be translated and refined into valid judgements of λΠ[sPe]. In this section, when the type system is omitted from typing judgements, refinement judgements use type system $\mathfrak{S}$+[sPe] and judgements without refinement use λΠ[sPe].

Lemma 26. *For any well-formed context* Γ*, any well-sorted type* A *and any term* u *such that* $\Gamma \vdash u : A \rightsquigarrow u'$*, the two following propositions hold.*

1. *If* $\Gamma, x : A \vdash t \rightsquigarrow t' : B$ *then* $\Gamma \vdash \{u/x\}\, t : \{u'/x\}\, B \rightsquigarrow e$ *and* $e \simeq_{\beta,[\mathrm{sPe}]} \{u'/x\}\, t'$.

2. *Let* B *be a well-sorted type. If* $\Gamma, x : A \vdash t : B \rightsquigarrow t'$*, then* $\Gamma \vdash \{u/x\}\, t : \{u'/x\}\, B \rightsquigarrow e$ *and* $e \simeq_{\beta,[\mathrm{sPe}]} \{u'/x\}\, t'$.

Proof. The two propositions are proved simultaneously by induction on the typing derivation. Parts of the proof will be presented as sequences of judgement annotated with a justification, like

$$\Gamma \vdash u \rightsquigarrow t : U \qquad \textit{inversion (4).}$$

Numbers on the right of the justification refer to equations (e.g. 'Eq. (β)'). If there is no reference, the justification applies to either the previous judgement, or it applies to a hypothesis in the statement.

- For rule R-VAR when $t \in \mathcal{X}$ and $t \neq x$ the conclusion is immediate: $t : B \in \Gamma$, thus $\{u'/x\}\, B = B$ and $e = t = \{u'/x\}\, t$.
- For rule R-VAR when $t = x$, we have $B = A = \{u'/x\}\, B$, $\Gamma, x : A \vdash t \rightsquigarrow x : A$ and $\{u/x\}\, t = u$ and $e = u'$ by definition.
- $$\frac{\Gamma, x : A \vdash T_0 : \star \rightsquigarrow R_0 \qquad \Gamma, x : A, z : R_0 \vdash T_1 \rightsquigarrow R_1 : s_1 \quad (\star, s_1, s_2) \in \mathcal{P}^{\lambda\Pi}}{\Gamma, x : A \vdash \Pi z : T_0, T_1 \rightsquigarrow \Pi z : R_0, R_1 : s_2}\ \text{R-PROD}$$

Induction hypothesis gives $\Gamma, x : A \vdash \{u/x\}\, T_0 : \star \rightsquigarrow S_0$ with $S_0 \simeq_{[\mathrm{sPe}],\beta} \{u'/x\}\, R_0$ and $\Gamma, z : \{u'/x\}\, R_0 \vdash \{u/x\}\, T_1 : \{u'/x\}\, s_1 \rightsquigarrow S_1$ with $S_1 \simeq_{[\mathrm{sPe}],\beta} \{u'/x\}\, R_1$. By inversion, we have $\Gamma, z : \{u'/x\}\, R_0 \vdash \{u/x\}\, T_1 \rightsquigarrow S_1' : s_1'$ and a coercion from s_1' to s_1 (note that the inferred type is named s_1' but we have no evidence that it is a sort). Because there is no coercion from types to sorts, we

$$\Gamma \vdash \{u/x\}\, T_0 : \star \rightsquigarrow S_0 \qquad \textit{induction hypothesis}$$
$$S_0 \simeq \{u'/x\}\, R_0 \qquad \textit{induction hypothesis} \tag{4.5}$$
$$\Gamma, z : \{u'/x\}\, R_0 \vdash \{u/x\}\, t_1 : \{u'/x\}\, R_1 \rightsquigarrow e \qquad \textit{i.h.} \tag{4.6}$$
$$e \simeq \{u'/x\}\, r_0 \qquad \textit{induction hypothesis} \tag{4.7}$$
$$\Gamma, z : \{u'/x\}\, R_0 \vdash \{u/x\}\, t_1 \rightsquigarrow t_1^* : T^* \qquad \textit{inversion Eq. (4.6)} \tag{4.8}$$
$$t_1^* : T^* \lessdot : \{u'/x\}\, R_1 \rightsquigarrow e \qquad \textit{inversion Eq. (4.6)} \tag{4.9}$$

Figure 4.5: Reasoning steps for the abstraction case of Lemma 26.

get $s_1 = s'$ and $S_1' = S_1$. Therefore we may conclude using rule R-CHECK with $\Gamma \vdash \Pi z : \{u/x\}\, T_0, \{u/x\}\, T_1 \rightsquigarrow \Pi z : S_0, S_1 : s_2$.

- $$\frac{\Gamma, x : A \vdash T_0 : \star \rightsquigarrow R_0 \qquad \Gamma, x : A, z : R_0 \vdash t_0 \rightsquigarrow r_0 : R_1}{\Gamma, x : A \vdash \lambda z : T_0, t_0 \rightsquigarrow \lambda z : R_0, r_0 : \Pi z : R_0, R_1}\ \text{R-ABST}$$

The sequence of judgement that lead to the conclusion is given in Fig. 4.5. We can apply rule SUB-FUN on judgement (4.9) of Fig. 4.5 to obtain

$$\lambda z : R_0, t_1^* : \Pi z : R_0, T^* \lessdot : \{u'/x\}\, \Pi z : R_0, R_1 \rightsquigarrow \lambda z : \{u'/x\}\, R_0, e \tag{4.10}$$

and conclude with SUB-RED where the conversion is given by Eq. (4.5) and finally Eq. (4.7).

- $$\frac{\Gamma, x : A \vdash t_1 \rightsquigarrow t_1' :_\Pi \Pi z : B_1, B_2 \qquad \Gamma, x : A \vdash t_2 : B_1 \rightsquigarrow t_2'}{\Gamma, x : A \vdash (t_1\, t_2) \rightsquigarrow (t_1'\, t_2') : \{t_2'/z\}\, B_2}\ \text{R-APPL}$$

Induction hypothesis on the second premise gives $\Gamma \vdash \{u/x\}\, t_2 : \{u'/x\}\, B_1 \rightsquigarrow q_2$ for some q_2 with $q_2 \simeq_{[\mathrm{sPe}],\beta} \{u'/x\}\, t_2'$. We then analyse how $\{u/x\}\, t_1$ behaves under constrained inference. Judgements (4.16) and (4.18) give $\Gamma \vdash \{u/x\}\, t_1 \rightsquigarrow r_1 :_\Pi \Pi z : R_1, R_2$.

If t_1' is not convertible with an abstraction (modulo $\simeq_{[\mathrm{sPe}],\beta}$), then q_1 is not convertible with an abstraction as well, and by Lemma 24, r_1 is not. Lemma 15 Page 93 gives $R_2 \simeq \{u'/x\}\, B_2$. By Lemmas 17 and 20 Pages 93 and 96 over Eqs. (4.17) and (4.18), $r_1 \simeq_{[\mathrm{sPe}],\beta} q_1$. By transitivity of $\simeq$ and Eq. (4.15), we obtain $r_1 \simeq \{u'/x\}\, t_1'$. We can conclude by an application of rule R-CHECK using rule R-CAST for the second premise since $\{r_2/z\}\, R_2 \simeq \{u'/x\}\, \{t_2'/z\}\, B_2$ (the substitution is not parallel).

If t_1' is an abstraction of the form $t_1' = \lambda z, t_{11}'$, then q_1 is convertible with an

$$\Gamma, x : A \vdash t_1 \rightsquigarrow \alpha_1 : T_1 \qquad \textit{inversion first premise} \tag{4.11}$$
$$T_1 \prec^* \Pi z : B_1, B_2 \qquad \textit{idem} \tag{4.12}$$
$$\alpha_1 : T_1 \lessdot \Pi z : B_1, B_2 \rightsquigarrow t_1' \qquad \textit{idem} \tag{4.13}$$
$$\Gamma \vdash \{u/x\}\, t_1 : \{u'/x\}\, \Pi z : B_1, B_2 \rightsquigarrow q_1 \qquad \textit{i.h. Eqs. (4.11) and (4.13)} \tag{4.14}$$
$$q_1 \simeq \{u'/x\}\, t_1' \qquad \textit{idem} \tag{4.15}$$
$$\Gamma \vdash \{u/x\}\, t_1 \rightsquigarrow s_1 : S_1 \qquad \textit{inversion Eq. (4.14)} \tag{4.16}$$
$$s_1 : S_1 \lessdot \{u'/x\}\, \Pi z : B_1, B_2 \rightsquigarrow q_1 \qquad \textit{inversion Eq. (4.14)} \tag{4.17}$$
$$S_1 \prec^* \Pi z : R_1, R_2 \qquad \textit{Lemma 23} \tag{4.18}$$
$$\Pi z : R_1, R_2 \lessdot \{u'/x\}\, \Pi z : B_1, B_2 \qquad \textit{Lemma 23} \tag{4.19}$$
$$R_1 \simeq \{u'/x\}\, B_1 \qquad \textit{Lemma 14 and Eq. (4.19)} \tag{4.20}$$

Figure 4.6: Initial proof steps for the application case of Lemma 26.

abstraction of the form $\lambda z, q_{11}$ where $q_{11} \simeq_{[\mathrm{sPe}],\beta} t_{11}'$. By Lemma 24, r_1 is also an abstraction of the form $r_1 = \lambda z, r_{11}$. Lemma 14 and Eq. (4.19) Pages 93 and 106 give

$$r_{11} : R_2 \lessdot \{u'/x\}\, B_2 \rightsquigarrow q_{11}$$

because coercion is stable by substitution, we get,

$$\{r_2/z\}\, r_{11} : \{r_2/z\}\, R_2 \lessdot \{r_2/z\}\, \{u'/x\}\, B_2 \rightsquigarrow \{r_2/z\}\, q_{11}$$

and we also have $\{r_2/z\}\, q_{11} \simeq \{r_2/z\}\, \{u'/x\}\, t_{11}'$ by stability of $\simeq$ and because $((\lambda z, r_{11})\, r_2) \hookrightarrow_\beta \{r_2/z\}\, r_{11}$, we can apply Lemma 19 Page 95 to obtain

$$((\lambda z, r_{11})\, r_2) : \{r_2/z\}\, R_2 \lessdot \{r_2/z\}\, \{u'/x\}\, B_2 \rightsquigarrow \{r_2/z\}\, q_{11}'$$

and $q_{11}' \simeq \{r_2/z\}\, q_{11}$. Now, remarking that

$$\{r_2/z\}\, \{u'/x\}\, t_{11}' \simeq \{u'/x\}\, \{t_2'/z\}\, t_{11}'$$

(substitutions are sequential and not parallel), and $((\lambda z, t_{11}')\, t_2') \hookrightarrow_\beta \{t_2'/z\}\, t_{11}'$, we get by transitivity of $\simeq$ that $q_{11}' \simeq \{u'/x\}\, ((\lambda z, t_{11}')\, t_2')$. We can conclude with rule R-CHECK.

$$\Gamma \vdash \{u/x\}\, t : \{u'/x\}\, T \rightsquigarrow e_1 \qquad \textit{induction hypothesis} \tag{4.21}$$
$$e_1 \simeq \{u'/x\}\, t'' \qquad \textit{induction hypothesis} \tag{4.22}$$
$$\Gamma \vdash \{u/x\}\, t \rightsquigarrow e_2 : E_2 \qquad \textit{inversion of Eq. (4.21)} \tag{4.23}$$
$$e_2 : E_2 <: \{u'/x\}\, T \rightsquigarrow e_1 \qquad \textit{inversion of Eq. (4.21)} \tag{4.24}$$
$$\{u'/x\}\, t'' : \{u'/x\}\, T <: \{u'/x\}\, B \rightsquigarrow e_3 \qquad \textit{stability (Lemma 18)} \tag{4.25}$$
$$e_3 \simeq \{u'/x\}\, r \qquad \textit{stability (Lemma 18)} \tag{4.26}$$
$$e_2 : E_2 <: \{u'/x\}\, B \rightsquigarrow e_4 \quad \textit{Lemma 17 and Eqs. (4.22), (4.24) and (4.25)} \tag{4.27}$$
$$e_3 \simeq e_4 \qquad \textit{Lemma 20 and Eqs. (4.24), (4.25) and (4.27)} \tag{4.28}$$

Figure 4.7: Proof steps for case R-CHECK of Lemma 26.

- $$\frac{\Gamma, x : A \vdash t \rightsquigarrow t'' : T \qquad t'' : T <: B \rightsquigarrow r}{\Gamma, x : A \vdash t : B \rightsquigarrow r}\ \text{R-CHECK}$$

Judgements (4.23) and (4.27) allow to apply rule R-CHECK to conclude $\Gamma \vdash \{u/x\}\, t : \{u'/x\}\, B \rightsquigarrow e_4$. By transitivity of $\simeq_{[\mathrm{sPe}],\beta}$, Eqs. (4.26) and (4.28), $e_4 \simeq_{[\mathrm{sPe}],\beta} \{u'/x\}\, r$. □

4.3 Preservation of typeability by the encoding

In Section 2.2 Page 36, we showed that any valid judgement of *PVS-Cert* can be encoded into a valid judgement of λΠ[Pe]. Similarly, we show that any valid judgement of *PVS-Core* can be encoded into a valid judgement of $\mathfrak{S}$+[sPe]. Furthermore, provided that holes are replaced with actual proofs, $\mathfrak{S}$+[sPe] outputs judgements that are valid in λΠ[sPe].

As usual, we must ensure that terms that compute in *PVS-Core* are translated to terms that compute in $\mathfrak{S}$. Otherwise we could not translate the conversion rule of *PVS-Core* as a conversion of the framework. We already know that the syntactic translation [−] preserves computation. In the presence of coercions, we must ensure that inserting coercions does not break β-redexes, and that refined β-redexes compute to refined β-reducts.

Lemma 27. *Let* Γ *be well-formed context,* t *and* u *two terms, and* $t \hookrightarrow_\beta u$.

For any well-sorted term A in Γ, if $\Gamma \vdash [t] : A \rightsquigarrow t'$ then $\Gamma \vdash [u] : A \rightsquigarrow u'$ and $t' \simeq_{[\mathrm{sPe}],\beta} u'$. If $\Gamma \vdash [t] \rightsquigarrow t' : T$, then $\Gamma \vdash [u] : T \rightsquigarrow u'$ and $t' \simeq_{[\mathrm{sPe}],\beta} u'$.

Proof. We can show easily that $[t] \hookrightarrow_\beta [u]$ because the encoding is shallow. Therefore, without loss of generality, we consider terms t and u in the image of $[-]$.

By induction on a context γ such that $t = \gamma[e]$ and the reduction happens at the head of e. For the base case, consider the empty γ, and t is of the form $t = ((\lambda x : T_1, t_1)\, t_2)$. Then $t \hookrightarrow_\beta u = \{t_2/x\}\, t_1$. By Lemma 10 Page 63, $\Gamma \vdash ((\lambda x : T_1, t_1)\, t_2) \rightsquigarrow ((\lambda x : T_1', t_1')\, t_2') : T$, so the refinement $((\lambda x : T_1', t_1')\, t_2') \hookrightarrow_\beta \{t_2'/x\}\, t_1'$. Lemma 26 Page 104 gives $u' \simeq_{[\mathrm{sPe}],\beta} \{t_2'/x\}\, t_1'$ for some u' with $\Gamma \vdash \{t_2/x\}\, t_1 : T \rightsquigarrow u'$.

For the inductive case, we assume that $t \hookrightarrow_\beta u$ and for each possible context γ, we verify that $\Gamma \vdash \gamma[t] : A \rightsquigarrow t'$, $\Gamma \vdash \gamma[u] : A \rightsquigarrow u'$ and $t' \simeq_{[\mathrm{sPe}],\beta} u'$. We note $\Box$ the hole of contexts. Let e be a term. The possible contexts are 1. $(\Pi x : \Box, e)$, 2. $(\Pi x : e, \Box)$, 3. $(\lambda x : \Box, e)$, 4. $(\lambda x : e, \Box)$, 5. $(e\, \Box)$, 6. $(\Box\, e)$.

For Items 1, 3 and 5, terms t and u are checked against the same type: either $\star$ or the domain of e for the application case. By induction hypothesis, the refinements of t and u are convertible.

For Item 2, A must be a sort in $\{\star, \Box\}$, we note it s_A. By inversion and Lemma 18 Page 95, the sort inferred from $\Pi x : e, u$ is s_A. By the shape of $\mathcal{P}^{\lambda\Pi}$, s_A is the sort inferred from both t and u. Therefore, the refinements t' and u' are both the inference of t and u, which are syntactically equal to the refinement obtained by checking them against s_A. By induction hypothesis, we get $t' \simeq_{[\mathrm{sPe}],\beta} u'$.

For Item 2, we have $t = \Pi x : e, T_1 \hookrightarrow_\beta u = \Pi x : e, U_1$ with $T_1 \hookrightarrow_\beta U_1$. Proof steps described in Fig. 4.8 allow to conclude $\Pi x : e, U_1' \simeq \Pi x : e, T_1'$ using Eq. (4.30).

For Item 4, assume $t = \lambda x : e, t_1 \hookrightarrow_\beta \lambda x : e, u_1$, and $\Gamma \vdash e \Rightarrow_{\mathcal{S}} \star$. Then we have $\lambda x : e, t_1' \simeq \lambda x : e, v_1$ by Eq. (4.34).

For Item 6, assume $(t_1\, e) \hookrightarrow_\beta (u_1\, e)$. If u_1^π is not an abstraction, then Eq. (4.40) and Lemma 15 give $V_2 \simeq_{[\mathrm{sPe}],\beta} R_2$ and $u_1^\pi \simeq_{[\mathrm{sPe}],\beta} t_1^\pi$. By stability by context of $\simeq_{[\mathrm{sPe}],\beta}$, $(t_1^\pi\, e) \simeq_{[\mathrm{sPe}],\beta} (u_1^\pi\, e)$ (e is checked against V_1 or R_1, which are convertible by Eq. (4.42)).

If u_1^π is an abstraction $u_1^\pi = \lambda x : V_1, u_{11}$, then we also have w_2 of the form $w_2 = \lambda x, w_{21}$ by Lemma 14 Page 93. Finally, following Fig. 4.11 we have $(u_1^\pi\, e) = ((\lambda x, u_{11})\, e)$, and we can apply rule R-CHECK on Eq. (4.44), and we can

$\Gamma \vdash T_1 \rightsquigarrow T_1' : s_1$ *inversion of premise*
$\Gamma \vdash U_1 : s_1 \rightsquigarrow V_1$ *induction hypothesis* (4.29)
$V_1 \simeq T_1'$ *idem* (4.30)
$\Gamma \vdash U_1 \rightsquigarrow U_1' : s_2$ *inversion Eq.* (4.29)
$U_1' : s_2 \lessdot : s_1 \rightsquigarrow V_1$ *idem*
$U_1' = V_1 \wedge s_1 = s_2$ *Lemma 18* (4.31)
$\Gamma \vdash \Pi x : e, U_1 \rightsquigarrow \Pi x : e, U_1' : s_2$ *rule* R-PROD
$V = \Pi x : e, U_1'$ *rule* R-CHECK *and Eq.* (4.31) (4.32)

Figure 4.8: Proof steps for Lemma 27, case $t = \Pi x : e, \Box$.

$\Gamma \vdash t_1 \rightsquigarrow t_1' : T_1$ *inversion*
$\Gamma \vdash u_1 : T_1 \rightsquigarrow v_1$ *induction hypothesis* (4.33)
$v_1 \simeq t_1'$ *induction hypothesis* (4.34)
$\Gamma \vdash u_1 \rightsquigarrow u_1' : U_1$ *inversion Eq.* (4.33) (4.35)
$u_1' : U_1 \lessdot : T_1 \rightsquigarrow v_1$ *idem*
$\lambda x, u_1' : \Pi x : e, U_1 \lessdot : \Pi x : e, T_1 \rightsquigarrow \lambda x, v_1$ *rule* SUB-FUN
$\Gamma \vdash \lambda x, u_1 \rightsquigarrow \lambda x, u_1' : \Pi x : e, U_1$ *rule* R-ABST *on Eq.* (4.35) (4.36)
$\Gamma \vdash \lambda x, u_1 : \Pi x : e, T_1 \rightsquigarrow \lambda x : e, v_1$ *rule* R-CHECK (4.37)

Figure 4.9: Proof steps for stability by context, Lemma 27, case $\lambda x : e, \Box$.

$\Gamma \vdash t_1 \rightsquigarrow t_1' : T_1$ *inversion*

$T_1 <:_\Pi \Pi x : R_1, R_2$ *idem*

$t_1' : T_1 <: \Pi x : R_1, R_2 \rightsquigarrow t_1^\pi$ *idem*

$\Gamma \vdash t_1 : \Pi x : R_1, R_2 \rightsquigarrow t_1^\pi$ *rule* R-CHECK

$\Gamma \vdash u_1 : \Pi x : R_1, R_2 \rightsquigarrow w_1$ *induction hypothesis* (4.38)

$w_1 \simeq_{[\mathrm{sPe}],\beta} t_1^\pi$ *idem* (4.39)

$\Gamma \vdash u_1 \rightsquigarrow u_1' : U$ *inversion Eq.* (4.38)

$\Gamma \vdash u_1 \rightsquigarrow u_1^\pi :_\Pi \Pi x : V_1, V_2$ *Lemma 23*

$u_1^\pi : \Pi x : V_1, V_2 <: \Pi x : R_1, R_2 \rightsquigarrow w_2$ (4.40)

$w_2 \simeq_{[\mathrm{sPe}],\beta} w_1$ *coherence* (4.41)

$V_1 \simeq_{[\mathrm{sPe}],\beta} R_1$ *Lemma 14 and Eq.* (4.40) (4.42)

Figure 4.10: Initial proof steps for Lemma 27, for the case $(\Box\, e)$.

$\lambda x, u_{11} : \Pi x : V_1, V_2 <: \Pi x : R_1, R_2 \rightsquigarrow \lambda x, w_{21}$

$\lambda x, u_{11} : \Pi x : R_1, V_2 <: \Pi x : R_1, R_2 \rightsquigarrow \lambda x, w_{21}$
rule SUB-RED *and Eq.* (4.42) (4.43)

$u_{11} : V_2 <: R_2 \rightsquigarrow w_{21}$ *Lemma 14*

$\{e/x\}\, u_{11} : \{e/x\}\, V_2 <: \{e/x\}\, R_2 \rightsquigarrow \{e/x\}\, w_{21}$ *stability Lemma 18*

$((\lambda x, u_{11})\, e) \hookrightarrow_\beta \{e/x\}\, u_{11}$ (4.44)

$((\lambda x, u_{11})\, e) : \{e/x\}\, V_2 <: \{e/x\}\, R_2 \rightsquigarrow c$ *Lemma 19* (4.45)

$c \simeq_{[\mathrm{sPe}],\beta} \{e/x\}\, w_{21}$ *idem 2*

Figure 4.11: Proof steps for stability by context, Lemma 27, case $(\Box\, e)$ in presence of β redexes.

conclude by transitivity and monotonicity of $\simeq_{[\mathrm{sPe}],\beta}$ since

$$c \simeq_{[\mathrm{sPe}],\beta} \{e/x\}\, w_{21} \longleftarrow_{\beta} ((\lambda x, w_{21})\, e) = (w_2\, e) \simeq_{[\mathrm{sPe}],\beta} (w_1\, e) \simeq_{[\mathrm{sPe}],\beta} (t_1^{\pi}\, e)\,. \square$$

Lemma 28. *Let M, N and A be terms. If $\Gamma \vdash [M] : A \rightsquigarrow M'$ and $M \hookrightarrow_{\beta}^{*} N$, then $\Gamma \vdash [N] : A \rightsquigarrow N'$ and $M' \simeq_{[\mathrm{sPe}],\beta} N'$.*

Proof. By induction on the number of computation steps $\hookrightarrow_{\beta}$ where the base case is handled by Lemma 27. $\square$

Lemma 29. *If A and B are two well formed types of* PVS-Core, *$\Gamma \vdash [A] : \mathtt{Type} \rightsquigarrow A'$, $\Gamma \vdash [B] : \mathtt{Type} \rightsquigarrow B'$, and $A \simeq_{\beta} B$, then $A' \simeq_{[\mathrm{sPe}],\beta} B'$.*

Proof. If $A = \mathtt{Type}$, then $B = \mathtt{Type}$ and $[A] = [B] = \mathtt{Type}$. Applying R-SIGN followed by R-CHECK, we obtain that $A' = B' = \mathtt{Type}$.

Otherwise, by confluence of $\hookrightarrow_{\beta}$, there is C such that $A \hookrightarrow_{\beta}^{*} C \hookrightarrow_{\beta}^{*} B$. By Lemma 28, let $\Gamma \vdash [A] : \mathtt{Type} \rightsquigarrow A'$, then $[\![\Gamma]\!] \vdash [C] : \mathtt{Type} \rightsquigarrow C'$ and $A' \simeq_{[\mathrm{sPe}],\beta} C'$.

A symmetrical argument yields $B' \simeq_{[\mathrm{sPe}],\beta} C'$ for an analogously defined B' where $[\![\Gamma]\!] \vdash [B] \rightsquigarrow B' : \mathtt{Type}$ and finally, $A' \simeq_{[\mathrm{sPe}],\beta} C' \simeq_{[\mathrm{sPe}],\beta} B'$. $\square$

Definition 38. Let Γ be a well-formed *PVS-Core* context. We note $\Gamma \mapsto \Gamma^{\circ}$ the translation from *PVS-Core* contexts to $\mathfrak{S}$ contexts defined by $\emptyset^{\circ} = \emptyset$ and $(\Gamma, x : A)^{\circ} = \Gamma^{\circ}, x : A'$ where $\Gamma^{\circ} \vdash [\![A]\!] : \star \rightsquigarrow A'$.

Theorem 2. *Let Γ be a context, t and A terms such that $\Gamma \vdash_{\overline{\mathrm{Po}}} t : A$. Then the three following propositions hold, $\vdash \Gamma^{\circ}$, $\Gamma^{\circ} \vdash [\![A]\!] : \star \rightsquigarrow A'$ and $\Gamma^{\circ} \vdash [t] : A' \rightsquigarrow t'$.*

Proof. The three propositions are shown simultaneously by induction on the typing derivation $\Gamma \vdash_{\overline{\mathrm{Po}}} t : A$.

For rule EMPTY page 29, there is nothing to do: the empty context is translated as the empty context.

For rule DECL, induction hypothesis gives $\vdash \Gamma^{\circ}$ and $\Gamma^{\circ} \vdash [T] : [\![s]\!] \rightsquigarrow T'$. Since variables are unchanged, we still have $v \notin \Gamma^{\circ}$. If $s = \mathtt{Type}$, then $[\![s]\!] = \mathtt{Type}$. By correctness of the refiner (Proposition 10 Page 67), $\Gamma^{\circ} \vdash T' : \mathtt{Type}$ and thus with rule SIGN, $\Gamma^{\circ} \vdash \mathtt{El}\,[T'] : \star$ which allows to conclude with rule DECL.

For rule VAR, induction hypothesis gives $\vdash \Gamma^{\circ}$. By definition of $[\![\cdot]\!]$ and $x \mapsto x^{\circ}$, there is A' such that $(x : A') \in \Gamma^{\circ}$.

For rule SORT, refer to the proof of Theorem 1 Page 46, and we can conclude by partial completeness of the type checker with refinement.

For rule SUBTYPE-ELIM, induction hypothesis gives $\Gamma^\circ \vdash (\texttt{psub}\ [A]\ [P]) : \texttt{Type} \rightsquigarrow (\texttt{psub}\ A'\ P')$ and $\Gamma^\circ \vdash [t] : (\texttt{El}\ (\texttt{psub}\ A'\ P')) \rightsquigarrow t'$. Transitivity of coercion rules Fig. 4.3 ensures that if $[t]$ can be checked against $(\texttt{El}\ (\texttt{psub}\ A'\ P'))$, then it can be checked against $(\texttt{El}\ (A'))$.

For rule SUBTYPE-INTRO, induction hypothesis gives $\Gamma^\circ \vdash [t] : A' \rightsquigarrow t'$ where $\Gamma^\circ \vdash [\![A]\!] : \star \rightsquigarrow A'$. Like above, transitivity of coercion rules Fig. 4.3 ensures that if $[t]$ can be checked against $(\texttt{El}\ A')$, then it can be checked against $(\texttt{El}\ (\texttt{psub}\ A'\ P'))$ with R-CHECK.

For rule PROD, we have $(s_1, s_2, s_3) = (\texttt{Type}, \texttt{Type}, \texttt{Type})$. In that case, induction hypothesis gives $\Gamma^\circ \vdash [A] : \texttt{Type} \rightsquigarrow A'$ and $(\Gamma, x : A')^\circ \vdash [B] : \texttt{Type} \rightsquigarrow B'$. By definition, $(\Gamma, x : A)^\circ = \Gamma^\circ, x : A''$ where $\Gamma^\circ \vdash [\![A]\!] : \star \rightsquigarrow A''$. Since s_1 is $\texttt{Type}$, $[\![A]\!] = (\texttt{El}\ [A])$ so by inversion of R-APPL, $A'' = A'$. With rules R-CHECK and R-ABST, we can derive $\Gamma^\circ \vdash \lambda x : (\texttt{El}\ A'), B' : \Pi x : (\texttt{El}\ A'), \texttt{Type} \rightsquigarrow \lambda x : (\texttt{El}\ A'), B'$. It allows us to conclude using SIGN with $(A' \rightsquigarrow (\lambda x, B'))$.

For rule ABST, we have $\Gamma \vdash A : \texttt{Type}$ and $\Gamma, x : A \vdash B : \texttt{Type}$ because $(\texttt{Type}, \texttt{Type}, \texttt{Type})$ is the only product rule of *PVS-Core* with $\texttt{Type}$ as last sort. Thus induction hypothesis gives 1. $\Gamma^\circ \vdash [A] : \texttt{Type} \rightsquigarrow A'$ along with $\Gamma^\circ, x : A' \vdash [B] : \texttt{Type} \rightsquigarrow B'$ and 2. $\Gamma^\circ, x : A' \vdash [t] : (\texttt{El}\ B') \rightsquigarrow t'$ (induction hypothesis hold because for each judgement, its inputs are well formed thanks to previous judgements). With rule R-SIGN and Item 1 we can derive 3. $\Gamma^\circ \vdash (\texttt{El}\ [A]) : \star \rightsquigarrow (\texttt{El}\ A')$ where $[\![A]\!] = (\texttt{El}\ [A])$. We can apply rule R-CHECK on Item 3 and Item 2 to obtain $\Gamma^\circ, x : (\texttt{El}\ A') \vdash \lambda x : [\![A]\!], [t] : \Pi x : (\texttt{El}\ A'), (\texttt{El}\ B') \rightsquigarrow \lambda x : (\texttt{El}\ A'), t'$. Finally, note that $[\![\Pi x : A, B]\!] = (\texttt{El}([A] \rightsquigarrow [B]))$, hence by rule R-SIGN, $\Gamma^\circ \vdash (\texttt{El}\ ([A] \rightsquigarrow [B])) : \star \rightsquigarrow (\texttt{El}\ (A' \rightsquigarrow B'))$. Furthermore, $(\texttt{El}(A' \rightsquigarrow (\lambda x, B'))) \hookrightarrow^*_{\beta, \mathcal{R}_{[\text{sPe}]}} \Pi x : \texttt{El}A', (\texttt{El}B')$ so we can use the rule R-CHECK to conclude $\Gamma^\circ, x : (\texttt{El}\ A') \vdash \lambda x : [\![A]\!], [t] : (\texttt{El}\ (A' \rightsquigarrow (\lambda x, B'))) \rightsquigarrow \lambda x : (\texttt{El}\ A'), t'$.

For rule APPL, the idea is to use rules R-APPL and R-CHECK. We first justify that we can use R-PROD-C. By induction hypothesis, $\Gamma^\circ \vdash [\Pi x : B_1, B_2] : \texttt{Type} \rightsquigarrow B'$. By definition of translation and inversion of typing, $B' = B'_1 \rightsquigarrow (\lambda x : (\texttt{El}\ B'_1), B'_2)$ and by Eq. (2.25) Page 51, $(\texttt{El}\ (B'_1 \rightsquigarrow (\lambda x : (\texttt{El}\ B'_1), B'_2))) \hookrightarrow \Pi x : (\texttt{El}\ B'_1), (\texttt{El}\ B'_2)$. By induction hypothesis,

$$\Gamma^\circ \vdash [t] : B' \rightsquigarrow t'. \tag{4.46}$$

By inversion, there are X' and X such that $\Gamma^\circ \vdash [t] \rightsquigarrow t' :_\Pi \Pi x : X', X$. We have, $X' \simeq (\texttt{El}\ B'_1)$ because either rule SUB-FUN is used or rule R-CAST. Hence we can derive $\Gamma^\circ \vdash [t\, u] \rightsquigarrow (t'\, u') : \{u'/x\}\, X$ using induction hypothesis $\Gamma^\circ \vdash u : (\texttt{El}\ B'_1) \rightsquigarrow u'$. Finally we must verify that we have $\Gamma^\circ \vdash [t\, u] : A' \rightsquigarrow$

t'' where $\Gamma^\circ \vdash [\{u/x\}\, B_2] : \texttt{Type} \rightsquigarrow A'$ and $A' \simeq \{u'/x\}\, B_2'$ by Lemma 26 Page 104. Either $\{u'/x\}\, X \simeq (\texttt{El}\ \{u'/x\}\, B_2')$, in which case there is nothing to do. Otherwise, we have covariance on the codomain of the type of t. This case is possible only if t is an abstraction because there is no subtyping on product types in *PVS-Core* (and $(t\, u)$ is thus a β-redex). By induction hypothesis Eq. (4.46) and inversion of rule R-CHECK and rule SUB-FUN, X is coercible to B_2, hence the application $(t'\, u')$ can be coerced from $\{u'/x\}\, X$ to $\{u'/x\}\, B_2'$ and we conclude with R-CHECK.

For rule SIGN, induction hypotheses allow to apply rule R-SIGN because for each judgement $(f[\boldsymbol{x} : \boldsymbol{A}] : B : s) \in \Sigma_{\mathrm{Po}}$, there is a symbol $(\hat{f}[\boldsymbol{x} : [\![\boldsymbol{A}]\!]] : [\![B]\!] : [\![s]\!]) \in \Sigma_{\mathrm{Pe}}$.

For rule CONV, s is either `Type` or `Kind` (`Prop` is not a sort). If it is `Kind`, then $A = B = \texttt{Type}$ because `Type` is the only inhabitant of `Kind`. Otherwise, we have $\Gamma \vdash B : \texttt{Type}$ induction hypothesis gives $\Gamma^\circ \vdash [t] : B' \rightsquigarrow t'$ where $\Gamma^\circ \vdash B : \texttt{Type} \rightsquigarrow B'$. The sort s_A of A is `Type` as well (see F. Gilbert 2018, Section 3.2) and $\Gamma^\circ \vdash [A] : [\![s_A]\!] \rightsquigarrow A'$. We have $[\![s_A]\!] = \texttt{Type}$, so by Lemma 29 Page 111, $A' \simeq_{[\mathrm{sPe}],\beta} B'$. We can use rule R-CHECK and coerce $[t]$ from its inferred type to B' and then from B' to A' to conclude $\Gamma^\circ \vdash t : A' \rightsquigarrow t'$. □

Any well-formed judgement of *PVS-Core* can be translated to a well-formed judgement in the encoding of *PVS-Cert*. However, this judgement may contain holes: each time the subtyping rule SUBTYPE-INTRO page 88 is used in the *PVS-Core* derivation, a coercion rule Section 4.2.1 that generates a hole is used to type the *PVS-Cert*-encoded judgement (note that this assertion is not accurate, one may build a derivation tree in *PVS-Core* with useless subtyping rules which will not have any counterpart in the typing derivation in $\mathfrak{S}$+[sPe]).

Completeness of the encoding of *PVS-Core* states that whenever a proposition of *PVS-Core* encoded in λΠ[sPe] is inhabited, then the original proposition is also inhabited in *PVS-Core*. Because *PVS-Cert* is a conservative extension of *PVS-Core* (ibid., Chapter 8), we do not need to prove more than the completeness of λΠ[sPe].

4.4 Conclusion

We have embedded simple type theory with implicit predicate subtyping (named *PVS-Core*) into the type system $\mathfrak{S}$ (defined in Chapter 3 Page 61). For this, a signature and a rewrite system to embed terms of *PVS-Core* in λΠmr in Section 4.1 has been provided.

In order to type check judgements with implicit subtyping, we provided a rewrite system to implement a suitable cast relation: whenever type A is a subtype of type B in *PVS-Core*, then the embedding of A can be cast to the embedding of B. The type system made of $\mathfrak{S}$ parametrised by the signature and rewrite system for the embedding and the cast relation is denoted $\mathfrak{S}$+[sPe].

We end the chapter by proving that the embedding of *PVS-Core* in $\mathfrak{S}$+[sPe] preserves typing.

Chapter 5

Translating *PVS*

Previous chapters described theoretic embeddings of terms from various systems to a logical framework. Although there was a focus on computability through bidirectional formalisms and decidable congruences, few applications have been shown.

In this section, we take a look at the translation of the standard library of the proof assistant *PVS* called 'Prelude' to *Dedukti*, an implementation of a logical framework with computation rules.

5.1 Computational logical frameworks

Dedukti (Deducteam 2022a) is an implementation[1] of λΠmr. It is used as a proof checker: given a signature, it returns whether it is is well-formed or not. Because *Dedukti* is by design minimal, it does not handle existential variables nor coercions. Therefore we use an alternative implementation *Lambdapi* (Deducteam 2022b) that can refine incomplete terms into complete *Dedukti* terms: *PVS* files are translated to *Lambdapi* sources, *Lambdapi* refines these terms and outputs *Dedukti* files.

Both *Lambdapi* and *Dedukti* interact with signatures which are coded as lists of typing declarations and rewrite rules. Figure 5.1 defines the syntax of typing

[1] At the time of writing, a specification for implementations of λΠmr is being developed. This specification is to be called *Dedukti*, whereas the legacy implementation, called *Dedukti* in this manuscript, has been renamed to 'the *dk* tool suite'. It contains a parser and a type checker (called *dkcheck*) for the *Dedukti* language.

$\langle id \rangle$ ::= ...
$\langle stmt \rangle$ ::= symbol $\langle id \rangle$: $\langle t \rangle$ [:= $\langle t \rangle$]? [begin admitted]?;
| rule $\langle t \rangle \hookrightarrow \langle t \rangle$;

Figure 5.1: BNF grammar of *Lambdapi* statements. Non terminal are written between angles ⟨like this⟩. Optional groups are written between square brackets followed by a question mark [like this]?. The class $\langle id \rangle$ is the class of identifiers, it is left unspecified (in practice, it may be the class of words formed with ASCII letters). The class $\langle t \rangle$ is the class of terms of type systems modulo (see Definition 2 Page 28) with holes $\Diamond$ (see Section 3.3 Page 82).

declarations in *Lambdapi*.

Semantically, a symbol declaration adds a typing judgement to the global signature. If the optional group [:= e] is used in the statement, then a rewrite rule from the name to expression e is also added to the signature. If the keyword 'begin admitted' is appended to the declaration, holes in the type or the definition are replaced by fresh axioms added to the signature. These fresh axioms (generated by *Lambdapi*) are translated as symbol declarations in *Dedukti*. A rewrite rule declaration adds a rewrite rule to the signature.

In this part, *PVS* code will be shown like this

```
stack: TYPE
% Here is a comment
```

5.2 Statements and theories

The translation of *PVS* terms is defined in of Fig. 4.2 Page 89, but the syntax of *PVS* slightly differs from the one we used to express the translation.

PVS' developments are split into *theories* which can be seen as modules or namespaces. Most basic theories consist of a name and a list of *statements*. There are four kinds of statements: declarations, definitions, axioms and theorems. For each kind of statement, we informally give its semantics, its syntax in *PVS* and its translation to *Lambdpi*.

Declarations The declaration of name `f` of type `A` states that `f` is an inhabitant of type `A`. A declaration is translated as a fresh typing judgement in a typing signature Σ that is accumulated through the translation. For instance,

```
double(x: real): real
```

declares that name `double` applied to a real number is a real number. It is translated to

$$\text{symbol } \texttt{double} : (\texttt{El}\ ([\texttt{real}] \rightsquigarrow (\lambda x, [\texttt{real}])));$$

Definitions Definitions are essential for practical developments, they allow to use names instead of large expressions. In general, the unfolding of definitions is represented by a dedicated reduction called δ (Owre and Shankar 1997b; Severi and Poll 1994; The Coq Development Team 2022) *Dedukti* does not handle definitions *per se*. However, since it has a liberal reduction, we may simply use it for definitions as well: A definition is translated as a typing judgement (just like declarations) with a unique rewrite rule which rewrites the name of the definition to the larger expression.

```
double(x: real) : real = x + x
```

declares that `double(x)` expands to `x + x`. It is translated as the following declarations

$$\text{symbol } \texttt{double} : (\texttt{El}\ ([\texttt{real}] \rightsquigarrow (\lambda x, [\texttt{real}]))) \coloneqq \lambda x, x + x;$$

Axioms An axiom states that a certain proposition is inhabited, but the inhabitant is not provided. Like declarations, they are translated as fresh typing judgements.

```
foo : POSTULATE Bar
```

declares that `foo` is a proof of proposition `Bar`. Since `Bar` is a proposition, its type is `bool`. It is translated as

$$\text{symbol } \texttt{foo} : (\texttt{Prf}\ [\texttt{Bar}]);$$

Theorems Theorems are like definitions, but the type of the term is a proposition, and the term itself is seen as a proof of that proposition, with, in general,

no computational relevance. It is thus common in implementations to drop the expansion: what matters is that a proof has been given at some point.

```
foo : THEOREM Bar
```

defines `foo` as a proof of proposition `Bar`. Because there is no proof term in *PVS*, the proof is not given in the statement, it is asked to the user, and stored elsewhere as a tree of tactics. It is translated as a single judgement and a new hole

symbol `foo` : (`Prf` [`Bar`]) ≔ ◊ begin admitted;

5.3 *PVS* language features

So far, we have only dealt with a minimal kernel for *PVS*. *PVS* has more features which must be correctly encoded to be able to translate actual files. We give some of these features, and sketch encodings into *Lambdapi*.

5.3.1 Overloading

PVS allows to overload symbols (Owre, Shankar et al. 2020, Chapter 8) (this feature is sometimes called *ad-hoc polymorphism*). For instance, the minus symbol `-` is overloaded to be both a unary minus and a binary subtraction. Overloading is also used to provide both curried and uncurried version of some operators, or to pass arguments as theory parameters (see for instance the function `least_upper_bound?` defined in theories 'orders' and 'orders_alt').

Overloading can be removed by the translation. When a term is type checked by *PVS*, each 'name' (a 'name' is a *Lisp* structure which represents a variable or constant) contains a pointer to the statement in which it is declared. The translation takes into account both the original name and the declaration to create a new, unambiguous identifier in *Lambdapi*.

5.3.2 Theory parameters and polymorphism

Theories may depend on a context containing type variables as well as object variables. Consider the theory Fig. 5.2. All statements of the theory abstract over the parameter `t`, therefore, when not in the scope of the theory, objects of the theory 'stack' must instantiate the parameter `t`. This instantiation is performed in *PVS* by providing a context, for instance, the type

```
stack[t: TYPE]: THEORY
  stack: TYPE
  nil: stack
  push(e: t, s: stack): stack
END stack
```

Figure 5.2: Theory for stacks with polymorphism.

of stacks of booleans is `stack[bool].stack`. We could say that there are two different applications: one with brackets to instantiate theories' parameters and the other noted with parentheses being the usual application of λ-calculus. In the framework, we have one application only. Because there is no mechanism to implicitly abstract over statements in a module, theory parameters are translated as function arguments. The type declaration `stack` above is translated as 'symbol `stack` : Πt : `Type`, `Type`;' and `nil` is translated as 'symbol `nil` : Πt : `Type`, (`stack` t);'. Such an encoding is not conservative since the non applied term `stack` is not typeable in *PVS* while it is in the framework: Πt : `Type`, `Type` is not in the image of any embedding.

5.3.3 Logical connectives

The primitive components for logical content of *PVS* are falsity `false`, the equality `=` and a (polymorphic) ternary `IF`. Because of predicate subtyping, type checking branches may require to assume that the condition holds or that the negation holds. For example, the first branch of `IF(x /= 0, 1 / x, x)`, is well-typed only if we can prove `x /= 0` (by typing of the division operator `/`). Therefore, the `then` branch of a conditional expression `IF(P, then, else)` is type-checked in a *logical context* where proposition `P` is assumed true whereas the `else` branch is type checked in a logical context where `not P` is assumed.

We (arbitrarily) chose to use usual encodings of simple type theory using the implication and the universal quantification as primitive connectives (as done by F. Gilbert 2018, Section 2.1.2). The ternary (propositional) 'if' can be encoded by

symbol `if` :
Πp : (`El o`), (`El` ((`Prf` p) $\rightsquigarrow$ `o`)) $\to$ (`El` ((`Prf` $(\neg p)$) $\rightsquigarrow$ `o`)) $\to$ (`El o`) :=
$\lambda p, \lambda then, \lambda else, (p \Rightarrow then) \Rightarrow (\lambda x, (\neg p) \Rightarrow else)$;

The logical context is encoded by the context used in the type checking relation, and its extension is hence encoded by the dependent implication ($p \Rightarrow then$) where '*then*' binds a proof of the condition p in the expression used as first branch. Therefore, assuming a division operator

$$\text{symbol } / : (\texttt{El}\,(\texttt{real} \rightsquigarrow (\texttt{psub}\,(\lambda r, r \neq 0)) \rightsquigarrow \texttt{real}));$$

the latter function can be encoded in *PVS-Cert* with

$$\lambda x, (\texttt{if}\,(x \neq 0)\,(\lambda p : (\texttt{Prf}\,(x \neq 0)), (/\,1\,(\texttt{pair}\,x\,p)))\,(\lambda p, 1))\,.$$

5.3.4 Tuples and matching

PVS handles natively tuples and dependent tuples. A tuple is a heterogeneous fixed-length collection of elements. To distinguish between types and inhabitants, we will use *telescopes* (de Bruijn 1991) for types of tuples, and tuples for the objects. We may have dependent telescopes, which behave like chained dependent pairs, and non dependent telescopes. Owre and Shankar 1997b provide the typing rules for pairs but not for general telescopes. Tuples are noted (x, y, z) and telescopes $[x : A, y : B, z : C]$ or $[A, B, C]$ if B does not depend on x and C does not depend on x nor y. We infer naturally typing rules for telescopes,

$$\frac{\Gamma \vdash A : \texttt{TYPE} \qquad \Gamma, x : A \vdash [\boldsymbol{B}] : \texttt{TYPE}}{\Gamma \vdash [x : A, \boldsymbol{B}] : \texttt{TYPE}}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash (\boldsymbol{b}) : [\boldsymbol{B}]}{\Gamma \vdash (a, \boldsymbol{b}) : [x : A, \boldsymbol{B}]}$$

In contrast with binary cartesian products, $[A, [B, C]]$ is different from $[A, B, C]$.

In idiomatic *PVS*, telescopes are used abundantly in function declarations. Therefore functions are seldom curried. To currify *PVS*, we may use some procedure defined by

$$\textsc{Curry}([D_0, \boldsymbol{D}] \to R) = D_0 \to \textsc{Curry}([\boldsymbol{D}] \to R)$$
$$\textsc{Curry}([D_1, D_2] \to R) = D_1 \to D_2 \to R.$$

But this method fails to currify correctly the defined type `PRED` such that `PRED[t] = [t -> bool]`, in particular when `PRED[[t, t]]` appears (this function is defined in the theory 'defined_types' of the Prelude).

To ease the definition of functions, *PVS* pattern-matches the components of arguments that are tuples. To define a function of two arguments, that is, a function that takes a tuple of two elements as argument, the syntax `f(x, y) = x + y` is used, instead of the more cumbersome `f(x) = x`1 + x`2` (where `x`n` is the `n`th element of tuple `x`). We thus have to define a matching operator on tuples as well. Let `A.N` be the type of (unary) integers, `T n` be the type of telescopes of length `n` and `code` a function that injects telescopes into type codes (`code` is thus of type $\Pi n : \texttt{A.N}, (\texttt{T}\, n) \to \texttt{Type}$ where the first argument is left implicit).

```
symbol match [l: A.N] [ret: Set] [tt: T l] (arg: El (code tt)):
  El (mkarr tt ret) → El ret;
rule match nil $e ↪ $e
with match (&cons $x $y) $f ↪ match $y ($f $x)
with match (cons $x $y) $f ↪ match $y ($f $x);
```

Function `mkarr` is defined by

```
injective symbol mkarr [n: A.N]: T n → Set → Set;
rule mkarr nil! $Ret ↪ $Ret
with mkarr (&cons! $X $Q) $Ret ↪ arrd $X (λ x, (mkarr ($Q x) $Ret))
with mkarr (cons! $X $Q) $Ret ↪ arr $X (mkarr $Q $Ret);
```

so that, denoting `[A,B]` for a (non dependent) telescope and `A ~> B` for an encoded arrow, `mkarr [A, B] ret` returns `A ~> B ~> ret`, that is, `mkarr` transforms a telescope into a function type with `ret` for codomain. With these functions, we are able to translate a statement such as

```
XOR(A: bool, B: bool) = (A /= B)
```

by

symbol `XOR` :
$(\texttt{El}\ [\texttt{[A, B -> bool]}]) :=$
$\lambda x, (\texttt{match}\, x\, (\lambda A\, B : (\texttt{El}\ [\texttt{Prop}]), [\texttt{A /= B}]));$

where the variables `A` and `B` of the body `A /= B` are captured by the variables of the variables A and B of the abstraction.

5.3.5 Bounded quantification

Since there is a notion of subtypes in *PVS*, a possible extension is to allow theories to quantify over all subtypes of a given type. A subtype is declared in theory parameters with `[A TYPE FROM B]` to state that `A` is a subtype of `B`. Semantically, it means that `A` can be substituted by any type `C` such that the conjunct of the predicates that define `C` imply the conjunct of predicates defining `B`.

Contexts are extended with bindings of the form $X <: A$. The following rule is admissible,

$$\frac{X <: A \in \Gamma \qquad \Gamma \vdash t : X}{\Gamma \vdash t : A}$$

which allows to perform predicate subtyping on type variables. For instance, in context

$$\Gamma = \mathit{int} : \texttt{Type}, \mathit{nat} : \texttt{Type} := \texttt{psub}(\mathit{int}, (\lambda k, k \geq 0)), \\ X <: \mathit{nat}, x : X, \texttt{abs} : \mathit{int} \to \mathit{nat}$$

(where *nat* is defined) judgement '$\Gamma \vdash (\texttt{abs}\, x) : \mathit{nat}$' holds because x can be typed as *int* using the aforementioned rule: it is first typed from X to *nat* because $X <: \mathit{nat}$ is in Γ, then from *nat* to *int* by (syntactic) subtyping. We cannot ignore the judgement $X <: \mathit{nat}$ and translate it as $X : \texttt{Type}$, because there is no coercion from X to *nat*, so there is no t such that $x : (\texttt{El}\, X) <: [\![\mathit{nat}]\!] \rightsquigarrow t$ holds.

To ensure correctness, we can provide a `cast` operator defined as

$$\text{symbol } \texttt{cast} : \Pi A : \texttt{Type}, \Pi B : \texttt{Type}, \Pi x : (\texttt{El}\, A), (\texttt{El}\, B);$$

and we translate the previous example into $\Delta \vdash (\texttt{abs}\ (\texttt{cast}\, X\, \mathit{nat}\, x)) \Leftarrow [\![\mathit{nat}]\!]$ which in turn holds. However the `cast` operator is highly unsafe: it breaks completeness of the encoding since any type can be inhabited with `cast`. To avoid inhabiting any type, we can ask for a proof `p` that `X` is a subtype of `A` in `cast X A p x`.

There are two ways to provide such a proof `P`. Either the `cast` operator expects a semantical proof which states that the conjunct of predicates defining `X` implies the conjunct of predicates defining `A`. This requires an operator that retrieves the predicates that define a type. In the semantics manual, Owre and Shankar 1997b define the operator π for such a job. Or the operator expects

a syntactic proof which ensures that `X` can only be substituted by syntactic subtypes of `A`.

We chose to encode the syntactic constraint. For this, we consider a tree where types are nodes, and there is an edge between two types if one is a direct subtype of the other. For instance, there is an edge between `{x: A | P}` and `A`. The evidence that X is a subtype of A is then a path in the tree from X to A. Paths can be built with the system in Fig. 5.3. This proof system is encoded into the framework using a dedicated type

```
constant symbol Restriction: Set → Set → TYPE;
```

where an inhabitant of `Restriction A B` is an evidence that `A` is a subtype of `B`. Then the inference rules are encoded:

```
constant symbol Rest-refl (a: Set): Restriction a a;
constant symbol Rest-sub (a: Set) (p: El (a ~> prop)): Restriction (psub [a] p) a;
constant symbol Rest-trans (a b c: Set):
  Restriction a b → Restriction b c → Restriction a c;
constant symbol Rest-fun (d r0 r1: Set):
  Restriction r0 r1 → Restriction (d ~> r0) (d ~> r1);
constant symbol Rest-tuple [len: A.N]
                           (hd0: Set) (tl0: TL.T len)
                           (hd1: Set) (tl1: TL.T len):
  Restriction hd0 hd1 → Restriction (TL.code tl0) (TL.code tl1) →
        Restriction (TL.code (TL.cons! hd0 tl0)) (TL.code (TL.cons! hd1 tl1));
```

and we can define the cast operator

```
symbol cast (a: Set) (b: Set) (_: Restriction a b) (_:El a): El b;
```

If no computation is added on `cast`, we may encounter conversion issues, because, for instance, `(cast (psub A P) A (Rest-sub A P) x)` is not convertible with `(fst A P x)`. Therefore we add the following reduction rules where `TL` is a namespace for telescope-related operations, and in that namespace, `car` retrieves the head of a telescope, `cdr` retrieves its tail and `code` transforms a telescope into a type code,

```
rule cast _ _ (Rest-refl _) $x ↪ $x;
rule cast _ _ (Rest-trans $a $b $c $prf-ab $prf-bc) $x ↪
  cast $b $c $prf-bc (cast $a $b $prf-ab $x);
rule cast _ _ (Rest-sub _ _) $x ↪ fst $x;
rule cast _ _ (Rest-fun _ $r0 $r1 $proof) $f ↪ λ x, cast $r0 $r1 $proof ($f x);
rule cast _ _ (Rest-tuple $h0 $t0 $h1 $t1 $proof-hd $proof-tl) $l ↪
  TL.cons (cast $h0 $h1 $proof-hd (TL.car $l))
          (cast (TL.code $t0) (TL.code $t1) $proof-tl (TL.cdr $l));
```

These reductions pattern-match on the proof provided, which allows to have an orthogonal, type-preserving rewrite system. Again, just like in *PVS*, subtyping is covariant on the codomain but neither covariant nor contravariant in the domain because of rule REST-FUN.

$$\text{REST-REFL}\ \frac{}{A \leq A} \qquad \text{REST-SUB}\ \frac{}{\texttt{psub}(A, P) \leq A} \qquad \text{REST-TRANS}\ \frac{A \leq B \qquad B \leq C}{A \leq C} \qquad \text{REST-FUN}\ \frac{R \leq R'}{D \to R \leq D \to R'}$$

$$\text{REST-TUP}\ \frac{T_0 \leq T_0' \qquad (T_i)_{i \in \{1 \ldots\}} \leq (T_i')_{i \in 1 \ldots}}{\boldsymbol{T} \leq \boldsymbol{T}'}$$

Figure 5.3: Inference rules to derive subtyping judgements $A \leq B$.

Remark 11. At the time of writing, this encoding is not used in the transpiler yet. Indeed, since this encoding manipulates subtyping proofs, the translation function must introduce them. Therefore, any subtype binding of the form `X TYPE FROM Y` has to introduce two elements in the context, the type `X` as well as a proof that `X` is a subtype of `Y`. Because contexts may be extended only through abstractions or products in the framework both `X` and the proof of subtyping must be abstracted over.

For instance, the theory

```
min_nat[T: TYPE FROM nat]: THEORY
  min(S: (nonempty?[T])): {a | S(a) AND (FORALL x: S(x) IMPLIES a <=
       x)}
  [...]
END min_nat
```

would be translated as

$$\texttt{symbol min}: \Pi T : \texttt{Type}, (\texttt{Restriction}\, T\, \texttt{nat}) \to \Pi S : (\texttt{El}\ (\texttt{psub}\ (\texttt{nonempty?}\, T))), \ldots;$$

where `min` is a function that accepts three arguments: a type T, a proof that T is a subtype of `nat` and a set S.

Regarding the code of the transpiler, it is fairly easy to abstract over an additional parameter, but one has to remember which theory uses bounded quantification. The translator should thus inspect the declarations of the constants and the theory they are declared in to see whether it uses bounded quantification.

5.3.6 Records

Records (Owre, Shankar et al. 2020, Section 5.11) are native types in *PVS*, and can be anonymous. For instance, given the expression `(# x := 4, y := 5 #)`, *PVS* infers the record type `[# x : nat, y : nat #]`. We do not have yet an encoding for records, so such judgements cannot be translated. However, we may be able to axiomatise declared record types. For instance, assuming a record type `rc : [# a: T1, b: T2 #]` is declared in *PVS*, we can declare a fresh type, a constructor and the projections ,

```
constant symbol rc: Set;
constant symbol make_rc (_: El T1) (_: El T2): El rc;
constant symbol a_of_rc : El (rc ~> T1);
constant symbol b_of_rc : El (rc ~> T2);
```

and then translate `(# a := e1, b := e2 #)` by `make_rc a b`; given a record value `v`, projection `v`a` is translated as `a_of_rc v` and `v`b` as `b_of_rc v`.

However, because *PVS* allows anonymous record types, the translation has to axiomatise each anonymous record type occurrence and ensure that two equivalent anonymous record type are also equivalent in the framework.

5.3.7 Fixpoints and inductive types

Inductive types (Pierce 2002) are allowed in *PVS*. For instance, the transitive closure of a relation `R`, defined in the theory 'relations' of the Prelude reads as follows

```
TC(R)(x, y): INDUCTIVE bool = R(x,y) OR (EXISTS z: TC(R)(x,z) AND
    TC(R)(z,y))
```

Recursive functions (Owre, Shankar et al. 2020, Section 3.4) can also be defined, such as the exponential in theory 'exponentiation':

```
expt(r, n): RECURSIVE real =
  IF n = 0 THEN 1 ELSE r * expt(r, n-1) ENDIF
MEASURE n;
```

A measure must be provided along with the definition to ensure that it is decreasing. For each recursive definition, a type correctness condition is issued to prove the recursive calls are smaller according to the measure.

Following the intuitions of Gaspard Férey 2021, a recursor can be defined with a `fix` operator:

```
symbol fix (a: Set)(r: Set)(meas:El a → Nat)
           (f: Π(x:El a),
               (Π(y:El a)(π:Prf(lenat (meas y) (meas x))), El r) → El r):
  El a → El r;
```

where `Nat` is the type of natural numbers and `lenat` is the usual order on natural numbers. The idea is to create a fixpoint such that $(\texttt{fix}\, f) = (f\ (\texttt{fix}\, f))$. But because we want it polymorphic, we must take as argument the codes of the domain `a` and codomain `r`. We also take as argument the measure `meas` used to ensure that the recursive call is issued on a smaller argument. Then we take as argument the function `f` itself. This function has type

$$\Pi x : (\texttt{El}\, a),\\ (\Pi y : (\texttt{El}\, a), \Pi \pi : (\texttt{Prf}\ (\texttt{lenat}\ (\texttt{meas}\, y)\ (\texttt{meas}\, x))), (\texttt{El}\, r)) \to \\ (\texttt{El}\, r)$$

where x is the current argument of the fixpoint $(\texttt{fix}\, f\, x)$, y is the value on which the recursive call occurs and π is a proof that the measure of y is smaller than the measure of x. We also define a computation rule

```
rule fix $a $r $meas $f $x ↪
     $f $x
        (λ (y: El $a) (π:Prf(lenat ($meas y) ($meas $x))),
         fix $a $r $meas $f y);
```

Let us see how the function behaves on the definition of the factorial:

```
symbol nat: Set;
rule (El nat) ↪ Nat;
symbol mul: El (nat ~> nat ~> nat);
symbol pred: El (nat ~> nat);
symbol pred-lower: Prf (∀ nat (λ n: El nat, (lenat (pred n) n)));
symbol fac* n fac ≔
  (if (eqnat n 0)
      1
      (mul n (fac (pred n) (pred-lower n))));
symbol fac n ≔ fix nat nat (λ x, x) fac* n;
```

We give a sequence of reduction of the term `(fac 5)`

1. We begin by unfolding the symbol `fac`,

```
fix nat nat (λ x, x) fac* 5
```

2. which produces a term that reduces to, by the fixpoint rule,

```
(λ n fac,
  if (eqnat n 0)
      1
      (mul n (fac (pred n) (pred-lower n))))
5
(λ(y:El nat) (π:Prf(lenat y 5)), fix nat nat (λ x, x) fac* y)
```

3. This new term β reduces to

```
if (eqnat 5 0)
   1
   (mul 5 ((λ y π, fix nat nat (λ x, x) fac* y) 4 (pred-lower 5)))
```

4. and further β reduces to

```
if (eqnat 5 0)
   1
   (mul 5 (fix nat nat (λ x, x) fac* 4))
```

The result is convertible with `if (eqnat 5 0) 1 (mul 5 (fac 4))`. We can see that the second β reduction (Item 4) may occur only if a proof of decreasingness is provided.

However, this method has a severe drawback: the termination of the fixpoint reduction rule depends on the reduction strategy. Indeed, we chose to perform a β reduction at Item 3, but we could have continued unfolding the reduction rule of the fixpoint. For that reason, this encoding is not used in the translation, and the computational content of recursive functions is removed.

5.3.8 Abstract datatypes

Owre and Shankar 1997a also define 'abstract datatypes'. Abstract datatypes allow one to define sructures that can be freely generated by a finite number of constructors, such as lists. Datatype declarations behave like macros, *i.e.* they are expanded (by *PVS*) to a theory.

5.4 Implementation

The code in charge of the translation of *PVS* specifications to *Lambdapi* signatures is a patch for *PVS*, written in *Common Lisp* (X3J13 Committee 1994). It is part of the *personoj* (Hondet 2022) suite. The code defines a new command for *PVS* that can be invoked from *PVS*' 'read-eval-print-loop'. The command takes as argument the name of a *PVS* theory and outputs its translation. The patch defines mainly a printing function `pp-dk` such that `(pp-dk strm obj)` prints the translation of PVS object `obj` onto stream `strm`.

An arbitrary theory from the Prelude may be printed with the commands given in Fig. 5.4

The translation has only been tested on the Prelude so far. In particular, in *PVS*, all theories of the Prelude are implicitly imported. Therefore, for any theory of the Prelude, its *PVS* version does not contain any import command

```
$ ./pvs -raw
[...]
* (get-theory "booleans")

#<Theory booleans>
* (pp-dk *standard-output* * t)

require open personoj.lhol personoj.logical personoj.pvs_cert
personoj.eq personoj.restrict personoj.coercions;
require personoj.telescope as TL;
require personoj.extra.arity-tools as A;
require open personoj.nat;
require open personoj.cast;
// Theory booleans
constant symbol prop: Set;
symbol prop: Set ≔ prop begin admitted;
constant symbol false: El prop begin admitted;
constant symbol true: El prop begin admitted;
constant symbol NOT: El (prop ~> prop) begin admitted;
constant symbol ¬: El (prop ~> prop) begin admitted;
constant symbol AND: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol &: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol ∧: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol OR: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol ∨: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol IMPLIES: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol =>: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol ⇒: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol WHEN: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol IFF: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol <=>: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
constant symbol ⇔: El ((TL.code (TL.double! prop prop)) ~> prop) begin admitted;
NIL
*
```

Figure 5.4: Commands to call the translator. The translator is called on the theory 'booleans'. Prompts starting with `$` are shell prompts, the ones starting with `*` are *Lisp* prompts. The output of each command directly follows the prompt up to the next prompt.

whereas its translation to *Lambdapi* imports all Prelude theories defined before it. Some theories of the Prelude are part of the encoding. These theories can be identified in the output as the ones with the prefix `personoj` in Fig. 5.4.

As described in Chapter 4 Page 87, any `pair` coercion gives birth to a hole (denoted $\Diamond$). In *Lambdapi*, these holes are materialised by existential variables. For a declaration or definition to be well-typed in *Lambdapi*, it must not contain any existential variable, that is, they must have been instantiated to some term. Because holes that stand for missing proofs cannot be instantiated automatically, *Lambdapi* has been modified to generate fresh symbols, which can be interpreted as axioms, to replace holes. These axioms are materialised as constant definitions when exported to *Dedukti*.

Notes on the development workflow Developing in *Lisp* may feel odd in regard with other languages. Readers may consult the blogpost of Losh 2018 for an overview of the tooling used to program in *Lisp*. *PVS* is itself a *Lisp* image with additional functions to parse *PVS* files, prove propositions interactively, assert whether two *PVS* terms are equal &c. Documentation may be obtained using usual *Lisp* introspection facilities. For instance, the function `describe` can be used

```
$ ./pvs -raw
[...]
* (describe 'lambda-expr)
PVS:LAMBDA-EXPR
  [symbol]

LAMBDA-EXPR names the standard-class #<STANDARD-CLASS PVS:LAMBDA-EXPR>:
  Class precedence-list: LAMBDA-EXPR, BINDING-EXPR, EXPR, SYNTAX,
                         STANDARD-OBJECT, SB-PCL::SLOT-OBJECT, T
  Direct superclasses: BINDING-EXPR
  Direct subclasses: RECURSIVE-DEFN-CONVERSION, TUPTYPE-CONVERSION,
                     RECTYPE-CONVERSION, FUNTYPE-CONVERSION,
                     FIELDEX-LAMBDA-EXPR, LET-LAMBDA-EXPR, SET-EXPR,
                     LAMBDA-CONVERSION, LAMBDA-EXPR-WITH-TYPE
  No direct slots.
* (describe 'binding-expr)
PVS:BINDING-EXPR
  [symbol]

BINDING-EXPR names the standard-class #<STANDARD-CLASS PVS:BINDING-EXPR>:
  Class precedence-list: BINDING-EXPR, EXPR, SYNTAX, STANDARD-OBJECT,
                         SB-PCL::SLOT-OBJECT, T
  Direct superclasses: EXPR
  Direct subclasses: QUANT-EXPR, LAMBDA-EXPR
  Direct slots:
    OP
      Initargs: OP, :OP
      Readers: OP
      Writers: (SETF OP)
```

```
BINDINGS
  Initargs: BINDINGS, :BINDINGS
  Readers: BINDINGS
  Writers: (SETF BINDINGS)
EXPRESSION
  Initargs: EXPRESSION, :EXPRESSION
  Readers: EXPRESSION
  Writers: (SETF EXPRESSION)
COMMAS?
  Initargs: COMMAS?, :COMMAS?
  Readers: COMMAS?
  Writers: (SETF COMMAS?)
CHAIN?
  Initargs: CHAIN?, :CHAIN?
  Readers: CHAIN?
  Writers: (SETF CHAIN?)
```

Data structures for expressions are defined in file `src/classes-expr.lisp` of *PVS* distribution 7.1, and the declarations are defined in `src/classes-decl.lisp`.

5.5 Results

We translated parts of the Prelude of *PVS*. The whole Prelude contains 1000 propositions split among 133 theories. We are able to translate and type check 881 propositions split among 85 theories. There are theories that have not been translated because they use either abstract datatypes, records or recursive definitions. Some theories that use bounded quantification (particularly that instantiate bounded types) could not be translated either, and we failed to translate a few other theories for still unclear reasons. Judgements[2] are not translated. On the other hand, some type correctness conditions are translated as propositions. The translation contains 77 propositions that are type correctness conditions, therefore we have a total of 952 translated propositions. We observe that terms are substantially larger with explicit predicate subtyping. Table 5.1 shows that some theories explode in size when predicate subtyping is made explicit. Indeed, even though half of the translation in *Dedukti* are less than 2.17 times bigger than the translation in *Lambdapi*, theories in *Dedukti* are in average 21.3 times bigger, suggesting that there are extreme values in the distribution, as shown by the important standard deviation of 63.0. The more predicate subtypes are nested, the more the size of terms increases (because of explicit coercions). Furthermore, Table 5.1 does not count axioms used to instantiate type correctness conditions, which would further increase the size

[2]As defined in Section 1.1.2 Page 12.

of *Dedukti* files (the translated theory 'real_defs' contains 2742 such axioms generated from 53 declaration and definitions). With these axioms taken into account, the average ratio soars up to 366, the standard deviation to 1390 and the higest ratio to 8830. Therefore, theories like 'sets' that have at most one level of predicate subtyping (that is, (`psub` $t\ p$) but not (`psub` (`psub` $t\ p$) q)) have a least expansion ratio than theories like 'naturalnumbers' which contains up to four levels of predicate subtyping: 'sets' grows from 15Kio[3] (in *Lambdapi*) to 44Kio (in *Dedukti*) while 'naturalnumbers' grows from 11Kio to 4.7Mio. In consequence, type checking theories takes more time with *Lambdapi* than with *PVS*. In particular, type checking in *PVS* does not take subtyping into account, it only checks that arguments have an appropriate maximal supertype (that is, if `f` is a function whose domain is `psub`(`bool`, p), then (`f e`) is well-typed when the maximal supertype of `e` is `bool`). Table 5.1 shows the size of some theories and their type checking time while Table 5.3 displays the number of statements and the number of generated axioms of the same theories. We can clearly see that type checking time is not proportional to the size of the theory: among the listed theories, 'real_defs' is the longest to type check but not the biggest in size. Type checking is sensitive to the shape of terms, and in particular to how nested predicate subtyping is. We also see that *Dedukti* takes in general less time to type check theories than *Lambdapi*, except for 'extra_real_props' whose number of axioms per entity ratio is substantially higher than the ratios of other theories. We can infer that type checking expanded terms is generally faster than type checking and refining terms, except for some pathological cases where subtyping generates a considerable amount of type correctness conditions. In that case, because we expect type correctness conditions to be redundant, sharing should reduce drastically the size of the theory (and its type checking time). Furthermore, Type checking a file requires to load all previous signatures which can take a substantial amount of time, especially in *Lambdapi* where signatures are heavier. For instance, the *Lambdapi* signature for 'real_defs' weighs 520Mio whereas its *Dedukti* counterpart weighs 249Mio. We suspect signatures are heavier in *Lambdapi* because *Bindlib* (the library used to implement binders) uses higher order abstract syntax and thus signatures contain closures; where *Dedukti* uses de Bruijn indices. Readers concerned with performance should rather look at *Dedukti* (Deducteam 2022a) or more specifically *Kontroli* by Färber 2022 which can handle bigger files than *Lambdapi*.

[3] A kibioctet (abbreviated Kio) is $2^{10} = 1024$ octets, a mebioctet (abbreviated Mio) is 2^{20} octets, or 1024Kio.

Table 5.1: Distribution of the ratios between the size of theories translated to *Dedukti* (with explicit predicate subtyping) and the size of theories translated to *Lambdapi* (with implicit predicate subtyping). The first cell indicates that theories translated to *Dedukti* are in average 21.3 times bigger than the theories translated to *Lambdapi*. Columns labelled '25%', '50%' and '75%' are the first, second and third quartiles. Axiom declarations for type correctness conditions that appear in *Dedukti* files are not counted.

Mean	Std. dev	Min	25%	50%	75%	Max
21.3	63.1	0.761	1.47	2.17	5.20	386

Table 5.2: Size and type checking time of some theories of *PVS*' 'Prelude' listed in topological order. The first line indicates that the theory 'functions' weighs 8Kio when translated to *Lambdapi* (with implicit predicate subtyping), 22Kio when translated to *Dedukti* (with explicit predicate subtyping), *Lambdapi* takes 0.04 seconds (wall clock time) to type check (and refine) the translation and *Dedukti* takes 0.00 second as well to type check the (refined) translation. Theories have been type checked on a processor 'Intel Core i5-8265U' with 15GiB of random access memory.

Theory	LSize (Kio)	DSize (Kio)	LTime (s)	DTime (s)
functions	7	22	0.04	0.00
orders	10	25	0.05	0.00
sets	15	44	0.08	0.01
sets_lemmas	45	190	0.13	0.03
naturalnumbers	11	4700	6.3	1.3
real_defs	61	310 000	100	91
real_props	130	21 000	20	8.4
extra_real_props	170	220 000	55	120

Table 5.3: Amount of statements. The first line indicates that the theory 'functions' contains 14 statements (a statement is either a declaration, a definition or a proposition) and type checking the theory generates 2 type correctness conditions.

Theory	Statements	Axioms
functions	14	2
orders	23	3
sets	52	25
sets_lemmas	119	127
naturalnumbers	34	416
real_defs	53	2742
real_props	220	3582
extra_real_props	62	6258

5.6 Conclusion

In this section, we applied the theoretical work of preceding chapters to translate the standard library of *PVS* to *Lambdapi*, an implementation of $\mathfrak{S}$+[sPe]. Our theory does not handle yet all features used by *PVS* in its standard library. The implementation supports some of these features although their embedding has not been studied extensively, and other features are not supported at all. We are also able to make subtyping explicit, which allows us to analyse the size increase of terms when making subtyping explicit.

Chapter 6

Exporting *PVS* proofs

As a proof assistant, a substantial part of *PVS* is dedicated to the manipulation of proofs. Historically, we distinguish several ways to store proofs. For the de Bruijn criterion, proofs ought to be stored in full to be checked by other independent checkers. The *LCF* architecture by Milner 1972 rather records the inference rules used to perform a proof. A proof is verified by rerunning these inference rules on the theorem, and ensuring that these inference rules are properly used. The trust we have on a *LCF*-style system depends on the kernel that implements the inference rules of the logic. The more this kernel is complex, the less we may trust it. *PVS* is closer to *LCF* style proof assistants, but it is also highly automated and its kernel is large.

This chapter explores ways to translate *PVS* proofs to logical frameworks, in particular λΠmr implemented by *Dedukti* and *Lambdapi*.

6.1 Proof representations

To understand how proofs are represented in *PVS*, we will describe succinctly how users interact with their specification through *PVS*.

To define a theory in *PVS*, users mainly write definitions and specify properties of these definitions as propositions. There is no proof object in theories. Propositions are proved interactively: when type checking the theory, *PVS* asks users to prove theorems in a new window. This window shows a prompt, where users may write tactics to prove the proposition incrementally. Finally, when the proposition is proved, the window is closed and the proof script is recorded

into another file (out of the theory).

Such a workflow emphasises the separation between defining a theory, and proving it. There is no notion of proof in the *PVS* specification language, and *PVS* does not specify any proof language, besides a (substantial) set of tactics (Shankar et al. 2021).

On the other hand, logical frameworks based on λΠmr use the same representation for proofs and terms: proofs are terms whose types are propositions. This principle—called the Curry-de Bruijn-Howard correspondence—is embedded into the encoding of *PVS-Cert* (Fig. 2.4 Page 39) through the mapping `Prf`: propositions are terms of type (`El o`), and π is a proof of P if $\vdash \pi : (\texttt{Prf}\ P)$ (see Blanqui, Dowek et al. 2021, page 4).

For instance, if P is a proposition, we have $\vdash \lambda x : (\texttt{Prf}\ P), x : (\texttt{Prf}\ (P \Rightarrow P))$ (because $(\texttt{Prf}\ P) \to (\texttt{Prf}\ P) \simeq_{[\mathrm{Pe}]} (\texttt{Prf}\ (P \Rightarrow P))$), so $\lambda x : (\texttt{El}\ P), x$ is a proof of $P \Rightarrow P$. In *PVS*, the proposition $P \Rightarrow P$ is proved by the sequence of tactics `(flatten propax)` (sequences are written as S-expressions).

Small LCF-style systems have already been encoded successfully into *Dedukti*, such has *HOL-Light* (Assaf and Burel 2015). In such encodings, there is one symbol declaration per inference rule of the logic, and the application of an inference rule is translated as the application of a symbol. Because *PVS* is a large and highly automated program, it is difficult to justify how tactics are applied.

However, internally, *PVS* represents proofs as trees whose nodes are labelled with propositions. Each node of the tree is labelled with a proposition, and there is an edge from node n_1 to node n_2 when the application of a certain tactic on the proposition of n_1 produce the proposition of node n_2. In practice, tactics often produce several 'sub goals' (a goal is a proposition to be proved), and proofs are therefore rather trees than lists. Proof trees are available in the *Lisp* environment once the proof has been rerun on the initial statement: intermediate goals are not saved, only the tactics are.

In the rest of this chapter, propositions are expressed in *PVS-Cert*. To simplify notations, we will write '$P \Rightarrow Q$' instead of '$\Pi x : P, Q$' when P and Q are propositions (that is, their type is `Prop`). We remind that if '$\Gamma \vdash h : P \Rightarrow Q$' and '$\Gamma \vdash i : P$', then '$\Gamma \vdash (h\, i) : Q$': proofs of implications are functions and therefore implication are eliminated by term application.

6.2 Proof scripts to incomplete terms

The objective is thus to design a function ϕ which maps proof trees to proof terms. If τ is a proof tree, we note $\tau :: P$ to state that P is the root of tree τ. For any tree $\tau :: P$, let P' be defined by $\vdash [P] : (\texttt{El o}) \rightsquigarrow P'$ (where $[-]$ is the object translation defined in Fig. 4.2 Page 89), the proof of tree τ is $\phi(\tau)$ such that $\vdash \phi(\tau) : (\texttt{Prf}\ P')$.

PVS implements a classical sequent calculus (Owre and Shankar 1997b), but inference steps between nodes of the trees are generally more complex than the application of an inference rule of the sequent calculus.

Since there is *a priori* no way to determine which inference rule is used, the idea is to replace inference rules by holes. For instance, assume tree τ is of the form

$$\dfrac{\dfrac{\ldots}{Q_1} \quad \dfrac{\ldots}{Q_2}}{P}$$

(which can also be represented by the symbolic expression $(P\,(Q_1\ \ldots)\ (Q_2\ \ldots)))$. Then we know that a proof π_P is obtained from proofs π_{Q_1} of Q_1 and π_{Q_2} of Q_2, but we do not know how. Therefore, provided we have a proof $\pi_\rightarrow$ of $Q_1 \Rightarrow Q_2 \Rightarrow P$, we can conclude $\pi_P = \left(\pi_\rightarrow\, \pi_{Q_1}\, \pi_{Q_2}\right)$.

We define a function $|-|$ from trees of propositions to trees of propositions that transforms a tree of goals to a tree containing the justification of inference steps as labels. The notation $(\Rightarrow Q_i)_i \Rightarrow P$ is short for $Q_1 \Rightarrow Q_2 \Rightarrow \ldots \Rightarrow P$ or just P if the sequence $(Q_i)_i$ is empty.

$$\left|\dfrac{(\tau_i :: Q_i)_i}{P}\right| = \dfrac{|\tau_i|}{(\Rightarrow Q_i)_i \Rightarrow P}$$

Trees can be annotated with proofs, such trees are noted

$$\dfrac{\dfrac{\ldots}{\rho_1 : Q_1} \quad \dfrac{\ldots}{\rho_2 : Q_2}}{\pi : P}$$

We define the function $\|-\|$ that transforms proof-annotated proof trees to terms by

$$\left\|\dfrac{(\tau_i)_i}{\pi : P}\right\| = \left(\pi\ \|\tau_i\|_i\right) \tag{6.1}$$

For the following function, we introduce the notion of *named* holes. Named holes are variables of a countable set $\mathcal{Y}$ noted $?x$ that are handled like regular holes by the type checker (with rule B-HOLE page 83). A proof problem $\mathcal{P}$ is a set of judgements of the form $\{\vdash ?x : P\}$ which states that $?x$ is a hole that should be replaced by a term of type P. The function $(-)^\bullet$ annotates a proof tree with named holes:

$$\left(\frac{(\tau_i)_i}{P}\right)^\bullet = \text{let } (\chi_i, \mathcal{P}_i)_i = ((\tau_i)^\bullet)_i \text{ in } \left(\frac{(\chi_i)_i}{?h : P}, \{\vdash ?h : P\} \bigcup_i \mathcal{P}_i\right) \quad (6.2)$$

where $?h$ must be fresh and the 'let in' construct binds, for each i, $(\tau_i)^\bullet$ to elements χ_i and $\mathcal{P}_i$. The result is a pair made of a tree whose labels are annotated with a typed hole and a proof problem containing the type of the hole.

The following lemma provides some specifications for the procedures.

Definition 39. An indexing function $f : \mathcal{Y} \to \mathcal{X}$ is an injection that can be extended to a mapping $\hat{f} : \mathcal{T}(\mathcal{X} \cup \mathcal{Y}) \to \mathcal{T}(\mathcal{X})$ defined by

$$\hat{f}(t\,u) = \left(\hat{f}(t)\,\hat{f}(u)\right) \qquad \hat{f}(\lambda x : T, t) = \lambda x : \hat{f}(T), \hat{f}(t)$$
$$\hat{f}(\Pi x : T, U) = \Pi x : \hat{f}(T), \hat{f}(U) \qquad \hat{f}(x) = x$$
$$\hat{f}(?x) = x$$

where the generated x must be fresh in the case of named holes. To simplify notations, we do not distinguish indexings f from their extension $\hat{f}$.

Lemma 30.
- *Procedures $|-|$, $\|-\|$ and $(-)^\bullet$ terminate.*
- *For any proof tree $\tau :: P$, with $(\chi, \mathcal{P}) = (|\tau|)^\bullet$, for any indexing I, let Δ be such that for each judgement $\{\vdash ?x : P\}$ in $\mathcal{P}$, then $(I(?x) : P) \in \Delta$, then $\Delta \vdash I(\|\chi\|) : P$.*

Proof. For each of these procedures, the recursive calls are operated on sub-trees of the argument. We conclude by finiteness of proof trees.

By induction on the tree τ. Assume τ is of the form

$$\frac{\chi_1 :: Q_1 \quad \cdots \quad \chi_n :: Q_n}{P}$$

By definition, $|\tau|$ is

$$\frac{|\chi_1| :: R_1 \quad \cdots \quad |\chi_n| :: R_n}{Q_1 \Rightarrow ... \Rightarrow Q_n \Rightarrow P}$$

and $(\tau)^\bullet$ yields a proof problem $\mathcal{P} = \mathcal{Q}_1 \cup ... \cup \mathcal{Q}_n$ where for all i, $(\chi'_i, \mathcal{Q}_i) = (|\chi_i|)^\bullet$; and a tree τ'

$$\frac{\chi'_1 :: (?i_1 : R_1) \quad \cdots \quad \chi'_n :: (?i_n : R_n)}{?h : Q_1 \Rightarrow ... \Rightarrow Q_n \Rightarrow P}$$

and $\|\tau'\| = (?h\ \|\chi'_1\|\ ...\ \|\chi'_n\|)$.

By induction hypothesis, there are $(\Delta_i)_i$ such that for all i, $\Delta_i \vdash I(\|\chi'_i\|) : Q_i$. We define Δ such that for all judgement $\{\vdash ?x : P\} \in \mathcal{P}$, $(I(?x) : P) \in \Delta$. Therefore, we can derive $\Delta \vdash I(?h) : Q_1 \Rightarrow ... \Rightarrow Q_n \Rightarrow P$, and thus, by the application rule, we get $\Delta \vdash I(?h)\,(I(\|\chi'_1\|))\ ...\ (I(\|\chi'_n\|)) : P$ where terms $I(\|\chi'_i\|)$ are typeable in Δ because by definition, for any i, $\Delta_i \subseteq \Delta$. □

The main motivation behind these procedures is that inference rules are easier to prove than successive goals: given a proof tree τ, it is easier to prove labels of $|\tau|$ than labels of τ itself.

Procedures $|-|$ and $\|-\|$ have been implemented in *personoj* (Hondet 2022): the *Lisp* function `pprint-proof` prints the proof of a formula. Each inference step $\frac{\boldsymbol{Q}}{P}$ is translated to a let-binding of the form 'let $v : \bigwedge \boldsymbol{Q} \Rightarrow P := ?P$ in' as shown in Fig. 6.1 Page 140. In *Lambdapi* source files, proofs are not different from definitions: given a proposition named `prop` whose statement is the type P, its translated proof π, the proposition is declared along with its proof with

```
symbol prop : (Prf P) := π;
```

If π contains holes (which will almost always be the case), type checking this declaration will fail because all existential variables must be instantiated for type checking to succeed. To solve these existential variables, we can take advantage of the proof mode of *Lambdapi* which allows to prove sub-goals incrementally using tactics.

6.3 Filling gaps

Using the procedures defined in the previous section, cross checking *PVS* proofs amount to solve the proof obligations generated by the annotation function $(-)^\bullet$.

These proof obligations can obviously be solved manually, the cross checking procedure is thus semi-automatic: previous procedures transform propositions into smaller sub-goals that are easier to prove.

```
* (let ((*current-context* *prelude-context*))
    (pprint-proof "xor_def"))
[...]
let _v2 : Prf (∀ [prop] (λ (A: El prop),∀ [prop] (λ (B: El prop),= [prop]
(TL.double [prop] [prop] (xor_def.XOR (@TL.double prop prop A B))
(@if prop A (λ _v3: Prf A, ¬ B) (λ _v4: Prf (¬ A), B))))) = _ in
_v2
"_v2"
```

Figure 6.1: Translation of the proof of the formula `xor_def` from the theory `xor_def`. *PVS*' prompt is preceded by `*`. The proof has only one inference step which is converted into the let-binding `_v2`. The final proof term is also `_v2`.

To obtain a fully automatic cross checking procedure, we may call automated solvers (El Haddad 2021). After all, *PVS* proofs often use a tactic `grind` that calls an SMT solver, so we may be able to replicate this behaviour. There are several issues to solve for this strategy to pay off:

1. proofs from *PVS* use predicate subtyping, there is no SMT solver that understand predicate subtyping[1];
2. *PVS* is higher order, higher order proof search is difficult;
3. the substitution with which lemmas are instantiated is not recorded when they are called through highly automated tactics[2] (*e.g.* the `auto-rewrite` tactic).

Automated provers used in proof assistants such as *Isabelle*'s 'sledgehammer' (Paulson and Blanchette 2012) encode higher order problems into first order logic (Czajka and Kaliszyk 2018; Robinson 1969). There are native higher order SMT solvers as well such as *Satallax* by Brown 2012 or *Leo-III* (Steen 2021; Steen and Benzmüller 2018) (see Benzmüller and Miller 2014, for more examples). There are also recent development to extend SMT solvers to support higher order theorem proving without higher order unification (Barbosa et al. 2019). On the other hand, Bentkamp et al. 2021 use higher order unification to handle proofs in higher order logic.

Item 3 is likely to need *PVS* to be edited in order to record substitutions used to instantiate lemmas. Otherwise, the framework may implement some form of proof search among already proved lemmas.

[1] with the notable exception of Yices 1, which is no longer maintained

[2] but lemmas and type correctness conditions themselves are recorded in proof states.

F. Gilbert 2018 showed that *PVS-Cert* is a conservative extension of simple type theory: a proposition P is inhabited in simple type theory if and only if it is inhabited in *PVS-Cert*. The translation $[\![-]\!]$ defined in (ibid., p. 129) translates *PVS-Cert* expressions to simple type theory expression. We will note $\langle - \rangle$ instead of $[\![-]\!]$ to avoid confusions.

For instance,

$$\langle \forall x : \mathtt{psub}(A, P), Q \rangle = \forall x : \langle A \rangle , (\langle P \rangle\ x) \Rightarrow \langle Q \rangle$$

where term Q may use $\pi_r(A, P, x)$, which has no counterpart in simple type theory. The translation adds a hypothesis of $(\langle P \rangle\ x)$ that will be used in $\langle Q \rangle$ instead of $\pi_r(A, P, x)$.

We can map function $\langle - \rangle$ over proof trees so that labels do not use predicate subtyping. Assuming proofs of a $\langle - \rangle$-mapped proof tree are filled, we want *in fine* to obtain a proof of the initial statement. Assuming proofs are found for propositions without predicate subtyping, we still have to build a proof of the initial statement which may contain predicate subtyping. A procedure $\langle - \rangle^{-1}$ mapping proofs of propositions of the form $\langle P \rangle$ to proofs of P is required to prove the initial goal.

Conjecture 3. *Let* $\mathcal{T}_{\mathrm{Pe}} = \mathcal{T}(\{\mathtt{Prop}, \mathtt{Type}, \mathtt{Kind}\}, \{\mathtt{psub}, \mathtt{pair}, \pi_\ell, \pi_r\})$. *Then there is a function*

$$\phi : \mathcal{T}_{\mathrm{Pe}} \times \mathcal{T}(\{\mathtt{Prop}, \mathtt{Type}, \mathtt{Kind}\}) \to \mathcal{T}_{\mathrm{Pe}}$$

such that, for all P, *whenever* $\vdash \pi : \langle P \rangle$, *then* $\phi(P, \pi)$ *reduces to a proof term in* $\mathcal{T}_{\mathrm{Pe}}$ *and* $\vdash \phi(P, \pi) : P$.

Related works Procedure $|-|$ is a minimal version of the extraction phase handled by *Ekstrakto* (El Haddad 2021; Haddad, Burel and Blanqui 2019). While *personoj* handles directly *PVS* proof trees, *Ekstrakto* processes problems in the *TSTP* format (Sutcliffe 2017). For each proof step in the *TSTP* file, it generates a *TPTP* problem to justify inferences of the trace file. *Ekstrakto* is capable of reconstructing proofs provided that *TPTP* sub-problems are solved by automated theorem provers that generate *Dedukti* proofs like *ZenonModulo* (Delahaye et al. 2013) or *ArchSat* (Bury 2019). The proof reconstruction phase performs the same task as function $\|-\|$ where proofs are *Lambdapi* constants. Each *TPTP* file gives birth to a *Lambdapi* file, in contrast to our procedure which generates a symbol declaration for each proof trace. Proofs to be found are modelled with existential variables by procedure $(-)^\bullet$ which is lighter

than using the module system provided by *Lambdapi*: for each sub-problem, *Ekstrakto* expects its proof to be stored in a term named `delta`. The existential quantification is thus performed at the level of modules rather than terms.

Results from the works of El Haddad 2021 are promising, but *Ekstrakto* only handles formulæ in clausal normal form.

6.4 Conclusion

This section completes the translation process: while Chapter 5 only dealt with definitions and propositions, in this chapter, we sketched a procedure to build proof terms in $\mathfrak{S}$+[sPe] from the proof traces we can get from *PVS*. We saw that transforming a *PVS* proof trace into a proof term amounts to certify that the deduction steps *PVS* performs are sound. Finding such certificates amounts to finding proofs in simple type theory with predicate subtyping. Fortunately, F. Gilbert 2018 showed that predicate subtyping can be safely eliminated from propositions, and thus the problem reduces to finding proofs in simple type theory.

Chapter 7

Conclusion

The goal of this document was to show how predicate subtyping *à la PVS* can be encoded in an extension of λΠmr with coercion insertion.

We showed how to encode certificates of typing derivations for *PVS-Core* (simple type theory with predicate subtyping) (Chapter 2). We proved the encoding preserves typing. In order to type check not only certificates but terms of *PVS-Core*, we set up a new type checker featuring existential variables and coercion insertion (Chapter 3). We showed that certificates for *PVS-Core* can be generated using coercions, and that typing in *PVS-Core* is preserved (Chapter 4). We discussed which features are missing from *PVS-Core* to be able to encode developments extracted from the standard library of *PVS* (Chapter 5). We specified a procedure to extract proofs associated to *PVS* specifications and reduce them to easier sub-problems, that can hopefully be solved by automated provers (Chapter 6).

Perspectives

The equational theory encoded by equations Eqs. (2.2), (2.3) and (β) pages 31, and 33 is a subset of the relation $\equiv_*$ defined in (ibid.). Indeed, $\equiv_*$ contains surjective pairing '$\mathtt{pair}(t, p, \pi_\ell(t, p, e), \pi_r(t, p, e)) \equiv_* e$' while our encoded congruence $\simeq_{[\mathrm{Pe}]}$ does not. We could add surjective pairing to $\simeq_{[\mathrm{Pe}]}$, but 1. the proof of Lemma 1 Page 34 does not hold anymore and 2. relation $\hookrightarrow_{\beta,\pi_\ell,\mathrm{SP}}$ (where $\hookrightarrow_{\mathrm{SP}}$ is surjective pairing) is not confluent on untyped terms. See Section 4.1 Page 88 for more information on surjective pairing.

Completeness of the encoding can only be conjectured. References to tackle completeness are given in Section 2.3 Page 42.

We chose to encode predicate subtyping using a constant `psub`. We could also encode predicate subtyping like existential quantifiers in λ-calculi,

$$(\texttt{El}\ (\texttt{psub}\, a\, p)) \hookrightarrow \Pi z : \texttt{Type}, (\Pi x : (\texttt{El}\, a), (\texttt{Prf}\ (p\, x)) \to (\texttt{El}\, z)) \to (\texttt{El}\, z)\,.$$

We provided a confluent rewrite relation to decide our equational theory $\simeq_{[\mathrm{Pe}]}$, but its termination is conjectured in Section 2.4.1 Page 49. More broadly, we implemented a weak form of proof irrelevance where only an argument of `pair` is proof irrelevant. It might be interesting to study full proof irrelevance in λΠme, where all propositions are considered equal.

Regarding the refiner, we chose not to include a dedicated checking rule for abstractions. Including it may allow the coercion judgement to not depend on the term, since we would remove rule SUB-FUN page 92. In consequence, the proof of Lemma 26 Page 104 should be reviewed.

No criteria have been provided to implement mechanical checks for typing preservation for the rewrite rules for coercions. Algorithms from (Blanqui 2020; Saillard 2015) could be adapted.

Similarly, criteria to check termination of the coercion system, assuming termination of the main rewrite system, could be studied. We saw that as soon as subtyping becomes transitive, we need recursive coercion rules, and coercion eliminators which we encoded with a non-linear rewrite rule. We may be able to remove this elimination rule by providing an algorithmic subtyping relation without reflexivity taking inspiration from Pierce 2002. Coercion would then be eliminated with rules of the shape '$(\kappa\, \iota\, \iota\, x) \hookrightarrow x$' where ι is an encoded sort of the source language (a maximal type in the terminology of *PVS*). The resulting subtyping strategy would become close to the one exposed by Owre and Shankar 1997b where terms are systematically cast to their maximal supertype. One could also look at λ*Prolog* (Dunchev et al. 2015; Felty et al. 1988) to replace rewriting in order to implement coercion systems.

F. Gilbert proposes several extensions for *PVS-Cert* in order to encompass more features of *PVS*. These features can be found in the standard library of *PVS* and are thus discussed in Chapter 5 Page 115, but their encoding is not proved to preserve typing, nor to be complete. Polymorphism, tuple types and record types are ubiquitous in *PVS*, and thus could be proposed (for the encoding of records in λΠmr, see Cauderlier and Dubois 2014). Proper encoding of recursive structures (Giménez 1994) could be used to encode recursive definitions, inductive types and 'datatypes' (Owre and Shankar 1997a). Bounded

quantification (Pierce 2002) would also be worthwhile since it is used in *PVS*' standard library.

Implementations The refiner has been implemented in *Lambdapi* and integrated in the main codebase (see Deducteam 2022b, version 2.1.0) but the coercion algorithm is at the moment prototypical. Furthermore, for now, right elements of pairs (`pair` $t\,u$) are systematically holes (that are refined into existential variables). Since all type correctness conditions are translated, many of these holes could probably be translated as instantiations of type correctness conditions. One could try to instantiate automatically these holes, searching through already declared propositions.

The proof extraction mechanism (Chapter 6) remains to be implemented completely and tested on the Prelude. Functions $|-|$, $\|-\|$ and $(-)^\bullet$ have been implemented by Hondet 2022, noting that *personoj* leaves 'holes' for existential variables, and it is *Lambdapi* that builds proof problems when transforming these holes into existential variables. Function $\langle - \rangle^{-1}$ and $\langle - \rangle$ remain to be both formalised and implemented. To take full advantage of the ability of *Lambdapi* to process proof scripts, procedures $\langle - \rangle^{-1}$ and $\langle - \rangle$ could be coded as tactics themselves. However, these procedures could not be coupled with the tactic `why3` anyway since the latter does not return proof terms yet, it adds axioms in the signature. One could also design a procedure that works in parallel: proof obligations could be output in another signature and proved separately using *ad-hoc* mechanisms (just like *PVS* stores proofs beside specifications). A prototype has been designed in earlier versions of (ibid.).

Index

Bibliography

Abadi, Martín and Luca Cardelli (1995). 'A Theory of Primitive Objects: Second-Order Systems'. In: *Sci. Comput. Program.* 25.2-3, pp. 81–116. DOI: 10.1016/0167-6423(95)00010-0. URL: https://doi.org/10.1016/0167-6423(95)00010-0.

AbdelGawad, Moez A. (2014). 'Subtyping in Java with Generics and Wildcards is a Fractal'. In: *CoRR* abs/1411.5166. arXiv: 1411.5166. URL: http://arxiv.org/abs/1411.5166.

Abel, Andreas and Thierry Coquand (June 2020). 'Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality'. In: *Logical Methods in Computer Science* Volume 16, Issue 2. DOI: 10.23638/LMCS-16(2:14)2020. URL: https://lmcs.episciences.org/6606.

Abel, Andreas, Thierry Coquand and Miguel Pagano (2011). 'A Modular Type-checking algorithm for Type Theory with Singleton Types and Proof Irrelevance'. In: *Log. Methods Comput. Sci.* 7.2. DOI: 10.2168/LMCS-7(2:4)2011. URL: https://doi.org/10.2168/LMCS-7(2:4)2011.

The Agda Team (Dec. 2021). *Agda Manual*. Version 2.6.2.1. URL: https://agda.readthedocs.io/en/v2.6.2.1/index.html.

Asperti, Andrea, Wilmer Ricciotti and Claudio Sacerdoti Coen (2014). 'Matita Tutorial'. In: *J. Formaliz. Reason.* 7.2, pp. 91–199. DOI: 10.6092/issn.1972-5787/4651. URL: https://doi.org/10.6092/issn.1972-5787/4651.

Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen and Enrico Tassi (2012). 'A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions'. In: *Log. Methods Comput. Sci.* 8.1. DOI: 10.2168/LMCS-8(1:18)2012. URL: https://doi.org/10.2168/LMCS-8(1:18)2012.

[SW] Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen and Enrico Tassi, *Matita* 2018. VCS: https://github.com/LPCIC/matita, SWHID:

⟨swh:1:dir:766d45eced3d73664ce3548352024f9c6022d362;origin=https://github.com/LPCIC/matita;visit=swh:1:snp:e9b4ab1512b17932f600788af1e1b97f25fbeef9;anchor=swh:1:rev:794ed25e6e608b2136ce7fa2963bca4115c7e175⟩.

Aspinall, David and Adriana B. Compagnoni (2001). 'Subtyping dependent types'. In: *Theor. Comput. Sci.* 266.1-2, pp. 273–309. DOI: 10.1016/S0304-3975(00)00175-4. URL: https://doi.org/10.1016/S0304-3975(00)00175-4.

Assaf, Ali (Sept. 2015). 'A framework for defining computational higher-order logics'. Theses. École polytechnique. URL: https://pastel.archives-ouvertes.fr/tel-01235303.

Assaf, Ali and Guillaume Burel (2015). 'Translating HOL to Dedukti'. In: *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*. Ed. by Cezary Kaliszyk and Andrei Paskevich. Vol. 186. EPTCS, pp. 74–88. DOI: 10.4204/EPTCS.186.8. URL: https://doi.org/10.4204/EPTCS.186.8.

Avizienis, Algirdas (1985). 'The N-Version Approach to Fault-Tolerant Software'. In: *IEEE Trans. Software Eng.* 11.12, pp. 1491–1501. DOI: 10.1109/TSE.1985.231893. URL: https://doi.org/10.1109/TSE.1985.231893.

Baader, Franz and Tobias Nipkow (1998). *Term rewriting and all that*. Cambridge University Press. ISBN: 978-0-521-45520-6.

Barbosa, Haniel et al. (2019). 'Extending SMT Solvers to Higher-Order Logic'. In: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*. Ed. by Pascal Fontaine. Vol. 11716. Lecture Notes in Computer Science. Springer, pp. 35–54. DOI: 10.1007/978-3-030-29436-6_3. URL: https://doi.org/10.1007/978-3-030-29436-6%5C_3.

Barendregt, Hendrik Pieter, Wil Dekkers and Richard Statman (2013). *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press. ISBN: 978-0-521-76614-2. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.

Barendregt, Henk and Kees Hemerik (1990). 'Types in Lambda Calculi and Programming Languages'. In: *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*. Ed. by Neil D. Jones. Vol. 432. Lecture Notes in Computer Science. Springer, pp. 1–35. DOI: 10.1007/3-540-52592-0_53. URL: https://doi.org/10.1007/3-540-52592-0%5C_53.

Barendregt, Henk and Adrian Rezus (1983). 'Semantics for Classical AUTOMATH and Related Systems'. In: *Inf. Control.* 59.1-3, pp. 127–147. DOI: 10.1016/S0019-9958(83)80033-3. URL: https://doi.org/10.1016/S0019-9958(83)80033-3.

Barras, Bruno (Nov. 1999). 'Auto-validation d'un système de preuves avec familles inductives'. PhD thesis. Université Paris 7.

Barras, Bruno and Bruno Bernardo (2008). 'The Implicit Calculus of Constructions as a Programming Language with Dependent Types'. In: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings.* Ed. by Roberto M. Amadio. Vol. 4962. Lecture Notes in Computer Science. Springer, pp. 365–379. DOI: 10.1007/978-3-540-78499-9_26. URL: https://doi.org/10.1007/978-3-540-78499-9%5C_26.

Benthem Jutting, L. S. van, James McKinna and Robert Pollack (1993). 'Checking Algorithms for Pure Type Systems'. In: *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers.* Ed. by Henk Barendregt and Tobias Nipkow. Vol. 806. Lecture Notes in Computer Science. Springer, pp. 19–61. DOI: 10.1007/3-540-58085-9_71. URL: https://doi.org/10.1007/3-540-58085-9%5C_71.

Bentkamp, Alexander et al. (2021). 'Superposition with Lambdas'. In: *CoRR* abs/2102.00453. arXiv: 2102.00453. URL: https://arxiv.org/abs/2102.00453.

Benzmüller, Christoph and Dale Miller (2014). 'Automation of Higher-Order Logic'. In: *Computational Logic.* Ed. by Jörg H. Siekmann. Vol. 9. Handbook of the History of Logic. Elsevier, pp. 215–254. DOI: 10.1016/B978-0-444-51624-4.50005-8. URL: https://doi.org/10.1016/B978-0-444-51624-4.50005-8.

Blanqui, Frédéric (Sept. 2001). 'Théorie des types et réécriture'. english version: http://hal.inria.fr/inria-00105525/. Theses. Université Paris Sud - Paris XI. URL: https://tel.archives-ouvertes.fr/tel-00105522.

— (2005). 'Definitions by rewriting in the Calculus of Constructions'. In: *Math. Struct. Comput. Sci.* 15.1, pp. 37–92. DOI: 10.1017/S0960129504004426. URL: https://doi.org/10.1017/S0960129504004426.

— (2020). 'Type Safety of Rewrite Rules in Dependent Types'. In: *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference).* Ed. by Zena M. Ariola. Vol. 167. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum

für Informatik, 13:1–13:14. DOI: 10.4230/LIPIcs.FSCD.2020.13. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.13.

Blanqui, Frédéric, Gilles Dowek et al. (2021). 'Some Axioms for Mathematics'. In: *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*. Ed. by Naoki Kobayashi. Vol. 195. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:19. DOI: 10.4230/LIPIcs.FSCD.2021.20. URL: https://doi.org/10.4230/LIPIcs.FSCD.2021.20.

Blanqui, Frédéric, Thérèse Hardin and Pierre Weis (2007). 'On the Implementation of Construction Functions for Non-free Concrete Data Types'. In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 95–109. ISBN: 978-3-540-71316-6.

Bouhoula, Adel, Jean-Pierre Jouannaud and José Meseguer (2000). 'Specification and proof in membership equational logic'. In: *Theor. Comput. Sci.* 236.1-2, pp. 35–132. DOI: 10.1016/S0304-3975(99)00206-6. URL: https://doi.org/10.1016/S0304-3975(99)00206-6.

Boyer, Robert S. and J. Strother Moore (Dec. 1979). *A Computational Logic*. Ed. by Thomas A. Standish. Academic Press. ISBN: 9781483277882.

Brown, Chad E. (2012). 'Satallax: An Automatic Higher-Order Prover'. In: *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*. Ed. by Bernhard Gramlich, Dale Miller and Uli Sattler. Vol. 7364. Lecture Notes in Computer Science. Springer, pp. 111–117. DOI: 10.1007/978-3-642-31365-3_11. URL: https://doi.org/10.1007/978-3-642-31365-3%5C_11.

[SW Rel.] Bury, Guillaume, *Archsat* version v1.1, Sept. 2019. LIC: MIT. VCS: https://github.com/Gbury/archsat, SWHID: ⟨swh:1:dir:9b6d26ca7327080671981480ea4e4df0efbca8d4;origin=https://github.com/Gbury/archsat;visit=swh:1:snp:146ff404d9ba1c03d40a711a55bd28201faa8511;anchor=swh:1:rev:c76bad0dd0f1d6393a16648404fbf8a7b0aaca47⟩.

Cardelli, Luca (1984). 'A Semantics of Multiple Inheritance'. In: *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*. Ed. by Gilles Kahn, David B. MacQueen and Gordon D. Plotkin. Vol. 173. Lecture Notes in Computer Science. Springer, pp. 51–67. DOI: 10.1007/3-540-13346-1_2. URL: https://doi.org/10.1007/3-540-13346-1%5C_2.

Cargill, Thomas A. (1991). 'Controversy: The Case Against Multiple Inheritance in C++'. In: *Comput. Syst.* 4.1, pp. 69–82. URL: http://www.usenix.org/publications/compsystems/1991/win%5C_cargill.pdf.

Cauderlier, Raphaël and Catherine Dubois (2014). 'Objects and Subtyping in the Lambda-Pi-Calculus Modulo'. In: *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*. Ed. by Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau. Vol. 39. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 47–71. DOI: 10.4230/LIPIcs.TYPES.2014.47. URL: https://doi.org/10.4230/LIPIcs.TYPES.2014.47.

Chrzaszcz, Jacek (2003). 'Modules in Coq Are and Will Be Correct'. In: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*. Ed. by Stefano Berardi, Mario Coppo and Ferruccio Damiani. Vol. 3085. Lecture Notes in Computer Science. Springer, pp. 130–146. DOI: 10.1007/978-3-540-24849-1_9. URL: https://doi.org/10.1007/978-3-540-24849-1%5C_9.

Church, Alonzo (1940). 'A Formulation of the Simple Theory of Types'. In: *J. Symb. Log.* 5.2, pp. 56–68. DOI: 10.2307/2266170. URL: https://doi.org/10.2307/2266170.

Coen, Claudio Sacerdoti and Enrico Tassi (2009). 'Nonuniform Coercions via Unification Hints'. In: *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009*. Ed. by Tom Hirschowitz. Vol. 53. EPTCS, pp. 16–29. DOI: 10.4204/EPTCS.53.2. URL: https://doi.org/10.4204/EPTCS.53.2.

Collins, Allan M. and M. Ross Quillian (1969). 'Retrieval time from semantic memory'. In: *Journal of Verbal Learning and Verbal Behavior* 8.2, pp. 240–247. ISSN: 0022-5371. DOI: https://doi.org/10.1016/S0022-5371(69)80069-1. URL: https://www.sciencedirect.com/science/article/pii/S0022537169800691.

Constable, Robert L. et al. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice Hall. ISBN: 978-0-13-451832-9. URL: http://dl.acm.org/citation.cfm?id=10510.

Coquand, Thierry and Gérard P. Huet (1988). 'The Calculus of Constructions'. In: *Inf. Comput.* 76.2/3, pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

Courant, Judicaël (1997). 'A Module Calculus for Pure Type Systems'. In: *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*. Ed. by Philippe de Groote. Vol. 1210. Lecture Notes in Computer Science. Springer, pp. 112–128. DOI: 10.1007/3-540-62688-3_32. URL: https://doi.org/10.1007/3-540-62688-3%5C_32.

Cousineau, Denis and Gilles Dowek (2007). 'Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo'. In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, pp. 102–117. DOI: 10.1007/978-3-540-73228-0_9. URL: https://doi.org/10.1007/978-3-540-73228-0%5C_9.

Curien, Pierre-Louis and Roberto Di Cosmo (1996). 'A Confluent Reduction for the lambda-Calculus with Surjective Pairing and Terminal Object'. In: *J. Funct. Program.* 6.2, pp. 299–327. DOI: 10.1017/S0956796800001696. URL: https://doi.org/10.1017/S0956796800001696.

Czajka, Lukasz and Cezary Kaliszyk (2018). 'Hammer for Coq: Automation for Dependent Type Theory'. In: *J. Autom. Reason.* 61.1-4, pp. 423–453. DOI: 10.1007/s10817-018-9458-4. URL: https://doi.org/10.1007/s10817-018-9458-4.

de Bruijn, Nicolaas Govert (1972). 'Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem'. In: *Indegationes Mathematicae (Proceedings)* 75.5, pp. 381–392. ISSN: 1385-7528. DOI: 10.1016/1385-7528(72)90034-0.

— (1991). 'Telescopic Mappings in Typed Lambda Calculus'. In: *Inf. Comput.* 91.2, pp. 189–204. DOI: 10.1016/0890-5401(91)90066-B. URL: https://doi.org/10.1016/0890-5401(91)90066-B.

— (1994). 'Some Extensions of Automath: The AUT-4 Family'. In: *Selected Papers on Automath*. Ed. by R.P. Nederpelt, J.H. Geuvers and R.C. de Vrijer. Vol. 133. Studies in Logic and the Foundations of Mathematics. Elsevier, pp. 283–288. DOI: https://doi.org/10.1016/S0049-237X(08)70209-X. URL: http://www.sciencedirect.com/science/article/pii/S0049237X0870209X.

[SW Rel.] Deducteam, *Dedukti* version 8f68bb15, May 2022. LIC: CECILL-B. VCS: https://github.com/Deducteam/Dedukti, SWHID: ⟨swh:1:dir:3370713ab51947c391e9edc50c3cbdf346aafb1d;origin=https://github.com/Deducteam/Dedukti;visit=swh:1:snp:598eb05f3c67cf121c4252330f6b581c77b76057;anchor=swh:1:rev:c65e7e66fe4bb8285db1856b8fe2a2532da2bd50⟩.

[SW Rel.] Deducteam, *Lambdapi* version 2.2.0, 18th Mar. 2022. LIC: CECILL-B. VCS: https://github.com/Deducteam/lambdapi, SWHID: ⟨swh:1:rel:da8496ba40aff4487947c2c0dc3393394f010165;origin=https://github.com/Deducteam/lambdapi;visit=swh:1:snp:4746d104ac26fd901e807a427806b3ee6b8f5118⟩.

Delahaye, David et al. (2013). 'Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo'. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Ed. by Kenneth L. McMillan, Aart Middeldorp and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer, pp. 274–290. DOI: 10.1007/978-3-642-45221-5_20. URL: https://doi.org/10.1007/978-3-642-45221-5%5C_20.

Descartes, René (June 1637). *Discours de la Méthode*. French. URL: https://www.gutenberg.org/ebooks/13846.

Dowek, Gilles (2017). 'Models and Termination of Proof Reduction in the lambda Pi-Calculus Modulo Theory'. In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis et al. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 109:1–109:14. DOI: 10.4230/LIPIcs.ICALP.2017.109. URL: https://doi.org/10.4230/LIPIcs.ICALP.2017.109.

Dunchev, Cvetan et al. (2015). 'ELPI: Fast, Embeddable, \lambda Prolog Interpreter'. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. Lecture Notes in Computer Science. Springer, pp. 460–468. DOI: 10.1007/978-3-662-48899-7_32. URL: https://doi.org/10.1007/978-3-662-48899-7%5C_32.

Dunfield, Jana and Neel Krishnaswami (2019). 'Bidirectional Typing'. In: *CoRR* abs/1908.05839. arXiv: 1908.05839. URL: http://arxiv.org/abs/1908.05839.

Dutertre, Bruno (2014). 'Yices 2.2'. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_49.

El Haddad, Yacine (Sept. 2021). 'Integrating Automated Theorem Provers in Proof Assistants'. Theses. Université Paris-Saclay. URL: https://tel.archives-ouvertes.fr/tel-03387912.

[SW] Färber, Michael, *Kontroli* 2022. LIC: GPL-3.0-or-later. VCS: https://github.com/01mf02/kontroli-rs, SWHID: ⟨swh:1:dir:252f87ab6290c9ddc0a2a8e1993adc8dd67bc3df;origin=https://github.com/01mf02/kontroli-rs;visit=swh:1:snp:c069cffb811c1cd023bd5de77e7af04bbc48214f;anchor=swh:1:rev:52f4a715bc4b1580b2b59ef40061218cefbec678⟩.

Felicissimo, Thiago (2022). 'Adequate and computational encodings in the logical framework Dedukti'. URL: https://raw.githubusercontent.com/thiagofelicissimo/my-files/master/adequate-and-computational-long.pdf.

Felty, Amy P. et al. (1988). 'Lambda-Prolog: An Extended Logic Programming Language'. In: *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*. Ed. by Ewing L. Lusk and Ross A. Overbeek. Vol. 310. Lecture Notes in Computer Science. Springer, pp. 754–755. DOI: 10.1007/BFb0012882. URL: https://doi.org/10.1007/BFb0012882.

Férey, Gaspard (Nov. 2021). 'Higher-Order Confluence and Universe Embedding in the Logical Framework'. Theses. Université Paris-Saclay. URL: https://tel.archives-ouvertes.fr/tel-03418761.

Férey, Gaspard and Jean-Pierre Jouannaud (Sept. 2021). 'Confluence in Non-Left-Linear Untyped Higher-Order Rewrite Theories'. In: *PPDP 2021 - 23rd International Symposium on Principles and Practice of Declarative Programming*. Tallin, Estonia. DOI: 10.1145/NNNNNNN.NNNNNNN. URL: https://hal.inria.fr/hal-03126115.

Ferreira, Francisco and Brigitte Pientka (2014). 'Bidirectional Elaboration of Dependently Typed Programs'. In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*. Ed. by Olaf Chitil, Andy King and Olivier Danvy. ACM, pp. 161–174. DOI: 10.1145/2643135.2643153. URL: https://doi.org/10.1145/2643135.2643153.

Frege, Gottlob (1879). *Begriffsschrift*. Lubrecht & Cramer, p. 124. ISBN: 978-3487-0062-39.

Futatsugi, Kokichi et al. (1985). 'Principles of OBJ2'. In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*. Ed. by Mary S. Van Deusen, Zvi Galil and Brian K. Reid. ACM Press, pp. 52–66. DOI: 10.1145/318593.318610. URL: https://doi.org/10.1145/318593.318610.

Gilbert, Frédéric (Apr. 2018). 'Extending higher-order logic with predicate subtyping : application to PVS'. Theses. Université Sorbonne Paris Cité. URL: https://tel.archives-ouvertes.fr/tel-02058937.

Gilbert, Gaëtan et al. (Jan. 2019). 'Definitional Proof-Irrelevance without K'. In: *Proc. ACM Program. Lang.* 3.POPL. DOI: 10.1145/3290316. URL: https://doi.org/10.1145/3290316.

Giménez, Eduardo (1994). 'Codifying Guarded Definitions with Recursive Schemes'. In: *Types for Proofs and Programs, International Workshop*

TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers. Ed. by Peter Dybjer, Bengt Nordström and Jan M. Smith. Vol. 996. Lecture Notes in Computer Science. Springer, pp. 39–59. DOI: 10.1007/3-540-60579-7_3. URL: https://doi.org/10.1007/3-540-60579-7%5C_3.

Girard, J. Y. (1971). 'Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types'. In: 63, pp. 63–92.

Guillaume Burel, Ali Assaf and et al. (2016). 'Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory'.

Haddad, Mohamed Yacine El, Guillaume Burel and Frédéric Blanqui (2019). 'EKSTRAKTO A tool to reconstruct Dedukti proofs from TSTP files (extended abstract)'. In: *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*. Ed. by Giselle Reis and Haniel Barbosa. Vol. 301. EPTCS, pp. 27–35. DOI: 10.4204/EPTCS.301.5. URL: https://doi.org/10.4204/EPTCS.301.5.

Harper, Robert, Furio Honsell and Gordon D. Plotkin (1993). 'A Framework for Defining Logics'. In: *J. ACM* 40.1, pp. 143–184. DOI: 10.1145/138027.138060. URL: https://doi.org/10.1145/138027.138060.

Harper, Robert and Daniel R. Licata (2007). 'Mechanizing metatheory in a logical framework'. In: *Journal of Functional Programming* 17, pp. 613–673.

Harper, Robert and John C. Mitchell (1999). 'Parametricity and Variants of Girard's *J* Operator'. In: *Inf. Process. Lett.* 70.1, pp. 1–5. DOI: 10.1016/S0020-0190(99)00036-8. URL: https://doi.org/10.1016/S0020-0190(99)00036-8.

Hoare, C. A. R. (1969). 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[SW Rel.] Hondet, Gabriel, *Personoj* version 0.1, Feb. 2022. LIC: CECILL-B. VCS: https://github.com/Deducteam/personoj, SWHID: ⟨swh:1:rel:294cd408c629b69c53ac4c42ccef4346e6357583;origin=https://github.com/Deducteam/personoj;visit=swh:1:snp:eae7073a2f0316041a1b8962b9f8573bc8fbfd1d⟩.

Hondet, Gabriel and Frédéric Blanqui (2020). 'The New Rewriting Engine of Dedukti'. In: *CoRR* abs/2010.16115. arXiv: 2010.16115. URL: https://arxiv.org/abs/2010.16115.

— (2021). 'Encoding of Predicate Subtyping with Proof Irrelevance in the λΠ-Calculus Modulo Theory'. In: *CoRR* abs/2110.13704. arXiv: 2110.13704. URL: https://arxiv.org/abs/2110.13704.

Hurd, Joe (2001). 'Predicate Subtyping with Predicate Sets'. In: *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*. Ed. by Richard J. Boulton and Paul B. Jackson. Vol. 2152. Lecture Notes in Computer Science. Springer, pp. 265–280. DOI: 10.1007/3-540-44755-5_19. URL: https://doi.org/10.1007/3-540-44755-5%5C_19.

ISO (2018). *ISO/IEC 9899:2018: C17: Programming languages — C*. Tech. rep. International Organization for Standardization. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

Kaufmann, Matt and J. Strother Moore (1997). 'An Industrial Strength Theorem Prover for a Logic Based on Common Lisp'. In: *IEEE Trans. Software Eng.* 23.4, pp. 203–213. DOI: 10.1109/32.588534. URL: https://doi.org/10.1109/32.588534.

Klop, Jan Willem (1980). 'Combinatory reduction systems'. PhD thesis. Univ. Utrecht.

Klop, Jan Willem, Vincent van Oostrom and Femke van Raamsdonk (1993). 'Combinatory Reduction Systems: Introduction and Survey'. In: *Theor. Comput. Sci.* 121.1&2, pp. 279–308. DOI: 10.1016/0304-3975(93)90091-7. URL: https://doi.org/10.1016/0304-3975(93)90091-7.

Knight, John C. and Nancy G. Leveson (1986). 'An Experimental Evaluation of the Assumption of Independence in Multiversion Programming'. In: *IEEE Trans. Software Eng.* 12.1, pp. 96–109. DOI: 10.1109/TSE.1986.6312924. URL: https://doi.org/10.1109/TSE.1986.6312924.

— (1990). 'A reply to the criticisms of the Knight & Leveson experiment'. In: *ACM SIGSOFT Softw. Eng. Notes* 15.1, pp. 24–35. DOI: 10.1145/382294.382710. URL: https://doi.org/10.1145/382294.382710.

Knuth, D. and P. Bendix (1983). 'Simple Word Problems in Universal Algebras'. In.

Lennon-Bertrand, Meven (2021). 'Complete Bidirectional Typing for the Calculus of Inductive Constructions'. In: *CoRR* abs/2102.06513. arXiv: 2102.06513. URL: https://arxiv.org/abs/2102.06513.

[SW Rel.] Lepigre, Rodolphe, *Bindlib* version 6.0.0, 28th Feb. 2022. LIC: LGPL-3.0-only. VCS: https://github.com/rlepigre/ocaml-bindlib, SWHID: ⟨swh:1:rel:fb275298fee0e52fb0c384bd84647de61a0558e7;origin=https://github.com/rlepigre/ocaml-bindlib;visit=swh:1:snp:90a86c4c79b665c68439ad91aad83a5dd1b05152⟩.

Leroy, Xavier et al. (2022). *The OCaml Manual*. Inria. URL: https://ocaml.org/manual/index.html.

Losh, Steve (Aug. 2018). *A Road to Common Lisp*. URL: https://stevelosh.com/blog/2018/08/a-road-to-common-lisp.

Lovas, William and Frank Pfenning (2010). 'Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance'. In: *Log. Methods Comput. Sci.* 6.4. DOI: 10.2168/LMCS-6(4:5)2010. URL: https://doi.org/10.2168/LMCS-6(4:5)2010.

Luo, Zhaohui (1990). 'An extended calculus of constructions'. PhD thesis. University of Edinburgh, UK. URL: http://hdl.handle.net/1842/12487.

Luo, Zhaohui, Sergei Soloviev and Tao Xue (2013). 'Coercive subtyping: Theory and implementation'. In: *Inf. Comput.* 223, pp. 18–42. DOI: 10.1016/j.ic.2012.10.020. URL: https://doi.org/10.1016/j.ic.2012.10.020.

McBride, Conor (Aug. 2018). *Basics of bidirectionalism*. URL: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/.

Miller, Dale (1991). 'A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification'. In: *J. Log. Comput.* 1.4, pp. 497–536. DOI: 10.1093/logcom/1.4.497. URL: https://doi.org/10.1093/logcom/1.4.497.

[SW] Miller, Dale and Gopalan Nadathur, *Teyjus* July 2019. LIC: GPL-3.0-or-later. VCS: https://github.com/teyjus/teyjus, SWHID: ⟨swh:1:dir:4a1885e0a944cb84f267b1737801eb170e53be51⟩.

Milner, Robin (1972). 'Implementation and applications of Scott's logic for computable functions'. In: *Proceedings of ACM Conference on Proving Assertions About Programs, Las Cruces, New Mexico, USA, January 6-7, 1972*. ACM, pp. 1–6. DOI: 10.1145/800235.807067. URL: https://doi.org/10.1145/800235.807067.

Moura, Leonardo Mendonça de et al. (2015). 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26. URL: https://doi.org/10.1007/978-3-319-21401-6%5C_26.

Muñoz, César (1997). *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. Research Report RR-3309. Projet COQ. INRIA. URL: https://hal.inria.fr/inria-00073380.

Norell, Ulf (Sept. 2007). 'Towards a practical programming language based on dependent type theory'. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology.

Owre, Sam, John Rushby et al. (Oct. 1998). 'PVS: An Experience Report'. In: *Applied Formal Methods—FM-Trends 98*. Ed. by Dieter Hutter et al. Vol. 1641. Lecture Notes in Computer Science. Boppard, Germany: Springer-Verlag, pp. 338–345. URL: http://www.csl.sri.com/papers/fmtrends98/.

Owre, Sam, John M. Rushby and Natarajan Shankar (1992). 'PVS: A Prototype Verification System'. In: *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*. Ed. by Deepak Kapur. Vol. 607. Lecture Notes in Computer Science. Springer, pp. 748–752. DOI: 10.1007/3-540-55602-8_217. URL: https://doi.org/10.1007/3-540-55602-8%5C_217.

Owre, Sam and Natarajan Shankar (June 1997a). *Abstract Dtatypes in PVS*. Computer Science Laboratory, SRI International. 333, Ravenswood avenue, Menlo Park, CA 94025.

— (Aug. 1997b). *The Formal Semantics of PVS*. Tech. rep. SRI-CSL-97-2. Menlo Park, CA: Computer Science Laboratory, SRI International.

Owre, Sam, Natarajan Shankar et al. (Aug. 2020). *PVS Language Reference*. Version 7.1. Computer Science Laboratory, SRI International. Menlo Park, CA.

Paulson, Lawrence C. and Jasmin Christian Blanchette (2012). 'Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers'. In: *IWIL 2010. The 8th International Workshop on the Implementation of Logics*. Ed. by Geoff Sutcliffe, Stephan Schulz and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, pp. 1–11. DOI: 10.29007/36dt. URL: https://easychair.org/publications/paper/wV.

Peano, Giuseppe (1889). *Arithmetices principia, nova methodo exposita*.

Pfenning, Frank (2001). 'Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory'. In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, pp. 221–230. DOI: 10.1109/LICS.2001.932499. URL: https://doi.org/10.1109/LICS.2001.932499.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8.

Pierce, Benjamin C. and David N. Turner (2000). 'Local type inference'. In: *ACM Trans. Program. Lang. Syst.* 22.1, pp. 1–44. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

Pol, Jaco van de (2001). 'Just-in-time: On Strategy Annotations'. In: *Electron. Notes Theor. Comput. Sci.* 57, pp. 41–63. DOI: 10.1016/S1571-0661(04)00267-1. URL: https://doi.org/10.1016/S1571-0661(04)00267-1.

Pottinger, Garrel (1981). 'The Church-Rosser theorem for the typed λ-calculus with surjective pairing'. In: *Notre Dame J. Formal Log.* 22.3, pp. 264–268. DOI: 10.1305/ndjfl/1093883461. URL: https://doi.org/10.1305/ndjfl/1093883461.

Robinson, John Alan (1969). 'Mechanizing higher-order logic'. In: Machine Intelligence 4. Ed. by Bernard Meltzer and Donald Michie, pp. 151–172.

Rushby, John M., Sam Owre and Natarajan Shankar (1998). 'Subtypes for Specifications: Predicate Subtyping in PVS'. In: *IEEE Trans. Software Eng.* 24.9, pp. 709–720. DOI: 10.1109/32.713327. URL: https://doi.org/10.1109/32.713327.

Russell, Bertrand (1903). *The Principles of Mathematics*. Cambridge University Press. ISBN: 978-1-313-30597-6.

Sacerdoti Coen, Claudio and Enrico Tassi (Mar. 2011). 'Nonuniform Coercions via Unification Hints'. In: *Electronic Proceedings in Theoretical Computer Science* 53, pp. 16–29. ISSN: 2075-2180. DOI: 10.4204/eptcs.53.2. URL: http://dx.doi.org/10.4204/EPTCS.53.2.

Saillard, Ronan (2015). 'Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique)'. PhD thesis. Mines ParisTech, France. URL: https://tel.archives-ouvertes.fr/tel-01299180.

Saïbi, Amokrane (1999). 'Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories. (Formalization of Mathematics in Type Theory. Generic tools of Modelisation and Demonstration. Application to Category Theory)'. PhD thesis. Pierre and Marie Curie University, Paris, France. URL: https://tel.archives-ouvertes.fr/tel-00523810.

Salvesen, Anne and Jan M. Smith (1988). 'The Strength of the Subset Type in Martin-Löf's Type Theory'. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, pp. 384–391. DOI: 10.1109/LICS.1988.5135. URL: https://doi.org/10.1109/LICS.1988.5135.

Severi, Paula and Erik Poll (1994). 'Pure Type Systems with Definitions'. In: *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*. Ed. by Anil Nerode and Yuri V. Matiyasevich. Vol. 813. Lecture Notes in Computer Science. Springer, pp. 316–328. DOI: 10.1007/3-540-58140-5_30. URL: https://doi.org/10.1007/3-540-58140-5%5C_30.

Shankar, Natarajan et al. (Aug. 2021). *PVS Prover Guide*. Computer Science Laboratory, SRI International. 333, Ravenswood avenue, Menlo Park, CA 94025.

Sozeau, Matthieu (2006). 'Subset Coercions in Coq'. In: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lecture Notes in Computer Science. Springer, pp. 237–252. DOI: 10.1007/978-3-540-74464-1_16. URL: https://doi.org/10.1007/978-3-540-74464-1%5C_16.

[SW] Steen, Alexander, *Leo-III 1.6* version v1.6, Oct. 2021. LIC: BSD-3-Clause. DOI: 10.5281/zenodo.5571355, URL: https://doi.org/10.5281/zenodo.5571355, VCS: https://github.com/leoprover/Leo-III.

Steen, Alexander and Christoph Benzmüller (2018). 'The Higher-Order Prover Leo-III'. In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. Ed. by Didier Galmiche, Stephan Schulz and Roberto Sebastiani. Vol. 10900. Lecture Notes in Computer Science. Springer, pp. 108–116. DOI: 10.1007/978-3-319-94205-6_8. URL: https://doi.org/10.1007/978-3-319-94205-6%5C_8.

Stump, Aaron (2003). 'Subset Types and Partial Functions'. In: *Automated Deduction – CADE-19*. Ed. by Franz Baader. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 151–165. ISBN: 978-3-540-45085-6.

Sutcliffe, G. (2017). 'The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0'. In: *Journal of Automated Reasoning* 59.4, pp. 483–502.

Tannen, Val et al. (1991). 'Inheritance as Implicit Coercion'. In: *Inf. Comput.* 93.1, pp. 172–221. DOI: 10.1016/0890-5401(91)90055-7. URL: https://doi.org/10.1016/0890-5401(91)90055-7.

[SW Rel.] Tassi, Enrico and Claudio Sacerdoti Coen, *ELPI* June 2022. LIC: LGPL-2.1-only. VCS: https://github.com/LPCIC/elpi, SWHID: ⟨swh:1:rel:2f11f4206379f0fbb7fd9d9dda6044959f88a76f;origin=https://github.com/LPCIC/elpi;visit=swh:1:snp:61a86249237e73bdd471d7f3cdaf5df1b3a689bc⟩.

[SW] The Coq Development Team, *The Coq Proof Assistant* version 8.15, Jan. 2022. DOI: 10.5281/zenodo.5846982, URL: https://doi.org/10.5281/zenodo.5846982.

Thiré, F. and G. Férey (2019). *Proof Irrelevance and Predicate Subtyping in Dedukti*. https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Main/bo

oks-of-abstracts-TYPES2019.pdf, p. 106. Abstract of a talk given at the TYPES conference.

Thiré, François (Dec. 2020). 'Interoperability between proof systems using the logical framework Dedukti'. Theses. Université Paris-Saclay. URL: https://hal.archives-ouvertes.fr/tel-03224039.

Werner, Benjamin (2008). 'On the Strength of Proof-irrelevant Type Theories'. In: *Log. Methods Comput. Sci.* 4.3. DOI: 10.2168/LMCS-4(3:13)2008. URL: https://doi.org/10.2168/LMCS-4(3:13)2008.

Whitehead, Alfred North and Bertrand Russell (1997). *Principia Mathematica to *56*. 2nd ed. Cambridge Mathematical Library. Cambridge University Press. DOI: 10.1017/CBO9780511623585.

[SW] X3J13 Committee, *Common Lisp HyperSpec* 1994. URL: http://clhs.lisp.se.

Appendix A

Typing rules of $\mathfrak{S}$

The typing rules of system $\mathfrak{S}$ defined in Chapter 3 (Page 61) are given in Fig. A.1 (Page 168). They are parametrised by

- a signature Σ to declare and type symbols;
- a rewrite system $\mathcal{R}$ used in the congruence (in R-CAST), in the cast relation R-COERCE;
- a coercion system $\mathcal{C}$ used in the cast relation R-COERCE,
- a subtype projection $\prec$ used to apply coercions on the head of applications in R-PROD-C.

Notation $\mathcal{P}^{\lambda\Pi}$ abbreviates the product rules of λΠme,

$$\mathcal{P}^{\lambda\Pi} = \{(\star,\star,\star),(\star,\Box,\Box)\}.$$

R-SORT
$$\frac{}{\Gamma \vdash \star \rightsquigarrow \star : \Box}$$

R-VAR
$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : A}$$

R-ABST
$$\frac{\Gamma \vdash A : \star \rightsquigarrow A' \qquad \Gamma, x : A' \vdash t \rightsquigarrow t' : B}{\Gamma \vdash \lambda x : A, t \rightsquigarrow \lambda x : A', t' : \Pi x : A', B}$$

R-PROD
$$\frac{\Gamma \vdash A : \star \rightsquigarrow A' \qquad \Gamma, x : A' \vdash B \rightsquigarrow B' :_{\mathcal{S}} s \qquad (\star, s, s) \in \mathcal{P}^{\lambda\Pi}}{\Gamma \vdash \Pi x : A, B \rightsquigarrow \Pi x : A', B' : s}$$

R-APPL
$$\frac{\Gamma \vdash t \rightsquigarrow t' :_{\Pi} \Pi x : A_1, A_2 \qquad \Gamma \vdash u : A_1 \rightsquigarrow u'}{\Gamma \vdash (t\, u) \rightsquigarrow (t'\, u') : \{u'/x\}\, A_2}$$

R-SIGN
$$\frac{f[\boldsymbol{x} : \boldsymbol{A}] : B : s \in \Sigma \qquad \left(\Gamma \vdash t_i : \left\{t'_j/x_j\right\}_{j<i} A_i \rightsquigarrow t'_i\right)_i}{\Gamma \vdash (f\, \boldsymbol{t}) \rightsquigarrow (f\, \boldsymbol{t'}) : \{\boldsymbol{t'}/\boldsymbol{x}\}\, B}$$

R-PROD-C
$$\frac{\Gamma \vdash t \rightsquigarrow t' : A \qquad A \prec^* \Pi x : A_1, A_2 \qquad t' : A \lessdot: \Pi x : A_1, A_2 \rightsquigarrow t''}{\Gamma \vdash t \rightsquigarrow t'' :_{\Pi} \Pi x : A_1, A_2}$$

R-SORT-C
$$\frac{\Gamma \vdash t \rightsquigarrow t' : s \qquad s \in \{\star, \Box\}}{\Gamma \vdash t \rightsquigarrow t' :_{\mathcal{S}} s}$$

R-CHECK
$$\frac{\Gamma \vdash t \rightsquigarrow t' : B \qquad t' : B \lessdot: A \rightsquigarrow t''}{\Gamma \vdash t : A \rightsquigarrow t''}$$

R-CAST
$$\frac{A \simeq_{\beta,\mathcal{R}} B}{t : A \lessdot: B \rightsquigarrow t}$$

R-COERCE
$$\frac{(\kappa\, A\, B\, t) \hookrightarrow^{+}_{\beta,\mathcal{R},\mathcal{C}} t' \qquad \kappa \notin t'}{t : A \lessdot: B \rightsquigarrow t'}$$

Figure A.1: Inference rules for system $\mathfrak{S}$.

Appendix B

Encoding of normalisation counter-example

Here follows the encoding of the normalisation counter example of Section 3.2.4 provided in (Abel and Coquand 2020). It is encoded in simple type theory (see Section 2.2.1) with an equality on propositions. The development is written in *Dedukti* version 2.6.

```
Set: Type.
def El : Set -> Type.
arr : Set -> Set -> Set.
[T, U] El (arr T U) --> El T -> El U.

o: Set.
def Prf : El o -> Type.
imp : El (arr o (arr o o)).
[P, Q] Prf (imp P Q) --> Prf P -> Prf Q.
all : (T: Set) -> (El (arr (arr T o) o)).
[T, P] Prf (all T P) --> (x: El T) -> Prf (P x).

eq_o : (El (arr o (arr o o))).

def bot : El o := all o (x => x).
def neg (A: El o) := imp A bot.
def top := neg bot.
```

```
def cast : Prf (all o (A => all o (B => imp (eq_o A B) (imp A B)))).
[A, e, x] cast A A e x --> x.

def delta : Prf top := z => z top z.

def omega : Prf (neg (all o (A => all o (B => eq_o A B)))) :=
  h => A => cast top A (h top A) delta.

def Omega : Prf (neg (all o (A => all o (B => eq_o A B)))) :=
  h => delta (omega h).
```

In the code above, for any term `h` whose type is

```
h: Prf (all o (A => all o (B => eq_o A B)))
```

term `Omega h` does not terminate because of the reduction sequence

```
(Omega h) := delta (omega h) := omega h top (omega h)
  := cast top top (h top top) delta (omega h)
  --> delta (omega h)
```

where `:=` represents definition unfolding and `-->` represents reduction.