UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Thèse

Présentée et soutenue publiquement pour l'obtention du titre de

DOCTEUR DE L'UNIVERSITÉ DE LORRAINE

Mention: Informatique

Automatisation de preuves et synthèse de types pour la théorie des ensembles dans le contexte de TLA$^+$

par Hernán VANZETTO

8 décembre 2014

Membres du jury

Rapporteurs:

M. Sylvain CONCHON	PR	Université Paris-Sud
M. David DELAHAYE	MCF HDR	Conservatoire National des Arts et Métiers

Examinateurs:

M. Jasmin BLANCHETTE	Chercheur	Technische Universität München, Allemagne
M. Kaustuv CHAUDHURI	CR INRIA	INRIA Saclay Île-de-France (co-directeur)
M. Horatiu CIRSTEA	MCF HDR	Université de Lorraine, LORIA
M. Leslie LAMPORT	Chercheur	Microsoft Research, États-Unis
M. Stephan MERZ	DR INRIA	INRIA Grand-Est, LORIA (directeur)
M. David PICHARDIE	PR	École Normale Supérieure de Rennes

Loria Laboratoire lorrain de recherche en informatique et ses applications

Laboratoire Lorrain de Recherche en Informatique et ses Applications

Resumé

Cette thèse présente des techniques efficaces pour déleguer des obligations de preuves TLA$^+$ dans des démonstrateurs automatiques basées sur la logique du premier ordre non-sortée et multi-sortée. TLA$^+$ est un langage formel pour la spécification et vérification des systèmes concurrents et distribués. Sa partie non-temporelle basée sur une variante de la théorie des ensembles Zermelo-Fraenkel permet de définir des structures de données. Le système de preuves TLAPS pour TLA$^+$ est un environnement de preuve interactif dans lequel les utilisateurs peuvent vérifier de manière deductive des propriétés de sûreté sur des spécifications TLA$^+$. TLAPS est un assistant de preuve qui repose sur les utilisateurs pour guider l'effort de preuve, il permet de générer des obligations de preuve puis les transmet aux vérificateurs d'arrière-plan pour atteindre un niveau satisfaisant d'automatisation.

Nous avons développé un nouveau démonstrateur d'arrière-plan qui intègre correctement dans TLAPS des vérificateurs externes automatisés, en particulier, des systèmes ATP et solveurs SMT. Deux principales composantes constituent ainsi la base formelle pour la mise en oeuvre de ce nouveau vérificateur. Le premier est un cadre de traduction générique qui permet de raccorder à TLAPS tout démonstrateur automatisé supportant les formats standards TPTP/FOF ou SMT-LIB/AUFLIA. Afin de coder les expressions d'ordre supérieur, tels que les ensembles par compréhension ou des fonctions totales avec des domaines, la traduction de la logique du premier ordre repose sur des techniques de réécriture couplées à une méthode par abstraction. Les théories sortées telles que l'arithmétique linéaire sont intégrés par injection dans la logique multi-sortée. La deuxième composante est un algorithme pour la synthèse des types dans les formules (non-typées) TLA$^+$. L'algorithme, qui est basé sur la résolution des contraintes, met en oeuvre un système de type avec types élémentaires, similaires à ceux de la logique multi-sortée, et une extension avec des types dépendants et par raffinement. Les informations de type obtenues sont ensuite implicitement exploitées afin d'améliorer la traduction. Cette approche a pu être validé empiriquement permettant de démontrer que les vérificateurs ATP/SMT augmentent de manière significative le développement des preuves dans TLAPS.

Mots-clés: verification formelle, démonstration automatique des théorèmes, théorie des ensembles, systèmes de types.

Proof automation and type synthesis for set theory in the context of TLA$^+$

Abstract

This thesis presents effective techniques for discharging TLA$^+$ proof obligations to automated theorem provers based on unsorted and many-sorted first-order logic. TLA$^+$ is a formal language for specifying and verifying concurrent and distributed systems. Its non-temporal fragment is based on a variant of Zermelo-Fraenkel set theory for specifying the data structures. The TLA$^+$ Proof System TLAPS is an interactive proof environment in which users can deductively verify safety properties of TLA$^+$ specifications. While TLAPS is a proof assistant that relies on users for guiding the proof effort, it generates proof obligations and passes them to backend verifiers to achieve a satisfactory level of automation.

We developed a new back-end prover that soundly integrates into TLAPS external automated provers, specifically, ATP systems and SMT solvers. Two main components provide the formal basis for implementing this new back-end. The first is a generic translation framework that allows to plug to TLAPS any automated prover supporting the standard input formats TPTP/FOF or SMT-LIB/AUFLIA. In order to encode higher-order expressions, such as sets by comprehension or total functions with domains, the translation to first-order logic relies on term-rewriting techniques coupled with an abstraction method. Sorted theories such as linear integer arithmetic are homomorphically embedded into many-sorted logic. The second component is a type synthesis algorithm for (untyped) TLA$^+$ formulas. The algorithm, which is based on constraint solving, implements one type system for elementary types, similar to those of many-sorted logic, and an expansion with dependent and refinement types. The obtained type information is then implicitly exploited to improve the translation. Empirical evaluation validates our approach: the ATP/SMT backend significantly boosts the proof development in TLAPS.

Keywords: formal verification, theorem proving, set theory, type systems

Acknowledgments

Many people deserve my gratitude for making this thesis possible.

I especially want to thank my advisor Stephan Merz. His kindness and technical insights make working with him a real pleasure. Most of the previous work that later resulted in this thesis was made with his collaboration. In addition, he suggested several corrections and commented on drafts of this document.

I want to thank the thesis committee, in particular the reading members Sylvain Conchon and David Delahaye for accepting to review this document. I would like to thank Leslie Lamport for the very useful discussions and his remarks on TLA$^+$, as well as for his comments on the first chapters. Kaustuv Chaudhuri and Jasmin Blanchette also provided comments and corrections.

Many thanks to my colleagues from the VeriDis and Mosel teams at LORIA in Nancy, with whom I had the pleasure to share many good moments, and my colleagues from the TLA$^+$ project at the Microsoft Research-INRIA lab in Saclay, for the technical discussions that we had and their readiness to help.

Special thanks to Christoph Weidenbach for receiving me in his team during the last months of my thesis preparation.

Last but not least, I want to thank my parents, Carlos and Avelina, my sister, and my brothers for their continuous support. I want to deeply thank Ioanna for her love, encouragement and patience.

Contents

List of Figures

Chapter 0.

Résumé etendu

Introduction

TLA$^+$ [Lam02] est un langage de spécification formelle, conçu initialement pour modéliser et analyser des algorithmes et des systèmes concurrents et répartis. L'un de ses objectifs est de décrire et de raisonner sur des systèmes en utilisant des notions mathématiques simples. Les fondements logiques de TLA$^+$ résultent du croisement de la logique temporelle des actions (TLA) [Lam94a], une variante de la logique temporelle linéaire, et d'une variante de la théorie des ensembles standard Zermelo-Fraenkel avec l'axiome du choix (ZFC). Ce dernier langage est accepté par la plupart des mathématiciens comme étant la base pour la formalisation des mathématiques. TLA permet de décrire le comportement dynamique des systèmes, alors que ZFC sert pour définir les structures de données. En tant que langage de spécification, TLA$^+$ fournit une syntaxe concrète pour des constructions de haut niveau, comme les modules, les déclarations de constantes et de variables, et les définitions d'opérateurs aidant à structurer les formules TLA. Un trait saillant de TLA$^+$ est son caractère *non-typé*, au point de ne pas même distinguer les formules (expressions booléennes) des termes (expressions non-booléennes).

Récemment, TLA$^+$ a été étendu par une notation pour écrire des preuves hiérarchiques [CDLM08, CDL$^+$]. L'assistant de preuve dédié à TLA$^+$ nommé TLAPS (TLA$^+$ Proof System) [CDLM08, CDLM10] est un environnement de preuve interactif auquel l'utilisateur peut soumettre une preuve de propriétés de spécifications TLA$^+$. TLAPS peut être utilisé à partir de l'outil TLA$^+$ Toolbox, un environnement intégré de développement pour TLA$^+$ basé sur Eclipse et qui inclut des outils tels que le model-checker TLC [YML99] et un traducteur du langage algorithmique de haut niveau PlusCal [Lam08] vers TLA$^+$. TLAPS est construit autour d'un noyau appelé le Proof Manager, qui génère des obligations de preuve correspondant aux différentes étapes de la preuve fournie par l'utilisateur, puis les transmet à des vérificateurs automatiques.

Avant les travaux décrits dans cette thèse, TLAPS incluait trois démonstrateurs automatiques, ou *backends*, avec des capacités différentes : Zenon [BDD07], un démonstrateur fondé sur la méthode de tableaux pour la logique du premier ordre avec égalité et qui inclut des extensions pour raisonner sur des ensembles et des fonctions de TLA^+ ; Isabelle/TLA^+, un encodage fidèle de TLA^+ dans l'assistant de preuve Isabelle [WPN08], qui fournit des méthodes de preuve automatisées basées sur le raisonnement en logique de premier ordre et sur la réécriture ; et un démonstrateur automatique appelé SimpleArithmetic qui met en oeuvre une procédure de décision pour l'arithmétique de Presburger. Isabelle/TLA^+ et Zenon fournissent un soutien très limité pour raisonner en arithmétique, tandis que SimpleArithmetic ne manipule que des formules arithmétiques pures, obligeant l'utilisateur à décomposer manuellement les preuves jusqu'à ce que les obligations de preuve se situent dans ces fragments respectifs.

Dans ce travail, nous introduisons un nouveau démonstrateur générique pour TLAPS qui est basé sur des systèmes ATP (demonstrateurs de théorèmes automatiques) et des solveurs SMT (Satisfaisabilité Modulo Théories). Les traductions sur lesquelles repose ce démonstrateur font intervenir deux systèmes de types et des algorithmes correspondants pour la construction de types pour des expressions TLA^+. Les systèmes ATP et SMT sont basés respectivement sur la logique classique du premier ordre avec égalité (FOL) et la logique du premier ordre multi-sortée (MS-FOL). Les solveurs SMT ont suscité un intérêt particulier pendant la dernière décennie parce qu'ils combinent un raisonnement en logique du premier ordre sans quantificateurs, des mécanismes d'instantiation de quantificateurs et des procedures de décision pour des théories fondamentales pour la vérification, telles que des fragments décidables de l'arithmétique, des tableaux, ou des vecteurs de bits. Bien que beaucoup de nos études de cas concernent des systèmes repartis, nous considérons des systèmes critiques au sens large.

Motivation TLA^+ s'appuie fortement sur la modélisation de données en utilisant des ensembles et des fonctions. Les n-uplets et les records, qui apparaissent très souvent dans des spécifications TLA^+, sont définis à partir des fonctions. Des expressions dans lesquelles interviennent à la fois des opérateurs de la logique du premier ordre, des constructions ensemblistes, des fonctions et des opérateurs arithmétiques surviennent fréquemment dans les preuves des propriétés de sûreté des spécifications TLA^+. La logique du premier ordre offre un compromis entre expressivité et raisonnement automatisé efficace. Nous nous concentrons donc sur deux classes différentes de démonstrateurs automatiques adaptés à ce langage : les systèmes ATP et les solveurs SMT. Beaucoup d'obligations de preuve faisant un usage intensif de fonctions non interprétées et de formules quantifiées peuvent être traitées par les systèmes ATP. Les obligations de preuve incluant des expressions arithmétiques peuvent souvent être traitées par des solveurs SMT.

Défis L'objectif principal de cette thèse est d'améliorer de manière sure et efficace les capacités d'automatisation du système de preuves de TLA$^+$. Nous nous concentrons uniquement sur le fragment non-temporel du langage, ce qui est suffisant pour démontrer les propriétés de sûreté, y compris les invariants inductifs et les conditions de raffinement [AL91]. En de termes pratiques, moins de 5% d'une spécification TLA$^+$ typique contient des expressions temporelles, donc la partie non-temporelle d'une spécification représente la majeure partie de l'effort de vérification, et la vérification de propriétés de sûreté ne nécessite qu'un raisonnement trivial en logique temporelle. Pour traiter ces cas, TLAPS inclut depuis très récemment une procédure de décision pour la logique temporelle propositionnelle [DKL$^+$14].

Traduire les expressions TLA$^+$ en des expressions de la logique du premier ordre représente le premier défi de l'intégration auquel nous nous sommes confrontés. Même si quelques-unes des techniques d'encodage utilisées peuvent être trouvées dans des outils similaires pour d'autres langages de spécification, les particularités de TLA$^+$ rendent cette traduction non-triviale :

- La syntaxe de TLA$^+$ permet d'écrire des expressions absurdes : par exemple, $3 \cup \text{TRUE}$ dénote une valeur, mais cette valeur n'est pas spécifiée. En outre, il n'y a pas de distinction syntaxique entre les termes et les formules.
- La théorie des ensembles de Zermelo-Fraenkel ne permet pas d'axiomatisation finie en logique du premier ordre, contrairement à d'autres formalisations de la théorie des ensembles telles que von Neumann-Bernays-Gödel (NBG). Plus précisément, les axiomes de compréhension et de remplacement de ZF permettent l'introduction de nouveaux ensembles à travers des schémas d'axiomes, c'est-à-dire, d'axiomes paramétrés par un prédicat. Par conséquent, ces objets ne peuvent pas être codés directement en logique du premier ordre.
- Les fonctions, qui sont définies de manière axiomatique, sont totales sur leur domaine. Cela signifie qu'une fonction appliquée à un élément de son domaine renvoie la valeur attendue mais, pour tout autre argument, le résultat de l'application de la fonction renvoie une valeur non spécifiée. De la même manière, le comportement des opérateurs arithmétiques n'est pas spécifié en dehors des arguments numériques.
- Le langage TLA$^+$ comporte un opérateur primitif de choix déterministe, correspondant à l'opérateur de Hilbert ε, qui est difficile à encoder dans un cadre classique du premier ordre.

Peut-être plus éprouvant dans la définition de la traduction est le fait que TLA$^+$ est un langage non-typé. Cette caractéristique résulte en un langage très expressif et flexible pour l'écriture des spécifications ; par contre, il rend assez difficile l'automatisation du raisonnement. Ne pas avoir de types fait partie de l'idiosyncrasie de TLA$^+$. La justification de cette décision prise lors de la conception du langage se trouve dans un article de Lamport et Paulson [LP99]. Essentiellement, ils affirment qu'une discipline de types dans un langage de spécification de haut niveau limite le

pouvoir expressif du langage. En outre, les types compliquent la tâche des utilisateurs lors de la rédaction des specifications. TLA$^+$ est orienté vers l'utilisateur, dans le sens où le langage se concentre à fournir aux utilisateurs un compromis entre un langage de spécification simple mais très expressif. Une discipline de types peut conduire à une détection précoce des erreurs de syntaxe, mais plus important pour notre propos, elle est bénéfique aux capacités déductives des outils de raisonnement. Si elle peut être évitée, la tâche de traiter les complications de l'absence d'un système de type doit reposer sur les développeurs des outils, au lieu de forcer les utilisateurs à adapter leurs spécifications au fardeau des types. Puisque les langages d'entrée des solveurs SMT requièrent l'identification de sortes, un deuxième défi consiste à affecter automatiquement des types aux sous-expressions d'une obligation de preuve TLA$^+$. Nous utiliserons indistinctement les termes *type* et *sorte*.

Integration de démonstrateurs automatiques de premier ordre dans TLAPS

L'intégration de concepts de la théorie des ensembles dans la logique du premier ordre est souvent contre nature. Pour encoder des expressions du second ordre, notamment liées aux axiomes de compréhension et de remplacement, la première partie de la traduction consiste en une phase de pré-traitement et d'optimisation qui fait appel à des techniques de réécriture de termes, couplées avec une méthode d'abstraction pour gérer efficacement des expressions non-basiques. Par exemple, l'axiome définissant l'union des ensembles conduit à la règle de réécriture

$$x \in e_1 \cup e_2 \longrightarrow x \in e_1 \vee x \in e_2.$$

De façon similaire, la règle

$$S = \{a, b\} \longrightarrow \forall x : x \in S \Leftrightarrow x = a \vee x = b$$

est une conséquence de l'axiome d'extensionalité, appliqué à un type particulier d'ensemble. Nous décrivons une collection de telles règles de réécriture ; afin de nous assurer de leur correction vis à vis de la sémantique de TLA$^+$, les équivalences sous-jacentes à chacune de ces règles ont été démontrées en utilisant Isabelle/TLA$^+$. Les règles de réécriture, cependant, présentent de nombreuses formules quantifiées supplémentaires, qui sont difficiles à manipuler par les solveurs.

Le premier pas de la traduction consiste en un processus appelé Boolification qui rend possible la distinction entre des expressions booléennes et non-booléennes. Pour les expressions non-basiques, c'est-à-dire, les expressions qui ne peuvent pas être réduites à la logique du premier ordre à travers la réécriture, la méthode d'abstraction les remplace par un nouveau symbole, puis elle ajoute une nouvelle hypothèse définissant le symbole. En conséquence, cette méthode produit des formules

équisatisfiables mais pas nécessairement équivalentes à la formule d'origine. Par example, considérons l'obligation de preuve

$$\forall x \colon P(\{x\}) \Leftrightarrow P(\{x\} \cup \{x\})$$

qui est valide. Les sous-expressions non-basiques $\{x\}$ et $\{x\} \cup \{x\}$ sont remplacées par de nouveaux constantes $k_1(x)$ et $k_2(x)$. Après cette étape d'abstraction, la formule devient alors :

$$\begin{array}{l} \wedge\ \forall y_1 \colon k_1(y_1) = \{y_1\} \\ \wedge\ \forall y_2 \colon k_2(y_2) = \{y_2\} \cup \{y_2\} \\ \Rightarrow \forall x \colon P(k_1(x)) \Leftrightarrow P(k_2(x)). \end{array}$$

Il est désormais possible d'appliquer les règles de réécriture correspondant à l'extensionalité qui élargissent les égalités dans les définitions des expressions non-basiques. Le mécanisme qui combine la réécriture de termes avec l'abstraction permet également aux backends de traiter avec succès des expressions CHOOSE (opérateur de choix de Hilbert) et des expressions TLA$^+$ de construction de fonctions, correspondant aux λ-abstractions. Dans notre traduction nous traitons les fonctions comme si elles étaient des fonctions partielles, encodant les applications à des arguments dans le domaine de la fonction par une fonction α et les autres par une fonction ω. Ainsi, $f[x]$ est traduit comme $\alpha(f, x)$ quand $x \in \text{DOMAIN } f$, et comme $\omega(f, x)$ sinon.

La deuxième partie de la traduction est un encodage bien-fondé des formules TLA$^+$ appartenant au fragment du premier ordre dans les logiques du premier ordre non-sortée et multi-sortée. ce stade, les obligations de preuve ont déjà été transformées en des formules du premier ordre dans la phase précédente. Lorsque le langage cible est non-sorté, nous encodons les opérateurs booléens et non-booléens respectivement comme des fonctions ou des prédicats, tout en respectant leurs arités. Par exemple, les opérateurs binaires comme $\cup$ et $\in$ sont représentés dans le langage cible par la fonction union et le prédicat in, tous les deux d'arité 2. De la même manière, lors de l'encodage dans un langage sorté, nous représentons l'univers des valeurs de TLA$^+$ avec la sorte prédéfinie Bool pour les propositions, et nous déclarons une nouvelle sorte U pour les autres expressions. Par conséquent, les opérateurs de TLA$^+$ correspondent à des fonctions ou prédicats non interprétés qui prennent des arguments de type U.

Pour résoudre le problème de l'intégration d'un langage non-typé tel que TLA$^+$ dans un langage typé, nous avons mis en place une méthode qui revient à déléguer l'inférence de types aux solveurs. En ce qui concerne les expressions appartenant à des sortes interprétées par le langage cible (comme les expressions arithmétiques), la traduction repose sur un plongement homomorphique : la sorte interprétée est plongée dans l'univers TLA$^+$ par une fonction injective, et les opérateurs interprétés s'étendent de manière homomorphique sur l'image de ce plongement. Par example, les opérateurs arithmétiques de TLA$^+$ sont defines en utilisant la function injective

$\mathsf{int2u} : \mathsf{Int} \to \mathsf{U}$. L'addition est encodée par la fonction $\mathsf{plus} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$ dont le comportement sur les entiers est décrit par l'axiome

$$\forall m^{\mathsf{Int}}, n^{\mathsf{Int}}.\ \mathsf{plus}(\mathsf{int2u}(m), \mathsf{int2u}(n)) = \mathsf{int2u}(m + n),$$

où le symbole $+$ à droite représente l'addition primitive des solveurs SMT sur les entiers.

La correction de l'encodage est immédiate : tous les axiomes sur les ensembles, les fonctions, les records, les n-uplets etc. sont des théorèmes dans la théorie de TLA$^+$ de fond qui existe dans l'encodage en Isabelle. La correction de l'encodage des opérateurs arithmétiques par plongement repose sur le fait que l'arithmétique de TLA$^+$ coïncide avec celle des solveurs SMT sur les entiers. En faisant abstraction des instances de l'axiome d'extensionalité, la traduction est complète dans le sens que la traduction d'une obligation de preuve valide en logique de TLA$^+$ est une formule valide en logique du premier ordre. Par défaut, la traduction n'inclut des instances d'extensionalité que pour des ensembles spécifiques, des domaines fonctionnels et des fonctions. Pour raisonner sur l'égalité dans sa forme la plus générale, l'utilisateur devra explicitement ajouter l'axiome de extensionalité à la preuve. Nous avons démontré que les étapes intermédiaires de pré-traitement et l'encodage de formules basiques sont correctes et que leur mise en œuvre termine.

Notre traduction gère un fragment utile de TLA$^+$, y compris la théorie des ensembles, des fonctions, des expressions arithmétiques, n-uplets, records, et l'opérateur du choix. Certaines restrictions existent concernant les expressions bien formées pris en charge par la traduction. Par exemple, après l'étape de Boolification, la traduction rejettera des expressions absurdes comme $x \wedge 42$. Un ensemble comme $\{x, y\}$ peut être traduit lorsque ses composants ont la même sorte. De même, le co-domaine des fonctions, y compris les n-uplets et les records, doit être d'une sorte précise dans le langage cible. Par exemple, une fonction comme

$$[x \in \mathit{Int} \mapsto \textsc{if}\ c\ \textsc{then}\ x\ \textsc{else}\ \textsc{false}]$$

sera rejetée.

Synthèse de types pour TLA$^+$

Au-delà des débats récurrents sur l'utilisation ou non des types pour formaliser les mathématiques ou pour spécifier les systèmes logiciels [LP99], nous observons que les types, considérés simplement comme une classification des éléments d'un langage, se posent tout naturellement dans la théorie des ensembles. Motivés par l'intégration des démonstrateurs automatiques basées sur la logique multi-sortée,

nous avons défini deux systèmes de types pour TLA$^+$. Un des avantages de l'approche SMT pour la démonstration automatique est que la logique MS-FOL partitionne l'univers en types, à la différence de la logique non-typée FOL. Les systèmes de types pour TLA$^+$ visent à synthétiser l'information de type à partir de formules non-typées pour mieux classifier le domaine de discours. Pour une obligation de preuve donnée, des types sont synthétisés à partir de ce que nous appelons des *hypothèses de type*, qui proviennent souvent d'invariants de typage, naturellement utilisés en TLA$^+$. Lorsque la synthèse de types réussit, on obtient des annotations de types au dessus d'un langage de spécification non-typé, pour obtenir le meilleur des deux approches.

Considérons la traduction en SMT-LIB de la formule TLA$^+$

$$\forall x \colon x \in \mathit{Int} \Rightarrow x + 0 = x.$$

Comme décrit précédemment, la traduction plonge le terme 0 dans le type universel U comme $\mathsf{int2u}(0)$ et encode l'addition en utilisant une fonction axiomatisée. Si nous pouvions détecter l'information de type appropriée à partir de la formule TLA$^+$ originale, spécifiquement si la variable quantifiée x était de type entier, nous pourrions traduire cette formule en SMT-LIB comme

$$\forall x^{\mathsf{Int}}.\, \mathsf{in}(\mathsf{int2u}(x), \mathsf{tla\_Int}) \Rightarrow x + 0 = x,$$

en utilisant l'opérateur SMT intégré $+$ au lieu de l'opérateur axiomatisé $\mathsf{plus} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$. Nous pourrons faire encore mieux et appliquer des règles de réécriture conditionnelles basées sur les informations de type. Si la variable x est connue pour être de type entier, noté $x \colon \mathsf{Int}$, le processus de réécriture déclenchera l'application des règles

$$x \in \mathit{Int} \xrightarrow{x:\mathsf{Int}} \textsc{true} \quad \text{et} \quad x + 0 \xrightarrow{x:\mathsf{Int}} x.$$

La formule résultante après le pré-traitement sera simplement TRUE.

Notre premier système de types pour TLA$^+$, appelé $\mathcal{T}_1$, inclut des types élémentaires, tels que les entiers, les booléens et les types fonctionnels, qui reflètent de façon naturelle les sortes de MS-FOL, et un constructeur des types Set qui permet la stratification des ensembles. Ce système de type est assez limité et ne peut, dans certains cas, pas exprimer des informations adéquates de type. Il donne lieu à un problème décidable de construction de types, au prix d'une sur-approximation des types des expressions TLA$^+$. Ainsi, des contrôles supplémentaires doivent être ajoutés à l'encodage, par exemple pour vérifier que l'argument d'une fonction appartient bien au domaine de la fonction.

Prenons comme exemple la formule (non valide en TLA$^+$)

$$f = [x \in \{1,2,3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1$$

où $[x \in \{1,2,3\} \mapsto x * x]$ est la fonction avec domaine $\{1,2,3\}$ et qui renvoie le carré de son argument. Le système $\mathcal{T}_1$ représente le type de f comme une fonction de Int

vers Int. Même si l'argument 0 n'est pas dans le domaine de f, le type Int sera attribué à l'expression $f[0]$. La correction de la traduction est maintenue grâce à l'ajout de la *condition de domaine* $0 \in \text{DOMAIN } f$ à la formule générée par la traduction, et cette condition est fausse dans notre exemple. La construction des types nous permet d'attribuer un type à toute sous-expression qui apparaît dans l'obligation de preuve. La traduction non-typée encode toute expression de la forme $f[x]$ par l'expression conditionnelle

$$\text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f, x) \text{ ELSE } \omega(f, x)$$

ce qui oblige le solveur à considérer les deux cas possibles, et de raisonner sur les opérateurs supplémentaires α et ω. Le calcul du domaine d'une fonction TLA$^+$ encodé dans une logique du premier ordre n'est pas toujours facile, et peut faire en sorte que les démonstrateurs échouent à démontrer les conditions correspondantes. Par ailleurs, la conception d'un système de type approprié est compliquée par le fait que certaines formules, comme $f[x] \cup \{\} = f[x]$, sont en fait valides indépendamment du fait que la condition de domaine soit vraie ou non.

Les observations ci-dessus nous incitent à utiliser un système de type plus expressif. Nous introduisons un deuxième système de type appelé $\mathcal{T}_2$ qui étend $\mathcal{T}_1$ et est basé sur des types dépendants et par raffinement [FP91, XP99]. En utilisant les types par raffinement, le type de l'expression DOMAIN f dans l'exemple ci-dessus est $\{x : \mathsf{Int} \mid x = 1 \vee x = 2 \vee x = 3\}$, ce type représentant la collection des éléments x avec le type de base Int qui satisfont au prédicat de raffinement $x = 1 \vee x = 2 \vee x = 3$. Pendant la construction des types, le système tentera de démontrer $x = 0 \Rightarrow x \in \text{DOMAIN } f$, et cette preuve va échouer. Par conséquent, la traduction va retomber à l'encodage non-typé (qui à son tour ne prouvera pas la formule, comme il se doit). Dans de nombreux exemples concrets, la condition de domaine $x \in \text{DOMAIN } f$ peut être établi au cours de la construction des types, conduisant à des obligations de preuve plus courtes et plus simples.

Cependant, le problème de synthèse de type pour ce système est indécidable. Les formules pour lesquels l'inférence de type échoue sont toujours converties selon l'encodage "non typé", et peuvent donc être démontrées par les solveurs du premier ordre. Puisque TLA$^+$ est basée sur la théorie des ensembles de Zermelo-Fraenkel, nous pensons que cette approche peut s'appliquer plus largement à la demonstration de théorèmes dans d'autres langages basés sur la théorie des ensembles. Un système de types avec des types par raffinement est très expressif et en fait assez proche de la théorie des ensembles elle-même, donnant lieu à des obligations de preuve qui sont indécidables. Plus précisément, l'égalité entre les deux types $\{x : \tau \mid \phi_1\}$ et $\{x : \tau \mid \phi_2\}$ est réduit à démontrer $\phi_1 \Leftrightarrow \phi_2$, et le sous-typage entre ces types conduit à démontrer $\phi_1 \Rightarrow \phi_2$, toujours dans un contexte où x est de type τ.

Introduire des types dans la théorie des ensembles représente le passage d'un paradigme à l'autre, dans le sens où, inévitablement, les systèmes de types restreignent le fragment d'expressions accepté. Néanmoins, nous pouvons profiter de cela et utiliser les systèmes de type pour la détection : (i) des expressions mal formées ou in-

cohérentes, c'est-à-dire, des expressions qui ne possèdent pas de valeur sémantique définie, comme par exemple $3 + \text{TRUE}$, (ii) des expressions parfois utiles qui sont sémantiquement correctes mais ne peuvent être traduites dans le langage cible, comme par exemple $\{3, \text{TRUE}\}$, et (iii) des expressions dont la valeur ne correspond probablement pas à celle prévue par l'utilisateur, comme par exemple $f[x]$ lorsque $x \notin \text{DOMAIN } f$ (seulement dans $\mathcal{T}_2$).

Evaluation empirique

Nous pensons que notre objectif initial d'améliorer les capacités d'automatisation de TLAPS a été réalisé avec succès. Des résultats encourageants montrent que les systèmes ATP et les solveurs SMT améliorent significativement les performances du démonstrateur interactif TLAPS pour la vérification des obligations de preuve "peu profondes", ainsi que pour des formules plus complexes, y compris, par exemple, des expressions arithmétiques linéaires dans le cas des solveurs SMT. La taille des preuves interactives, ainsi que le temps nécessaire pour trouver des preuves automatiques peuvent être réduits sensiblement en utilisant le nouveau démonstrateur. Nous considérons que la réduction de la taille des preuves est l'élément le plus important, car il reflète le nombre d'interactions effectuées par l'utilisateur. Par ailleurs, les deux classes de démonstrateurs automatiques considérés, c'est à dire ATP et SMT, s'avèrent être complémentaires pour différentes obligations de preuve.

Notre expérience avec la mise en oeuvre de l'algorithme de construction des types adaptés aux backends ATP/SMT a été très positive : les types sont déduits avec succès pour la grande majorité des obligations de preuve que nous avons vu en pratique. Le système de type décidable $\mathcal{T}_1$ ne peut qu'être avantageux par rapport à l'encodage non typé, hormis un coût très réduit nécessaire à la construction des types. Lorsque la synthèse de types est applicable, la traduction génère des formules beaucoup plus simples que l'encodage non typé et introduit notamment beaucoup moins de quantificateurs. Ceci conduit à une augmentation significative du nombre des obligations de preuve que les deux backends basés sur ATP et SMT peuvent gérer sans intervention humaine. Puisque le système de type $\mathcal{T}_2$ est un raffinement de $\mathcal{T}_1$, $\mathcal{T}_2$ réussit pour toutes les obligations pour lesquelles $\mathcal{T}_1$ réussit tout en simplifiant davantage la taille des formules générées, ce qui augmente encore plus le nombre d'obligations de preuve qui peuvent être traitées. Les améliorations introduites par $\mathcal{T}_2$ sont particulièrement appréciables pour les spécifications qui contiennent un nombre important d'applications de fonctions. Ces expressions se produisent fréquemment dans des obligations de preuves générées à partir de spécifications TLA^+ étant donné que les fonctions, les n-uplets et les records sont des structures de données fondamentales.

Contributions

Cette thèse présente des techniques correctes et efficaces pour traduire des obligations de preuve TLA$^+$ en langages d'entrée de démonstrateurs automatiques basées sur la logique du premier ordre non-sortée et multi-sortée. Un nouveau backend ATP/SMT intègre des démonstrateurs automatiques externes à TLAPS.

La première contribution de cette thèse est une proposition pour l'intégration de systèmes ATP et de solveurs SMT dans des preuves TLA$^+$, mise en œuvre au sein de TLAPS. Les démonstrateurs externes sont intégrés en tant qu'*oracles*, ce qui signifie que nous les considérons dignes de confiance. L'intégration comporte deux volets principaux. Le premier est une traduction correcte d'un fragment important de formules TLA$^+$ en des logiques du premier ordre non-sortée et multi-sortée. La théorie des ensembles de TLA$^+$ est codée en utilisant une sorte unique (permettant en particulier de représenter des ensembles d'ensembles), et les théories primitives des solveurs, comme l'arithmétique, sont plongées homorphiquement dans cet univers. Le deuxième volet de l'intégration englobe des techniques de pré-traitement et d'optimisation pour rendre la traduction efficace et réalisable pour toutes les constructions de TLA$^+$, y compris l'opérateur CHOOSE. Parmi ces techniques figure un moteur de réécriture de termes confluent qui transforme des expressions TLA$^+$ en une forme normale, couplé avec une méthode d'abstraction qui introduit de nouveaux symboles pour représenter des expressions TLA$^+$ complexes pour lesquelles la traduction de base ne s'applique pas directement.

La deuxième contribution principale est la définition formelle de deux systèmes de types pour la théorie des ensembles non-typée, couvrant toutes les expressions de TLA$^+$ pertinentes. Dans cette approche, un algorithme de synthèse de types basé sur des contraintes affecte des types aux expressions TLA$^+$. Cette synthèse de types reste invisible pour l'utilisateur de TLA$^+$, les types n'étant utilisés qu'en interne pour simplifier les formules générées par la traduction. Dans une première approche à la traduction que nous appelons "non typée", nous utilisons une sorte unique pour représenter toutes les expressions de TLA$^+$, avec un plongement des théories primitives du solveur. De cette façon, l'inférence de type est essentiellement déléguée aux solveurs par l'encodage.

Le système de type $\mathcal{T}_1$ pour TLA$^+$ repose sur des types élémentaires. Ce système de type est simple et donne lieu à un algorithme efficace pour la construction de types, mais il renvoie des approximations des types des expressions TLA$^+$. Le deuxième système $\mathcal{T}_2$ étend le système précédent avec des types dépendants et par raffinement, ce qui saisit très précisément les valeurs représentées par les termes de TLA$^+$, au prix de l'indécidabilité de la construction des types. Alors que l'encodage non typé peut être utilisé pour traiter les expressions non-typables, les types précis permettent opportunément de simplifier les traductions. En particulier, les conditions de domaine pour le système de type $\mathcal{T}_1$ deviennent superflues pour $\mathcal{T}_2$. Nous avons démontré la correction des systèmes de type dans le sens où les annotations de types

ne changent pas la validité des expressions originales TLA$^+$. Ces techniques pour la construction de types pour des langages non typés peuvent être d'intérêt au-delà de TLA$^+$.

Les résultats précédents fournissent la base formelle pour la mise en oeuvre d'un nouveau backend générique pour TLAPS basé sur des démonstrateurs ATP et SMT. Tout démonstrateur externe qui accepte les langages d'entrée standard des systèmes ATP et des solveurs SMT, c'est-à-dire, les formats TPTP/FOF et SMT-LIB (plus spécifiquement la logique AUFLIA), peut être branché à TLAPS. Nous fournissons des preuves expérimentales montrant que notre approche est réalisable et utile en pratique.

Plusieurs des contributions décrites ici ont été présentées lors de conférences et de workshops internationaux. Ce mémoire de thèse consolide et améliore les idées et les résultats préliminaires présentés dans les publications [MV12a], [MV12b] et [MV14].

Chapter 1.

Introduction

Contents

Distributed algorithms lie at the heart of modern operating systems, network protocols, real-time controllers, and other safety-critical systems [Lyn96, FM03]. Their task is highly complex because they are autonomously executed without a centralized control, interacting one with the other, and sometimes also with the environment, sharing resources such as memory or services over a network. Concurrency seems to be unavoidable in distributed systems, introducing non-determinism and problems such as race conditions between the interleaving processes. An exponential number of execution scenarios is thus plausible, making concurrent systems particularly difficult to reason about, even for experienced users. For all these reasons, traditional validation methods of software quality assessment, such as testing and simulation, are not sufficient and they should be complemented with formal verification techniques.

The formal verification approach essentially amounts to abstracting the system into a precise, unambiguous specification, then finding the properties that the system must meet to be correct, and finally proving these properties against the specification with mathematical certainty [WLBF09]. Just by using a formal language to write down their specifications, software developers can gain a lot of confidence in their systems, even before starting to write a proof. But formal specifications allow us to go further and mechanize the verification process.

Since the seminal works of Floyd [Flo67] and Hoare [Hoa69] we know how to formally reason about programs as *state-transition* systems. Ashcroft [Ash75] extended

state-based reasoning to concurrent programs through the fundamental concept of invariance, which considers states from a global point of view, instead of locally as in sequential programs. An *invariant* is a property that holds true in all reachable states of a system [Lam77]. With the introduction of *temporal logic* to computer science by Pnueli [Pnu77], reasoning about the properties of concurrent algorithms found a strong basis.

Temporal logic allows us to express two kinds of properties of concurrent systems: *safety properties* express what the system is allowed to do, while *liveness properties* state that something must occur within the system [Lam77, Lam83]. Examples of safety properties are: partial correctness (the program does not produce a wrong answer), mutual exclusion (two processes are not in their critical section at the same time), and dead-lock freedom (the program does not reach a deadlock state). Examples of liveness properties are: termination (the program does eventually terminate) and starvation freedom (a process eventually receives service).

That said, important distributed safety-critical algorithms are still presented as pseudo-code or through semi-formal specifications, and their properties are proved by hand. For example, consider the case of Chord [SMK+01, SMLN+03], a well-known distributed protocol that maintains a decentralized hash-table over a peer-to-peer network. Introduced in 2001, this algorithm is significant because it is frequently used as a foundation for research on distributed hash-tables and it is widespread implemented in many applications.[1] Many different versions of the algorithm exist, each presented informally, their invariants having inconsistencies and ambiguities, and their correctness proofs having undefined terms and unstated assumptions [Zav12].

Unfortunately, no published version of Chord is correct, as shown by Zave [Zav12] in 2012. By compiling from various sources a definitive version of the algorithm, and applying to it a "lightweight" verification approach, Zave found a number of subtle errors in all of the algorithm's published alternatives, either in the pseudo-code or in the invariants. The approach to formal methods called *lightweight modeling* [Jac12] consists in constructing a small abstract model of the key parts of the system, and then analyzing the model with a fully automated tool, such as a model-checker. In this case, the model was written in Alloy and the verification tool was the Alloy Analyzer [Jac12].

The *model-checking* approach [DKW08, GV08, Hol91] to formal verification is a standard technique for state-based transition systems that performs exhaustive state-space exploration on a finite abstraction of the system. Given a bounded model of a system and a property one wants to verify, the model-checker analyzes all reachable states to verify that the property is valid in all of them. In case the property

[1] The paper that introduced Chord [SMK+01], with over 3.000 citations, is the fourth most cited paper in Computer Science (according to Citeseer), and it won SIGCOMM's 2011 Test-of-Time award.

is not satisfied, it can report the trace of states that led to the property violation, which provides valuable information to make corrections to the specification. This method has been very successful in many research and industrial applications of hardware and software systems [GV08]. One of the main reasons is in part due to its one-click invocation feature: the user just needs to point the property he wants to verify, invoke the model-checker and wait for the results. There are two methods to represent states. Explicit-state model-checkers enumerate states, and use graph algorithms to explore the states. Symbolic model-checkers drastically reduce the state space by implicitly representing set of states, originally analyzed using binary decision diagrams [BCM+92], later using SAT solvers [BCCZ99].

TLA$^+$ and the limitations of model-checking

In this work, our working platform for applying formal methods is TLA$^+$ [Lam02], a general-purpose formal specification language. It was designed by Leslie Lamport during the 1990s, originally to formally analyze concurrent and distributed algorithms and systems. One of the language's design goals is to describe and reason about systems by using simple mathematics. The logical foundation of TLA$^+$ is a combination of the Temporal Logic of Actions (TLA) [Lam94a], which is a variant of linear temporal logic, and a variant of standard Zermelo-Fraenkel set theory with the axiom of choice (ZFC), the language accepted by most mathematicians as the standard basis for formalizing mathematics. The first one allows the description of the dynamic system behavior, and the latter is used for specifying data structures. As a full-fledged specification language, TLA$^+$ provides a concrete syntax supporting high-level constructs, like modules, declaration of constants and variables, and operator definitions that help to give structure to the TLA formulas. A salient characteristic of TLA$^+$ is that it is *untyped*, to the point of not even distinguishing Boolean from non-Boolean expressions.

TLC [YML99] is an explicit-state model checker for TLA$^+$ that has been successfully applied to the verification of software and hardware systems in many industrial projects. For example we can mention: cache-coherency protocols for microprocessors at Compaq [LMTY02] and Intel [TYBK02, BL03], and fault-tolerant distributed algorithms at Amazon [New14], among others. However, the model-checking approach has some limitations, and TLA$^+$ users require proof-based verification tools [New14].[2]

Model-checking requires much less effort and less expertise in the verification domain from the user than writing a fully formal proof of correctness. Nevertheless, the combinatorial "explosion" of states makes this approach not scalable for many

[2] Newcombe [New14], regarding TLC: "Even when using such constraints [to bound the state-space] the model-checker still has to explore billions of states. We doubt that model-checking can go much further than this. Therefore, for our most critical algorithms we wish to also use proofs."

industrial-strength specifications. Users need to construct suitable abstractions of the system, usually resulting in small models, for which finite instances can effectively be verified.

Regarding the Chord algorithm, Zave provides in a subsequent paper [Zav14] a formal specification of a correct version of the algorithm, together with a proof of its correctness. However, because the formal analysis is based on model-checking, the proof is restricted to systems with up to 8 nodes in the network and considers that nodes can maintain a routing table of up to two nodes. The generalized proof was completed only informally,[3] falling short of providing complete confidence in its correctness.

Another representative example is the non-blocking multi-threaded algorithm by Detlefs et al. [DFG$^+$00] named Snark. This algorithm implements a shared two-sided queue based on a machine-primitive operation called double-compare-and-swap (DCAS), which performs an atomic access to two memory locations. The original paper concludes saying that the informal proof of correctness is "complex and delicate" and that "we are not sure that we can wholeheartedly recommend" that the proposed DCAS operation be implemented on real multiprocessor machines. It turns out that a few years following the algorithm's publication, it was discovered that the algorithm was incorrect. The original publication included a detailed semi-formal proof of correctness. In an attempt to formalize that proof for a master's thesis project, Doherty [DDG$^+$04] found a previously undetected bug. This time, the verification methodology was a formal proof in the PVS [ORSSC99] proof assistant. Two corrected and verified versions were proposed. Reportedly [Hol14], each proof attempt, for both the original algorithm and the corrected versions, took several months.[4]

The Chord and Snark case examples provide evidence that we cannot trust our intuition when reasoning about concurrent algorithms. Mechanically-checked proofs seem to be the solution. Like informal, hand-written proofs are no substitute for model checking, model checking is no substitute for formal proofs. Push-button methods have an enormous value for finding bugs through counter-examples, but full correctness assurance is needed for safety-critical systems.

[3] Zave [Zav14], regarding the proof of the corrected version of Chord: "In general, the claim that this constitutes a proof is based on the empirical small scope hypothesis, which says that most bugs can be illustrated by small examples [Jac12]. In particular, there is no evidence that longer successor lists would exhibit new problems. Concerning the number of nodes, although many new behaviors were found by increasing the number of nodes from 5 to 6, no new behaviors were ever found by increasing the number from 6 to 7; this makes 8 look safe as a limit."

[4] Afterwards, Doherty used the Spin model-checker [Hol97] to demonstrate the bug he had already discovered [Lam06]. The algorithm was later formalized by Lamport [Lam06] as a way to test the PlusCal language [Lam08], a high-level algorithm language based on TLA$^+$. Writing the specification and finding the algorithm's flaws with TLC took him just a couple of days.

Automated reasoning and interactive proofs

By following the laws of formal logic, automated deduction aims at mechanizing reasoning. A deductive system is a collection of axioms and inference rules for a logical language, which allows the derivation of theorems. An automated theorem prover is a computer program that implements a deductive system in order to generate proofs. There are many classes of theorems provers, including SAT solvers for propositional logic, provers for classical first-order logic with equality (FOL), which we will refer in the following as automated theorem proving (ATP) systems, and satisfiability modulo theories (SMT) solvers, which are usually based on many-sorted first-order logic (MS-FOL). In the last decade, SMT solvers have attracted particular interest because they combine quantifier-free first-order reasoning with quantifier instantiation and decision procedures for theories relevant to verification, such as decidable fragments of arithmetic, arrays, or bit-vectors.

In richer mathematical domains, higher-order languages increase undecidability, thus automation becomes less tractable. In the semi-automatic or interactive approach to automated deduction, the user —usually a domain expert— guides the theorem-proving effort either by writing declarative proofs or by systematically applying proof-tactics that simplify the conjecture until it becomes trivial. These systems are thus called interactive *proof assistants* [Wie06]. The lower levels of the proof can be discharged as *proof obligations* to be verified by automated provers integrated to the interactive system as back-end tools. Essential components of such systems are libraries of already proven results and powerful back-end provers to achieve a satisfactory level of automation.

While the application of (semi-)automated reasoning techniques to solve engineering problems is relatively recent, they are traditionally applied by researchers to solve open questions in mathematics and logic —even when mathematicians are still reluctant to accept mechanical proofs. Examples of ground-breaking proofs carried out using semi-automated theorem provers are: the four-color theorem in Coq [Gon08], the formalization of the operating-system micro-kernel seL4 in Isabelle/HOL [KEH+09], or the very recently finished proof for the longstanding Kepler conjecture (which by now should be called Kepler's theorem) in HOL Light [HHM+11].

Recently, TLA$^+$ has been extended by a notation for writing hierarchical proofs [CDLM08, CDL+]. The TLA$^+$ Proof System TLAPS [CDLM08, CDLM10] is an interactive proof environment in which users can deductively verify properties of TLA$^+$ specifications. TLAPS can be used from within the TLA$^+$ Toolbox, an Eclipse-based framework for the development of TLA$^+$ specifications, supported by tools like the TLC model checker [YML99], and a translator from the high-level algorithm language PlusCal [Lam08] to TLA$^+$. TLAPS is built around an application called the Proof Manager, which generates proof obligations, and passes them to back-end verifiers.

At the time of starting this work, there were three available back-end provers with different capabilities: Zenon [BDD07], a tableau prover for first-order logic with equality that includes extensions for reasoning about sets and TLA$^+$ functions; Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle proof assistant [WPN08], which provides automated proof methods based on first-order reasoning and rewriting; and a back-end called SimpleArithmetic that implements a decision procedure for Presburger arithmetic. The Isabelle and Zenon back-ends have very limited support for arithmetic reasoning, while SimpleArithmetic handles only pure arithmetic formulas, requiring the user to manually decompose the proofs until the corresponding proof obligations fall within the respective fragments.

Proof support for TLA$^+$ is more recent than the TLC model-checker. Therefore, fewer case studies and applications exists for the moment. We can still mention the verification of the Paxos consensus algorithm [Lam11], the Memoir [PLD$^+$11] security architecture, and the Pastry [LMW11, LMW$^+$12] algorithm, which, as Chord [SMK$^+$01], is a protocol for distributed hash-tables over a ring topology.

In this work, we introduce a new generic back-end prover for TLAPS based on ATP systems and SMT solvers, which is coupled with type systems and a type construction algorithm for TLA$^+$. Although many of our case studies fall in the category of distributed systems, we consider safety-critical systems in the general sense.

1.1. Motivation

TLA$^+$ heavily relies on modeling data using sets and functions. Tuples and records, which occur very often in TLA$^+$ specifications, are defined as functions. Assertions mixing first-order logic with sets, functions and arithmetic expressions arise frequently in safety proofs of TLA$^+$ specifications. First-order logic offers a fair tradeoff between expressiveness and efficient automated reasoning. We thus focus on two different classes of automated theorem provers based on first-order logic: ATP systems and SMT solvers. Many TLA$^+$ proof obligations making heavy use of uninterpreted functions and quantified formulas can be handled by ATP systems. Proof obligations including arithmetic expressions can often be handled by SMT solvers.

The everyday use of TLAPS provides us with the examples that we use to illustrate our motivations to improve its proving capabilities.

Example case one One of the first proofs ever carried out in TLAPS was the proof that (a simplified version of) the well-known Bakery algorithm [Lam74] implements mutual exclusion. Leslie Lamport wrote the proof as a way to test TLAPS. At that moment, the only back-end provers available in TLAPS were Isabelle/TLA$^+$

and Zenon. Arithmetic and string reasoning was obtained by manually adding the required axioms to the interactive proof.

The standard approach to prove that a safety property is an invariant is finding a stronger inductive invariant that logically implies the one we want to prove. As a comment in the specification, Lamport wrote that the inductive proof "is about 600 lines long (down from 1400 for the original version) and takes about 6 minutes to process [it] (down from about 40 minutes on [my] laptop for the original version)". The Proof Manager generates 223 proof obligations corresponding to the lower levels of the proof hierarchy.

He further comments about proving Bakery's inductive invariant:

> The proof of the last theorem took me about a week. I wrote it the way I usually write such proofs, first breaking it down into checking that each action preserves the invariant, then for each action checking that it preserves each conjunct of the invariant. [...] No matter how trivial the reasoning, Zenon and Isabelle require that quantifiers in a conclusion be eliminated. And Isabelle sometimes needs to be led by the nose.

Even if a decision procedure for arithmetic had been available at that moment, the proof would have been written practically with the same proof structure and in almost the same length. Moreover, the deductive capabilities of Zenon and Isabelle have not improved significantly since.

Now, we can replace the 600 lines of proof by a significantly shorter one. A user would have to write one level of interactive proof to decompose the conjecture at its first syntactical level, which basically corresponds to the disjunction of the eight actions of the algorithm (this step can be also achieved with a couple of mouse-clicks in the Toolbox). No extra lemmas are required, just the explicit expansion of the necessary definitions. By instructing the Proof Manager to use the new backend, the solvers succeed to find the proofs in a few seconds, depending on the SMT solver that was invoked.

Example case two Memoir [PLD+11] is a generic security architecture, formally verified by Douceur et al. [DLP+11] with TLAPS, when the first versions of the ATP and SMT backends were not yet publicly available. Briefly, they demonstrated that Memoir's protocols preserve the property of state continuity for protected modules of the architecture, meaning that a module's state remains persistently and completely inviolate. Memoir's specification can be considered to be a medium or large specification by TLA$^+$ standards: it contains 61 top-level definitions and 182 LET-IN definitions in three refinement levels, with 3, 8, and 9 actions at each refinement level. The specification makes heavy use of records, but the proofs do not require any arithmetic reasoning. The verification of the safety, type-correctness and refinement properties encompass 74 proofs of lemmas and theorems from which 5816 proof obligations are generated.

The proofs were originally hand-written and then transcribed to TLA$^+$ proofs. Different users have different styles of writing proofs, being hand-written or machine-checked. The TLA$^+$ proofs for Memoir are very detailed and could likely have been compressed somewhat further with the available back-end provers at that moment, although not as much as with the ATP and SMT backends.

For instance, the type-correctness proof of the first level specification is split in 424 smaller proof steps taking less than 8 seconds to prove it. The same interactive proof can now be discharged by the default ATP system or SMT solver in one line of proof step in less than 2 seconds. The user time and effort to write the TLA$^+$ proof is thus tremendously reduced.

Example case three Mathematical problems are conceptually deeper than proving safety properties about algorithms. Almost always, they include at least some basic arithmetic. As a toy example, consider a trivial theorem about the cardinality of sets, informally stated as: "any finite set whose cardinality is equal to one is a singleton". In TLA$^+$, it is written as follows:

$$\textsc{theorem}\ \ \forall S : \mathit{IsFiniteSet}(S) \land \mathit{Cardinality}(S) = 1 \Rightarrow \exists m : S = \{m\}$$

The constant $\mathit{Cardinality}$ is a unary operator, and $\mathit{IsFiniteSet}$ is defined as:

$$\mathit{IsFiniteSet}(S) \triangleq \exists n \in \mathit{Nat} : \exists f : \mathit{IsBijection}(f, 1\,..\,n, S)$$

In turn, each conjunct that defines the predicate $\mathit{IsBijection}(f, S, T)$ states that f is (i) a function with domain S and codomain T, (ii) injective, and (iii) surjective:[5]

$$\begin{aligned}\mathit{IsBijection}(f, S, T) \triangleq\ & \land\ f \in [S \to T] \\ & \land\ \forall x, y \in S : (x \neq y) \Rightarrow (f[x] \neq f[y]) \\ & \land\ \forall y \in T : \exists x \in S : f[x] = y\end{aligned}$$

The definition of cardinality is given axiomatically, that is, a formula given as an axiom over the TLA$^+$ logic defines its semantics. The standard TLA$^+$ module defines the following formula named $\mathit{CardinalityAxiom}$. It expresses that for any variable S that is a finite set, the cardinality of S is equal to some variable n if and only if n is a natural number and there exists a bijection between the interval from 1 to n and the set S:

$$\begin{aligned}&\textsc{axiom}\ \mathit{CardinalityAxiom} \triangleq \\ &\quad \forall S : \mathit{IsFiniteSet}(S) \Rightarrow \\ &\qquad \forall n : (n = \mathit{Cardinality}(S)) \Leftrightarrow n \in \mathit{Nat} \land \exists f : \mathit{IsBijection}(f, 1\,..\,n, S)\end{aligned}$$

[5] It is conventional in TLA$^+$ to write conjunctions and disjunctions as multi-line bulleted lists, for better presentation and to save parentheses.

After inspecting the theorem statement and the above definitions, one can deduce that the definition of $IsFiniteSet$ is irrelevant for the proof's validity (finiteness is just required as a condition to apply $CardinalityAxiom$). As a user writing the proof, one would expect to prove the theorem just by using $CardinalityAxiom$ as a fact and expanding the definition of $IsBijection$. In TLA$^+$ this is expressed by writing after the theorem statement:

BY $CardinalityAxiom$ DEF $IsBijection$

However, the default provers called by the Proof Manager, that is, Zenon and Isabelle, fail to find a proof. Then, the user would have to decompose the proof in many steps and prove them separately. The proof would look like this:

$\langle 1 \rangle$ SUFFICES ASSUME NEW S, $IsFiniteSet(S)$, $Cardinality(S) = 1$
PROVE $\exists m : S = \{m\}$
OBVIOUS
$\langle 1 \rangle 1$. $1 \,..\, 1 = \{1\}$
BY $SimpleArithmetic$
$\langle 1 \rangle 2$. PICK $f : IsBijection(f, 1 \,..\, 1, S)$
BY $CardinalityAxiom$
$\langle 1 \rangle 3$. $S = \{f[1]\}$
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ DEF $IsBijection$
$\langle 1 \rangle 4$. QED
BY $\langle 1 \rangle 3$

This detailed proof is straight-forward, but it requires the user to decompose it in five steps to prove trivial arithmetic facts such as step $\langle 1 \rangle 1$, where the expression $a \,..\, b$ represents the set of integer numbers comprised between the expressions a and b.

This example suggests that an automated prover handling simultaneously first-order logic and arithmetic could be very useful in practice. As a matter of fact, the back-end prover based on SMT solvers is able to discharge the proof obligation generated from the above single-line TLA$^+$ proof.

1.2. Challenges

The main goal of this thesis is to soundly and efficiently improve the automation capabilities of the proof development performed with the TLA$^+$ Proof System. We focus only on the non-temporal fragment of the language, which is enough for proving safety properties, including inductive invariants and refinement mappings [AL91]. In practical terms, less than 5% of a typical TLA$^+$ specification contain temporal expressions, therefore the non-temporal part of a specification accounts for the major part of the verification effort. Only trivial temporal-logic reasoning is needed for

safety. To handle these cases, we can use a new decision procedure for propositional temporal logic [DKL+14], very recently integrated in TLAPS.

The translation from TLA$^+$ to first-order logic presents the first challenge of the integration. Even if some of the employed encoding techniques can be found in similar tools for other specification languages, the particularities of TLA$^+$ make the translation non-trivial:

- The TLA$^+$ syntax allows to write absurd expressions such as $3 \cup \text{TRUE}$, which denotes a value, though unspecified. Moreover, there is no syntactical distinction between terms and formulas.
- Zermelo-Fraenkel set theory is not finitely axiomatizable in first-order logic, unlike other formalizations of set theory such as von Neumann-Bernays-Gödel (NBG). Specifically, the ZF axioms of set comprehension and replacement allow the introduction of new sets through axioms schemas, i.e. axioms parameterized by a predicate. Therefore, those set objects cannot be encoded directly in first-order logic.
- Functions, which are defined axiomatically, are total and have a domain. This means that a function applied to an element of its domain has the expected value but, for any other argument, the function application still has a value, but unspecified. In the same way, the behavior of arithmetic operators is specified only for arguments that are integers or positive integers.
- TLA$^+$ is equipped with a deterministic choice operator as a primitive element of the language. It corresponds to Hilbert's ε operator, which is difficult to encode in a classical first-order setting.

Perhaps more challenging is the fact that TLA$^+$ is an untyped language. Being untyped makes a language very expressive and flexible for writing specifications, but it also makes automated reasoning quite challenging. Not having types is part of the idiosyncrasy of TLA$^+$. The rationale behind this fundamental design principle can be found in a paper by Lamport and Paulson [LP99]. Basically, they argue that a type discipline in a high-level specification language restricts what can be expressed in that language. Moreover, types complicate the task of the users writing the specifications. TLA$^+$ is user oriented, in the sense that it focuses on providing users a compromise of a simple, yet very expressive, specification language. A type discipline can lead to an early detection of syntactic errors, but more importantly for our purposes, it benefits the deductive capabilities of the reasoning tools. If it can be avoided, the task of dealing with the complications of a lack of a type system should rest on the tool developers, instead of forcing the users to adapt their specifications to the burden of types. Since SMT input languages are sorted, a second challenge consists in automatically assigning types to the sub-expressions of a TLA$^+$ proof obligation. In the following we will use the terms *type* and *sort* interchangeably.

1.3. Related work

Over the past years there have been several efforts to integrate interactive and automated theorem provers. One of the main components of any integration is the definition of the translation from the logic of the specification language to the input language of the external provers. Most automated theorem provers are based on (typed or untyped) first-order logic, while the proof assistants are generally founded on variants of Church's higher-order logic or Martin-Löf's type theory, which are foundational alternatives to set theory. Therefore, many integrations for these kind of systems exist in the literature, for instance in Coq [AF06, AGST10, BCP11, AFG+11], in Isabelle/HOL [Hur03, MP06, FMM+06, PB10, BBN11, BBP13], in HOL Light [KU13], in PVS [Dd06], or in ACL2 [RH06]. Only a few formalisms are based on set theory, namely Z [Spi92], B [Abr10], Mizar [lM93], Isabelle/ZF [Pau93], and Metamath [Meg07].

Specification formalisms almost always exploit types in some way, either explicitly or implicitly. Except TLA$^+$ and Isabelle/ZF (described below), the only other formal language that does not have any kind of type system, as far as we know, is ACL2 [KMM00]. In short, ACL2 is an applicative programming language based on Common Lisp. In an untyped language like set theory, one writes $x \in Int$ to express that a variable x is an integer. In ACL2, one relies on the predicate $(\mathsf{integerp}\ x)$, which is true if and only if x is an integer number.

In the rest of this section we compare TLA$^+$ and TLAPS with three formal languages and their respective tools and proof environments: the B method, Mizar, and Isabelle/ZF. We consider these three systems as the most relevant and established set theory-based languages [Wie06]. Apart from being based on some variant of set theory, they integrate external first-order automated provers. The technical details about the translation to first-order languages and how they treat types will be given respectively in the chapters 4 and 5, where that information is relevant.

B The (classical) B and Event-B methods [Abr10] are modeling notations founded on the concept of refinement to represent systems at different abstraction levels. Like TLA$^+$, B models represent abstract state machines. The B languages are also based on Zermelo-Fraenkel set theory, although in a somewhat weaker version, because terms and functions have monomorphic types in the style of MS-FOL, thus greatly simplifying the translations to SMT languages. Another difference is that functions are defined as binary relations, as is typical in set theory.

Deductive proofs for B models are mainly performed to verify safety properties and consistency between refinement levels. The Eclipse-based Rodin toolset [ABH+10] for Event-B generates proof obligations that can either be proved interactively or they can be discharged to SMT solvers, which are used as oracles. Rodin incorporates two plugins to translate to SMT languages: ppTrans by Konrad et al. [KV12] and, another

one incidentally called SMT solvers by Deharbe et al. [DFGV12]. Atelier-B is a similar tool aimed at the verification of industrial B models. Mentre et al. [MMFA12] proposed Why3 as an interface to discharge Atelier-B proof obligations to different SMT solvers. The recent BWare project by Delahaye et al. [DDMM14] aims at building a generic verification platform based on Why3 as a proxy for discharging proof obligations to first-order provers and SMT solvers. Unlike previous tools, BWare adopts a skeptic approach. It requires the external theorem provers to return proof objects, with the goal of checking them independently in logical frameworks such as Coq or Dedukti.

Pro-B [PL12] is another related platform which includes an animator, a model-checker and a constraint solver for Event-B. It relies on Kodkod, the Alloy Analyzer's [Jac12] backend, to do constraint solving over the first-order fragment of the language, and on the Pro-B kernel for the rest. It can translate TLA$^+$ modules to Event-B [HL12] (and back) allowing TLA$^+$ users to use the Pro-B tools.

Mizar The longstanding Mizar project [lM93] provides one of the largest available libraries of formalized mathematics, mechanically verified by the Mizar checker. Its architecture, and the way it integrates external provers, is quite different than that of TLAPS. But as a set-theoretic language, it has some points in common with the underlying logic of TLA$^+$. The Mizar language is based on first-order predicate logic extended with schematic axioms to be able to deal with the comprehension and replacement axiom schemes of ZF set theory, and with Hilbert's choice operator.[6] Semantically all objects are sets, but contrary to standard set theory axiomatizations, the Mizar language is typed [Wie07]. Every quantified or declared variable is accompanied, possibly implicitly, by a type annotation, even when types do not play any foundational role in the system. In particular, it implements a powerful type system with dependent types.

Earlier Mizar tools MoMM and Proof Advisor based on first-order theorem provers, apply machine learning techniques to search the entire MML library for similar already-proved theorems, which are then provided as hints to prove new conjectures. The more recent MizAR (Automated Reasoning for Mizar) tool suite [Urb08], attaches several automated reasoning and presentation tools to the Mizar system. It relies on a web service, inspired by SystemOnTPTP [Sut00], that invokes external ATP systems. The ATP integration is also exploited to do cross-verification of the Mizar library in different theorem provers. There is no integration of Mizar with SMT solvers that we know of.

Isabelle Isabelle [WPN08] is an LCF-style theorem prover based on a generic logical framework called Isabelle/Pure. Object-logics such as Isabelle/TLA$^+$, Isabelle/ZF

[6] Mizar's set theory is called Tarski-Grothendieck set theory, which is ZF plus an extra axiom from which the Axiom of Choice can be derived.

[Pau93], and Isabelle/HOL [NPW02] are instantiations over the meta-theory of Isabelle/Pure. Isabelle/ZF axiomatizes classical Zermelo-Fraenkel set theory with the Axiom of Choice, and it is mainly used for formalizing pure mathematics. Except for terms and formulas, which are defined as different syntactical entities, Isabelle/ZF is untyped. On the other hand, Isabelle/HOL is an encoding of polymorphic higher-order logic.

The proof language of Isabelle, which is called Isar [Wen07], as well as that of TLA$^+$, are hierarchical and declarative, allowing the combination of structured proofs with the invocation at the lowest proof levels of automatic theorem provers and decision procedures. Isar and TLA$^+$ proofs have some differences that encourage different styles of proof development in each case. For instance, Isar users have more control on how to invoke the back-end tools, and they can accumulate facts instead of citing proof steps by their name, allowing a more linear, rather than hierarchical, proof development. Isabelle is equipped with several native automatic proof tools, including decision procedures for arithmetic, a term-rewriting engine called Simplifier [Nip89], and a tableau prover [Pau99].

Sledgehammer [PB10] is a sophisticated tool for Isabelle/HOL that automates the invocation of external first-order ATP systems and SMT solvers. The underlying logic of Isabelle/HOL entails a quite different encoding into first-order languages than set theory. This is one of the reasons why Sledgehammer is not available for Isabelle/ZF. Despite that, most Isabelle/HOL formulas fall in the first-order domain, using only a few higher-order features [MP08].

Unusually, but inspired by a previous integration of the first-order prover Metis and HOL4 [Hur03], the first implementation of the translation to the input format of automated theorem provers was allowed to be unsound: types were not attached to every term, only enough type information was encoded to enforce correct type class reasoning [MP06]. This is sufficient because proofs are rechecked by Isabelle's inference kernel, thus soundness is not crucial. The current implementation safely erases most type information by inferring type monotonicity [BK11, CLS11], resulting in a sound encoding [BBN11]. The automated provers can be executed in parallel, either locally or remotely via the SystemOnTPTP [Sut00] web service.

One of Sledgehammer's most important features is one-click invocation, which is also inherent to the TLA$^+$ proof language and to the architecture of TLAPS. The difference is that Sledgehammer implements a symbol-based relevance filter that heuristically collects a potentially useful set of a few hundred lemmas from existing libraries to attach to the translations. TLAPS libraries are currently under development, so users have to manually identify relevant facts and add these facts themselves to the interactive proof.

1.4. Contributions

The first contribution of this thesis is a solution to the integration of ATP systems and SMT solvers to the TLA$^+$ proof environment (Chapter 4). The external theorem provers are integrated as *oracles*, meaning that we consider them sound and bug-free (at least for now). The integration has two main components. The first one is a sound translation of a significant fragment of TLA$^+$ formulas to unsorted and many-sorted first-order logic. The set theoretical fragment of the language is encoded using a unique sort (allowing set of sets), while sorted theories like arithmetic are homomorphically injected into the single-sorted formulas. However, set theoretical objects that are defined by axiom schemas cannot be finitely encoded in first-order logic. For instance, in the set $\{x \in S : P(x)\}$, which contains the elements of S satisfying P, the predicate P would be quantified as a second-order variable. The second component of the integration encompasses preprocessing and optimization techniques to make the translation efficient and feasible for all TLA$^+$ constructs, including the CHOOSE operator. Among them, a confluent term-rewriting engine that takes TLA$^+$ expressions to *basic* normal form is coupled with an abstraction method that removes complex and untranslatable TLA$^+$ expressions from the proof obligation.

The second main contribution is the formal definition of two type systems for untyped set theory, extended to all relevant TLA$^+$ expressions (Chapter 5). In this approach, a constraint-based type synthesis algorithm takes a TLA$^+$ expression and decorates it with types. This happens behind the scenes: TLA$^+$ users never see any type annotation, which is used only internally by TLAPS to improve the above translation. In a first approach to the translation, which we call "untyped", we use only one sort to encode all TLA$^+$ expressions except for sorted theories. In this way, type inference is essentially delegated to the solvers through the encoding.

The first type system for TLA$^+$ is based on elementary types, i.e. types that are on a par with the sorts of many-sorted first-order logic. This type system is simple and results in a decidable algorithm for constructing types, but it over-approximates the values of TLA$^+$ expressions. Thus, extra checks need to be added to the encoding, for instance to make sure that a function's argument belongs to the function's domain. The second type system expands the previous one with dependent and refinement types, which captures very precisely the values represented by the TLA$^+$ terms, at the expense of undecidability of type construction. While the above "untyped" encoding can be used to deal with the non-typable expressions, the precise types opportunely allow to further simplify the translations. In particular, the domain checks for the previous type system are no longer needed. This technique for untyped languages may be of independent interest.

Additionally, the above results have been realized in a new generic back-end prover based on ATP systems and SMT solvers for TLAPS. We say it is generic because it is not tailored to improve the performance of a specific prover. Instead, any external

prover that accepts the standard input languages of ATP systems and SMT solvers can be plugged to TLAPS. We provide experimental evidence showing that our approach is feasible in practice (Chapter 6).

Several of the contributions described here have been presented at conferences and workshops. The work in this thesis consolidates and improves upon preliminary ideas and results presented in the publications [MV12a], [MV12b], and [MV14].

1.5. Overview

The rest of this thesis is organized as follows:

- Chapter 2 describes the TLA$^+$ specification language, the TLA$^+$ proof language and TLA$^+$'s proof tools. First, we formally describe the logic underlying the TLA$^+$ fragment under consideration in this document. As a minor contribution, we formally describe the syntax and semantics of the set theoretic fragment of TLA$^+$, and analyze to which variant of ZFC set theory TLA$^+$ belongs. By way of a running toy example, we present a typical specification with its properties, and then we describe the proof language, and the TLA$^+$ Proof System.
- Chapter 3 briefly describes the main languages, concepts, and techniques of automated theorem provers based on first-order logic. In particular, we describe the underlying unsorted and many-sorted first-order logics in which the theorem provers operate, their standard input languages, and the internal workings of the ATP systems and SMT solvers.
- Chapter 4 presents the generic integration framework for automated theorem provers in TLAPS and its main component, the translation to unsorted and many-sorted first-order logics.
- Chapter 5 describes the two new type systems for TLA$^+$, including typing rules, soundness results, and the constraint-based type synthesis algorithm.
- Chapter 6 presents experimental evaluation of the ATP and SMT back-end prover, which implement the encoding methods and the type systems of the previous two chapters.
- Chapter 7 concludes and gives directions for future work.

The main results of this thesis are in Chapters 4 and 5, which can be read independently.

Chapter 2.

TLA$^+$ and its proof tools

Contents

> One good definition is worth three theorems.
>
> Alfred Adler, 1972

The specification language TLA$^+$ [Lam02] provides a concrete syntax on top of its logical foundation, the temporal logic of actions (TLA), which describes dynamic behaviors of state-transition systems. Specifically, TLA is a variant of linear-time temporal logic [AM96] coupled with a modal priming operator $'$ for specifying the state transitions. A *state* of a system is an assignment of values to variables $x_1, \ldots, x_n$. Each event or *step* of the system is defined as a transition from one state to another. An *action* asserts a relation between two states through a transition predicate on variables $x_1, \ldots, x_n$ and $x'_1, \ldots, x'_n$. Given a pair of states (x, x'), the unprimed variable x refers to the value of x in the first state, and the primed variable x' refers to the value of x in the second state. TLA$^+$ specifications are organized in *modules*, which belong to the extra-logical part of the language. TLA is parameterized by an underlying first-order language. In the case of TLA$^+$, that language is a variant of Zermelo-Fraenkel set theory with the axiom of choice, which is used to specify the systems' data structures.

In addition, TLA$^+$ is extended with a notation for writing hierarchical proofs. The TLA$^+$ Proof System (TLAPS) interactive environment interprets those proofs to generate proof obligations, which are discharged to be proved by back-end verifiers.

Primed expressions, modules and macro definitions are irrelevant in our context because the Proof Manager application "flattens out" these expressions before generating the proof obligations.

Chapter overview Section 2.1 presents the underlying logic of TLA$^+$ over which we will work in the rest of this document. In Section 2.2, with the help of a toy example, we describe a TLA$^+$ specification to introduce some basic concepts. In Section 2.3, we continue developing the running example to present the TLA$^+$ Proof System and to show how it generates proof obligations.

2.1. Underlying logic

The logical foundations of TLA$^+$ are the TLA logic and a variant of ZFC set theory. We are interested in the language of the proof obligations generated by the TLA$^+$ Proof System, in particular its non-temporal fragment. In TLA$^+$ terminology, we are only considering the language at the *state* level,[1] which includes constants, constant operators, and unprimed variables. A proof obligation, which we define in detail below, is a TLA$^+$ sequent containing state-level expressions and possibly primed variables or primed operators. If a primed variable x' appears in a proof obligation, we simply consider it as a different symbol than the variable symbol x. The same applies for primed operators.

For a complete presentation of the TLA$^+$ language, we refer the reader to [Lam02, Sec. 16], which can be considered the "official" description of the language. The language we describe is an extension of first-order logic with equality. The main difference with standard fist-order logic is that predicate and function symbols belong to the single entity of TLA$^+$ *operators*. We start by defining the syntax and semantics of the first-order fragment over which the set theoretical axioms are defined. Then, we add the choice operator and gradually append other constructs.

TLA$^+$ syntax

We assume given two non-empty, infinite, and disjoint collections

- $\mathcal{V}$ of *variable* symbols, and

[1] In TLA, expressions are categorized in four levels, hierarchically ordered as follows: *constant* $\subseteq$ *state* $\subseteq$ *transition* $\subseteq$ *temporal*. Constant-level expressions contain only rigid variables (a.k.a. constants). Flexible variables belong to the state level. Primed expressions may occur at the transition level. A temporal-level expression may contain any TLA operator, i.e. it is a temporal formula.

- $\mathcal{O}$ of *operator* symbols,[2]

and a function

- $ar : \mathcal{O} \to \mathbb{N}$ that assigns a non-negative number, the *arity*, to each symbol in $\mathcal{O}$.

Together, they define a TLA$^+$ signature $\langle \mathcal{V}, \mathcal{O}, ar \rangle$ that fixes an alphabet of non-logical symbols. We consider $\mathcal{V}$ and $\mathcal{O}$ as fixed collections of built-in and user-defined symbols, pre-existing in each generated proof obligation. Note that these "variables" are not the state variables mentioned before, but variables of ordinary logic (more precisely, the rigid variables of modal logic), and they are among the "constants" of TLA$^+$.

The only syntactical category in the language is the expression. Only for presentational purposes we distinguish among them terms, formulas and set-objects. An *expression* e is inductively defined by the following grammar:

$$\begin{array}{lll} e ::= v \mid w(e, ..., e) & & \text{(terms)} \\ \quad \mid \textsc{false} \mid e \Rightarrow e \mid \forall v\colon\ e \mid e = e \mid e \in e & & \text{(formulas)} \\ \quad \mid \{\} \mid \{e, e\} \mid \textsc{subset}\ s \mid \textsc{union}\ s \mid \{v \in e : e\} \mid \{e : v \in e\} & & \text{(sets)} \end{array}$$

A *term* is a variable symbol v in $\mathcal{V}$ or an arity-consistent application of an operator symbol w in $\mathcal{O}$ to expressions. Nullary operator symbols c are called *constants*, and are written c instead of $c()$.

Formulas are built from FALSE, implication and universal quantification, and from the binary operators $=$ and $\in$ which are the two non-logical primitive symbols of the language, set aside the set objects. From these formulas, we can define the familiar constant TRUE, the unary connective $\neg$, the binary connectives $\wedge$, $\vee$, $\Leftrightarrow$, and the existential quantifier $\exists$.

In standard set theory, sets are constructed from axioms that state their existence. Here, we add the set constructors as primitive objects of the language, and later we define their semantics axiomatically. The *set objects* are the empty set, the pairing set, the power set, the generalized union, and two forms of set comprehension. The enumeration set $\{e_1, \ldots, e_n\}$, with $n \geq 1$, can be defined using pairs and UNION. Since TLA$^+$ is a set-theoretic language, every expression —including formulas, functions, numbers, etc.— denotes a set.

Other familiar set expressions can be defined from the above: $e_1 \cup e_2$ (union), $e_1 \cap e_2$ (intersection), and $e_1 \setminus e_2$ (set difference). TLA$^+$ also defines the abbreviations:

$$\begin{array}{llll} x \neq y \triangleq \neg(x = y) & S \subseteq T \triangleq \forall x \in S\colon x \in T & \exists x \in S\colon e \triangleq \exists x\colon x \in S \wedge e \\ x \notin y \triangleq \neg(x \in y) & \textsc{boolean} \triangleq \{\textsc{true}, \textsc{false}\} & \forall x \in S\colon e \triangleq \forall x\colon x \in S \Rightarrow e \end{array}$$

[2] TLA$^+$ operator symbols correspond to the standard function and predicate symbols of first-order logic but we reserve the term "function" for TLA$^+$ functional values.

The definition of free variables is the usual one for first-order logic over the set of variable symbols $\mathcal{V}$. We note $FV(e)$ the free variables of an expression e. We additionally define variable substitution in the usual way, and substitution for operators when applied only to variables. These definitions are for terms and formulas only. The extension to set objects and other expressions is straightforward.

Definition 1 (Variable substitution)**.** *Given two* TLA$^+$ *expressions* e_1 *and* e_2*, and* x *a free variable of* e_1*. The expression* $e_1[x \leftarrow e_2]$ *is inductively defined as follows on* e_1*:*

$$
\begin{aligned}
x[x \leftarrow e_2] &\triangleq e_2 \\
y[x \leftarrow e_2] &\triangleq y \\
o(e_1, \ldots, e_n)[x \leftarrow e_2] &\triangleq o(e_1[x \leftarrow e_2], \ldots, e_n[x \leftarrow e_2]) \\
\textsc{false}[x \leftarrow e_2] &\triangleq \textsc{false} \\
(\phi_1 \Rightarrow \phi_2)[x \leftarrow e_2] &\triangleq (\phi_1[x \leftarrow e_2]) \Rightarrow (\phi_2[x \leftarrow e_2]) \\
(\forall y : \phi)[x \leftarrow e_2] &\triangleq \forall z : (\phi[y \leftarrow z][x \leftarrow e_2]) \qquad \textit{(where } z \textit{ is a fresh variable)} \\
(e = f)[x \leftarrow e_2] &\triangleq (e[x \leftarrow e_2]) = (f[x \leftarrow e_2]) \\
(e \in f)[x \leftarrow e_2] &\triangleq (e[x \leftarrow e_2]) \in (f[x \leftarrow e_2])
\end{aligned}
$$

TLA$^+$ first-order semantics

The standard semantics of TLA$^+$ offers three different alternatives to interpret expressions. We adopt the *liberal* interpretation, as defined in [Lam94b, Sec. 16.1.3], in contrast to the *moderate* and to the *conservative* interpretations. In the liberal interpretation, an expression like $42 \Rightarrow \{\}$ always has a truth value, but it is not specified if that value is true or false. In the conservative and moderate interpretations, the value of $42 \Rightarrow \{\}$ is completely unspecified. Only in the moderate and liberal interpretation, the expression $\textsc{false} \Rightarrow \{\}$ has a Boolean value, and that value is true. In the liberal interpretation, all the ordinary laws of logic, such as commutativity of $\wedge$, and any tautology of propositional or predicate logic, are valid, even when the arguments are not guaranteed to be Boolean.

We formally define the first-order semantics of TLA$^+$ as an extension to the standard semantics of first-order logic with equality. Let a TLA$^+$ *model* or *structure* $\mathcal{M}$ be a triple $\langle \mathcal{D}, v, \mathcal{I} \rangle$, which is composed of:

- a countably infinite set $\mathcal{D}$ called the *domain* or *universe of discourse*, which contains two fixed, distinguished symbols T and F denoting truth values, such that $\mathsf{T} \neq \mathsf{F}$,
- a *valuation* function $v \colon \mathcal{V} \to \mathcal{D}$ that assigns to each non-logical variable symbol an element in the domain, and
- an *interpretation* function $\mathcal{I}$ that assigns to each operator symbol $o \in \mathcal{O}$ a function $\mathcal{I}(o) \colon \mathcal{D}^{ar(o)} \to \mathcal{D}$.

With this in mind, we define the *truth* valuation of TLA$^+$ expressions in the standard Tarskian way, with the aid of a function assigning *universal* values to expressions. In the following, let $v \oplus (x \mapsto d)$ be the valuation function equal to v for all variables in $\mathcal{V}$ except for x, where it maps x to d.

Definition 2 (Truth value and universal value)**.** *We define the interpretation of* TLA$^+$ *expressions with two mutually recursive valuation functions, both defined under a model* $\mathcal{M} = \langle \mathcal{D}, v, \mathcal{I} \rangle$*. The* truth value *of an expression* e*, noted* $truth_{\mathcal{M}}(e)$ *or* $truth_{\mathcal{D},v,\mathcal{I}}(e)$*, is defined by:*

$$
\begin{array}{ll}
truth_{\mathcal{M}}(x) & = val_{\mathcal{M}}(x) \textit{ iff } val_{\mathcal{M}}(x) \textit{ is } \mathsf{T} \textit{ or } \mathsf{F} \\
truth_{\mathcal{M}}(o(e_1, \ldots, e_n)) & = val_{\mathcal{M}}(o(e_1, \ldots, e_n)) \textit{ iff } val_{\mathcal{M}}(o(e_1, \ldots, e_n)) \textit{ is } \mathsf{T} \textit{ or } \mathsf{F} \\
truth_{\mathcal{M}}(\textsc{false}) & = \mathsf{F} \\
truth_{\mathcal{M}}(\phi_1 \Rightarrow \phi_2) & = \mathsf{F} \textit{ iff } truth_{\mathcal{M}}(\phi_1) \textit{ is } \mathsf{T} \textit{ and } truth_{\mathcal{M}}(\phi_2) \textit{ is } \mathsf{F}\textit{, or } \mathsf{T} \textit{ otherwise} \\
truth_{\mathcal{M}}(\forall x : \phi) & = \mathsf{T} \textit{ iff } truth_{\mathcal{D},\mathcal{I},v \oplus (x \mapsto d)}(\phi) \textit{ is } \mathsf{T} \textit{ for all } d \in \mathcal{D}\textit{, or } \mathsf{F} \textit{ otherwise} \\
truth_{\mathcal{M}}(e_1 = e_2) & = \mathsf{T} \textit{ iff } val_{\mathcal{M}}(e_1) \textit{ is equal to } val_{\mathcal{M}}(e_2)\textit{, or } \mathsf{F} \textit{ otherwise}
\end{array}
$$

and the universal value *of an expression* e*, noted* $val_{\mathcal{M}}(e)$*, is defined by:*

$$
\begin{array}{ll}
val_{\mathcal{M}}(x) & = v(x) \\
val_{\mathcal{M}}(o(e_1, \ldots, e_n)) & = \mathcal{I}(o)(val_{\mathcal{M}}(e_1), \ldots, val_{\mathcal{M}}(e_n)) \\
val_{\mathcal{M}}(\textsc{false}) & = \mathsf{F} \\
val_{\mathcal{M}}(\phi_1 \Rightarrow \phi_2) & = truth_{\mathcal{M}}(\phi_1 \Rightarrow \phi_2) \\
val_{\mathcal{M}}(\forall x : \phi) & = truth_{\mathcal{M}}(\forall x : \phi) \\
val_{\mathcal{M}}(e_1 = e_2) & = truth_{\mathcal{M}}(e_1 = e_2)
\end{array}
$$

The truth and universal value of any other expression is unspecified. The truth value of a variable or an applied operator is defined only when the universal value is T or F. The universal value of any expression is an element of $\mathcal{D}$, while the truth value, if defined, is either T or F.

We say that an expression e is *valid in a model* $\mathcal{M} = \langle \mathcal{D}, v, \mathcal{I} \rangle$, noted $\mathcal{M} \models e$, iff $truth_{\mathcal{M}}(e)$ is T under $\mathcal{M}$. If $truth_{\mathcal{M}}(e)$ is F, we note it $\mathcal{M} \not\models e$. An expression e is *valid*, noted $\models e$, iff $truth_{\mathcal{D},v,\mathcal{I}}(e) = \mathsf{T}$ in every model $\langle \mathcal{D}, v, \mathcal{I} \rangle$, that is, for any domain $\mathcal{D}$, interpretation $\mathcal{I}$, and valuation v. An expression e is called *satisfiable* iff there exists a model $\mathcal{M}$ such that $\mathcal{M} \models e$. Otherwise, the expression e is *unsatisfiable*.

TLA$^+$ set theory

We axiomatically build TLA$^+$ set theory on our first-order language, where equality is an interpreted relation, by adding the primitive concept of set membership. We first take the essential axiom of extensionality:

$$\text{(extensionality)} \qquad (\forall x : x \in S \Leftrightarrow x \in T) \Rightarrow S = T \tag{2.1}$$

The truth and universal values for set expressions are unspecified. In order to give them a meaning, we also take as axioms the universal closure of the propositions:

$$\text{(power set)} \qquad S \in \textsc{subset}\ T \Leftrightarrow \forall x \in S\colon x \in T \tag{2.2}$$

$$\text{(union)} \qquad x \in \textsc{union}\ S \Leftrightarrow \exists T \in S\colon x \in T \tag{2.3}$$

$$\text{(empty set)} \qquad x \in \{\} \Leftrightarrow \textsc{false} \tag{2.4}$$

$$\text{(pairing)} \qquad x \in \{e_1, e_2\} \Leftrightarrow x = e_1 \vee x = e_2 \tag{2.5}$$

together with the following axiom schemas:

$$\text{(comprehension}_1\text{)} \qquad x \in \{y \in S\colon P(y)\} \Leftrightarrow x \in S \wedge P(x) \tag{2.6}$$

$$\text{(comprehension}_2\text{)} \qquad x \in \{e(y)\colon y \in S\} \Leftrightarrow \exists y \in S\colon x = e(y) \tag{2.7}$$

where P and e are schematic variables, meaning that they can be instantiated by countably infinite expressions. In a second-order setting, P and e would be simply second-order variables.

TLA$^+$ as a variant of ZF set theory It is usually said that the set theory of TLA$^+$ is a *variant* of Zermelo-Fraenkel (ZF) set theory. In the following, we briefly analyze and try to clarify to which set-theoretical variant TLA$^+$ belongs. The rest of this sub-section can be safely ignored.

The first syntactical difference, as mentioned before, is that the standard axioms assert the existence of the sets, while TLA$^+$ set objects are part of its logical symbols. The axioms of Zermelo (Z) set theory are: extensionality (the axiom 2.1), power-set (corresponding to 2.2), union (2.3), pairing (2.5),[3] bounded comprehension (2.6), and infinity. The axiom of *infinity* says that there exists a set I such that

$$\{\} \in I \wedge \forall x \in I\colon x \cup \{x\} \in I.$$

That is, I is constructed from the empty set (which can be interpreted as the number 0), and further elements (interpreted as successive numbers) are added inductively. The axiom of infinity corresponds in TLA$^+$ to the set Nat of natural numbers, which we will add later to our fragment. Originally, Zermelo included the axiom of choice (AC) in his set theory, now usually labelled ZC set theory. Incidentally, the axioms (2.1-2.6) plus the axiom of infinity, called MacLane set theory, is a suitable fragment to formalize large parts of mathematics [Lan86].

Z set theory can be extended to ZF set theory by adding the axiom of regularity and the axiom schema of replacement. The axiom of *regularity* (also called *foundation*) says that $\in$ is a well-founded relation on $\mathcal{D}$. Equivalently, it states that every non-empty set S contains an element that is disjoint from S:

$$\forall S\colon S \neq \{\} \Rightarrow \exists x \in S\colon x \cap S = \{\}.$$

[3] Originally, it was the axiom of elementary sets, i.e. singletons. Now, a singleton $\{x\}$ is usually encoded as a pair $\{x, x\}$.

This is the axiom's most common presentation. From it, we can derive, for instance, the property that no set is a member of itself.[4] As stated by Kunen [Kun80, Chap. 3], the axiom of foundation is "totally irrelevant" in ordinary mathematics, having "no real application". It is only useful, but not essential, to prove some properties about ordinals. Therefore, it is safe to omit the axiom of regularity in our encoding of TLA$^+$ problems in first-order logic. ZF without regularity is commonly labelled ZF$^-$.

The standard axiom schema of *replacement* says that, given an expression S and a binary predicate ϕ, such that ϕ is *single-valued* for any x in S, i.e.

$$\forall x \in S\colon \forall y, z\colon \phi(x, y) \wedge \phi(x, z) \Rightarrow y = z,$$

then there exists a set object $\mathcal{R}(S, \phi)$, and that $x \in \mathcal{R}(S, \phi) \Leftrightarrow \exists y \in S : \phi(x, y)$. The axiom schema of *bounded comprehension* (also called *separation* or *specification*, corresponding to 2.6) is implied by the axiom schema of replacement and the axiom of empty set (2.4). Furthermore, both set comprehension objects (referred in 2.6 and 2.7) are instances of the axiom schema of replacement: taking the two single-valued predicates

$$\phi_1(x, y) \triangleq x = y \wedge P(y) \qquad \text{and} \qquad \phi_2(x, y) \triangleq x = e(y),$$

we can define

$$\{y \in S : P(y)\} \triangleq \mathcal{R}(S, \phi_1) \qquad \text{and} \qquad \{e(y) : y \in S\} \triangleq \mathcal{R}(S, \phi_2).$$

Other constructs

Choice Another primitive expression of standard TLA$^+$ is Hilbert's choice operator ε, written $\textsc{choose}\ x\colon P(x)$, that denotes an arbitrary but fixed value x such that $P(x)$ is true, if such value exists. If no such x exists, then the expression has a completely arbitrary value.

$$e ::= \cdots \mid \textsc{choose}\ x\colon P(x)$$

The semantics of CHOOSE are expressed by the following axiom schemas. In particular, the first one gives an alternative way of defining quantifiers, and the second one expresses CHOOSE's extensionality by assigning the same witness value to equivalent formulas P and Q.

$$\big(\exists x\colon P(x)\big) \Leftrightarrow P\big(\textsc{choose}\ x\colon P(x)\big) \tag{2.8}$$

$$\big(\forall x\colon P(x) \Leftrightarrow Q(x)\big) \Rightarrow \big(\textsc{choose}\ x\colon P(x)\big) = \big(\textsc{choose}\ x\colon Q(x)\big) \tag{2.9}$$

[4] Proof: in the case where S is $\{x\}$, for some set x, the only element in $\{x\}$ is x, which by regularity must be disjoint from $\{x\}$: $x \cap \{x\} = \{\}$. Then, it is not possible to have $x \in x$.

The CHOOSE operator with the axiom (2.8) is equivalent to Hilbert's ε operator, which is non-deterministic, thereby representing a form of the axiom of choice [AZ13]. From axiom (2.9) note that, if for some predicate P, the formula $\exists x\colon P(x)$ does not hold, or equivalently, $\forall x\colon P(x) \Leftrightarrow \text{FALSE}$ holds, then

$$(\text{CHOOSE } x\colon P(x)) = (\text{CHOOSE } x\colon \text{FALSE}).$$

Consequently, the expression CHOOSE $x\colon$ FALSE and all its equivalent forms represent a unique value.

In the following, we extend the TLA$^+$ grammar with expressions defined axiomatically on top of set theory.

Functions As a first extension of this purely set-theoretic fragment, we now introduce (total) functions.

$$e ::= \cdots \mid e[e] \mid \text{DOMAIN } e \mid [v \in e \mapsto e] \mid [e \text{ EXCEPT } ![e] = e] \mid [e \to e]$$

In principle, all well-formed terms denote sets, but some of those expressions are used as *functions*, as they are called in TLA$^+$. Functions are then those terms f that satisfy a special predicate $IsAFcn(f)$. The predicate

$$IsAFcn(f) \triangleq f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

characterizes the TLA$^+$ value of f as being a function. Note that it is possible to quantify over (terms representing) functions.

In standard set theory, functions are defined as binary relations (i.e. sets of pairs) restricted so that each element of the domain is mapped to a unique element in the range of the relation. TLA$^+$ instead introduces functions axiomatically using three primitive constructs. The expression $f[e]$ denotes the result of applying the function f to the expression e, DOMAIN f denotes the domain of f, and the expression $[x \in S \mapsto e]$ denotes the function f with domain S such that $f[x] = e$, for any $x \in S$. For $x \notin S$, the value of $f[x]$ is unspecified. In this way, no function application of any expression to any other expression can be discarded as syntactically incorrect.

Functions are governed by the axiom

$$\begin{aligned} f = [x \in S \mapsto e] \quad \Leftrightarrow \quad & \land\ IsAFcn(f) \\ & \land\ \text{DOMAIN } f = S \\ & \land\ \forall y \in S\colon f[y] = e[x \leftarrow y] \end{aligned} \tag{2.10}$$

From this axiom and set extensionality (2.1) we can derive the property of *function extensionality*:

$$\begin{aligned} \forall f, g\colon\ & \land\ IsAFcn(f) \land IsAFcn(g) \\ & \land\ \text{DOMAIN } f = \text{DOMAIN } g \\ & \land\ \forall x \in \text{DOMAIN } f\colon f[x] = g[x] \\ & \Rightarrow f = g \end{aligned} \tag{2.11}$$

In addition, the construct for function-update $[f \text{ EXCEPT } ![d] = e]$ denotes the function $\hat{f}$ equal to f except that $\hat{f}[d] = e$ when $d \in \text{DOMAIN } f$, and the expression $[S \to T]$ denotes the set of all functions with domain S and codomain T:

$$[f \text{ EXCEPT } ![d] = e] \triangleq [y \in \text{DOMAIN } f \mapsto \text{IF } y = d \text{ THEN } e \text{ ELSE } f[y]] \tag{2.12}$$

$$[S \to T] \triangleq \{[x \in S \mapsto y] : y \in T\} \tag{2.13}$$

Arithmetic TLA$^+$'s grammar is further extended with arithmetic expressions.

$$e ::= \cdots \mid 0 \mid 1 \mid 2 \mid \ldots \mid Int \mid -e \mid e + e \mid e - e \mid e * e \mid e \,\%\, e \mid e \div e \mid e < e$$

Natural numbers are primitive symbols, Int denotes the set of integer numbers, and the operators $+$, $-$, $*$, and $<$ are interpreted in the standard way when their arguments are integers. When a is an integer and b a positive integer, $a \div b$ and $a \,\%\, b$ denote the integer quotient and the remainder between a and b, respectively. The standard modules also define the set of natural numbers Nat (as the set "CHOOSE a set satisfying Peano's axioms") and the interval between two integer numbers $a \mathinner{..} b \triangleq \{n \in Int : a \leq n \land n \leq b\}$, where $a \leq b$ is defined as $a = b \lor a < b$.

Miscellaneous constructs Finally, TLA$^+$ includes conditional expressions, strings, tuples and records. The syntactic category c represents single characters and s represents strings.

$$\begin{aligned} e ::= {} & \cdots \\ & \mid \text{IF } e \text{ THEN } e \text{ ELSE } e \\ & \mid \text{CASE } e \to e \,\Box\, \cdots \,\Box\, e \to e \mid \text{CASE } e \to e \,\Box\, \cdots \,\Box\, e \to e \,\Box\, \text{OTHER } e \\ & \mid \langle e, \ldots, e \rangle \mid e \times \ldots \times e \\ & \mid \text{“}c \ldots c\text{”} \\ & \mid e.s \mid [s \mapsto e, \ldots, s \mapsto e] \mid [s : e, \ldots, s : e] \end{aligned}$$

The semantics of the conditional constructs IF-THEN-ELSE and CASE are formally defined in terms of CHOOSE:

$$\begin{aligned} &\text{IF } c \text{ THEN } e_1 \text{ ELSE } e_2 \triangleq \\ &\quad \text{CHOOSE } v : (c \Rightarrow (v = e_1)) \land (\neg c \Rightarrow (v = e_2)) \end{aligned} \tag{2.14}$$

$$\begin{aligned} &\text{CASE } c_1 \to e_1 \,\Box\, \cdots \,\Box\, c_n \to e_n \triangleq \\ &\quad \text{CHOOSE } v : (c_1 \land (v = e_1)) \lor \cdots \lor (c_n \land (v = e_n)) \end{aligned} \tag{2.15}$$

$$\begin{aligned} &\text{CASE } c_1 \to e_1 \,\Box\, \cdots \,\Box\, c_n \to e_n \,\Box\, \text{OTHER } e \triangleq \\ &\quad \text{CASE } c_1 \to e_1 \,\Box\, \cdots \,\Box\, c_n \to e_n \,\Box\, \neg(c_1 \lor \cdots \lor c_n) \to e \end{aligned} \tag{2.16}$$

In TLA$^+$, an n-tuple $\langle e_1, \ldots, e_n \rangle$ is a function whose domain is the interval of integer numbers from 1 to n, where $\langle e_1, \ldots, e_n \rangle[i] = e_i$, for $1 \leq i \leq n$. Thus, the function application construct $\_[\_]$ is used for tuple projection as well.

$$\langle e_1, \ldots, e_n \rangle \triangleq [y \in 1..n \mapsto \textsc{case}\ (y = 1) \to e_1 \ \square \ldots \ \square\ (y = n) \to e_n] \tag{2.17}$$

$$S_1 \times \ldots \times S_n \triangleq \{\langle y_1, \ldots, y_n \rangle : y_1 \in S_1, \ldots, y_n \in S_n\} \tag{2.18}$$

The 0-tuple $\langle\,\rangle \triangleq [x \in \{\} \mapsto \{\}]$ is the unique function having an empty domain.

TLA$^+$ defines a string to be a tuple of characters. Although there is no special syntax for writing characters, the TLA$^+$ semantics does specify that characters are different from one another.

Records are functions whose domain is a finite set of strings, representing the record fields. Record selection $r.h$ is a shorthand for $r[\text{“}\mathsf{h}\text{”}]$, for any expression r and field identifier h. The following rules define explicit record construction and the set of records, respectively.

$$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \triangleq [y \in \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \mapsto \textsc{case}\ (y = \text{“}\mathsf{h_1}\text{”}) \to e_1 \ \square \ldots \ \square\ (y = \text{“}\mathsf{h_n}\text{”}) \to e_n] \tag{2.19}$$

$$[h_1 : S_1, \ldots, h_n : S_n] \triangleq \{[h_1 \mapsto y_1, \ldots, h_n \mapsto y_n] : y_1 \in S_1, \ldots, y_n \in S_n\} \tag{2.20}$$

2.2. Specifications

In order to give a better understanding of the nature of TLA$^+$ proofs described in the following section, we show how does a TLA$^+$ specification and its properties look like by way of a small well-known example. Our case study is the proof that Peterson's algorithm [Pet81] implements mutual exclusion, meaning that no two processes are in their critical sections at the same time.

In order to make the algorithm easier to assimilate for a TLA$^+$ novice, we present it first in PlusCal [Lam08], a high-level algorithm language that is based on TLA$^+$. The actual TLA$^+$ specification is described in the following sub-section. The PlusCal code is written inside a comment in a TLA$^+$ module. The variables and operators declared, defined, or imported in that module can be freely used in the PlusCal code. The PlusCal translator, accessible through a Toolbox menu, parses the PlusCal code and automatically generates a TLA$^+$ specification, which is what we reason about.

```
--algorithm Peterson {
  variables flag = [i ∈ {0,1} ↦ FALSE], turn = 0;
  process (proc ∈ {0,1}) {
      a0: while (TRUE) {
      a1:   flag[self] := TRUE;
      a2:   turn := Not(self);
     a3a:   if (flag[Not(self)]) {goto a3b} else {goto cs};
     a3b:   if (turn = Not(self)) {goto a3a} else {goto cs};
      cs:   skip; \* critical section
      a4:   flag[self] := FALSE;
          } \* end while
  } \* end process
} \* end algorithm
```

Figure 2.1.: Peterson's algorithm in PlusCal

Peterson's algorithm allows two processes, called 0 and 1, to share a single resource —the access to the critical section— while communicating only through shared memory, represented by global variables. The algorithm's code in PlusCal is shown in Figure 2.1 and should be easy enough for the reader to figure out its meaning with the following brief explanation. The operator Not is defined in the TLA$^+$ module as

$$Not(i) \triangleq \text{IF } i = 0 \text{ THEN } 1 \text{ ELSE } 0$$

The variables statement declares the global variables and their initial values. To specify a multiprocess algorithm, it is necessary to specify what its atomic actions are. In PlusCal, an execution from one label to the next constitutes an atomic action, which in turn corresponds to exactly one TLA$^+$ action. Comments are written between the characters * and the end of the line.

2.2.1. Specification structure

The TLA$^+$ specification of an algorithm or a computer system is represented by a single temporal formula, often named $Spec$,[5] stating a predicate on behaviors. A *behavior* is an infinite sequence of states (i.e. an assignment of values to variables), with the *steps* of a behavior being its successive pairs of states. The core of a specification consists of a predicate that describes the initial state, traditionally named

[5] A typical TLA$^+$ specification follows the structure given by $Spec$, as in the example's translation. The language and the back-end provers do not impose any restriction in the way one constructs specifications but, by convention and common practice, it is convenient to follow $Spec$'s structure. Moreover, for a language as expressive as TLA$^+$, it is difficult to have verification tools that can cope with any possible formula of the language.

$$
\begin{array}{l}
\textsc{variables}\ \mathit{flag},\ \mathit{turn},\ \mathit{pc} \\
\mathit{vars} \triangleq \langle \mathit{flag},\ \mathit{turn},\ \mathit{pc} \rangle \\
\\
\mathit{Init} \triangleq \land\ \mathit{flag} = [i \in \{0,\,1\} \mapsto \textsc{false}] \\
\qquad\quad \land\ \mathit{turn} = 0 \\
\qquad\quad \land\ \mathit{pc} = [\mathit{self} \in \{0,\,1\} \mapsto \text{“a0”}] \\
\\
a0(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a0”} \\
\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a1”}] \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{flag},\ \mathit{turn} \rangle \\
\\
a1(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a1”} \\
\qquad\quad \land\ \mathit{flag}' = [\mathit{flag}\ \textsc{except}\ ![\mathit{self}] = \textsc{true}] \\
\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a2”}] \\
\qquad\quad \land\ \mathit{turn}' = \mathit{turn} \\
\\
a2(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a2”} \\
\qquad\quad \land\ \mathit{turn}' = \mathit{Not}(\mathit{self}) \\
\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a3a”}] \\
\qquad\quad \land\ \mathit{flag}' = \mathit{flag} \\
\\
a3a(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a3a”} \\
\qquad\quad \land\ \textsc{if}\ \mathit{flag}[\mathit{Not}(\mathit{self})] \\
\qquad\qquad\quad \textsc{then}\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a3b”}] \\
\qquad\qquad\quad \textsc{else}\ \ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“cs”}] \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{flag},\ \mathit{turn} \rangle \\
\\
a3b(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a3b”} \\
\qquad\quad \land\ \textsc{if}\ \mathit{turn} = \mathit{Not}(\mathit{self}) \\
\qquad\qquad\quad \textsc{then}\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a3a”}] \\
\qquad\qquad\quad \textsc{else}\ \ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“cs”}] \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{flag},\ \mathit{turn} \rangle \\
\\
cs(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“cs”} \\
\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a4”}] \\
\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{flag},\ \mathit{turn} \rangle \\
\\
a4(\mathit{self}) \triangleq \land\ \mathit{pc}[\mathit{self}] = \text{“a4”} \\
\qquad\quad \land\ \mathit{flag}' = [\mathit{flag}\ \textsc{except}\ ![\mathit{self}] = \textsc{false}] \\
\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“a0”}] \\
\qquad\quad \land\ \mathit{turn}' = \mathit{turn} \\
\\
\mathit{proc}(\mathit{self}) \triangleq a0(\mathit{self}) \lor a1(\mathit{self}) \lor a2(\mathit{self}) \lor a3a(\mathit{self}) \lor a3b(\mathit{self}) \lor cs(\mathit{self}) \lor a4(\mathit{self}) \\
\\
\mathit{Next} \triangleq \exists\, \mathit{self} \in \{0,\,1\} : \mathit{proc}(\mathit{self}) \\
\\
\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

Figure 2.2.: Specification of Peterson's algorithm, translated from PlusCal code of Figure 2.1 (slightly simplified and pretty-printed version)

$Init$, and an action formula, named $Next$, that states a *next-state* relationship between unprimed and primed variables, describing how the states can change.

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

Specifically, the behaviors satisfying $Spec$ are the sequences starting in a state satisfying $Init$, and where each of the sequences' steps either satisfies $Next$ or else leaves the values of the declared variables $vars$ unchanged. The symbol $\Box$ is the ordinary *always* operator of linear-time temporal logic, and $[Next]_{vars}$ abbreviates $Next \vee \text{UNCHANGED } vars$. In turn, UNCHANGED $vars$ abbreviates $vars' = vars$, and a primed expression such as $vars'$ denotes a copy of the expression in which all state variables v are replaced by primed state variables v'. Leaving all variables unchanged allows "stuttering steps", which makes refinement and composition of specifications simpler.[6]

Figure 2.2 shows the generated TLA$^+$ translation for the PlusCal code of Figure 2.1. The variable pc (from "program control") is added by the PlusCal translator to explicitly record the point of execution of the program, specified by the labels, for each process. For example, the process i is executing code at the label cs iff $pc[i]$ equals the string "cs". The allowed transitions are stated by the action formulas $proc(0)$ and $proc(1)$, where each $proc(self)$ is the disjunction of seven formulas: one for each label in the body of the process. The formula $a0(self)$ specifies the state change performed by process $self$ executing an atomic action starting at label $a0$, and similarly for the other six labels.

2.2.2. Safety properties and invariants

In our running example, the goal is to prove that Peterson's algorithm actually satisfies mutual exclusion. This property states that the two processes never both have control at label cs, as expressed by the formula:

$$MutualExclusion \triangleq (pc[0] \neq \text{"cs"}) \vee (pc[1] \neq \text{"cs"})$$

The predicate $MutualExclusion$ is an invariant of the algorithm if and only if it is true in all reachable states. The assertion that Peterson's algorithm implements mutual exclusion is formalized in TLA$^+$ as:

$$\text{THEOREM } Spec \Rightarrow \Box MutualExclusion$$

The keyword THEOREM asserts a conjecture which can be optionally followed by a TLA$^+$ proof. TLA$^+$ also provides the keyword AXIOM to assert assumptions.

The standard method for proving this invariance property is to find an inductive invariant Inv that implies $MutualExclusion$. An inductive invariant is one that is

[6] The described *Spec* formula asserts only safety properties. In order to prove liveness properties, we would have to conjoin *fairness conditions* to *Spec*.

true in the initial state and whose truth is preserved by the next-state relation. The inductive invariant Inv is defined as the conjunction of two formulas:

$$Inv \triangleq TypeOK \land I$$

The conjunct $TypeOK$ is a *type-correctness* invariant, asserting that all relevant variables have values of the expected "types", or, more precisely, their values are elements of the expected sets.

$$\begin{aligned} TypeOK \triangleq\ & \land\ pc \in [\,\{0,1\} \to \{\text{“a0”}, \text{“a1”}, \text{“a2”}, \text{“a3a”}, \text{“a3b”}, \text{“cs”}, \text{“a4”}\}\,] \\ & \land\ turn \in \{0,1\} \\ & \land\ flag \in [\,\{0,1\} \to \textsc{boolean}\,] \end{aligned}$$

In TLA$^+$, type-correctness is not imposed by the language; it is just another property of the system's specification formula. In practice, it is customary that the first thing the user does after declaring the variables in a TLA$^+$ module is to write the type invariant for every declared variable occurring in the specification. Once proved, this invariant is used as a hypothesis in other theorems.

The type invariant provides valuable information about the variables: their values belong to a certain set that remains constant throughout all reachable states of the system specification. The "types" of a TLA$^+$ type invariant can go beyond the customary types of standard programming languages, due to the flexibility of set theory. In Chapter 5, we will use this information to infer types (in the traditional sense) for the variables.

The second conjunct, I, is the interesting one that explains why Peterson's algorithm implements mutual exclusion.

$$\begin{aligned} I \triangleq\ & \forall i \in \{0,1\}: \\ & \quad \land\ pc[i] \in \{\text{“a2”}, \text{“a3a”}, \text{“a3b”}, \text{“cs”}, \text{“a4”}\} \Rightarrow flag[i] \\ & \quad \land\ pc[i] \in \{\text{“cs”}, \text{“a4”}\} \Rightarrow \land\ pc[Not(i)] \notin \{\text{“cs”}, \text{“a4”}\} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \land\ pc[Not(i)] \in \{\text{“a3a”}, \text{“a3b”}\} \Rightarrow turn = i \end{aligned}$$

A central part of any proof development is devoted to the discovery of inductive invariants. The user writing the proof should be knowledgeable enough about the algorithm or the system he modeled to come up with the right formulas. This is a time-consuming and non-trivial process, and it may require many iterations, strengthening the invariant formulas until obtaining one that is inductive. Although an important subject, it is not relevant for this thesis, as our focus is on proving, and not finding, these kind of properties.

2.3. Proofs

The TLA$^+$ specification language was extended by a notation for writing hierarchical proofs [Lam95, CDLM08, CDL$^+$]. Specifications, their properties, and proofs

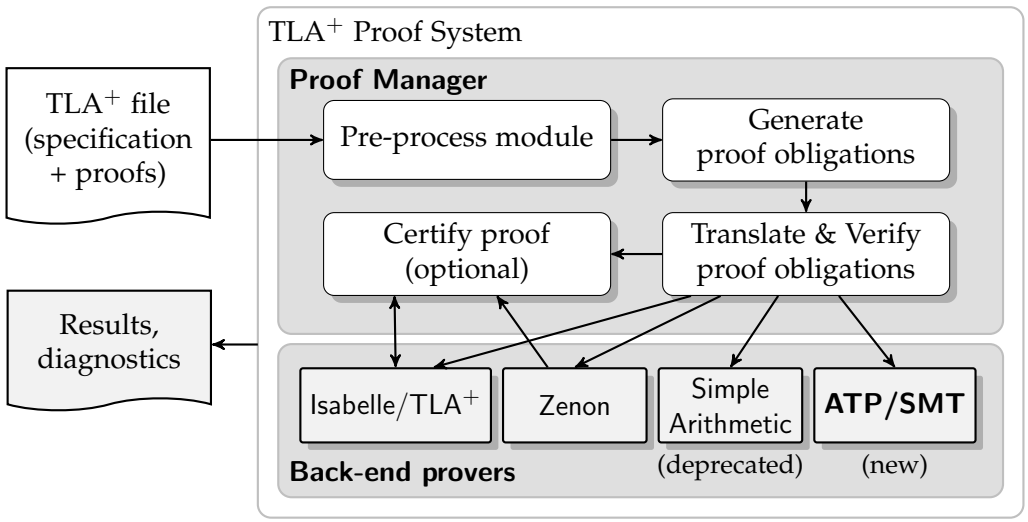


Figure 2.3.: TLA$^+$ Proof System architecture

are built on top of the same core language with the aim of building a homogeneous verification platform. This idea was realized in the tool TLA$^+$ Proof System (TLAPS) [CDLM10], an interactive proof environment in which users can deductively verify properties of TLA$^+$ specifications. TLAPS can be used from within the TLA$^+$ Toolbox, an Eclipse-based framework for the development of TLA$^+$ specifications, that provides tools like the TLC model checker and the PlusCal translator.

As in most interactive theorem provers, automatic theorem provers underpin TLAPS. Figure 2.3 illustrates a schematic view of its architecture: TLAPS is built around an application called Proof Manager, which interprets the proofs occurring in the TLA$^+$ module provided by the user, expands the necessary module and operator definitions, distributes the priming operator, generates corresponding proof obligations, and passes them to external automated verifiers, which are integrated to TLAPS as *back-end provers*.

2.3.1. Back-end provers

At the time of starting this work, there were three main backends available in TLAPS:

- Isabelle/TLA$^+$, a faithful encoding of the logic of TLA$^+$ in the Isabelle proof assistant [WPN08], which provides automated proof methods based on first-order reasoning and rewriting.
- Zenon [BDD07], a tableau prover for first-order logic with equality that includes extensions for TLA$^+$ for reasoning about sets and functions. It does not

support arithmetic, but Hilbert's ε operator, corresponding to TLA$^+$'s CHOOSE, is part of its core logic.

- A back-end called SimpleArithmetic, now deprecated, that implemented a decision procedure (Cooper's algorithm) for (quantifier-free) linear integer arithmetic, also known as Presburger arithmetic.

Beyond its integration as a semi-automatic back-end, Isabelle/TLA$^+$ serves as the most trusted back-end prover. Accordingly, Isabelle/TLA$^+$ is used as well for certifying proof scripts produced by other back-end provers. As a consequence, the precise TLA$^+$ semantics implemented by the other back-end provers should follow exactly the definitions in Isabelle/TLA$^+$. When possible, back-end provers are expected to produce a detailed Isar proof script that can be checked by Isabelle/TLA$^+$. Currently, only the Zenon back-end exports certifiable proofs as Isar scripts.

In this thesis we have developed a new back-end prover for ATP systems and SMT solvers. The back-ends available prior to the work presented here also included a generic translation to the input language of SMT solvers that focused on quantifier-free formulas of linear arithmetic (not shown in Figure 2.3). This SMT back-end was occasionally useful because the other back-ends perform quite poorly on obligations involving arithmetic reasoning. However, it covered a rather limited subset of TLA$^+$, namely a fragment of first-order logic and linear arithmetic.

At the time of writing this work, a newly developed back-end for propositional temporal logic (PTL) was released [DKL$^+$14]. It can discharge formulas containing simple propositional and temporal logic but, for the moment, it is not good at reasoning about complex liveness properties.

2.3.2. Proof language

The TLA$^+$ proof language is hierarchical and declarative. *Hierarchical* proofs correspond to a natural deduction proof tree of TLA$^+$ sequents with local scopes. They enable a user to decompose a complex proof into smaller steps until they become provable by one of the available back-end provers. Unlike other interactive proof assistants [Wie06], TLAPS has been designed around a proof language that is independent of any specific proof back-end. Thus, we say that the proof language is *declarative*, in the sense that leaf proof steps may indicate what facts are needed to prove a certain goal, but it does not allow the user to specify the exact procedure to be followed by the backends.

A proof step includes a *goal* formula to be proved and a local *context* that determines the goal's validity, which in turn may contain constant and variable symbol declarations, assumptions, and already established facts. The Proof Manager tracks contexts, which are modified by the non-leaf steps, and generates proof obligations

corresponding to each leaf proof. A *proof obligation* is a self-contained sequent formula composed of the proof goal and its local context. TLA$^+$ sequents are of the form

$$\text{ASSUME } h_1, \ldots, h_n \text{ PROVE } c,$$

where $h_1, \ldots, h_n$ are TLA$^+$ expressions, sequents, or declarations of new constant and variable symbols, and where c is an expression. In a sequent, it is possible to introduce new operator symbols of arity greater than zero. However, we only support second-order operators in the first-level sequent, where they can be introduced to the context as Skolem functions.

Non-leaf proof steps

TLA$^+$ proofs are generally written top-down. Each proof in the hierarchy ends with a QED step that asserts the goal of that proof; the QED step for the top level asserts the statement of the theorem (unless changed by other non-leaf steps such as SUFFICES in the same level, as explained below). Because proof obligations are independent of one another, steps can be written and checked in any order. Yet, the QED step is usually proved first to be sure that the unproved steps that precede it are sufficient for establishing the validity of the proof.

Figure 2.4 presents the correctness proof of mutual exclusion for Peterson's algorithm. In this proof, the first-level QED step follows easily from steps $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and $\langle 1 \rangle 3$, and by using the PTL back-end, which can successfully handle trivial temporal formulas such as this one. Step $\langle 1 \rangle 2$ asserts that the truth of Inv is preserved by the next-state relation. Its proof continues in the usual hierarchical style. For a complete description of the TLA$^+$ proof language, see [CDLM08]; in the following we briefly describe the proof constructs used in the example, which are anyway the most commonly used in TLA$^+$ proofs. SUFFICES changes the goal by reducing the proof of the current assertion to that of a supposedly simpler one. For instance, step $\langle 2 \rangle 1$ asserts that to prove the current goal, it suffices to assume that Inv and $[Next]_{vars}$ are true and then to prove Inv'. The step PICK $x \in S : P(x)$ performs $\exists$-elimination by introducing to the context a new variable $x \in S$ satisfying $P(x)$. In our example, step $\langle 3 \rangle 2$ refers to the facts $i \in \{0, 1\}$ and $proc(i)$. The proof construct CASE splits the current goal into separate proof cases. In the example, the step identifier $\langle 3 \rangle 3$ used as a fact refers to the assumption $i = j$.

"Obvious" proof steps

A user works on a proof splitting it into cases and refining intermediate steps, until a point where he considers that the leaf steps are *obvious*, that is, from his perspective, the leaf steps are simple enough that they can be discharged by the automated back-end provers provided by TLAPS. The Proof Manager does not expand the definitions

```
THEOREM Spec ⇒ □MutualExclusion
⟨1⟩1. Init ⇒ Inv
  BY DEFS Init, Inv, TypeOK, I
⟨1⟩2. Inv ∧ [Next]_vars ⇒ Inv'
  ⟨2⟩1. SUFFICES ASSUME Inv, Next
                 PROVE  Inv'
    BY DEFS vars, Inv, TypeOK, I
  ⟨2⟩2. TypeOK'
    BY ⟨2⟩1 DEFS Inv, TypeOK, Next, proc, a0, a1, a2, a3a, a3b, cs, a4, Not
  ⟨2⟩3. I'
    ⟨3⟩1. SUFFICES ASSUME NEW j ∈ {0, 1}
                   PROVE   I!(j)'
      BY DEF I
    ⟨3⟩2. PICK i ∈ {0, 1} : proc(i)
      BY ⟨2⟩1 DEF Next
    ⟨3⟩3. CASE i = j
      BY ⟨2⟩1, ⟨3⟩2, ⟨3⟩3 DEFS Inv, I, TypeOK, proc, a0, a1, a2, a3a, a3b, cs, a4, Not
    ⟨3⟩4. CASE i ≠ j
      BY ⟨2⟩1, ⟨3⟩2, ⟨3⟩4 DEFS Inv, I, TypeOK, proc, a0, a1, a2, a3a, a3b, cs, a4, Not
    ⟨3⟩5. QED
      BY ⟨3⟩3, ⟨3⟩4
  ⟨2⟩4. QED
    BY ⟨2⟩2, ⟨2⟩3 DEF Inv
⟨1⟩3. Inv ⇒ MutualExclusion
  BY DEFS MutualExclusion, Inv, I, Not
⟨1⟩4. QED
  BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, PTL
```

Figure 2.4.: Proof of mutual exclusion of Peterson's algorithm using only Isabelle/TLA$^+$ and Zenon as back-end provers

occurring in a proof obligation unless directed to do so. The definitions and facts must be cited explicitly by the user. The proof construct to achieve this is

```
BY hs DEFS ds
```

which allows the user (i) to add to the leaf proof's contexts a list hs of facts (including hypotheses assumed in the current context), and (ii) to expand the definitions of the operators given in the list ds. In this way, the user has some control about the size of the search space that the back-end provers will need to handle. The OBVIOUS proof construct is a shorthand for BY TRUE, that is, it simply invokes the default back-end prover in the current goal with its current context.

TLAPS uses Zenon and Isabelle as its default back-end provers, first trying Zenon and then trying Isabelle if Zenon fails to find a proof. The user still has the possibility

to bypass the default behavior in order to invoke a specific verifier, or to give a longer timeout to a desired back-end prover. For this purpose, special pragma identifiers can be added to the list of facts hs. Some common options are *Zenon* and *Isa* to call Zenon and the Isabelle/TLA$^+$ back-ends. Their respective parameterized directives *ZenonT(t)* and *IsaT(t)* instruct the Proof Manager to give the provers a timeout of t seconds. With the introduction of the new back-end prover, the user now has the option to call the default ATP system with ATP, or the default SMT solver with SMT. Pragmas to invoke specific provers with a specific timeout also exist.

Example The following sequent is the proof obligation that the Proof Manager generates from step $\langle 3 \rangle 3$ in our running example. The actual formula is too large to display it in a single page, so we show only the formula where the definitions of Inv, $proc$, $a0$, and Not are unexpanded.

$$
\begin{array}{ll}
\textsc{assume} & \textsc{new variable}\ flag, \\
& \textsc{new variable}\ turn, \\
& \textsc{new variable}\ pc, \\
& \textsc{new constant}\ j \in \{0, 1\}, \\
& \textsc{new constant}\ i \in \{0, 1\}, \\
& Inv, \\
& proc(i), \\
& i = j \\
\textsc{prove} & \land\ pc'[j] \in \{\text{“a2”}, \text{“a3a”}, \text{“a3b”}, \text{“cs”}, \text{“a4”}\} \Rightarrow flag'[j] \\
& \land\ pc'[j] \in \{\text{“cs”}, \text{“a4”}\} \Rightarrow \\
& \qquad \land\ pc'[Not(j)] \notin \{\text{“cs”}, \text{“a4”}\} \\
& \qquad \land\ pc'[Not(j)] \in \{\text{“a3a”}, \text{“a3b”}\} \Rightarrow turn = j
\end{array}
$$

The symbols 0 and 1 are treated as constants, and no reasoning about arithmetic is required in this case, except the fact $0 \neq 1$. The expanded version of this self-contained formula is exactly what the Proof Manager sends to the back-end provers —in this case to Zenon, that proves it in a few seconds.

In summary, the Proof Manager transforms a proof into a collection of proof obligations to be verified by back-end provers. When a prover fails to find a proof, the user may need to examine the failed proof obligation, and then, either to invoke another specialized back-end prover, or to create another level in the proof tree to decompose the current assertion into sub-steps, simplifying the proof search space. Even if we limit ourselves to non-temporal formulas, TLAPS is able to handle the verification of safety properties. Non-temporal proof obligations are usually “shallow”, but they require many details to be checked. Interactive proofs can become quite large without powerful back-end provers that can cope with a significant fragment of the language.

Chapter 3.

Automated theorem proving

Contents

> [...] nos esse quasi nanos, gigantium humeris incidentes [...]
>
> Bernard of Chartres, as cited by John of Salisbury, 1159.

In this chapter we summarize some essential concepts of automated theorem proving (ATP), including the main formalisms in which the problems are expressed, the calculus they implement, and two important standard input formats in the ATP community. The goal of an automated theorem prover is to decide the validity of a conjecture formula ϕ by deducing ϕ as a logical consequence of a set of hypothesis. Or, equivalently, to show that the set of hypotheses together with the negation of ϕ is inconsistent, giving a proof by refutation. If a proof does not exist, the prover would ideally report a model or counter-example for ϕ.

Chapter overview Section 3.1 describes CNF, FOL and MS-FOL, the logical languages that underpin most first-order automated theorem provers like ATP systems and SMT solvers, of which we give some insights of their internal workings in Sections 3.2. Section 3.3 presents the input formats TPTP for ATP systems, and SMT-LIB for SMT solvers.

3.1. Unsorted and many-sorted logics

In this section we describe the syntax and semantics of first-order logic (FOL), first-order and propositional clause normal form (CNF), and many-sorted first-order logic (MS-FOL).

3.1.1. First-order logic

The language FOL is the classical (unsorted) first-order logic with equality.

FOL syntax We assume given three non-empty, infinite, and disjoint collections

- $\mathcal{V}$ of *variable* symbols,
- $\mathcal{F}$ of *function* symbols,[1] and
- $\mathcal{P}$ of *predicate* symbols.

In addition, we assume the total function

- $ar : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ that assigns a natural number, the *arity*, to each symbol in $\mathcal{F}$ and $\mathcal{P}$.

Together, they define the FOL signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{P}, ar \rangle$ that fixes an alphabet of non-logical symbols. Nullary function (respectively predicate) symbols are called *constants* (respectively *propositional variables*).

FOL has two syntactical categories: terms t and formulas ϕ.

$$t ::= x \mid f(t, \ldots, t)$$
$$\phi ::= \bot \mid \phi \Rightarrow \phi \mid \forall x. \phi \mid t = t \mid p(t, \ldots, t)$$

A *term* t is either a variable symbol x in $\mathcal{V}$, or an arity-consistent application of a function symbol f in $\mathcal{F}$ to terms. A term not containing any variable is called a *ground* term. A *formula* ϕ is either the falsehood symbol $\bot$, an implication between two formulas, a universal quantification of a formula, an equality between terms, or an arity-consistent application of a predicate symbol p in $\mathcal{P}$ to terms. The last two kind of formulas are called *atoms*.

Additionally, we can define the familiar constant $\top$ (truth), the connectives $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, and the existential quantifier $\exists$. In particular, universal and existential quantifiers can be freely nested. The definitions of free variables and variable substitution are the standard ones. By $FV(\phi)$, we note the free variables of a formula ϕ, and by $\phi_1[x \leftarrow \phi_2]$, the formula ϕ_1 where the free variable x is substituted by the formula ϕ_2.

[1] CNF, FOL, and MS-FOL functions should not to be confused with TLA$^+$ functions, see Section 2.1.

FOL semantics A FOL *model* or *structure* $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$ is composed of

- a non-empty set $\mathcal{D}$ called the *domain* or *universe of discourse*,
- a *valuation* function $v : \mathcal{V} \to \mathcal{D}$ that assigns to each variable an element in the domain, and
- an *interpretation* function $\mathcal{I}$ that assigns to each function symbol f in $\mathcal{F}$ a function $\mathcal{I}(f) : \mathcal{D}^{ar(f)} \to \mathcal{D}$, and to each predicate symbol p in $\mathcal{P}$ a set $\mathcal{I}(p) \subseteq \mathcal{D}^{ar(f)}$.

The interpretation of first-order formulas is defined in the standard way [End01]. The valuation of a term t under a model $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$, noted $val_M(t)$, is defined by:

$$val_M(x) = v(x)$$
$$val_M(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_M(t_1), \ldots, val_M(t_n))$$

The truth value of a formula ϕ under a model $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$, noted $M \models \phi$ or $\mathcal{D}, v, \mathcal{I} \models \phi$, is inductively defined as:

$$\begin{array}{ll} M \not\models \bot & \\ M \not\models \phi_1 \Rightarrow \phi_2 & \text{iff } M \models \phi_1 \text{ and } M \not\models \phi_2 \\ M \models \forall x.\, \phi & \text{iff } \mathcal{D}, v \oplus (x \mapsto d), \mathcal{I} \models \phi \text{ for all } d \in \mathcal{D} \\ M \models t_1 = t_2 & \text{iff } val_M(t_1) \text{ is equal to } val_M(t_2) \\ M \models p(t_1, \ldots, t_n) & \text{iff } \langle val_M(t_1), \ldots, val_M(t_n) \rangle \text{ is a member of } \mathcal{I}(p) \end{array}$$

where $v \oplus (x \mapsto d)$ denotes a valuation that is equal to v for all variables in $\mathcal{V}$ except in x, and mapping x to d. We say that a formula ϕ is *valid in a model* M iff $M \models \phi$ holds. A formula ϕ is *valid*, noted $\models \phi$, iff $M \models \phi$ for all models M, that is, for every domain, valuation, and interpretation. A formula ϕ is called *satisfiable* iff there exists a model M such that $M \models \phi$. Otherwise, ϕ is called *unsatisfiable*.

3.1.2. Clausal normal form

First-order clause normal form (CNF) is a fragment of FOL where quantifier-free FOL formulas are represented in conjunctive normal form as a set of *clauses*.

A CNF signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{P}, ar \rangle$ is defined in the same way as in FOL. The language CNF has four syntactical categories: terms t, atoms a, literals ℓ, and clauses C.

$$t ::= x \mid f(t, \ldots, t) \qquad a ::= t = t \mid p(t, \ldots, t)$$
$$\ell ::= a \mid \neg a \qquad C ::= \ell \vee \ldots \vee \ell$$

Terms and atoms have the same definitions as FOL terms and atoms. A *literal* ℓ is either an atom (a positive literal) or the negation of an atom (a negative literal).

A *clause* C is a set of literals that represents the disjunction of its elements. Thus, the empty clause $\bot$ is interpreted as falsehood. A clause with one literal is called a *unit clause*. A CNF formula ϕ is a set of clauses $C_1, \ldots, C_n$, where variables are interpreted universally.

Propositional CNF At their core, most automated theorem provers handle formulas in the propositional fragment. A *propositional* CNF formula is a ground, i.e. variable-free, first-order CNF formula. The following definitions are used below to describe the DPLL algorithms [NOT06]. A model M in propositional CNF is a truth assignment (possibly partial) represented as sequence of literals, where each literal ℓ or its negation $\neg\ell$ can occur only once in a model. The *negation* of a literal ℓ, written $\neg\ell$, denotes $\neg a$ if ℓ is the atom a, and a if ℓ is $\neg a$. A literal ℓ is true in M (noted $\ell \in M$) iff ℓ occurs in M; it is false iff $\neg\ell$ occurs in M ($\neg\ell \in M$); otherwise it is undefined. A clause $C \triangleq \ell_1 \vee \ldots \vee \ell_n$ is true in M ($M \models C$) iff $\ell_i \in M$ for some i in $1 \,..\, n$. We say that a clause C is *conflicting* in model M ($M \not\models C$) when M evaluates C to false. A formula $\phi \triangleq C_1, \ldots, C_n$ is true or satisfied by M ($M \models \phi$) iff $M \models C_i$ for all i in $1 \,..\, n$. If no model satisfies ϕ, then ϕ is *unsatisfiable*. We write $C_1, \ldots, C_n \models C$ when C is true in all models of $C_1, \ldots, C_n$.

Given the standard interpretation for the propositional CNF symbols, the Boolean satisfiability (SAT) problem for a propositional formula ϕ is to decide if there exists a assignment M such that $M \models \phi$. Validity is the dual of unsatisfiability: a formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable. Thus, a theorem prover, which is a validity checker, just needs to check for the unsatisfiability of the formula or, equivalently, to derive the empty clause $\bot$ from the given clauses using some appropriate calculus, such as semantic tableaux or one of the variants of the DPLL algorithm described below.

3.1.3. Many-sorted logic

In logical formalizations, it is often natural to categorize objects in many different classes, or sorts. For example, when formalizing geometry it is common to think about points and lines as two different categories. The logic underlying SMT solvers, many-sorted first-order logic (MS-FOL) [End01], offers this possibility at the expense of some expressiveness [Coh87], but bringing significant benefits to automated theorem provers: the type discipline implicitly avoids pointless inferences by not having to evaluate ill-sorted formulas. Sorts cut the search space by eliminating useless branches, even from infinite to finite spaces [FRZ04].

The MS-FOL language extends FOL by introducing an extra syntactical category of sorts, i.e. basic types. The set $\mathcal{V}$ of variable symbols is partitioned by sorts, while functions and predicates range over sorts as well.

MS-FOL syntax We assume given four non-empty enumerable infinite, and disjoint collections

- $\mathcal{S}$ of *atomic sort* symbols,
- $\mathcal{V} = \bigcup_{\sigma \in \mathcal{S}} \mathcal{V}_\sigma$, the (enumerable) union of sets $\mathcal{V}_\sigma$ of *variable* symbols of sort σ,
- $\mathcal{F}$ of *function* symbols, and
- $\mathcal{P}$ of *predicate* symbols.

In addition, we assume given the functions

- $ar : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ assigning an *arity* to each symbol in $\mathcal{F}$ and $\mathcal{P}$, and
- $\theta : \mathcal{F} \cup \mathcal{P} \to \mathcal{S}^*$ assigning to functions f in $\mathcal{F}$ a value in $\mathcal{S}^{ar(f)+1}$ (the Cartesian product with $ar(f)+1$ dimensions), and to predicates p in $\mathcal{P}$ a value in $\mathcal{S}^{ar(p)}$.

Together, they define a MS-FOL signature $\langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, ar, \theta \rangle$.[2]

The language MS-FOL has three syntactical categories: sorts, (well-sorted) terms, and formulas. A *sort* σ is just an atomic sort symbol in $\mathcal{S}$. Note that the values $\theta(f)$ and $\theta(p)$ associated to a function symbol f and a predicate symbol p, when $ar(p) > 1$, are no sorts.

$$t ::= x \mid f(t, \ldots, t)$$
$$\phi ::= \bot \mid \phi \Rightarrow \phi \mid \forall x^\sigma . \phi \mid t = t \mid p(t, \ldots, t)$$

A *term* t of sort σ is either a variable symbol x in $\mathcal{V}_\sigma$, or a sort-consistent application of a sorted function symbol f with $\theta(f) = \langle \sigma_1, \ldots, \sigma_n, \sigma \rangle$ in $\mathcal{F}$ to terms of sort $\sigma_1, \ldots, \sigma_n$. A *formula* ϕ is built as in unsorted FOL except that quantified variables are annotated with a sort, equality is ad-hoc polymorphic over the sorts (equalities between terms that have different sorts are ill-sorted), and predicates applied to terms must be sort-consistent, that is, a predicate symbol p in $\mathcal{P}$ with $\theta(p) = \langle \sigma_1, \ldots, \sigma_n \rangle$ should be applied to terms of sort $\sigma_1, \ldots, \sigma_n$.

MS-FOL semantics The semantics of MS-FOL is analogous to FOL with the obvious modifications corresponding to the presence of multiple sorts. Hence, the definitions are more technically involved [End01, Man05]. That said, the sort machinery is in principle not needed because sorts can be encoded using unary predicates by relativizing quantified sorted variables. *Relativization* is the traditional method to encode a multi-sorted language into a single-sorted one [End01, Man05]. For every atomic sort $\sigma \in \mathcal{S}$, a *characteristic predicate* P_σ that represents the set of values having that sort. A MS-FOL formula is relativized by systematically replacing the quantified sorted formulas $\forall x^\sigma . \phi$ by a FOL formula $\forall x . P_\sigma(x) \Rightarrow \phi$, where P_σ is

[2] The arity of a function or predicate symbol f can be recovered from $\theta(f)$ but we keep it in the signature to maintain MS-FOL as an extension of FOL.

introduced to the set $\mathcal{P}$ as a fresh unary predicate symbol, for each sort σ. Moreover, these predicates partition the universe of atomic sorts in disjoint sets. For each pair of sorts $\langle \sigma_1, \sigma_2 \rangle$, the encoding introduces an extra axiom

$$\forall x, y.\, P_{\sigma_1}(x) \wedge P_{\sigma_2}(y) \Rightarrow \neg(x = y).$$

Lemma 1 (Relativization is sound). $\models \forall x^{\sigma}.\, \phi$ *implies* $\models \forall x.\, P_{\sigma}(x) \Rightarrow \phi$ *[End01, Man05].*

Consequently, all the main results of the unsorted logic extend to the many-sorted case. Moreover, if the set $\mathcal{S}$ contains only one sort, the logic MS-FOL becomes single-sorted, thus equivalent to FOL.

3.2. Automated theorem provers

In order to give the reader an idea of the kind of formulas that ATP systems and SMT solvers can handle, we informally give a compact description of their components and internal workings. In addition, we first overview some basic concepts of rewriting systems, used by ATP systems and applied later in the following chapter.

3.2.1. Rewriting systems and equational reasoning

Rewriting systems

An abstract rewriting system (ARS) models step by step activities like object (e.g. term) transformations or stepwise execution of computations. Formally, an ARS is a pair $\langle A, \longrightarrow \rangle$ consisting of a set A of objects and a binary relation $\longrightarrow$ on A, i.e. $\longrightarrow \subseteq A \times A$, called rewriting or reduction relation.

The following definitions take place in the context of some arbitrary, but fixed, system $\langle A, \longrightarrow \rangle$, most of them taken from Baader and Nipkow [BN99]. When $\langle a, b \rangle \in \longrightarrow$ for $a, b \in A$, we write $a \longrightarrow b$ and we say that there is a *step* from a to b. The reflexive transitive closure of $\longrightarrow$ is noted $\overset{*}{\longrightarrow}$. We write $a \overset{*}{\longrightarrow} b$, or $b \overset{*}{\longleftarrow} a$, if there is some finite *path* from a to b. An ARS is *terminating* or *Noetherian* if there is no infinite descending chain $a_0 \longrightarrow a_1 \longrightarrow \cdots$ of objects a_i. Then, $\longrightarrow$ is *well-founded*. An object a is in *normal form* if it is non-reducible, meaning that there is no b such that $a \longrightarrow b$. The object b is a *normal form of* a if $a \overset{*}{\longrightarrow} b$ and b is in normal form. Objects a_1 and a_2 are *joinable*, noted $a_1 \downarrow a_2$, if there is an object b such that $a_1 \overset{*}{\longrightarrow} b \overset{*}{\longleftarrow} a_2$. An ARS is *confluent* (equivalently, *Church-Rosser*) if $a_1 \overset{*}{\longleftarrow} b \overset{*}{\longrightarrow} a_2 \Rightarrow a_1 \downarrow a_2$, that is, the system defines at most one

normal form for each object. A relation $\longrightarrow$ is *locally confluent* if, for every object b, $a_1 \longleftarrow b \longrightarrow a_2 \Rightarrow a_1 \downarrow a_2$, that is, it is confluent restricted to one-step divergences.

Lemma 2 (Newman's)**.** *A terminating* ARS *is confluent if it is locally confluent. [BN99]*

Now we consider rewriting systems whose objects are terms, as defined in (first-order) CNF. An *equation* $s \approx t$ is a pair $\langle s, t \rangle \in T \times T$, where T is the set of all terms.[3] Identities can transform terms into other "equivalent" terms by replacing instances of the left-hand side (lhs) with the corresponding instances of the right-hand side (rhs), and vice-versa. Let $Var(x)$ be the set of variables occurring in term x. An equation $s \approx t$ where s is not a variable and $Var(s) \supseteq Var(t)$ is called a *rewrite rule* and is usually written $s \longrightarrow t$. A *term rewriting system* (TRS) is a set of rewrite rules. A *redex* (reducible expression) is an instance of the lhs of a rewrite rule. Rewriting or reducing the redex means replacing it with the corresponding instance of the rhs of the rule. In Section 4.4.1, we will define a TRS on the set of well-formed expressions of TLA$^+$. Our goal will be to prove the fundamental properties of termination and congruence for that system.

Finally, we can define the concept of critical pair, which is crucial for proving TRS confluence, and also in the superposition calculus. Let $a|_p$ denote the sub-term of a at a given position p of its syntactic tree, and $a[b]_p$ the term a such that b replaces $a|_p$. Consider two rules $a_1 \longrightarrow b_1$ and $a_2 \longrightarrow b_2$ whose variables have been renamed such that the rules do not have variables in common. Let p be a position in the syntactic tree of a_1 such that $a_1|_p$ is not a variable, and let σ be a most general unifier (mgu) of $a_1|_p$ and a_2, that is, the *superposition* of the left-hand sides of both rewriting rules. Then, the pair $\langle b_1\sigma, (a_1[b_2]_p)\sigma \rangle$ is a *critical pair*. For example, consider the rules $f(f(x,y),z) \longrightarrow f(x,f(y,z))$ and $f(g(a),a) \longrightarrow b$. By unifying the non-variable sub-term $f(x,y)$ of the left-hand sides of both rules, we obtain the mgu $\{x \mapsto g(a), y \mapsto a\}$. The formula $f(f(g(a)),a),z)$ can be reduced to the two elements of the critical pair $\langle f(g(a),f(a,z)),f(b,z) \rangle$.

Theorem 3 (Critical pairs)**.** *A* TRS *is locally confluent if and only if all its critical pairs are joinable [Hue80, KB70].*

Combining the critical pair theorem (3) with Newman's lemma (2), we obtain:

Corollary 4. *A terminating* TRS *is confluent if and only if all its critical pairs are joinable.*

[3] Formally, T is $T(\mathcal{V}, \mathcal{F})$, the term algebra generated by the set of variable symbols $\mathcal{V}$ and the signature function symbols $\mathcal{F}$.

Completion and orderings

A completion procedure, originally by Knuth and Bendix [KB70], takes a set E of equations between terms and applies the inference rules $\Longrightarrow_{KB}$ to derive a confluent term rewriting system, provided a well-founded ordering $\succ$ on terms. Initially, the rules are applied to the pair $E, \emptyset$, where the second element represents the set of rewriting rules. We describe only the two main inference rules:

$$\begin{array}{lll} \text{(Orient)} & E \cup \{s \approx t\}, R \Longrightarrow_{KB} E, R \cup \{s \longrightarrow t\} & \text{if } s \succ t \\ \text{(Deduce)} & E, R \Longrightarrow_{KB} E \cup \{s \approx t\}, R & \text{if } \langle s, t \rangle \text{ is a critical pair of } R \end{array}$$

Rule (Orient) transforms an equation $s \approx t$, if $s \succ t$, into the rewriting rule $s \longrightarrow t$. Rule (Deduce) detects reductions $s \longleftarrow u \longrightarrow t$, i.e. critical pairs of $\longrightarrow$, and derives a new equation $s \approx t$ to add it to E. During the process, some equations are simplified, and trivial ones such as $s \approx s$, which cannot be oriented, are deleted. The procedure succeeds if it generates a final pair $\emptyset, R$, where the final set R is a set of rewriting rules equivalent to E.

The standard completion algorithm fails or runs forever if an equation can neither be oriented or deleted. For example, theories with commutativity usually cannot be represented as terminating systems. Knuth-Bendix completion ensures termination provided the ordering is total on ground terms [BDP89]. A completion algorithm using this kind of ordering is called an *unfailing* completion. For instance, the Knuth-Bendix ordering $\succ_{\mathsf{KB}}$ makes a lexicographic comparison of terms employing a weight function. Assuming some basic requirements, the ordering $\succ_{\mathsf{KB}}$ is a well-founded total order on ground terms [KB70]. The rules $\longrightarrow$ in R constructed in this way guarantee that $\longrightarrow \subseteq \succ_{\mathsf{KB}}$, ensuring that R is terminating. Since all critical pairs are joinable, the rewriting system R is also confluent, by Corollary 4.

Resolution and equational reasoning

Resolution is one of the main computational methods in automated theorem proving. The DPLL algorithm is better suited than resolution for deciding the unsatisfiability of propositional CNF formulas, but resolution can be easily extended to first-order clauses using unification [Rob65]. The (non-ground) resolution calculus is based on the (binary) resolution and factoring inference rules:

$$\frac{B \vee A \qquad C \vee \neg A'}{(B \vee C)\sigma}\ \textsc{Res} \qquad \frac{B \vee A \vee A'}{(B \vee A)\sigma}\ \textsc{Factor} \quad \text{if } \sigma = mgu(A, A')$$

Given a first-order CNF clause set, the two rules are applied until either the empty clause is deduced, or the clause set becomes *saturated* with respect to both rules, that is, the rules are applied fairly and exhaustively. Unification generalizes equality of

ground atoms to unifiability of general atoms while producing only those clause instances required at each inference step. Resolution is a *complete refutation* procedure for first-order CNF [Rob65]: if the formula ϕ is unsatisfiable, then fair derivations from ϕ will eventually derive the empty clause.

One of the many variants of resolution [BG01] is ordered resolution. Let $\succ$ be a total and well-founded ordering on ground atoms, such as the Knuth-Bendix ordering. A literal ℓ is called *strictly maximal* in a clause $C \stackrel{\Delta}{=} \ell, \ell_1, \ldots, \ell_n$ (noted $C < \ell$) iff there exists a ground substitution σ such that $\ell\sigma \succ \ell_i\sigma$, for $i \in 1\,..\,n$. *Ordered resolution* overlaps only maximal literals in a clause by extending the rule Res with the additional conditions $B\sigma < A\sigma$ and $C\sigma < \neg A'\sigma$, and the rule Factor with the condition $B\sigma < A\sigma$. In this way, the improved calculus restricts the number of inferred clauses, most of which are irrelevant or redundant.

The next step is to consider equality reasoning. Congruence axioms expressing the properties of equality can be added explicitly to the conjecture formula but this results impractical for resolution-based methods. Instead, theorem provers incorporate dedicated inference rules for equations. Paramodulation [RW83] is a refutationally complete method that combines resolution, with the following rule, which essentially generalizes equality substitution by including unification:

$$\frac{B \vee s \approx t \qquad C}{(B \vee C[t]_p)\sigma}\,\textsc{Param} \qquad \text{if } \sigma = mgu(s, C|_p)$$

As in basic resolution, paramodulation requires optimization strategies to control the way it generates new clauses.

The superposition calculus [BG94, NR95] is a paramodulation-based inference system, parameterized by a well-founded total ordering $\succ$ on ground terms. Inferences are restricted to involve only left-hand sides of possible rewrite steps, that is, only the big terms (in the sense of the above definition) with respect to an order $\succ$ are considered. To cover all possible cases, different inference rules are required. One of them is, for instance, the superposition rule for positive equational literals:

$$\frac{B \vee s \approx t \qquad C \vee u \approx v}{(B \vee C \vee u[t]_p \approx v)\sigma}\,\textsc{Sup}^{+} \qquad \begin{array}{l}\text{if } \sigma = mgu(s, u|_p) \text{ such that} \\ \quad (i)\ B\sigma < s\sigma \approx t\sigma, C\sigma < u\sigma \approx v\sigma, \\ \quad (ii)\ s\sigma \succ t\sigma, \text{ and } u\sigma \succ v\sigma\end{array}$$

The equation $(u[t]_p)\sigma \approx v\sigma$ is a critical pair resulting from overlapping the premises. Superposition essentially combines ordered resolution (represented in the conditions (i)), and unfailing completion (represented in the conditions (ii)), while holding the property of refutational-completeness. State-of-the-art ATP systems, like E [Sch13], SPASS [Wei99] or Vampire [KV13], implement variants of superposition.

3.2.2. SAT and SMT solving

SAT solvers

The DPLL procedure implemented by most SAT solvers attempts to build a model M for a given formula $\phi \triangleq C_1, \ldots, C_n$ in propositional CNF. We describe variants of the algorithm as a state transition system in the Abstract DPLL framework of Nieuwenhuis et al. [NOT06]. The binary relation $\Longrightarrow$ on states describes the algorithms. A state is either Fail or a pair of the form $M \parallel \phi$. In the sequence of literals composing a model M, literals can appear as ℓ^d to indicate that ℓ is a *decision* literal. The classical DPLL procedure is given by five conditional transition rules:

$$
\begin{array}{llll}
(\mathsf{UP}) & M \parallel \phi, C \vee \ell & \Longrightarrow & M\, \ell \parallel \phi, C \vee \ell \quad \text{if } \mathit{undef}(\ell, M) \text{ and } M \not\models C \\
(\mathsf{PL}) & M \parallel \phi & \Longrightarrow & M\, \ell \parallel \phi \quad \text{if } \mathit{undef}(\ell, M), \ell \in \phi \text{ and } \neg\ell \notin \phi \\
(\mathsf{D}) & M \parallel \phi & \Longrightarrow & M\, \ell^d \parallel \phi \quad \text{if } \mathit{undef}(\ell, M) \text{ and } \ell, \neg\ell \in \phi \\
(\mathsf{BT}) & M_1\, \ell^d\, M_2 \parallel \phi, C & \Longrightarrow & M_1\, \neg\ell \parallel \phi, C \quad \text{if } M_1\, \ell^d\, M_2 \not\models C \text{ and } \mathit{no\text{-}dec}(M_2) \\
(\mathsf{FI}) & M \parallel \phi, C & \Longrightarrow & \mathsf{Fail} \quad \text{if } M \not\models C \text{ and } \mathit{no\text{-}dec}(M)
\end{array}
$$

where we write $\ell \in \phi$ when literal ℓ occurs in some clause of ϕ; $\mathit{def}(\ell, M)$ means that ℓ, possibly as ℓ^d, or $\neg\ell$ occur in the sequence M, i.e. ℓ is defined in M; we write $\mathit{undef}(\ell, M)$ when not $\mathit{def}(\ell, M)$; and $\mathit{no\text{-}dec}(M)$ means that model M contains no decision literals.

In order to decide the satisfiability of ϕ, the algorithm applies the transition rules to the initial state $\emptyset \parallel \phi$ (with an empty model), until no further rule is applicable. If the final state is Fail, resulting from a conflicting clause and no decision literals (rule (FI)), then the formula ϕ is unsatisfiable. If the final state is $M \parallel \phi'$, then M is a model of ϕ. The DPLL algorithm performs a search in the space of the partial truth assignments based on three main satisfiability-preserving transformations:

- unit clause propagation (UP): if ℓ is a unit clause derived from a clause C, that is, all remaining literals in C are false, then ℓ should be true in the model,
- pure literal (PL): if a literal ℓ is pure in ϕ, that is, it occurs in ϕ while its negation does not, and the truth value of ℓ is undefined, then ℓ can be assigned to true in M, and
- case-splitting, which considers two separate problems by adding ℓ and $\neg\ell$ as new unit clauses, corresponding to executing a depth-first search by deciding on a value of some literal (D) and by applying a backtracking mechanism (BT). Each time an unsatisfied clause, i.e. a *conflict*, is identified, the algorithm backtracks, undoing splitting steps until reaching an unexplored branch flagged as a decision literal ℓ^d.

Modern SAT solvers implement variants of the classical algorithm, which include techniques like back-jumping, conflict-driven learning, and restarts among others,

plus many optimizations, such as the use of highly efficient data structures, and heuristics to select splitting variables [Kul09, MS99]. A *basic* DPLL algorithm is given by the rules (UP), (D), (FI), and the *back-jumping* rule (BJ), which subsumes (BT) by modeling a more sophisticated, non-chronological backtracking mechanism. Since no pure literals are ever introduced, it is enough to restrict (PL) to preprocessing.

$$\text{(BJ)}\quad M_1\, \ell^d\, M_2 \parallel \phi, C_1 \implies M\, \ell' \parallel \phi, C_1$$
if $M_1\, \ell^d\, M_2 \not\models C_1$ and there is some clause $C_2 \vee \ell'$ such that:
$\phi, C_1 \models C_2 \vee \ell'$; $\mathit{undef}(\ell', M_1)$; $M_1 \not\models C_2$; and
$\ell \in \phi$ or $\neg\ell \in \phi$ or $\mathit{def}(\ell, M)$

Unlike chronological backtracking as modeled by rule (BT), back-jumping allows to backtrack to earlier levels of the proof tree, potentially pruning large portions of the search space. When no further derivations are possible with (UP) and (D), the (BJ) transition rule searches for a *back-jump clause* $C_2 \vee \ell'$, a logical consequence of the conflicting clause ϕ, C_1, enabling a unit propagation of ℓ' at the level of $M_1\, \ell^d$. In order to build the back-jump clause, its literals are chosen among the negation of the decision literals.

To avoid future similar conflicts, another common enhancement to the original DPLL algorithm is to add the back-jump clauses as learned *lemmas* to the clause set, by implementing what is usually called conflict-driven clause learning (CDCL). The CDCL algorithm [MSS99, MSLM09] extends basic DPLL with two additional rules:

$$\text{(L)}\quad M \parallel \phi \implies M \parallel \phi, C \quad \text{if } \phi \models C \text{ and each atom of } C \text{ occurs in } \phi \text{ or } M$$
$$\text{(F)}\quad M \parallel \phi, C \implies M \parallel \phi \quad \text{if } \phi \models C$$

The rule (L) models the "learning" of an arbitrary clause C, not only conflict-driven clauses, as long as C is a logical consequence of ϕ. In a similar way, rule (F) allows to "forget" redundant clauses or clauses that are not longer needed, not only those learned through (L).

The order in which the search space is explored heavily relies on the choice of the splitting variables. When the search is not making enough progress within a certain time, SAT solvers implement rule (R) that restarts the decision variables with the goal of exploring the search space in a different way (but the learned lemmas are kept).

$$\text{(R)}\quad M \parallel \phi \implies \emptyset \parallel \phi$$

The efficiency of different SAT solvers depend on how they implement the preprocessing methods and the details not modeled by above abstract rules. Although SAT solvers perform extremely well in practice, checking the unsatisfiability of CNF formulas is coNP-complete [BK09]. Is it known that every CDCL derivation starting from $\emptyset \parallel \phi$ terminates [MSLM09].

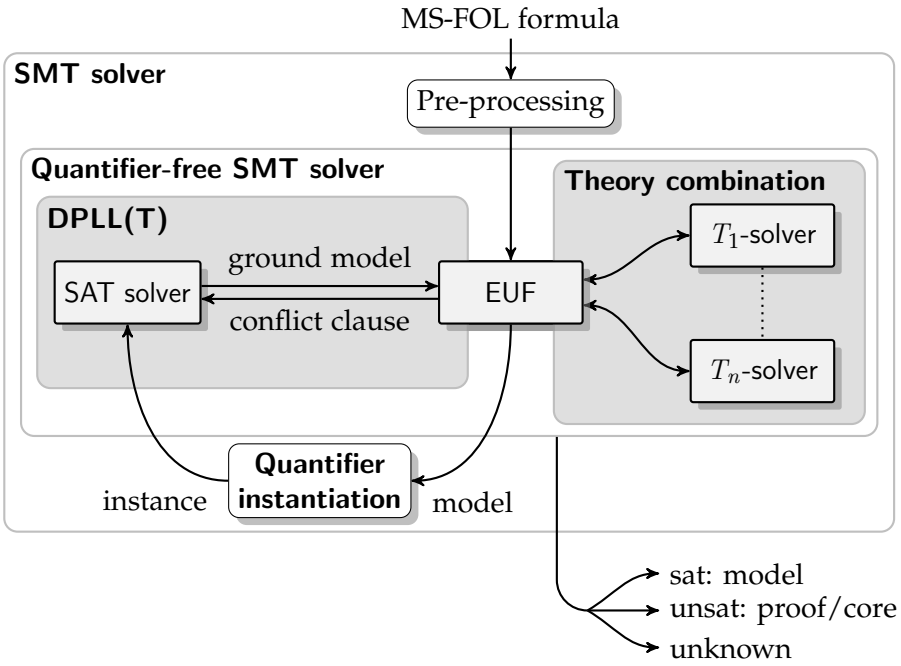


Figure 3.1.: Schematic view of an SMT solver architecture

Resolution-based theorem proving is at the heart of refutational theorem provers like SAT solvers. Any DPLL derivation starting from an unsatisfiable formula is equivalent to a tree-like refutation proof by (propositional) resolution. CDCL can polynomially simulate resolution [PD09].

Quantifier-free SMT solvers

Satisfiability modulo theories (SMT) solvers integrate a propositional engine, typically a SAT solver, with specialized T-solvers, that is, decision procedures for the different background theories T. The theory engine combines decision procedures for first-order theories, such as the theory of equality and uninterpreted functions (EUF), fragments of arithmetic (linear or non-linear, integer or real), arrays, bit vectors, and combinations thereof [BSST09]. All these components are interfaced through a congruence closure procedure for the EUF theory [NO80, BN99, NO05], as illustrated in Figure 3.1.

A *theory* T is a set of closed first-order formulas. Function symbols occurring in a theory T are *interpreted*. A first-order formula ϕ is *T-satisfiable* if $T \cup \{\phi\}$ is satisfiable, that is, if there exists a model of T, or *T-model*, that is also a model of ϕ. Otherwise, it is *T-unsatisfiable*. Function symbols in ϕ that are not in T are *uninterpreted*. Each T-solver implements efficient decision procedures for a theory T. The theory of linear integer arithmetic (LIA) is typically based on the Omega test [Pug91]

procedure, and that for linear real arithmetic (LRA) is based on the Simplex algorithm or the Fourier-Motzkin variable elimination approach [DCE73]. The theories of arrays and the conditional ite (if-then-else) operator are usually reduced to the EUF theory. For instance, in its most basic and common formalization, originally by McCarthy [Mcc62], arrays are characterized by the functions store, select, and the axioms

$$\forall a, i, v.\ \mathsf{select}(\mathsf{store}(a, i, v), i) \approx v, \text{ and}$$
$$\forall a, i, j, v.\ i \approx j \vee \mathsf{select}(\mathsf{store}(a, i, v), j) \approx \mathsf{select}(a, j).$$

The conditional ite operator on terms provided by most SMT solvers can be (naïvely) defined [KSJ09] by the axiom:

$$\forall c, t, u.\ \mathsf{ite}(\top, t, u) \approx t \wedge \mathsf{ite}(\bot, t, u) \approx u.$$

The DPLL(T) framework [NOT06, GHN+04], where T stands for a combination of theories, extends the propositional engine, that is, the SAT solver, by allowing it to interact with the theory engine through the exchange of models and conflict clauses. The SAT solver evaluates the propositional structure of a given formula ϕ, where each theory atom is considered as a propositional symbol. If it finds a partial model M, it invokes the theory reasoner. If M results T-unsatisfiabiable, the theory reasoner reports a conflict clause which is conjoined as a theory lemma to the propositional formula ϕ. The propositional engine incorporates this information and continues its search. This interplay is called the *lazy* approach to the SMT problem [BSST09, Seb07].[4] The process finishes when the SAT solver concludes that the given input formula is unsatisfiable, which is also T-unsatisfiable, or when the theory reasoner finds a T-model of ϕ. In the first case, the SMT solver outputs a proof of unsatisfiability, and some SMT solvers provide an unsatisfiable core, that is, the set of facts necessary to prove that the formula is unsatisfiable. In the latter case, the solver outputs the model it has found, which can be partial.

A DPLL modulo theories system can be defined by the rules (D), (FI), (UP), (BJ_T), (L_T), (F_T), and (R). The rules (BJ_T), (L_T), (F_T) are the same as their original counterparts in DPLL except that the conditions of the form $\phi \models C$, where ϕ is a formula and C is a clause, are replaced by $\phi \models_T C$, that is, these conditions are evaluated by the corresponding T-solver. The flexibility of this framework allows to introduce several efficiency-related techniques during the engines interaction. An *incremental* T-solver does not wait until the SAT solver finds a model: at regular intervals, it performs T-consistency checks while the model is still being built. An *online* SAT solver allows to backtrack to the sub-model M' of M where $M' \models \phi$ holds while $M \models \phi$ does not: when the theory engine reports a conflict lemma for the inconsistent model M, the SAT solver restarts the search from M' instead than from scratch.

[4] The *eager* SMT approach invokes the SAT solver with an encoding of the problem containing theories into an equisatisfiable propositional problem. In the lazy approach, T-satisfiability is checked only after a model at the propositional level is found.

The combination of an incremental T-solver with an online SAT solver is known to be crucial for efficiency [NOT06]. Another common method that greatly improves the search is theory propagation (TP).

$$(\mathsf{TP}) \quad M \parallel \phi \Longrightarrow M\,\ell \parallel \phi \quad \text{if } M \models_T \ell; (\ell \in \phi \text{ or } \neg\ell \in \phi); \text{ and } undef(\ell, M)$$

In order to guide the propositional search, rule (TP) assigns truth values to literals that the theory reasoner derives from the current model, before letting the SAT solver take a decision on those literals.

The theory engine of an SMT solver reasons about combinations of decision procedures for different theories, usually implemented over the Nelson-Oppen architecture [NO79]. In this combination method, the theory procedures communicate by exchanging equalities of variables through the EUF solver. Two decision procedures for different theories can be combined if the theories satisfy two conditions: (i) they are disjoint, meaning that they share equality as the only interpreted symbol, and (ii) they are stably infinite. A theory T is *stably infinite* if whenever a (quantifier-free) formula is T-satisfiable, then it is T-satisfiable in a model of the theory with an infinite universe.

Quantifier instantiation

State-of-the-art SMT solvers such as Alt-Ergo [CCKL08], CVC4 [BCD+11], veriT [BDODF09], Yices [Dd06] and Z3 [dB08], successfully decide the unsatisfiability of quantifier-free MS-FOL formulas with theories using the above architecture. However, even the most efficient solvers rely on refutationally incomplete methods for quantifier reasoning of pure first-order logic. Existential quantifiers are usually eliminated by Skolemization, i.e. by replacing the quantified variables by fresh symbols witnessing the satisfiability of the formula. In order to deal with universally quantified formulas $\forall x^{\sigma}.\,\phi$, SMT solvers implement *quantifier instantiation* methods,[5] which apply different heuristics to choose a set of ground instances $\{t_1, \ldots, t_n\}$ of the quantified sorted variable x. Then, the formulas $\phi[x \leftarrow t_i]$, which are logical consequences of the original formula, are added to the clause set to continue the quantifier-free search.

Instantiation methods focus on different approaches to search for ground instances: either they attempt to find a (complete) counter-model that satisfies the original formula (for instance, the finite-model finding algorithm [RTG+13]) or, conversely, they focus on finding just the instances that render the problem unsatisfiable. Among the latter, the most widely applied strategy is *pattern-based instantiation*, also known as E-matching [DNS05]. This heuristic finds ground terms that have the same shape

[5] It is possible by following [GBT09] to extend the above DPLL framework to model rules for quantifier instantiation, but it is not relevant in this simple discussion of the topic.

as sub-terms of the body of $\forall x^{\sigma}.\phi$, and uses them to guide the instantiation. For example, suppose the state in the EUF theory is of the form:

$$\{a \approx g(c)\} \parallel f(a,a) \not\approx b, \forall x. f(a, g(x)) \approx b$$

This method will find in the quantified formula the pattern $f(a, g(x))$ that matches the ground term $f(a, a)$ with the substitution $x \mapsto c$. As a result of applying the substitution in the quantifier, we obtain $f(a, g(c))$, which equals $f(a, a)$ module the set of equalities $\{a \approx g(c)\}$, causing a conflict at the ground level. Although relatively useful for some applications, pattern-based instantiation could generate an exponential number of matches (most of them useless), which may generally cause the solver's performance to degrade. Moreover, it is sensitive to the syntactic structure of ϕ, being particularly affected by equivalence-preserving transformations of the original formula. In order to limit the number of instances produced, users can guide the instantiation by providing *triggers*, which are sub-terms occurring in ϕ used as hints of potentially useful patterns [DCKP12].

Alternatively, the *model-based* approach for complete *quantifier instantiation* (MBQI) [GM09] uses model-checking techniques to recognize when a model of a quantified formula has been constructed, without explicitly generating all instances. MBQI is effective in finding solutions of satisfiable formulas and, in practice, it sometimes outperforms E-matching in solving unsatisfiable cases as well. For some fragments of first-order logic, the MBQI engine guarantees that the SMT solvers are decision procedures. For instance, Z3 handles the effectively propositional class of formulas (also called Bernays-Schönfinkel-Ramsey class), a decidable fragment that corresponds to formulas containing only constants, universal quantifiers, and predicate symbols, when written in prenex normal form [PdMB10].

Summing up, the quantifier instantiation procedure is incomplete, because not even a fair strategy can guarantee the generation of the ground instances that are necessary to derive the empty clause. In these cases, the SMT solver either runs indefinitely, overloading the problem with useless ground clauses, or it reports "unknown" together with a partial model satisfying only the grounded instances.

3.3. TPTP and SMT-LIB

TPTP and SMT-LIB provide respectively input languages for ATP systems and SMT solvers, which have become de facto standards in the automated reasoning community. In this section we describe the input formats that are relevant for our use of ATP systems and SMT solvers.

Notation The definitions of the TPTP formats were influenced by Prolog, while the SMT-LIB language has a Lisp-like syntax. This is convenient for the tools based on these languages, but not for human readability. For a better presentation, in this document we display TPTP and SMT-LIB problems in a pretty-printed format.

TPTP

The TPTP World [Sut10] provides the ATP community with the TPTP (Thousands of Problems for Theorem Provers) problem library, the TSTP proofs library, as well as languages for the exchange of problems and proofs between provers.

In the following, we briefly characterize some relevant TPTP languages that, except for some minor syntactic differences, can be hierarchically organized as the strict inclusion:

$$\text{CNF} \subset \text{FOF} \subset \text{TFF0}.$$

They are respectively based on the logics CNF, FOL, and MS-FOL.

CNF (Clausal Normal Form) The CNF language of TPTP [Sut09] corresponds to the CNF logic described above, where a problem is given as a set of clauses. In particular, variable identifiers should start with an upper-case letter, to distinguish them from function and predicate symbols, that start with a lower-case letter.

FOF (First-Order Form) The language FOF [Sut09] corresponds to untyped first-order logic with equality, that is, to FOL. The format FOF has become the de facto standard in the automated reasoning community, and it is the most widely supported TPTP format. Most modern ATP systems include efficient *clausifiers* that translate a FOL problem into an equisatisfiable CNF formula by introducing Skolem functions to encode existential quantifiers. Clausification is crucial: a naïve conversion to CNF may produce a formula whose size is exponential in the size of the original one [NW01, AW13]. By stating our problems in FOF, we benefit from the efficient clausifiers provided by the ATP systems.

TFF0 (Typed First-order Form with monomorphic types) TFF0 [SSCB12] is a many-sorted logic based on MS-FOL. Specifically, it is an extension of FOF with simple monomorphic types, interpreted arithmetic sorts for integers, rationals, and real numbers, and the usual arithmetic operators, including quotient functions with different interpretations. Unlike FOF, each variable, function and predicate symbol must be declared beforehand with their respective sorts.

SMT-LIB

The SMT-LIB [BST10] initiative provides a repository of benchmarks and a common input format for SMT solvers. An SMT-LIB benchmark file is a sequence of commands to interact with the solvers, declarations of sort and function symbols, and assertion of formulas over the resulting signature. An SMT-LIB problem is given as a list of assertions, where the conjecture is negated. Unlike TPTP, SMT-LIB requires all identifiers to be declared before using them.

The SMT-LIB language At the logical level, the SMT-LIB language is based on MS-FOL. Accordingly, each well-formed expression has a unique sort that has to respect the sorting discipline. SMT-LIB sorts extends MS-FOL atomic sort symbols in the set $\mathcal{S}$ to *sort terms* (including arities) built from symbols in $\mathcal{S}$. For example, it is possible to have a sort $\mathsf{List}(\mathsf{Array}(\mathsf{Int}, \mathsf{U}))$, where Int, U, List, and Array are sort constructors with arities 0, 0, 1 and 2, respectively.

In SMT-LIB, Bool is a predefined sort. In contrast to MS-FOL, a predicate is defined as a function that returns a Bool-sorted term, and SMT-LIB formulas are just terms of sort Bool, i.e. they are both included in the same syntactic category t.[6] In addition, SMT-LIB provides a conditional if-then-else operator as a term, noted ite, where the sort of the first argument is Bool, and the second and third arguments have the same sort:

$$t ::= \cdots \mid \mathsf{ite}(t, t, t)$$

SMT logics SMT-LIB problems are categorized in different *logics* that reflect the capabilities of the available SMT solvers to support certain background theories. An SMT logic is identified by a pre-established name to which are associated sort and function declarations, and possibly syntactic and semantic restrictions.

We are mainly interested in the logic AUFLIA that offers quantified formulas and arithmetic expressions, features that are ubiquitous in hardware and software verification problems.[7] The logic AUFLIA supports quantified formulas over the theories of arrays and linear integer arithmetic extended with free sort and function symbols. It provides a predefined sort Int and the usual arithmetic functions. Set theory is not currently supported natively by any pre-defined SMT-LIB logic.[8]

[6] Some SMT solvers like CVC3 do not allow quantification of variables with sort Bool.

[7] AUFLIA stands for Arrays, Uninterpreted Functions, Linear Integer Arithmetic. Other logic names in the same family labeled AUF[LN][RI]A include variants of Linear or Non-linear, Real or Integer Arithmetic.

[8] Kröning et al. [KRW09] proposed new logics for the theories of finite sets, lists, and maps, aimed at state-based specification languages, but they have not been realized in practice.

The languages TFF0 and AUFLIA are in essence the same, because they are both based on MS-FOL. In our TLA$^+$ encodings of Chapter 4, we will focus only on the translations to TPTP/FOF and AUFLIA fragments.

Chapter 4.

Integration of ATP systems and SMT solvers

Contents

> It's simple: you just take something and do something to it, and then do something else to it. Keep doing this, and pretty soon you've got something.
>
> Jasper Johns

Integrating an external automated theorem prover as a new back-end verifier for the TLAPS architecture requires to translate the TLA$^+$ proof obligations to the input language of the prover and then to invoke it. At this point, if the prover succeeds to find a proof before the predefined timeout, there is the option to take either a *faithful* or a *skeptical* approach. That is, either we trust the prover and accept the proof obligation as a theorem, or we obtain the proof script that the prover may produce, and re-check it in Isabelle/TLA$^+$, TLAPS's trusted kernel. In this chapter, we consider the external provers as sound *oracles*, making them part of the trusted base, so we can focus on the translation.

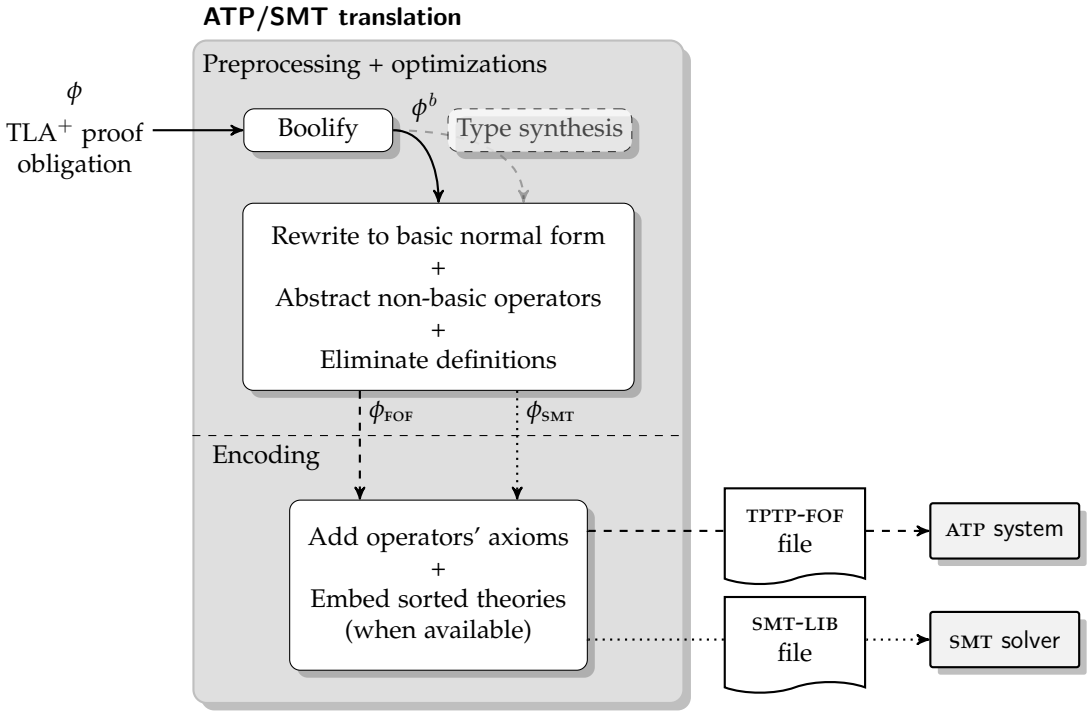


Figure 4.1.: Schematic view of the translation from TLA$^+$ to the input languages of ATPs and SMT solvers

Our main goal in this chapter is to define a sound translation from TLA$^+$ to the languages of ATP systems and SMT solvers. Based on the fact that the considered provers have a common logic as a baseline, i.e. first-order logic with equality, we propose a generic translation framework. In particular, we want to encode TLA$^+$ formulas into dialects of TPTP and SMT-LIB, the standard input formats of ATP systems and SMT solvers. Given a TLA$^+$ proof obligation, the system generates an equi-satisfiable formula tailored to the input format of a specific theorem prover.

The translation proceeds in two main steps. First, it preprocesses, massages and transforms the TLA$^+$ formula by encoding those expressions that the target solver cannot handle. At that point, the formula contains only expressions that have a native counterpart in the selected target language. Via a shallow embedding, it finally passes the resulting formula to the solver. For the many-sorted logics, we use only a single sort to encode all TLA$^+$ expressions. Thus, we call this the *untyped* encoding.

4.1. Translation overview

In order to simplify the presentation, we describe the ATPs and the SMT translations as a generic framework whose general architecture is presented in Fig. 4.1. The translation is divided in the preprocessing and optimization of the proof obligation, yielding an intermediate basic TLA$^+$ formula, and the subsequent encoding of this formula into a specific target language. Given a solver's input language such as TPTP/FOF or SMT-LIB/AUFLIA, we call a *basic* formula one that is formed by TLA$^+$ expressions that can be directly written in that target language. The intermediate subsets of basic TLA$^+$ contain elements that are in a one-to-one correspondence with the predefined operators of the target language. Once a TLA$^+$ formula is reduced to an equi-satisfiable basic formula, it can be straightforwardly passed to the solver by a syntactic rewriting. We are interested in two basic fragments:

- A basic TLA$^+$-FOF formula ϕ_{FOF} is composed only of TLA$^+$ terms and formulas, including equality and set membership relations, plus primitive arithmetic operators. These formulas are intended to be directly mapped to the language FOF of TPTP, which is essentially classical first-order logic with equality. Set theory can be axiomatized through an uninterpreted predicate for set membership, but arithmetic operators will remain just unspecified without any possible practical reasoning about them.
- The language TLA$^+$-SMT upgrades the TLA$^+$-FOF fragment by including IF-THEN-ELSE expressions. A basic TLA$^+$-SMT formula ϕ_{SMT} is intended to be translated to the many-sorted FOL of SMT-LIB, in particular to the logic AUFLIA, which supports quantifiers, the theory of arrays, and linear integer arithmetic.

A priori, the only known characteristic of the TLA$^+$ expressions is the arity of the operators. There is no way to distinguish expressions used as formulas from those used as terms. We need at least to differentiate those expressions that have a truth (Boolean) value to use them properly as formulas or predicates in (sorted or unsorted) first-order logic. By a process called *Boolification* it is possible to know if the truth value of an expression is Boolean or undefined. This is the first step in the translation and it will be described in Section 4.2.

The pre-processing step consists in the application of satisfiability-preserving transformations to the original TLA$^+$ formula ϕ to rephrase it using only basic constructs. After Boolification, the next step is the transformation of the Boolified formula ϕ^b with the purpose of producing an equi-satisfiable basic formula ϕ_{FOF} or ϕ_{SMT}, depending on the backend prover that TLAPS or the user has previously chosen as target. This and other optimization methods will be explained in Section 4.4. For instance, the main transforming method will be to take the formula to a basic normal form by applying rewriting rules derived from the language semantics (described in Section 4.4.1). If the formula contains non-basic expressions appearing in a form

where no rewriting rule can be applied, the formula has to be rephrased into a form where rewriting is possible (Section 4.4.3).

Once a proof obligation ϕ is reduced to the corresponding basic fragment, the translation to the solver's input language is just a syntax-directed mapping of expressions (Section 4.3). Mapping unsorted TLA$^+$ to an unsorted language like FOF is relatively trivial once we can distinguish terms and formulas. On the other side, saying that a language like TLA$^+$ is unsorted is equivalent to say that the language has only one extra sort in SMT-LIB, besides the built-in Bool. Then, we map all expressions having a truth value (i.e. T or F in our semantics) to the sort Bool, and we declare a new sort U (from *u*niversal) for the rest, regardless if they denote sets, functions, or numbers. Later, in the next chapter, we will study how to further partition U in more types to improve the solvers' performance. In the course of this chapter, it will be practically indistinct to declare variables and constants with sort U or not having sorts at all.

4.2. Boolification

In TLA$^+$, there is no syntactic distinction between Boolean and non-Boolean expressions. We are faced with a first task of differentiating those elements of $\mathcal{V}$ (the set of variable symbols) and $\mathcal{O}$ (the set of operator symbols) that are used as propositions and those that are not. In our interpretation of TLA$^+$ expressions (Section 2.1), logical expressions always have a truth (Boolean) value, and the arguments of logical operators always have a truth value as well. For instance, we interpret $\phi_1 \Rightarrow \phi_2$ as $(\phi_1 = \text{TRUE}) \Rightarrow (\phi_2 = \text{TRUE})$ and $\forall x : \phi$ as $\forall x : \phi = \text{TRUE}$.

For example, consider the expression $\forall x : (\neg\neg x) = x$, which is not a theorem and whose validity could be easily misinterpreted. Indeed, x is not known to be a proposition, it could have any value. For some of those values, say $x = 42$, it is not possible to know the truth value of the formula (the occurrence of x in $\neg\neg x$ is necessarily a Boolean value, but not the one in the right-hand side of the equality). However, $\forall x : (\neg\neg x) \Leftrightarrow x$ is valid because it is interpreted as

$$\forall x : (\neg\neg(x = \text{TRUE})) \Leftrightarrow (x = \text{TRUE}).$$

Observe that the value of $x = \text{TRUE}$ is a Boolean for any x: even $42 = \text{TRUE}$ is an unspecified Boolean value.

In order to identify the symbols used as propositions, we use the simple algorithm of Figure 4.2, which is mutually defined by the operator $[\![e]\!]^+$ that is applied when the expression e is considered as a formula, and by the operator $[\![e]\!]^-$ that is applied when e is considered a non-Boolean expression. The algorithm recursively traverses an expression searching for the arguments of every sub-expression. When it finds an expression e that is implicitly used as a Boolean, i.e. it could be interpreted as

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^+ & \triangleq & x^b & [\![w(e)]\!]^+ & \triangleq & w^b([\![e]\!]^-) \\
[\![e_1 \Rightarrow e_2]\!]^+ & \triangleq & [\![e_1]\!]^+ \Rightarrow [\![e_2]\!]^+ & [\![\forall x\colon e]\!]^+ & \triangleq & \forall x\colon [\![e]\!]^+ \\
[\![e_1 = e_2]\!]^+ & \triangleq & [\![e_1]\!]^- = [\![e_2]\!]^- & [\![e_1 \in e_2]\!]^+ & \triangleq & [\![e_1]\!]^- \in [\![e_2]\!]^- \\
[\![e_1[e_2]]\!]^+ & \triangleq & ([\![e_1]\!]^-[[\![e_2]\!]^-])^b & [\![\mathcal{E}]\!]^+ & \triangleq & \mathit{error}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\textsc{if } e_1 \textsc{ then } e_2 \textsc{ else } e_3]\!]^+ & \triangleq & \textsc{if } [\![e_1]\!]^+ \textsc{ then } [\![e_2]\!]^+ \textsc{ else } [\![e_3]\!]^+ \\
[\![\textsc{choose } x\colon e]\!]^+ & \triangleq & (\textsc{choose } x\colon [\![e]\!]^+)^b
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^- & \triangleq & x & [\![w(e)]\!]^- & \triangleq & w([\![e]\!]^-) \\
[\![e_1 \Rightarrow e_2]\!]^- & \triangleq & [\![e_1 \Rightarrow e_2]\!]^+ & [\![\forall x\colon e]\!]^- & \triangleq & [\![\forall x\colon e]\!]^+ \\
[\![e_1 = e_2]\!]^- & \triangleq & [\![e_1 = e_2]\!]^+ & [\![e_1 \in e_2]\!]^- & \triangleq & [\![e_1 \in e_2]\!]^+ \\
[\![e_1[e_2]]\!]^- & \triangleq & [\![e_1]\!]^-[[\![e_2]\!]^-] & [\![\textsc{domain } e]\!]^- & \triangleq & \textsc{domain } [\![e]\!]^- \\
[\![[x \in e_1 \mapsto e_2]]\!]^- & \triangleq & [x \in [\![e_1]\!]^- \mapsto [\![e_2]\!]^-] & [\![\{e_1, \ldots, e_n\}]\!]^- & \triangleq & \{[\![e_1]\!]^-, \ldots, [\![e_n]\!]^-\} \\
[\![\{x \in e_1 : e_2\}]\!]^- & \triangleq & \{x \in [\![e_1]\!]^- : [\![e_2]\!]^+\} & [\![\textsc{union } e]\!]^- & \triangleq & \textsc{union } [\![e]\!]^- \\
[\![\{e_1 : x \in e_2\}]\!]^- & \triangleq & \{[\![e_1]\!]^- : x \in [\![e_2]\!]^-\} & [\![\textsc{subset } e]\!]^- & \triangleq & \textsc{subset } [\![e]\!]^-
\end{array}
$$

$$
\begin{array}{rcl}
[\![\textsc{if } e_1 \textsc{ then } e_2 \textsc{ else } e_3]\!]^- & \triangleq & \textsc{if } [\![e_1]\!]^+ \textsc{ then } [\![e_2]\!]^- \textsc{ else } [\![e_3]\!]^- \\
[\![\textsc{choose } x\colon e]\!]^- & \triangleq & \textsc{choose } x\colon [\![e]\!]^+
\end{array}
$$

Figure 4.2.: Boolification algorithm: $[\![e]\!]^+$ processes the expression e as a formula, attaching a ^{b} symbol when finding a term, a function application, or a CHOOSE, and $[\![e]\!]^-$ considers e as a non-Boolean expression. The symbol $\mathcal{E}$ encompasses all non-Boolean expressions, such as sets $\{e_1, \ldots, e_n\}$, $\{x \in e_1 : e_2\}$, SUBSET e, DOMAIN e, ..., or functions $[x \in e_1 \mapsto e_2]$, etc. The algorithm is defined only for a relevant fragment, which includes terms, formulas, sets, primitive functions, IF-THEN-ELSE, and CHOOSE

$e = \textsc{true}$, it puts a superscript mark ^{b} on e, where $e^b \triangleq e = \textsc{true}$. This only applies if e is a term, a function application, or a CHOOSE expression. In particular, equality yields a Boolean value but it is not expected that its arguments are formulas. If a non-Boolean expression, like a set or a function, is tried to be Boolified, meaning that a formula is expected in its place, the algorithm aborts with a "type" error. In case e has a Boolean value, the value of e^b is equal to that of e.

In our examples, the Boolified versions of the formulas are respectively

$$\forall x\colon (\neg\neg x^b) \Leftrightarrow x^b \quad \text{and} \quad \forall x\colon (\neg\neg x^b) = x.$$

The (in)validity of these two formulas becomes evident with the ^{b} mark.

In practice, if all the occurrences of a variable or a non-standard operator x in the proof obligation appear as x^b, then x can be declared as Boolean or returning a Boolean. If that is not the case, x^b is encoded as $\mathsf{boolify}(x)$ where the coercion boolify is a predicate in FOF or an uninterpreted function of sort $\mathsf{U} \to \mathsf{Bool}$ in SMT-LIB. Then the above examples result in:

$$\forall x^{\mathsf{Bool}} \colon (\neg\neg x) \Leftrightarrow x \quad \text{and} \quad \forall x \colon (\neg\neg \mathsf{boolify}(x)) = x.$$

Finally, we can identify expressions e having a Boolean value. We write $e \colon \mathsf{Bool}$ if and only if the expression e has a ^{b} mark or if it is a formula, that is, an expression of the form FALSE, $e_1 \Rightarrow e_2$, $\forall x \colon \phi$, $e_1 = e_2$, $e_1 \in e_2$, or an expression defined from formulas. In any other case, we write $e \colon \mathsf{U}$.

4.3. Direct embbeding

The encoding method described in this section maps in an almost verbatim way Boolified TLA$^+$ formulas to corresponding formulas in the target language, without changing substantially the structure of the original formula. The embedding follows the same explanation for single-sorted and many-sorted logics. The goal is to encode TLA$^+$ expressions using essentially first-order logic and uninterpreted functions. Consequently, this embedding can only handle a fragment of the language. Specifically, the TLA$^+$ constructs containing second-order sub-expressions, such as $\{x \in S : P\}$, $[x \in S \mapsto e]$ or CHOOSE $x \colon P$, cannot be directly mapped to first-order sentences.

4.3.1. Set theory

First-order formulas with equality have a direct counterpart in any of the considered solvers. Given that each pair of symbols have equivalent semantics, it suffices to apply a *shallow embedding*, that is, a mapping of the TLA$^+$ quantifiers, logical connectors and the equality symbol to their corresponding entities in the solvers' input languages.

Declaring operators When the target language is unsorted, we encode the non-Boolean and Boolean operators respectively as functions and predicates, while respecting their arities. For example, binary operators like $\cup$ and $\in$ are represented in the target language by the function union and the predicate in, both of arity two. In a similar way, when encoding into a sorted language, we represent the universe of TLA$^+$ values with the predefined Bool sort and a newly declared sort U. Therefore,

TLA$^+$ operators are declared as unspecified function or predicate symbols with U-sorted arguments. In general, any non-Boolean operator p and a Boolean operator q, both of arity n, are represented respectively by the function and predicate symbols

$$\mathsf{p} : \underbrace{\mathsf{U} \times \ldots \times \mathsf{U}}_{n} \to \mathsf{U} \quad \text{and} \quad \mathsf{q} : \underbrace{\mathsf{U} \times \ldots \times \mathsf{U}}_{n} \to \mathsf{Bool}.$$

For example, the operators $\cup$ and $\in$ are encoded as the functions

$$\mathsf{union} : \mathsf{U} \times \mathsf{U} \to \mathsf{U} \quad \text{and} \quad \mathsf{in} : \mathsf{U} \times \mathsf{U} \to \mathsf{Bool}.$$

In this chapter, when encoding the variables and operators of a proof obligation, we make no use of other sorts, with the exception of a special sort for strings, as explained later.

Axiomatization We define the semantics of standard TLA$^+$ operators axiomatically. This means that we can give a meaning to the uninterpreted functions representing these operators by adding as hypotheses to the translation the universal closure of the axioms that define them. In the case of the elements of set theory, the only primitive operator is $\in$, so the function in will remain unspecified. The other set objects are defined in terms of $\in$, $=$ and first-order formulas, using the definitions (2.3)–(2.7) of Section 2.1. For example, the axiom for $\cup$ that corresponds to the formula (2.3) can be expressed in unsorted FOF as

$$\forall x, S, T.\ \ \mathsf{in}(x, \mathsf{union}(S, T)) \Leftrightarrow \mathsf{in}(x, S) \vee \mathsf{in}(x, T) \tag{4.1}$$

Note that sets are just values in the universe of discourse (represented by the sort U in the sorted translation), and it is therefore possible to represent sets of sets and to quantify over sets. The construct for set enumeration $\{e_1, \ldots, e_n\}$, with $n \geq 0$, is an n-ary expression, so we declare separate uninterpreted functions just for the arities that occur in the proof obligation, together with the corresponding axioms according to formula (2.5). For example, if a set of two elements appears in a proof obligation, then the following axiom would be added to the SMT-LIB file:

$$\forall x : \mathsf{U}, y_1 : \mathsf{U}, y_2 : \mathsf{U}.\ \ \mathsf{in}(x, \mathsf{setenum}_2(y_1, y_2)) \Leftrightarrow x = y_1 \vee x = y_2$$

where $\mathsf{setenum}_2 : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$ represents the sets with two arbitrary elements.

4.3.2. Sorted theories

In a pure first-order logic as that of the language FOF there is no theory of arithmetic. Still, we want to translate somehow the arithmetic expressions appearing in a proof obligation. If these expressions happen to not be relevant for the validity of the

formula, we would not be ruling out a potentially valid proof obligation. In these cases, we encode the set of integers Int and the integer literals as the constant symbols tla_Int, 0, 1, 2, etc. For example, the formula $3 \in Int$ is translated as $\mathsf{in}(3, \mathsf{tla\_Int})$. Since there are no axioms or arithmetic theory to reason about the symbols 3 and Int, it would not be possible for a first-order prover to verify the validity of the formula. The same applies for arithmetic operators. For instance, we map the addition operator $+$ to an uninterpreted binary function plus.

In order to reason about the theory of arithmetic, an automated prover demands type information, either generated internally, or provided explicitly in its language, as in SMT-LIB. The axioms that we have presented so far rely on first-order logic over uninterpreted function and predicate symbols over the single sort U. For arithmetic reasoning, we want to benefit from the solvers' native capabilities. We declare an unspecified, injective function $\mathsf{int2u} : \mathsf{Int} \to \mathsf{U}$ that embeds SMT integers into the sort representing TLA$^+$ values. Integer literals k are translated as $\mathsf{int2u}(k)$. The previous example $3 \in Int$ is now translated as $\mathsf{in}(\mathsf{int2u}(3), \mathsf{tla\_Int})$ and we have to add to the translation the axiom for Int:

$$\forall n : \mathsf{Int}.\ \ \mathsf{in}(\mathsf{int2u}(n), \mathsf{tla\_Int}) \tag{4.2}$$

Arithmetic operators over TLA$^+$ values are defined homomorphically over the image of int2u by axioms such as

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ \ \mathsf{plus}(\mathsf{int2u}(m), \mathsf{int2u}(n)) \ = \ \mathsf{int2u}(m + n) \tag{4.3}$$

where $+$ on the right-hand side denotes the built-in addition over SMT integers. For the other arithmetic operators, we define analogous axioms. For example, TLA$^+$ inequality $\leq$ and the interval operator .. are represented in a sorted language respectively by the functions $\mathsf{leq} : \mathsf{U} \times \mathsf{U} \to \mathsf{Bool}$ and $\mathsf{interval} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$, and their corresponding axioms are

$$\forall m, n : \mathsf{Int}.\ \ \mathsf{leq}(\mathsf{int2u}(m), \mathsf{int2u}(n)) \Leftrightarrow m \leq n \tag{4.4}$$

$$\forall m, n : \mathsf{Int},\ x : \mathsf{U}.\ \ \mathsf{in}(x, \mathsf{interval}(\mathsf{int2u}(m), \mathsf{int2u}(n))) \Leftrightarrow \exists k : \mathsf{Int}.\ x = \mathsf{int2u}(k) \wedge m \leq k \wedge k \leq n \tag{4.5}$$

Similarly, the values of TLA$^+$ integer division (idiv) and modulus (mod) operators are specified only when the second argument is a positive integer. This is reflected in the axioms (4.6) and (4.7), where the SMT symbols $\div$ and % correspond to interpreted functions in SMT-LIB.

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ \ n > 0 \Rightarrow \mathsf{idiv}(\mathsf{int2u}(m), \mathsf{int2u}(n)) \ = \ \mathsf{int2u}(m \div n) \tag{4.6}$$

$$\forall m : \mathsf{Int}, n : \mathsf{Int}.\ \ n > 0 \Rightarrow \mathsf{mod}(\mathsf{int2u}(m), \mathsf{int2u}(n)) \ = \ \mathsf{int2u}(m \ \% \ n) \tag{4.7}$$

In all these cases, type inference is, in some sense, delegated to the SMT solver. The link between SMT operations and their TLA$^+$ counterparts is effectively defined

```
1 axiom  isint(0)
2 axiom  isint(1)
3 axiom  ∀X, M, N. in(X, tla_Nat) ⇔ isint(X) ∧ leq(0, X)
4 axiom  ∀X, M, N. in(X, interval(M, N)) ⇔
                        isint(M) ∧ isint(N) ∧ isint(X) ∧ leq(M, X) ∧ leq(X, M)
5 axiom  ∀S. boolify(isFiniteSet(S))
             ⇒ ∀N. N = cardinality(S)
                     ⇔ in(N, tla_Nat)
                         ∧ ∃F. boolify(isBijection(F, interval(1, N), S))
6 conjecture  ∀S. boolify(isFiniteSet(S)) ⇒ in(cardinality(S), tla_Nat)
```

Figure 4.3.: TPTP/FOF encoding of $\forall S\colon \mathit{IsFiniteSet}(S) \Rightarrow \mathit{Cardinality}(S) \in \mathit{Nat}$ (in a pretty-printed presentation)

only for values in the range of the function int2u, and type inference is performed by the SMT solver during the proof attempt. This approach can be extended to other useful theories that are natively supported by some SMT solvers, such as arrays or algebraic datatypes, if needed.

4.3.3. Examples

Toy example one The cardinality of finite sets can be represented as a unary constant operator defined by the axiom $\mathit{CardinalityAxiom}$, as already described in page 19. Consider the following lemma and its proof:

> LEMMA $\mathit{CardinalityInNat} \triangleq \forall S\colon \mathit{IsFiniteSet}(S) \Rightarrow \mathit{Cardinality}(S) \in \mathit{Nat}$
> BY $\mathit{CardinalityAxiom}, \mathit{ATP}$

The definitions for predicates $\mathit{IsFiniteSet}$ and $\mathit{IsBijection}$ are irrelevant for the proof. The Proof Manager generates a proof obligation whose plain translation (without optimizations) to FOF is presented in Figure 4.3. Lines 3 and 4 give the axioms for tla_Nat and interval, respectively. They are not required for the validity of the lemma, but we include them in the translation simply because they occur in the obligation. Note that the function leq representing the operator $\leq$ is left unspecified. Line 5 is the translation of the axiom $\mathit{CardinalityAxiom}$. Finally, line 6 corresponds to the statement of the theorem.

Toy example two Consider the simple TLA$^+$ proof obligation

$$\forall x \in \mathit{Int}\colon x + 0 = x.$$

```
1 declare  int2u : (Int) U
2 declare  plus : (U U) U
3 assert  ∀m, n : Int. int2u(m) = int2u(n) ⇒ m = n
4 assert  ∀m, n : Int. plus(int2u(m), int2u(n)) = int2u(m + n)
5 assert ¬(∀x : U. (∃n : Int. x = int2u(n)) ⇒ plus(x, int2u(0)) = x)
```

Figure 4.4.: SMT-LIB encoding of $\forall x\colon x \in \mathit{Int} \Rightarrow x + 0 = x$ (in a pretty-printed presentation)

Its translation to SMT-LIB is shown in Figure 4.4: line 3 states the injectivity of int2u, line 4 corresponds to the axioms of addition, and line 5 to the proper (negated) proof obligation. Let us illustrate the interplay of the previous axioms on this concrete example. By Skolemization on line 5, the solver introduces a new constant, say n, of sort Int, such that $x = \mathsf{int2u}(n)$. It can then reason as follows:

$$\begin{aligned}
\mathsf{plus}(x, \mathsf{int2u}(0)) &= \mathsf{plus}(\mathsf{int2u}(n), \mathsf{int2u}(0)) && (x = \mathsf{int2u}(n)) \\
&= \mathsf{int2u}(n + 0) && \text{(by axiom 4.3)} \\
&= \mathsf{int2u}(n) && \text{(by the SMT arithmetic} \\
& && \text{decision procedure)} \\
&= x && (x = \mathsf{int2u}(n)).
\end{aligned}$$

4.4. Preprocessing and optimizations

The encoding described above has simple translation rules and is easy to implement but it has two limitations. First, the set theory of TLA$^+$ is not finitely axiomatizable. Therefore, some TLA$^+$ expressions cannot be encoded as first-order axioms. Namely, they are $\{x \in S : P\}$, $\{e : x \in S\}$, CHOOSE $x\colon P$ and $[x \in S \mapsto e]$, where the predicate P and the expression e, both of which may have x as free variable, become second-order variables when quantified. Secondly, the above encoding does not perform and scale well in practice; the back-end solvers are unable to prove even the simplest proof obligations. Consider for instance the TLA$^+$ formula $2 \in 0\,..\,3$, translated as

$$\mathsf{in}(\mathsf{int2u}(2), \mathsf{interval}(\mathsf{int2u}(0), \mathsf{int2u}(3))).$$

This formula is obviously provable using axiom (4.5) about integer intervals, but the automatic provers fail to find suitable instances of the axiom formula.

State-of-the-art SMT solvers provide *instantiation patterns* (see Section 3.2) to control the potential explosion in the number of ground terms generated for instantiating quantified variables, but we have not been able to come up with patterns to attach

to the axiom formulas that would significantly improve the performance, or at least to prove this simple theorem.[1]

What we do instead is to perform several transformations to the TLA$^+$ proof obligation to obtain an equi-satisfiable formula which, in its normalized basic form, could be straightforwardly passed to the solvers using the above encoding. Since these methods transform the structure of the proof obligation, the quality of the translation will have a notable impact on the success of the ensuing application of the automatic provers. We consider a translation better than another when it enables a system to find a proof or a counter-model, preferably in a shorter period of time. Since this is an undecidable criterion, there cannot be an optimal translation in this sense. As a result, we acquiesce in applying the following heuristics: the fewer non-basic expressions and the fewer quantified formulas a TLA$^+$ proof obligation has, the easier for the solvers to find a proof or a counter-model. By reducing the number of non-basic expressions, we are reducing at the same time the number of axioms and quantified formulas in the translation.

4.4.1. Normalization

We define a rewriting process that systematically expands the definitions of the non-basic operators. Instead of letting the solver find instances of the background axioms of the previous section, it applies the "obvious" instances of those axioms during the translation. In most cases, we can eliminate all non-basic operators, and therefore the solvers do not have to find suitable axiom instances. The definitions of non-primitive operators provide the first rewriting rules. For instance, the formula (2.3) yields the rule

$$x \in e_1 \cup e_2 \longrightarrow x \in e_1 \vee x \in e_2.$$

The above example $2 \in 0\,..\,3$ can be eventually translated to $2 \in \mathit{Int} \wedge 0 \leq 2 \wedge 2 \leq 3$ by matching the rewriting rule for integer intervals

$$x \in a\,..\,b \longrightarrow x \in \mathit{Int} \wedge a \leq x \wedge x \leq b,$$

thus avoiding an existential quantifier compared to the use of axiom (4.5).

Rewriting permits also to optimize the translation by reducing the number of non-basic operators occurring in a proof obligation, and thus reducing the number of required axioms. Other trivial rewriting rules, such as $x \in \{\} \longrightarrow \textsc{false}$ and $x \cup \{\} \longrightarrow x$, allow to further shorten the proof obligation. Since we can distinguish between Boolean and non-Boolean expressions, we add some trivial *conditional* rewriting rules for expressions of Boolean type, like

$$x \in \textsc{boolean} \overset{x:\mathsf{Bool}}{\longrightarrow} \textsc{true}.$$

[1] Böhme et al. [BBP13] observed the same behavior in his encoding of HOL formulas for Sledgehammer.

The situation is different for arithmetic expressions, where we can reduce $x + 0$ to x, as it appears in the example of Fig. 4.4, only if x is known to be an integer. To infer integer types, we will have to wait until the next chapter.

All defined rewriting rules apply equivalence-preserving transformations. We ensure the soundness of the rewriting method by proving that formulas corresponding to the rewriting rules are theorems in Isabelle/TLA$^+$, our trusted kernel. The theorems corresponding to rules $\phi_1 \longrightarrow \phi_2$ are: $\phi_1 \Leftrightarrow \phi_2$ when $\phi_1 : \mathsf{Bool}$ and $\phi_2 : \mathsf{Bool}$, and $\phi_1 = \phi_2$ in any other case. Most of these theorems are already part of Isabelle/TLA$^+$'s library, and, for those that are not, it is fairly trivial to prove their validity. The collection of rewriting rules are found in Appendix A.

Instantiating extensionality

The extensionality axiom for sets (2.1) quantify over variables ranging over the whole universe of discourse. Including this axiom by default in the translation forces the solvers to generate instances over all U-sorted values. To avoid attaching extensionality to the translation, we instantiate equality expressions $x = y$ whenever possible, that is, when x or y are not terms, with its corresponding extensionality property. In these cases, we say that we *expand* equality. Knowing the kind of an expression allows us to disambiguate the translation of equality, rather than lifting equality on the U sort. For each set object T we derive rewriting rules for expressions of the form $x = T$ and $T = x$, where x is a term. For instance, the following rules are derived from set extensionality and the axioms (2.3), (2.5), and (2.6), respectively.

$$S = T \cup R \longrightarrow \forall x : x \in S \Leftrightarrow x \in T \vee x \in R \tag{4.8}$$

$$S = \{a, b\} \longrightarrow \forall x : x \in S \Leftrightarrow x = a \vee x = b \tag{4.9}$$

$$S = \{x \in T : P(x)\} \longrightarrow \forall x : x \in S \Leftrightarrow x \in T \wedge P(x) \tag{4.10}$$

As detailed later, we define a corresponding rule for an instance of function extensionality, and analogous rules for the equality of two n-tuples and two records of the same shape.

Including the general extensionality axiom for sets (2.1) in the translation requires some experimentation to find a good balance between efficiency and completeness. For efficiency reasons we decided not to include it, and instead just apply these rewriting rules, rendering the translation incomplete. When required, the user will need to explicitly add the axiom to the TLA$^+$ proof.

Termination and confluence

All above rules of the form $\phi \longrightarrow \psi$ define a term rewriting system (TRS) noted (TLA$^+$, $\longrightarrow$), where $\longrightarrow$ is a binary relation over syntactically valid TLA$^+$ expres-

sions. We prove that $(\text{TLA}^+, \longrightarrow)$ has the properties of termination and confluence, according to the definitions in Section 3.2.

Theorem 5. $(\text{TLA}^+, \longrightarrow)$ *terminates.*

Proof. The most common way to prove termination is to embed the TRS into another reduction system that is known to terminate, typically the system $(\mathbb{N}, >)$ [BN99]. Thus, we embed $(\text{TLA}^+, \longrightarrow)$ through a monotone mapping $\mu\colon \text{TLA}^+ \to \mathbb{N}$ into $(\mathbb{N}, >)$. We just have to find an assignment μ such that $\mu(\phi) > \mu(\psi)$, for every rule $\phi \longrightarrow \psi$.

Some rewriting rules reduce the number of non-basic expressions, while others reduce the number of quantifiers, or rewrite a non-basic expression in terms of another one. We opt to assign to every TLA^+ symbol and construct e a coefficient $\mathcal{W}(e)$ through the relation $=_{\mathcal{W}}$, as shown in the Table 4.1. The lower the coefficient given to e, the more preference is given to e to be used in the right-hand side of the rules. Remember that what is considered non-basic, depends on the target language. Basic operators weigh 0, including $\in$ and arithmetic expressions, except for IF-THEN-ELSE expressions that are non-basic in TLA^+-FOF. Non-basic expressions have a higher coefficient than quantifiers, some more than others, depending if they are expressed in terms of quantifiers or other non-basic expressions.

Let $e|_p$ denote the sub-expression of e at some position p of its syntactic tree. We define the embedding ϕ of an expression e as the addition of the coefficients $\mathcal{W}(e|_p)$ for every position p in e:

$$\mu(e) \triangleq \mathcal{W}(e|_1) + \ldots + \mathcal{W}(e|_n)$$

For instance, the rule (4.10) weighs 3 in the left-hand side, which accounts only for $\{\_ \in \_ : \_\}$, and 1 for the right-hand side, corresponding to the quantifier. Given this coefficient assignment, one can easily check in all rewriting rules that the weight of the lhs is strictly larger than that of the rhs. Therefore, we can embed $(\text{TLA}^+, \longrightarrow)$ into a terminating system. □

The rewriting system $(\text{TLA}^+, \longrightarrow)$, as it was presented above, is not confluent. Consider the expression $S = T \cup \{\}$, which can be reduced to two different normal forms. One possible path is $S = T \cup \{\} \longrightarrow S = T$. Another possible path is

$$\begin{aligned} S = T \cup \{\} &\longrightarrow \forall x\colon x \in S \Leftrightarrow x \in T \vee x \in \{\} \\ &\longrightarrow \forall x\colon x \in S \Leftrightarrow x \in T \vee \textsc{false} \\ &\longrightarrow \forall x\colon x \in S \Leftrightarrow x \in T \end{aligned}$$

The first rules applied are $x \cup \{\} \longrightarrow x$ in the first case, and (4.8) in the second. In particular, the latter rule expands equality by instantiating extensionality. These two rules overlap, giving rise to the critical pair

$$\langle S = T,\ \forall x\colon x \in S \Leftrightarrow x \in T \vee x \in \{\} \rangle.$$

$v, w(\_, \ldots, \_) =_{\mathcal{W}} 0$	$0, 1, 2, \ldots =_{\mathcal{W}} 0$
$\textsc{false}, \textsc{true} =_{\mathcal{W}} 0$	$+, -, * =_{\mathcal{W}} 0$
$\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, = =_{\mathcal{W}} 0$	$<, \leq, \geq, > =_{\mathcal{W}} 0$
$\neq =_{\mathcal{W}} 1$	$Int, Nat =_{\mathcal{W}} 1$
$\forall, \exists \_ : \_ =_{\mathcal{W}} 1$	$\div, \% =_{\mathcal{W}} 2$
$\textsc{if} \_ \textsc{then} \_ \textsc{else} \_ =_{\mathcal{W}} 1$	$.. =_{\mathcal{W}} 2$
$\in =_{\mathcal{W}} 0$	$\textsc{domain} =_{\mathcal{W}} 1$
$\notin =_{\mathcal{W}} 1$	$\_[\_] =_{\mathcal{W}} 2$
$\{\}, \{\_, \ldots, \_\}, \subseteq, \cup, \cap, \setminus =_{\mathcal{W}} 2$	$[\_ \in \_ \mapsto \_] =_{\mathcal{W}} 2$
$\textsc{subset}, \textsc{union} =_{\mathcal{W}} 3$	$[\_ \textsc{except} \_ = \_] =_{\mathcal{W}} 3$
$\{\_ \in \_ : \_\}, \{\_ : \_ \in \_\} =_{\mathcal{W}} 3$	$[\_ \to \_] =_{\mathcal{W}} 3$

Table 4.1.: Weight coefficients of TLA$^+$ symbols and constructs for termination proof

Both expressions in the pair are derived from $S = T \cup \{\}$, which results from the unification of the left-hand side of the first rule and the non-variable sub-term $T \cup \{\}$ of the left-hand side of the latter. Since this critical pair is not joinable, we can assert that the system is non-confluent.

The non-confluence of $(\text{TLA}^+, \longrightarrow)$ comes from the extensionality expansion rules for sets. In the above example, the second path can be reduced to $x = y$ by contracting extensionality, that is, by applying the rule in the opposite direction, thus obtaining a common normal form. Accordingly, we add a rule to $(\text{TLA}^+, \longrightarrow)$ for the *contraction* of set extensionality

$$\forall z : z \in x \Leftrightarrow z \in y \longrightarrow x = y \tag{4.11}$$

which we apply with a higher priority than the expansion rules. The above proof of termination for this extended system is still valid. Now we can prove that $(\text{TLA}^+, \longrightarrow)$ plus the rule (4.11), which is still terminating, is confluent.

Theorem 6. $(\text{TLA}^+, \longrightarrow)$ *is confluent.*

Proof. By Corollary 4, it suffices to prove that all critical pairs are joinable. Thus, we just need to find the critical pairs $\langle \phi_1, \phi_2 \rangle$ between all combinations of rewriting rules, and then prove that ϕ_1 and ϕ_2 are joinable for each such pair. This is done by reducing the terms to some normal forms ϕ_1' and ϕ_2' and showing that they are syntactically equal, which is trivial. □

4.4.2. Functions

Semantically speaking, TLA$^+$ functions are special values in the universe of discourse where all elements denote sets. The TLA$^+$ function $[x \in S \mapsto e(x)]$ resembles a "bounded" λ-abstraction with the difference that the function application

$[x \in S \mapsto e(x)][y]$ reduces to the expected value $e(y)$ if the argument y is an element of S, as stated by the axiom (2.10). As a consequence, for example, the formula

$$f = [x \in \{1,2,3\} \mapsto x * x] \quad \Rightarrow \quad f[0] < f[0] + 1,$$

although syntactically well-formed, should not be provable. In particular, since 0 is not in the domain of f, we cannot deduce that $f[0]$ is an integer.

Function application

The application of f to x has two values depending on whether the *domain condition* $x \in \text{DOMAIN } f$ holds or not. We represent these values respectively using two special TLA$^+$ binary operators α and ω that take as arguments a (unary) function and its argument, as conditionally defined by

$$x \in \text{DOMAIN } f \Rightarrow \alpha(f,x) = f[x] \quad \text{and} \quad x \notin \text{DOMAIN } f \Rightarrow \omega(f,x) = f[x].$$

Using these definitions, we can derive the following theorem that gives a new defining equation for function application:

$$f[x] = \text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f,x) \text{ ELSE } \omega(f,x) \tag{4.12}$$

In this way, functions are just expressions that are conditionally related to their argument by α and ω. Moreover, from the axiom (2.10), and using the definition (4.12), we can deduce the theorems:

$$a \in S \Rightarrow \alpha([x \in S \mapsto e], a) = e[x \leftarrow a] \tag{4.13}$$

$$\text{DOMAIN } [x \in S \mapsto e] = S \tag{4.14}$$

The theorem (4.13) gives the meaning of α for explicit functions, while in any other case stays unspecified; ω is always unspecified. The theorem (4.14) encapsulates the definition of DOMAIN. These theorems were proved in Isabelle/TLA$^+$.

Consequently, the expressions $f[0]$ in the above example would be encoded as

$$\text{IF } 0 \in \text{DOMAIN } f \text{ THEN } \alpha(f,0) \text{ ELSE } \omega(f,0).$$

The solver would have to use the hypothesis to deduce that $\text{DOMAIN } f = \{1,2,3\}$, reducing the condition $0 \in \text{DOMAIN } f$ to false. The conclusion would result in the formula $\omega(f,0) < \omega(f,0) + 1$, which does not have any defined value, as expected.

Another example is the formula $f[x] = f[y]$ in a context where $x = y$ holds: the formula is valid irrespective of whether the domain conditions hold or not. Using the definition (4.12), it is encoded as the following equivalent formula, which is deducible just by the fact $x = y$:

$$\begin{gathered}\text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f,x) \text{ ELSE } \omega(f,x) \\ = \\ \text{IF } y \in \text{DOMAIN } f \text{ THEN } \alpha(f,y) \text{ ELSE } \omega(f,y)\end{gathered}$$

In accordance with the above, we encode TLA$^+$ functions as follows. A term symbol f representing a function will be declared as an element f without arguments. In a sorted language, it will be simply declared having sort U. As any set or U-sorted element, TLA$^+$ functions can be quantified over. We map the special operators α and ω to binary unspecified functions, or by declaring them with sort $\mathsf{U} \times \mathsf{U} \to \mathsf{U}$.

Explicit functions

Whenever possible, we try to avoid the encoding of function application as in the definition (4.12). From the formula (4.13), and by the definitions (4.12), (4.14), and that of EXCEPT (2.12), we deduce rewriting rules for the application of the two function constructs to an arbitrary argument a:

$$[x \in S \mapsto e][a] \longrightarrow \textsc{if } a \in S \textsc{ then } e[x \leftarrow a] \textsc{ else } \omega([x \in S \mapsto e], a) \tag{4.15}$$

$$\begin{aligned}[f \textsc{ except } ![x] = y][a] \longrightarrow \textsc{if } & a \in \textsc{domain } f \\ & \textsc{then if } a = x \textsc{ then } y \textsc{ else } \alpha(f, a) \\ & \textsc{else } \omega([f \textsc{ except } ![x] = y], a)\end{aligned} \tag{4.16}$$

These rules replace two non-basic operators (function application and the function expression) in the left-hand side by only one non-basic operator in the right-hand side (the function as argument of ω).

The expression $[x \in S \mapsto e(x)]$ cannot be mapped directly to a FOL expression. Even in sorted languages like MS-FOL, first-order functions have no notion of function domain other than the types of their arguments. Since we cannot represent this expression straightforwardly in first-order languages we replace any such expression with a fresh variable symbol $\hat{f}$ giving it a definition $\hat{f} = [x \in S \mapsto e(x)]$ in the appropriate context. This encoding renders the translation equisatisfiable to the original formula, although not logically equivalent. The defining formula for $\hat{f}$ appears in the left-hand side of the equality in axiom (2.10). Directed from left to right, it results in a rewriting rule replacing the function construct by a formula containing only basic operators:

$$\begin{aligned}f = [x \in S \mapsto e] \longrightarrow & \wedge \mathit{IsAFcn}(f) \\ & \wedge \textsc{domain } f = S \\ & \wedge \forall y \in S : \alpha(f, y) = e[x \leftarrow y]\end{aligned} \tag{4.17}$$

Observe that we have simplified $f[y]$ by $\alpha(f, y)$, because $y \in \textsc{domain } f$. This mechanism to encode explicit function expressions summarizes the essence of the abstraction method to deal with non-basic operators that we describe in the next section.

Optimization of function application

The rewriting rule (4.15) for $[x \in S \mapsto e][a]$ results in an expression containing $\omega([x \in S \mapsto e], x)$ that represents the unspecified value of the function application when the domain condition does not hold (i.e. $a \notin S$). During the preprocessing, the first argument in $\omega([x \in S \mapsto e], x)$ will be eventually abstracted, resulting in $\omega(k_f, x)$ where $k_f = [x \in S \mapsto e]$. Since the definition of k_f is never needed for reasoning about the unspecified function application, we could just coalesce any expression $\omega(f, x)$ into a new term symbol $\omega_{f,x}$, thus avoiding the introduction of k_f. Function application could then be defined as

$$f[x] = \text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f, x) \text{ ELSE } \omega_{f,x}$$

to replace the definition (4.12). We just have to be careful to assign the same symbol $\omega_{f,x}$ for any given arguments f and x, in the same way as in the encoding of the formula $f[x] = f[y]$ in the example on page 77. The same optimization applies to the rule (4.16) for $[f \text{ EXCEPT } ![x] = y][a]$.

Another possibility to encode function application $f[x]$ is simply to translate it as $\alpha(f, x)$ and to collect all domain conditions $x \in \text{DOMAIN } f$ appearing in the proof obligation to check them separately. The domain conditions can be either sent to automatic provers as side conditions of the proof obligation, or conjuncted with the conclusion of the proof obligation. The latter option is what we have done in a previous implementation of the backends, whose results appeared in [MV12a] and [MV12b].

Function extensionality

The extensionality property for functions (2.11) is required to prove that two functions are equal. Therefore, we add as an axiom to the translation the formula:

$$\begin{aligned} \forall f, g: \; & \wedge \; IsAFcn(f) \wedge IsAFcn(g) \\ & \wedge \; \text{DOMAIN } f = \text{DOMAIN } g \\ & \wedge \; \forall x \in \text{DOMAIN } g: \alpha(f, x) = \alpha(g, x) \\ & \Rightarrow f = g \end{aligned}$$

Again, note that $f[x]$ and $g[x]$ can be simplified using α. Unlike set extensionality, this formula is guarded, avoiding the instantiation of expressions that do not satisfy $IsAFcn$, that is, those that are not functions.

In order to prove that $\text{DOMAIN } f = \text{DOMAIN } g$, we would still need to add to the translation the set extensionality axiom, which we abstain from. Instead, the reasoning about domains can be solved by the axiom:

$$\begin{aligned} \forall f, g: \; & \wedge \; IsAFcn(f) \wedge IsAFcn(g) \\ & \wedge \; \forall x: x \in \text{DOMAIN } f \Leftrightarrow x \in \text{DOMAIN } g \\ & \Rightarrow \text{DOMAIN } f = \text{DOMAIN } g \end{aligned}$$

which is an instance of set extensionality for DOMAIN expressions, guarded for functions only.

4.4.3. Abstraction

The rewriting process significantly reduces the number of non-basic operators that occur in a proof obligation, but it is not always possible to obtain a formula in basic normal form just by applying rewriting rules. The difficulty stems from the fact that remaining non-basic sub-expressions do not occur in the same form as the left-hand sides of the rewriting rules. What we do instead is to resort to auxiliary steps that transform the non-basic formula into a form suitable for rewriting. We call *abstraction* of non-basic expressions the technique described here. It is required to deal with the non-axiomatizable TLA$^+$ constructs mentioned at the beginning of this section, such as the function in the first argument of $\omega([x \in S \mapsto e], a)$ generated by the rule (4.15), as well as any other non-basic expression.

Briefly, what this method does is, for every occurrence in a proof obligation of a non-basic expression ψ, it introduces in its place a fresh term y, and adds the formula $y = \psi$, as an assumption in the appropriate context. The new term acts as an "abbreviation" for the non-basic expression, and the equality acts as its "definition", paving the way for a transformation to a basic expression using the above rewriting rules. Formally, we designate a *definition* as a hypothesis of the form $y = \psi$, possibly universally closed, where y is a term and ψ is a non-basic expression such that $y \notin FV(\psi)$.

In order to obtain a basic normal form, we should be able to expand the definitions $y = \psi$, for any non-basic expression ψ. Thus, we rely on the rewriting rules instance of extensionality as defined above, which can match and expand definitions. Systematically applying them allows us to remove all non-basic TLA$^+$ expressions to obtain an equi-satisfiable basic formula. On the other hand, when expanding extensionality, completeness may be lost in the translation. Therefore, we add instances of extensionality contraction, as shown in the example below.

Before a more formal description of the method, we introduce some notation and define a substitution on TLA$^+$ operators. The symbol $\mathbf{x}$ is a shorthand for sequences of n variable symbols $x_1, \ldots, x_n$. The number of elements n is usually interpreted from the context, unless stated otherwise. The substitution $[\mathbf{x} \leftarrow \mathbf{y}]$ is a shorthand for $[x_1 \leftarrow y_1] \ldots [x_n \leftarrow y_n]$, when $|\mathbf{x}| = |\mathbf{y}|$, i.e. $\mathbf{x}$ and $\mathbf{y}$ have the same length.

Definition 3 (Operator substitution)**.** *Given two* TLA$^+$ *expressions* e_1 *and* e_2*, where* $FV(e_2) = \boldsymbol{y}$*, and an operator symbol* $o \in \mathcal{O}$ *of arity* $|\boldsymbol{y}|$*. The expression* $e_1[o(\boldsymbol{x}) \leftarrow e_2]$*,*

with $|\boldsymbol{x}| = |\boldsymbol{y}|$, is inductively defined as follows:

$$
\begin{array}{rcl}
x[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & x \\
p(\boldsymbol{z})[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & e_2[\boldsymbol{y} \leftarrow \boldsymbol{z}] \qquad \textit{(when } p = o \textit{ and } |\boldsymbol{x}| = |\boldsymbol{z}|\textit{)} \\
p(e_1, \ldots, e_n)[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & p(e_1[o(\boldsymbol{x}) \leftarrow e_2], \ldots, e_n[o(\boldsymbol{x}) \leftarrow e_2]) \quad \textit{(otherwise)} \\
\textsc{false}[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & \textsc{false} \\
(\phi_1 \Rightarrow \phi_2)[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & (\phi_1[o(\boldsymbol{x}) \leftarrow e_2]) \Rightarrow (\phi_2[o(\boldsymbol{x}) \leftarrow e_2]) \\
(\forall x : \phi)[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & \forall x : (\phi[o(\boldsymbol{x}) \leftarrow e_2]) \\
(e = f)[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & (e[o(\boldsymbol{x}) \leftarrow e_2]) = (f[o(\boldsymbol{x}) \leftarrow e_2]) \\
(e \in f)[o(\boldsymbol{x}) \leftarrow e_2] & \triangleq & (e[o(\boldsymbol{x}) \leftarrow e_2]) \in (f[o(\boldsymbol{x}) \leftarrow e_2])
\end{array}
$$

Note that the substitution $[o(\mathbf{x}) \leftarrow e_2]$ is effectively applied on operators matching the same identifier and having as arguments only variables symbols (second line). When this is the case, the expression $p(\mathbf{z})$ is replaced by e_2 where its free variables $\mathbf{y}$ are replaced by p's arguments.

Regarding the abstraction method more generally, consider n non-basic expressions $\psi_1, \ldots, \psi_n$, where $FV(\psi_i) = \mathbf{x}_i$ for $i \in 1 \mathinner{..} n$, and $\mathbf{x}_i$ denotes a sequence $x_1^i, \ldots, x_m^i$ of m variable symbols. Suppose some TLA$^+$ formula P with no free variables containing $k_1(\mathbf{x}_1), \ldots, k_n(\mathbf{x}_n)$ as sub-expressions (i.e. $\mathbf{x}_i$ are bounded inside P), where k_i are operator symbols with arity $|\mathbf{x}_i|$, that is, the number of free variables of ψ_i. We can represent a non-basic proof obligation as:

$$P[k_1(\mathbf{x}_1) \leftarrow \psi_1, \ldots, k_1(\mathbf{x}_n) \leftarrow \psi_n].$$

The abstraction of this expression is the equi-satisfiable formula

$$
\begin{array}{l}
\land\ \forall \mathbf{x}_1' : k_1(\mathbf{x}_1') = \psi_1[\mathbf{x}_1 \leftarrow \mathbf{x}_1'] \\
\land\ \ldots \\
\land\ \forall \mathbf{x}_n' : k_n(\mathbf{x}_n') = \psi_n[\mathbf{x}_n \leftarrow \mathbf{x}_n'] \\
\Rightarrow P
\end{array}
$$

where the $\psi_i[\mathbf{x}_1 \leftarrow \mathbf{x}_1']$ denotes the expression ψ_i with its free variables $\mathbf{x}_i$ substituted by fresh variable symbols $\mathbf{x}_i'$ for every i, where $\mathbf{x}_i'$ is a sequence of $|\mathbf{x}_i|$ new symbols.

Toy example Consider the valid proof obligation

$$\forall x : P(\{x\} \cup \{x\}) \Leftrightarrow P(\{x\}).$$

The non-basic sub-expressions $\{x\} \cup \{x\}$ and $\{x\}$ are replaced by fresh constant symbols $k_1(x)$ and $k_2(x)$. Then, the abstracted formula is:

$$
\begin{array}{l}
\land\ \forall y_1 : k_1(y_1) = \{y_1\} \cup \{y_1\} \\
\land\ \forall y_2 : k_2(y_2) = \{y_2\} \\
\Rightarrow \forall x : P(k_1(x)) \Leftrightarrow P(k_2(x)).
\end{array}
$$

The formula is now in a form where it is possible to apply the rewriting rules instance of extensionality, which expand the equalities in the definitions of the non-basic expressions.

In order to preserve satisfiability of the proof obligation, we have to add as hypotheses instances of extensionality contraction for every pair of definitions where extensionality expansion was applied. The equi-satisfiable formula in basic normal form is:

$$\begin{aligned}
&\wedge\ \forall z, y\colon z \in k_1(y) \Leftrightarrow z = y \vee z = y \\
&\wedge\ \forall z, y\colon z \in k_2(y) \Leftrightarrow z = y \\
&\wedge\ \forall y_1, y_2\colon (\forall z\colon z \in k_1(y_1) \Leftrightarrow z \in k_2(y_2)) \Rightarrow k_1(y_1) = k_2(y_2) \\
&\Rightarrow \forall x\colon P(k_1(x)) \Leftrightarrow P(k_2(x)).
\end{aligned}$$

Soundness We prove that the abstraction method is sound, i.e. that if the translated obligation is provable then the original obligation is valid, by choosing appropriate interpretations for the introduced operators. Intuitively, if $\mathcal{M}_1$ is a model of the original formula, we can construct a model $\mathcal{M}_2$ of the transformation by extending $\mathcal{M}_1$ with interpretations for the new operator symbols k_i that have the same universal value as the expression ψ_i. Then, k_i necessarily denotes the same value as ψ_i by its definition introduced as a hypothesis. We prove this result on one abstracted expression. Systematically applying this elementary case results in the general case for more than one non-basic expression.

Before the actual proof, we show two intermediate results and introduce some notation. The interpretation $\mathcal{I} \oplus (k \mapsto \lambda \mathbf{x}.\, f(\mathbf{x}))$ extends $\mathcal{I}$ for the operator symbol k with a function $\mathcal{I}(k)(\mathbf{x}) \triangleq f(\mathbf{x})$, the symbol $\mathbf{d}$ denotes some sequence of symbols $d_1, \ldots, d_n \in \mathcal{D}^n$, and $\mathbf{x} \mapsto \mathbf{d}$ for the mapping $x_1 \mapsto d_1, \ldots, x_n \mapsto d_n$, and the symbol $\doteq$ means "is equal to".

Lemma 7. *Let $vars(e)$ and $ops(e)$ be the set of variable and operator symbols occurring in an expression e. For any expression e,*

(A_1) *if two valuations v_1 and v_2 agree on $vars(e)$, meaning that $v_1(x) \doteq v_2(x)$, for every variable symbol x in $vars(e)$, and*

(A_2) *if two interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ agree on $ops(e)$, meaning that $\mathcal{I}_1(o)$ and $\mathcal{I}_2(o)$ have the same value given the same arguments, for all operator symbol o in $ops(e)$,*

then $val_{\mathcal{D},v_1,\mathcal{I}_1}(e) \doteq val_{\mathcal{D},v_1,\mathcal{I}_2}(e)$.

Proof. By induction on the structure of e. In particular, we distinguish three relevant cases.

$\langle 1 \rangle 1$. CASE e is a variable x, then $vars(e) = \{x\}$. It follows that $val_{\mathcal{D},v_1,\mathcal{I}_1}(x) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(x)$, by assumption (A_1).

⟨1⟩2. CASE e is an operator o of arity n applied to expressions $e_1, \ldots, e_n$, then $ops(e) = \{o\} \cup ops(e_1) \cup \ldots \cup ops(e_n)$. By induction hypothesis, we have that $val_{\mathcal{D},v_1,\mathcal{I}_1}(e_i) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(e_i)$, for all $i \in 1\,..\,n$, and by assumption (A_2), then $val_{\mathcal{D},v_1,\mathcal{I}_1}(o(e_1, \ldots, e_n)) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(o(e_1, \ldots, e_n))$.

⟨1⟩3. CASE e is of the form $e_1 \Rightarrow e_2$ then $vars(e) = vars(e_1) \cup vars(e_2)$ and $ops(e) = ops(e_1) \cup ops(e_2)$. Since the interpretations agree on the union of the two sets, they agree on each of $ops(e_1)$ and $ops(e_2)$. We can therefore apply the inductive hypothesis to conclude that $val_{\mathcal{D},v_1,\mathcal{I}_1}(e_1) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(e_1)$, and that $val_{\mathcal{D},v_1,\mathcal{I}_1}(e_2) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(e_2)$. Since the evaluation of e is a function of these sub-evaluations, $val_{\mathcal{D},v_1,\mathcal{I}_1}(e) \doteq val_{\mathcal{D},v_2,\mathcal{I}_2}(e)$.

⟨1⟩4. QED, by ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, and the missing cases which are similar to these ones. □

Lemma 8. *Consider a formula P with no free variables, an expression e with free variables $\boldsymbol{x}$, and an operator symbol $k \in \mathcal{O}$ with arity $|\boldsymbol{x}|$. Let $\mathcal{I}' \triangleq \mathcal{I} \oplus (k \mapsto \lambda \boldsymbol{x}.\, val_{\mathcal{D},v,\mathcal{I}}(e(\boldsymbol{x})))$. Then*

$$val_{\mathcal{D},v,\mathcal{I}}(P[k(\boldsymbol{x}) \leftarrow e]) \doteq val_{\mathcal{D},v,\mathcal{I}'}(P).$$

Proof. By induction on the structure of P. In particular, when P is e, we know, because of the way we have defined $\mathcal{I}'$, that:

$$val_{\mathcal{D},v,\mathcal{I}}(e) \doteq \mathcal{I}'(k)(val_{\mathcal{D},v,\mathcal{I}}(\mathbf{x})) \doteq val_{\mathcal{D},v,\mathcal{I}'}(k(\mathbf{x})).$$

For other kinds of expressions, the interpretations follow the structure of the expression so the result follows easily by the inductive hypothesis. □

Theorem 9 (Abstraction soundness)**.** *Given an expression P with no free variables and an expression e with free variables $\boldsymbol{x}$. Let $d \triangleq \forall \boldsymbol{x}' : k(\boldsymbol{x}') = e[\boldsymbol{x} \leftarrow \boldsymbol{x}']$ where k is a fresh operator symbol not in $\mathcal{O}$ of arity $|\boldsymbol{x}|$, and $\boldsymbol{x}'$ is a sequence $x'_1, \ldots, x'_{|\boldsymbol{x}|}$ of fresh variable symbols. Then $\models d \Rightarrow P$ iff $\models P[k(\boldsymbol{x}) \leftarrow e]$, i.e. the abstracted and the original formulas are equisatisfiable.*

Proof. We proceed by case analysis on the bi-implication.

⟨1⟩1. CASE $\models d \Rightarrow P$. Suppose a model $\mathcal{M} \triangleq \langle \mathcal{D}, v, \mathcal{I} \rangle$ satisfying $d \Rightarrow P$. The goal is to prove $\mathcal{M} \models P[k(\mathbf{x}) \leftarrow e]$.

⟨2⟩1. CASE $\mathcal{M} \not\models d$, i.e. the truth value of d is false for $\mathcal{M}$, then $\mathcal{M} \models d \Rightarrow a$ holds for any expression a.

⟨2⟩2. CASE $\mathcal{M} \models d$, i.e. $\mathcal{M}$ satisfies $\forall \mathbf{x}' \colon k(\mathbf{x}') = e[\mathbf{x} \leftarrow \mathbf{x}']$.

⟨3⟩1. $val_{\mathcal{D},v,\mathcal{I}}(k(\mathbf{x})) \doteq val_{\mathcal{D},v,\mathcal{I}}(e)$.

⟨4⟩1. From assumption ⟨2⟩2 and the definitions of d, val and $truth$, we know that: $truth_{\mathcal{D},v,\mathcal{I}}(\forall \mathbf{x}' \colon k(\mathbf{x}') = e[\mathbf{x} \leftarrow \mathbf{x}']) \doteq \mathsf{T}$ iff $val_{\mathcal{D},v',\mathcal{I}}(k(\mathbf{x}')) \doteq val_{\mathcal{D},v',\mathcal{I}}(e[\mathbf{x} \leftarrow \mathbf{x}'])$, for all $\mathbf{d} \in \mathcal{D}^n$, where v' is $v \oplus (\mathbf{x}' \mapsto \mathbf{d})$.

⟨4⟩2. QED. In particular ⟨4⟩1 holds when $\mathbf{x}'$ is instantiated with $\mathbf{x}$, i.e. the free variables of e. Hence, the goal ⟨3⟩1 follows by Lemma 7, because in this case, v' is just v.

⟨3⟩2. $\mathcal{M} \models P$, by ⟨1⟩1, ⟨2⟩2, and the definition of $truth$ for $\Rightarrow$.

⟨3⟩3. QED. $\mathcal{M}$ satisfies $P[k(\mathbf{x}) \leftarrow e]$, by ⟨3⟩1 and ⟨3⟩2.

⟨2⟩3. QED, by ⟨2⟩1 and ⟨2⟩2.

⟨1⟩2. CASE $\models P[k(\mathbf{x}) \leftarrow e]$. Suppose $P[k(\mathbf{x}) \leftarrow e]$ is satisfiable, say by a model $\mathcal{M}_1 \triangleq \langle \mathcal{D}, v, \mathcal{I} \rangle$. The goal is to prove $\models d \Rightarrow P$.

⟨2⟩1. With the aim of making the truth value of d always true, we build a new model $\mathcal{M}_2 = \langle \mathcal{D}, v, \mathcal{I}' \rangle$. Since k is a fresh symbol, we can fix a value for it. We choose the interpretation $\mathcal{I}' \triangleq \mathcal{I} \oplus (k \mapsto \lambda \mathbf{x}'.\, val_{\mathcal{D},v,\mathcal{I}}(e[\mathbf{x} \leftarrow \mathbf{x}']))$.

⟨2⟩2. $\mathcal{M}_2 \models d$. By the definition of d, we know that:

$$val_{\mathcal{D},v,\mathcal{I}'}(d) \doteq truth_{\mathcal{D},v,\mathcal{I}'}(\forall \mathbf{x}' \colon k(\mathbf{x}') = e[\mathbf{x} \leftarrow \mathbf{x}']).$$

By the definition of $truth$ for quantified formulas, it suffices to prove the following equivalence, for all $\mathbf{d} \in \mathcal{D}^n$, where v' is $v \oplus (\mathbf{x}' \mapsto \mathbf{d})$:

$$\begin{aligned} val_{\mathcal{D},v',\mathcal{I}'}(k(\mathbf{x}')) &\doteq \mathcal{I}'(k)(\mathbf{x}') && \text{(by definition of } val) \\ &\doteq val_{\mathcal{D},v,\mathcal{I}}(e[\mathbf{x} \leftarrow \mathbf{x}']) && \text{(by definition of } \mathcal{I}') \\ &\doteq val_{\mathcal{D},v',\mathcal{I}'}(e[\mathbf{x} \leftarrow \mathbf{x}']) && \text{(by Lemma 7; } \mathbf{x}' \text{ does not occur in } e) \end{aligned}$$

⟨2⟩3. $\mathcal{M}_2 \models P$, because $val_{\mathcal{D},v,\mathcal{I}}(P[k(\mathbf{x}) \leftarrow e]) \doteq val_{\mathcal{D},v,\mathcal{I}'}(P)$, from assumption ⟨1⟩2 and by Lemma 8.

⟨2⟩4. QED. Model $\mathcal{M}_2$ satisfies $d \Rightarrow P$, by the two following steps:

⟨3⟩1. $\mathcal{M}_2 \models d \wedge P$ follows from ⟨2⟩2 and ⟨2⟩3.

⟨3⟩2. $\mathcal{M}_2 \models d \wedge a$ iff $\mathcal{M}_2 \models d \Rightarrow a$, for any expression a, because the truth value of d is always T, by ⟨2⟩2.

⟨1⟩3. QED, by ⟨1⟩1 and ⟨1⟩2. □

The step ⟨1⟩1 of this proof shows that the validity of the abstracted formula implies the validity of the original formula, that is, they are logically equivalent. However,

the implication in the opposite direction, proved in step $\langle 2 \rangle 2$, does not hold, because the formulas are not satisfied exactly by the same models.

4.4.4. Eliminating definitions

In order to improve the encoding, we introduce an optimization procedure that eliminates *definitions*, in the sense of the preceding sub-section. The process consists in collecting definitions of the form $x = \psi$, and then simply substituting every occurrence of the term x by the non-basic expression ψ in the rest of the context, by applying the equality oriented as the rewriting rule $x \longrightarrow \psi$. It has the opposite effect of the abstraction method, where definitions are introduced and afterwards expanded to basic expressions. The definitions we want to eliminate typically occur in the original proof obligation, that is, they are not artificially introduced. In the next sub-section, we will explain the interplay between normalization, definition abstraction, and definition elimination.

The purpose of this transformation is to produce expressions that can eventually be normalized to their basic form. The restriction that x does not occur in ψ avoids rewriting loops and ensures termination of this process. For instance, the two equations $x = y$ and $y = x$ will be transformed into $y = y$, which cannot further be changed.[2] After applying the substitution, we can safely discard from the resulting formula the definition $x = \psi$, when x is a variable. However, we must keep the definition if x is an applied operator. Suppose we discard an assumption $\text{DOMAIN}\ f = S$, where the conclusion is $f \in [S \to T]$. Only after applying the rewriting rules, the conclusion will be expanded to an expression containing $\text{DOMAIN}\ f$, but the discarded fact required to simplify it to S will be missing.

Example Consider the following simplified paradigmatic example, that is representative of actual proofs.

$$f \in [Int \to Int] \ \land\ f' = [f\ \text{EXCEPT}\ [0] = 1] \ \Rightarrow\ f'[0] = 1.$$

The simplification process replaces f' by its definition $[f\ \text{EXCEPT}\ [0] = 1]$ in the conclusion, resulting in the formula:

$$f \in [Int \to Int] \ \Rightarrow\ [f\ \text{EXCEPT}\ [0] = 1][0] = 1.$$

Now the conclusion is ready to be taken to a basic normal form. After applying the rewriting rule (4.16) for function application of EXCEPT, the conclusion becomes:

$$\begin{array}{l}\big(\text{IF}\ 0 \in \text{DOMAIN}\ f \\ \quad \text{THEN}\ (\text{IF}\ 0 = 0\ \text{THEN}\ 1\ \text{ELSE}\ \alpha(f, 0)) \\ \quad \text{ELSE}\ \ \omega([f\ \text{EXCEPT}\ [0] = 1], 0)\big) = 1\end{array}$$

[2] The problem of efficiently eliminating definitions from propositional formulas is a major open question in the field of proof complexity. The definition-elimination procedure can result in an exponential increase in the size of the formula when applied naïvely [Avi03].

By the first hypothesis and the axiom (2.13) for the construct $[\_ \to \_]$, we know that DOMAIN $f = Int$, which permits the process to reduce the left-hand side of the equality to 1, and finally the conclusion to TRUE.

Compare the size of the formula we would have obtained without the simplification step. The normalization of the original formula would have included two extra quantifiers and introduced the operators α and ω:

$$
\begin{array}{ll}
\land & \land \mathit{IsAFcn}(f) \\
& \land \textsc{domain}\ f = \mathit{Int} \\
& \land \forall x \in \mathit{Int} : \alpha(f, x) \in \mathit{Int} \\
\land & \land \textsc{domain}\ f' = \textsc{domain}\ f \\
& \land (\textsc{if}\ 0 \in \textsc{domain}\ f'\ \textsc{then}\ \alpha(f', 0)\ \textsc{else}\ \omega(f', 0)) = 1 \\
& \land \forall x \in \textsc{domain}\ f \setminus \{0\} : \alpha(f', x) = \alpha(f, x) \\
\Rightarrow & (\textsc{if}\ 0 \in \textsc{domain}\ f'\ \textsc{then}\ \alpha(f', 0)\ \textsc{else}\ \omega(f', 0)) = 1
\end{array}
$$

Moreover, besides the encoding of this formula, the translation would need to include the axioms for DOMAIN, Int, set difference, and singleton set.

4.4.5. Pre-processing algorithm

Now we can put together the encoding techniques described in the above sections in a single algorithm that we call *Preprocess*. We define it formally as follows:

$$
\begin{array}{lll}
\mathit{Preprocess}(\phi) \triangleq \phi & & \mathit{Reduce}(\phi) \triangleq \phi \\
\quad \triangleright \mathit{Boolify} & & \quad \triangleright \textsc{Fix}\ (\mathit{Simplify} \circ \mathit{Rewrite}) \\
\quad \triangleright \textsc{Fix}\ \mathit{Reduce} & & \quad \triangleright \textsc{Fix}\ (\mathit{Abstract} \circ \mathit{Rewrite})
\end{array}
$$

Here, FIX $\mathcal{A}$ means that step $\mathcal{A}$ is executed until reaching a fixed point, the combinator $\triangleright$, used to chain actions on a formula ϕ, is defined as $\phi \triangleright f \triangleq f(\phi)$, and function composition $\circ$ is defined as $f \circ g \triangleq \lambda\phi.\, g(f(\phi))$.

The *Preprocess* algorithm takes a TLA$^+$ formula ϕ, Boolifies it (Section 4.2), and then applies repeatedly the step called *Reduce*, until reaching a fixed point, to take the formula to a basic normal form. Only then the resulting formula ϕ_{FOF} or ϕ_{SMT} is ready to be translated to the target language using the embedding of Section 4.3. In turn, *Reduce* first simplifies the given formula (4.4.4) and applies the rewriting rules (4.4.1) repeatedly, and then applies abstraction (4.4.3) followed by rewriting repeatedly. Observe that the simplification step is in some sense opposite to the abstraction step: the first one eliminates every definition $x = \psi$ by using it as the rewriting rule $x \longrightarrow \psi$, while the latter introduces a new symbol x in the place of a expression ψ and asserts $x = \psi$, where ψ is non-basic in both cases. Therefore, simplification should only be applied before abstraction, and each of those should be followed by rewriting (see example on page 85).

Since the *Preprocess* algorithm is composed of sound sub-steps, as explained in each sub-section, then the overall algorithm is sound. The algorithm terminates, meaning that it will always compute a basic normal formula, but with a caveat: we have to be careful that *Simplify* and *Abstract* do not repeatedly abstract and then simplify the same expression. *Simplify* does not produce non-basic expressions, but *Abstract* generates definitions that can be processed by *Simplify*, reducing them again to the original non-basic expression. That is the reason for *Rewrite* to come after every application of *Abstract*: the new definitions are rewritten, usually by an extensionality expansion rule. In short, termination depends on the existence of extensionality rewriting rules for each kind of non-basic expression that *Abstract* may catch. Then, for any TLA$^+$ expression ϕ there exists an equi-satisfiable basic expression ϕ_{FOF} or ϕ_{SMT} in normal form that the algorithm will compute.

On the other side, our encoding is not semantically complete. Even if we assume that the automated theorem provers are semantically complete, it may happen that the translation of a semantically-valid TLA$^+$ formula becomes invalid when encoded. Due to efficiency reasons, we decided not to include the axiom of set extensionality explicitly in the translation, as explained on page 74. However, if we decide to include the extensionality axiom, the encoding would become complete. As a partial solution to the incompleteness, we do add to the translation instances of extensionality for functions, function domains, records and tuples when possible.

4.5. Encoding other constructs

4.5.1. IF-THEN-ELSE

The TLA$^+$ expression IF c THEN t ELSE u can be conveniently mapped verbatim using SMT-LIB's conditional operator to $\mathsf{ite}(c, t, u)$. However, TPTP/FOF does not provide any similar feature, so we have to encode the IF-THEN-ELSE expressions as first-order formulas.

When both t and u are propositions or Boolified expressions, noted ϕ_1 and ϕ_2, we can apply the following equivalence-preserving transformation, according to the IF-THEN-ELSE definition (2.14):

$$\text{IF } c \text{ THEN } \phi_1 \text{ ELSE } \phi_2 \longrightarrow c \Rightarrow \phi_1 \wedge \neg c \Rightarrow \phi_2 \tag{4.18}$$

For the general case, i.e. when t and u are terms, we define a set of rewriting rules

$$\mathcal{P}(\text{IF } c \text{ THEN } t \text{ ELSE } u) \longrightarrow \text{IF } c \text{ THEN } \mathcal{P}(t) \text{ ELSE } \mathcal{P}(u) \tag{4.19}$$

where $\mathcal{P}$ is a placeholder for a predicate or an operator parameterized by some other expression a. The formula $\mathcal{P}(a, e)$ can take the form $a = e$, $a \in e$, $a \Rightarrow e$, $a[e]$, etc.

For example, when $\mathcal{P}(a, e) \equiv a = e$, the rule becomes

$$a = \textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u \longrightarrow \textsc{if}\ c\ \textsc{then}\ a = t\ \textsc{else}\ a = u.$$

These rules are to be read modulo symmetry of the equality symbol in the left-hand side. The purpose of the set of rules (4.19) is to distribute $\mathcal{P}$ on the subexpressions t and u while pulling out the IF expression from the non-Boolean operators. Eventually, $\mathcal{P}$ will be a Boolean operator, allowing the application of rule (4.18).

This naïve approach to encode conditional term-expressions could result in an exponential blow-up on the size of the original formula, mainly if the sub-expressions are IF-THEN-ELSE themselves. The above rules introduce redundancies: the condition c appears twice in the right-hand side of rule (4.18), and the expression a is also repeated in the rules (4.19). We apply a simple heuristic to abstract the redundant expression. If c or a are just variables, we leave the translation as it is. Otherwise, we abstract the repeated expressions by satisfiability-preserving transformations. For instance, rule (4.18) is modified to introduce a fresh Boolean variable z as an abbreviation of c:

$$\textsc{if}\ c\ \textsc{then}\ \phi_1\ \textsc{else}\ \phi_2 \longrightarrow \exists z^{\mathsf{Bool}} : (z \Leftrightarrow c) \wedge (z \Rightarrow \phi_1) \wedge (\neg z \Rightarrow \phi_2)$$

where $z \Leftrightarrow c$ can be simplified afterwards. This abstraction method is applied analogously to the rules (4.19).

The list of rewriting rules for IF-THEN-ELSE expressions (proved as theorems in Isabelle/TLA$^+$) appears in Appendix A.

4.5.2. Strings

In FOF, we treat every string as a constant, being careful of avoiding name clashes with the variables. Then, we assert that every literal string occurring in the proof obligation is different from each other. In SMT-LIB, we declare a dedicated new sort Str for strings. Accordingly, we use an injective function str2u : Str $\to$ U to lift string expressions. The polyadic SMT-LIB construct distinct allows to easily abbreviate the above requirement. This encoding does not allow us to implement the standard TLA$^+$ interpretation of strings, which are considered as a tuple of characters. Fortunately, characters are hardly used in practice.

4.5.3. Tuples and records

Some SMT solvers support certain data constructors that could facilitate the encoding of TLA$^+$ tuples and record expressions, like Z3's algebraic data types or Yices' built-in records and tuples. Instead, we adopt a more agnostic approach for

TPTP/FOF and SMT-LIB, which currently do not have predefined theories for these data structures.

Tuples and records are functions defined using complex set and CASE expressions (cf. 2.17 and 2.19). Instead of unfolding those definitions, we just treat them as any other non-basic expression. Consequently, we need rewriting rules with the left-hand side in the form of definitions to be applied after the abstraction process:

$$\begin{aligned} t = \langle e_1, \ldots, e_n \rangle \longrightarrow & \wedge\ \mathit{IsAFcn}(t) \\ & \wedge\ \textsc{domain}\ t = 1\,..\,n \\ & \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(t, i) = e_i \\ & \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(t, i)^b \Leftrightarrow e_i \end{aligned} \tag{4.20}$$

$$\begin{aligned} r = [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \longrightarrow & \wedge\ \mathit{IsAFcn}(r) \\ & \wedge\ \textsc{domain}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\ & \wedge\ \textstyle\bigwedge_{i \in 1\,..\,n} \alpha(r, \text{“}\mathsf{h_i}\text{”}) = e_i \quad (\text{when } e_i : \mathsf{U}) \\ & \wedge\ \textstyle\bigwedge_{i \in 1\,..\,n} \alpha(r, \text{“}\mathsf{h_i}\text{”})^b \Leftrightarrow e_i \quad (\text{when } e_i : \mathsf{Bool}) \end{aligned} \tag{4.21}$$

In order to preserve satisfiability for expressions considered as terms from those considered as formulas, we treat differently the elements e_i that are Booleans (noted $e_i : \mathsf{Bool}$) from those that are not (noted $e_i : \mathsf{U}$). Since records and tuples are functions, projection in both cases is defined as a function application. The expression $r.h$ is encoded as $r[\text{“}\mathsf{h}\text{”}]$, as defined by standard TLA$^+$. Tuple selection $t[i]$ is just a function application and therefore translated as such. For this reason, we are allowed to use α instead of function application in rules (4.20) and (4.21), because the projections' arguments belong to the tuple and record domains.

We identify a record $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ and a tuple $\langle e_1, \ldots, e_m \rangle$ by their structure. We call the *signature* of a record the list **h** of its field names $h_1, \ldots, h_n$ in lexicographic order, and that of a tuple its arity m. As in the case of function extensionality, the properties of tuple and record extensionality are lost when equality is expanded. Whenever rules (4.20) or (4.21) are triggered on a record of signature **h** or on a tuple of signature m, we add the following formulas as axioms to the translation:

$$\begin{aligned} \forall r_1, r_2 : & \wedge\ \mathit{IsAFcn}(r_1) \wedge \mathit{IsAFcn}(r_2) \\ & \wedge\ \textsc{domain}\ r_1 = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\ & \wedge\ \textsc{domain}\ r_2 = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\ & \wedge\ r_1.h_1 = r_2.h_1 \wedge \ldots \wedge r_1.h_n = r_2.h_n \\ & \Rightarrow r_1 = r_2 \end{aligned} \tag{4.22}$$

$$\begin{aligned} \forall t_1, t_2 : & \wedge\ \mathit{IsAFcn}(t_1) \wedge \mathit{IsAFcn}(t_2) \\ & \wedge\ \textsc{domain}\ t_1 = 1\,..\,m \\ & \wedge\ \textsc{domain}\ t_2 = 1\,..\,m \\ & \wedge\ t_1[1] = t_2[1] \wedge \ldots \wedge t_1[m] = t_2[m] \\ & \Rightarrow t_1 = t_2 \end{aligned} \tag{4.23}$$

This approach is limited to records and tuples whose signature can be determined in an obvious way. Otherwise, they would be treated as any other expression.

Finally, we use the following theorems about function reasoning as rewriting rules:

$$
\begin{aligned}
IsAFcn([h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]) &\Leftrightarrow \textsc{true} \\
IsAFcn(\langle e_1, \ldots, e_n \rangle) &\Leftrightarrow \textsc{true} \\
\textsc{domain}\ [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] &= \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
\textsc{domain}\ \langle e_1, \ldots, e_n \rangle &= 1 \mathinner{..} n \\
r \in [h_1 : S_1, \ldots, h_n : S_n] &\Leftrightarrow \land\ IsAFcn(r) \\
&\quad \land\ \textsc{domain}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
&\quad \land\ r.h_1 \in S_1 \land \ldots \land r.h_n \in S_n \\
t \in S_1 \times \ldots \times S_n &\Leftrightarrow \land\ IsAFcn(t) \\
&\quad \land\ \textsc{domain}\ t = 1 \mathinner{..} n \\
&\quad \land\ t[1] \in S_1 \land \ldots \land t[n] \in S_n
\end{aligned}
$$

4.5.4. CHOOSE

The CHOOSE operator of TLA$^+$ is notoriously difficult for automatic provers to reason about. Nevertheless, we can exploit CHOOSE expressions by using the axioms that define them. Recall axiom (2.8):

$$\big(\exists x \colon P(x)\big) \Leftrightarrow P\big(\textsc{choose}\ x \colon P(x)\big).$$

By introducing a definition for $\textsc{choose}\ x \colon P(x)$, we obtain the theorem

$$y = \big(\textsc{choose}\ x \colon P(x)\big) \ \Rightarrow\ \big((\exists x \colon P(x)) \Leftrightarrow P(y)\big) \tag{4.24}$$

where y is some fresh constant symbol. This formula can be conveniently used as a rewriting rule for CHOOSE expressions that occur negatively, in particular, as hypotheses of proof obligations.

Suppose the expression $\phi \triangleq \textsc{choose}\ x \colon P(x)$, with free variables $\mathbf{x}$, occurs in a proof obligation. The abstraction process replaces ϕ by a term $k(\mathbf{x})$, with fresh operator symbol k, and asserts the definition

$$\forall \mathbf{x} \colon k(\mathbf{x}) = \textsc{choose}\ x \colon P(x)$$

as an axiom, preparing an application of theorem (4.24) as a rewriting rule, which results in

$$\forall \mathbf{x} \colon (\exists x \colon P(x)) \Leftrightarrow P(k(\mathbf{x})).$$

Note that if the CHOOSE expression is Boolified, then its abbreviation $f(\mathbf{x})$ is Boolified as well in the axiom. As it is shown here, this formula is difficult for the solvers to handle, because it contains nested quantifiers. Except for some pre-processing that could still be applied inside the predicate P, we do not further optimize the formula, although this could be desirable for an efficient support of CHOOSE expressions.

Choice determinism Suppose an arbitrary pair of CHOOSE expressions

$$\phi_1 \triangleq \text{CHOOSE } x\colon P(x) \qquad \text{and} \qquad \phi_2 \triangleq \text{CHOOSE } x\colon Q(x)$$

where $FV(\phi_1) = \mathbf{x}$ and $FV(\phi_2) = \mathbf{y}$, with $\mathbf{x} \triangleq x_1, \ldots, x_n$, $\mathbf{y} \triangleq y_1, \ldots, y_m$. In order to express the determinism of CHOOSE (axiom (2.9)), it is necessary to check that formulas P and Q are equivalent, for each pair of expressions $\langle \phi_1, \phi_2 \rangle$ occurring in a proof obligation. By abstraction of ϕ_1 and ϕ_2, we obtain the axiomatic definitions

$$\forall \mathbf{x}\colon f_1(\mathbf{x}) = \text{CHOOSE } x\colon P(x) \qquad \text{and} \qquad \forall \mathbf{y}\colon f_2(\mathbf{y}) = \text{CHOOSE } x\colon Q(x),$$

where f_1 and f_2 are fresh operator symbols of arity n and m respectively. Then, we state the extensionality property for the pair $\langle f_1, f_2 \rangle$ as the axiom

$$\forall \mathbf{x}', \mathbf{y}'\colon \big(\forall x\colon P(x)[\mathbf{x} \leftarrow \mathbf{x}'] \Leftrightarrow Q(x)[\mathbf{y} \leftarrow \mathbf{y}']\big) \Rightarrow f_1(\mathbf{x}') = f_2(\mathbf{y}').$$

4.6. Related work

Some of the encoding techniques and methods presented in this chapter were already defined before or are simply folklore, but they have not been combined and studied in this way. Moreover, the idiosyncrasies of TLA$^+$ render their applicability non-trivial. For instance, TLA$^+$'s axiomatized functions with domains, including tuples and records, are deeply rooted in the language. Besides the works mentioned previously in Section 1.3, here we briefly review some other encoding methods.

Boolification In the encoding of Isabelle/HOL into SMT-LIB [BBP13], formulas and terms are separated by treating all expressions as terms (no predicates are declared in the translation) and injecting expressions expected to be formulas into a new sort isomorphic to Bool. This encoding, inspired from Spark [JP09], is left intentionally incomplete.

Set theory encoding Initially, inspired by a translation for Event-B by Déharbe [Déh10], we encoded sets by their characteristic predicate, which allows for the direct translation of the set membership relation. In this translation, elementary sets are represented as uninterpreted functions, and the expression $x \in S$ is encoded as $S(x)$, where $S\colon \mathsf{U} \to \mathsf{Bool}$. This encoding results in a more effective translation than ours, since sets and their elements have different sorts, which implicitly divides the proof's search space. However, in order to stay within the realm of first-order logic, set of sets cannot be handled.

The plug-in for Rodin called SMT solvers [DFGV12] implements two translation approaches for set theory. In the first one, an extension of the previous work mentioned

above [Déh10], simple sets (no set of sets allowed) are directly encoded as polymorphic λ-expressions. These expressions are non-standard and are only supported by the parser of the veriT SMT solver, which however can be used as a pre-processor to produce standard SMT-LIB for other solvers.

The second approach simply calls the ppTrans plug-in [KV12] which generates different SMT sorts for each basic set and every combination of sets (power sets or cartesian products) found in the proof obligation. Therefore, there is one membership operator for every declared set-sort, and constants representing set elements are declared with the corresponding set-sort. The advantage of this encoding is that it further partitions the search space, although it requires to know beforehand the types of the basic set. In TLA$^+$, this can only be achieved through type inference. Additionally, when the ppTrans plug-in detects that the proof obligation does not contain any set of sets, the translation is further simplified by encoding sets by their characteristic predicates.

Mentre et al. [MMFA12] proposed Why3 as an interface to discharge Atelier-B proof obligations to different SMT solvers, with similar results to those of Rodin's SMT plug-ins. Set theory, including extensionality, is axiomatized as a new Why3 theory, where sets have an abstract, polymorphic type $\mathsf{set}\,\alpha$. The Alt-Ergo [CCKL08] SMT solver is particularly useful for Why3 because it natively handles polymorphic first-order formulas. Function applications are represented by a flat binary function and then axiomatized. In the context of the BWare project, Conchon et al. [CI14] proposed many internal optimizations to improve the performance of Alt-Ergo, in order to discharge Atelier-B proof obligations obtained from industrial settings.

Recently, Delahaye et al. [DDG$^+$13] proposed a different approach to reason about set theory, instead of a direct encoding into first-order logic. The theory of deduction modulo is an extension of predicate calculus, which allows to rewrite terms as well as propositions, and which is well suited for proof search in axiomatic theories, as it turns axioms into rewrite rules. For example, Peano arithmetic or Zermelo set theory can be encoded without axioms, turning the proof search among the axioms into computations. First-order theorem provers extended to deduction modulo were implemented as Zenon Modulo [DDG$^+$13, JBDD12] and iProver Modulo [Bur11].

Abstraction The abstraction technique to deal with non-basic expressions may recall the Tseitin transformation [Tse68] that replaces all sub-formulas of a CNF formula for new variables and its definitions. The goal is to avoid the exponential explosion in the number of clauses during clausification, i.e. the conversion from a FOL formula to an equi-satisfiable CNF formula. The resulting formula is in a form called *definitional*-CNF. The abstraction method, just as Tseitin's transformation and also Skolemization, does not preserve logical equivalence because of the new introduced function symbols, but preserves satisfiability, and this is sufficient for refutation-

based theorem proving. Other more sophisticated CNF conversion techniques do not introduce fresh names for all sub-formulas [NW01, AW13, dMB08].

The MPTP system [Urb03] translates Mizar to TPTP/FOF. The Mizar language provides second-order predicate variables and abstract terms, that is, a generalization of objects generated from the replacement and comprehension axioms, such as the set $\{n - m$ where m, n is $Integer : n < m\}$. To encode these kind of expressions, the translation applies a pre-processing method called *deanonymization* that replaces them by fresh symbols, called Fraenkel functors, with their definitions at the top level. As our abstraction method, it is comparable to Skolemization, but no further detail is provided in [Urb03].

The analogue of abstraction in λ-calculus is the λ-lifting transformation, that eliminates free variables from λ-expressions by introducing additional parameters to let-definitions and λ-applications. Thus, bounded let-definitions can be moved out to a global scope. This technique is used to encode Isabelle/HOL functions into MS-FOL [BBP13].

Chapter 5.

Type systems for TLA$^+$

Contents

> No hay clasificación del universo
> que no sea arbitraria y conjetural.
> La razón es muy simple: no
> sabemos qué cosa es el universo.
>
> Jorge Luis Borges, 1942

In line with the foundations of classical mathematics, TLA$^+$'s underlying logic is untyped. The rationale behind this choice is explained in [LP99], where Lamport and Paulson ask themselves and promote the debate about whether specification formalisms should be typed. When considered just for specification purposes, there are many arguments in favor of untyped languages, especially high flexibility and expressiveness. On the other hand, the current most mature and successful interactive verification platforms are based on languages with strong type systems, such as type theory or high-order logic. Type checkers automate the assessment of explicit or implicit type assignments while, in untyped formalisms, facts giving a sense of type information have to be asserted and proved as lemmas —what in TLA$^+$ is called

type-correctness invariants. In particular, type invariants often go beyond of what can be expressed in standard type systems because type invariants should be defined such that they are inductive with respect to the next-state relation of the specification. In set theory in particular, type-checking needs to be conducted explicitly as set-membership reasoning.

Another argument in favor of untyped specification languages is that it is quite unreasonable to expect a fixed type system to be applicable to different kinds of systems, from high-level descriptions of cloud algorithms to low-level implementations in assembly language or even hardware. For "wide-spectrum" languages such as TLA$^+$, untypedness is a reasonable choice —even more so since TLA$^+$ is designed to support proofs of refinement (or implementation) between quite different levels of specifications.

All in all, the overall proofs that have to be performed during the verification process are essentially the same in both kind of formalisms. The difference is that users of untyped languages need to do some extra work to prove that the typing lemmas are invariants and to apply them afterwards, during the proof attempt, when type information is required. Although there is a legitimate case for untyped specification languages, there is overwhelming evidence for the fact that automated reasoning benefits from the classification of expressions through type disciplines. As Cardelli and Wegner [CW85] wrote:

> As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

To close the gap between the two approaches, we propose automated procedures to construct types for TLA$^+$ expressions. The final goal is to use the gathered type information to improve our encodings into both sorted and unsorted languages of automated theorem provers.

5.1. Introduction

The translation presented in the previous chapter introduces a substantial number of quantified formulas interpreted over just one sort U that degrade the performance of automated theorem provers. Type inference, required for instance by arithmetic operators, is implicitly delegated to the solvers through the encoding. We illustrated this reasoning in the translation to SMT-LIB of the TLA$^+$ formula $\forall x \colon x \in Int \Rightarrow x + 0 = x$, shown previously in Figure 4.4. In particular, that translation lifts 0 to the universal sort U as $\mathsf{int2u}(0)$, and it encodes addition using an axiomatized

function. If we could detect appropriate type information from the original TLA$^+$ formula, specifically that the bound variable x is of integer type, we could translate it to SMT-LIB as

$$\forall x^{\mathsf{Int}}.\, \mathsf{in}(\mathsf{int2u}(x), \mathsf{tla\_Int}) \Rightarrow x + 0 = x,$$

using the SMT built-in operator $+$ instead of the axiomatized $\mathsf{plus} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$. We could do even better and apply *conditional* rewriting rules based on the type information. If the variable x is known to be of an integer type, noted $x : \mathsf{Int}$, the rewriting process would trigger on the original formula the rules

$$x \in \mathit{Int} \xrightarrow{x:\mathsf{Int}} \textsc{true} \qquad \text{and} \qquad x + 0 \xrightarrow{x:\mathsf{Int}} x.$$

The resulting formula after preprocessing would be simply TRUE.

The above example motivates the definition of type systems for TLA$^+$ and associated algorithms to synthesize types.[1] By necessity, type systems impose restrictions on the admissible formulas, and one can therefore not expect type inference to succeed for all TLA$^+$ proof obligations. If no meaningful types can be built, the translation can fall back to the "untyped" encoding of the previous chapter. The question is then how expressive the type system should be in order to successfully handle a large class of TLA$^+$ formulas. In this chapter we propose two type systems for TLA$^+$, that we call $\mathcal{T}_1$ and $\mathcal{T}_2$, with different expressive power.

The type system $\mathcal{T}_1$ is composed of elementary types like integers, Booleans, and functional types, which reflect the MS-FOL sorts in a natural way, and a Set type constructor which allows the stratification of set objects. This type system is fairly restricted and, in certain cases, cannot express adequate type information. TLA$^+$ functions are total: a function applied to any expression has a value, which is unspecified if the argument is not in the function's domain. For practical purposes, we consider functions as being partial, since we can give a type only to those function applications $f[x]$ satisfying the domain condition $x \in \textsc{domain}\, f$ (cf. Section 4.4.2). For this reason, handling function applications in TLA$^+$ requires precise type information of the domain.

Consider the invalid TLA$^+$ formula that appeared before on page 77,

$$f = [x \in \{1, 2, 3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1 \tag{5.1}$$

The system $\mathcal{T}_1$ over-approximates the type of f as a function from Int to Int. Even if the argument 0 is not in the domain of f, the expression $f[0]$ would be given the type Int. This does not corrupt soundness because the *domain condition* would fail

[1] It is customary to use the term *type inference* to denote the problem of deducing the types of an expression in a static type system. The *type reconstruction* problem is, given an expression containing type variables in place of some or all type annotations, to find a ground assignment for those variables such that the resulting expression is well-typed. *Type synthesis* or *type construction* suggests that types are built from scratch, which is what is required by an untyped language like TLA$^+$.

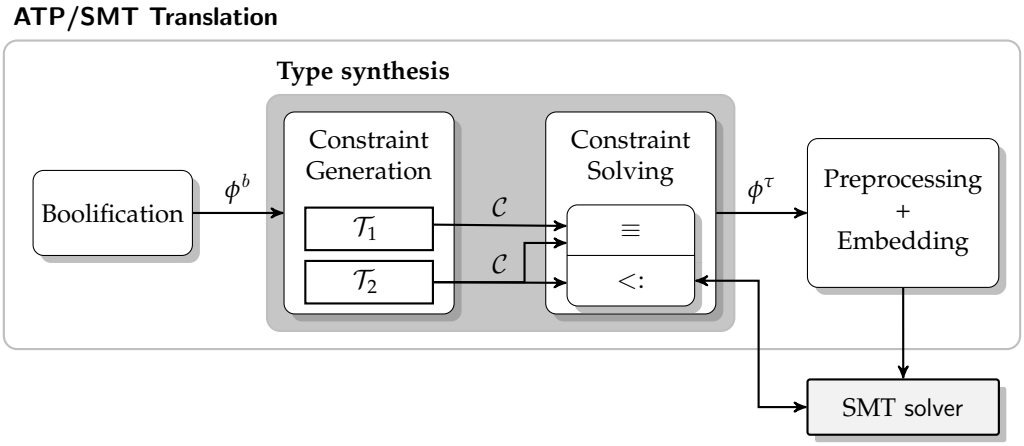


Figure 5.1.: Schematic view of type synthesis in the ATP/SMT backend

on $f[0]$. After type construction, we can assign a type to every sub-expression in the proof obligation. The untyped translation encodes an expression of the form $f[x]$ as the conditional IF $x \in$ DOMAIN f THEN $\alpha(f,0)$ ELSE $\omega(f,0)$, which compels the solver to prove or disprove $x \in$ DOMAIN f and to reason about the extra operators α and ω. Computing the domain of a TLA$^+$ function encoded in a first-order logic is not always easy, leading the provers to failed proof attempts. The design of an appropriate type system is further complicated by the fact that some formulas, such as $f[x] \cup \{\} = f[x]$, are actually valid irrespectively of whether the domain condition holds or not.

The above observations motivate the use of a more expressive type system. As an extension of $\mathcal{T}_1$, we introduce a second type system $\mathcal{T}_2$ based on dependent and refinement types [FP91, XP99]. Using refinement types, the type of DOMAIN f in the above example is $\{x : \mathsf{Int} \mid x = 1 \vee x = 2 \vee x = 3\}$, which represents the collection of those elements x with base type Int that satisfy the refinement predicate $x = 1 \vee x = 2 \vee x = 3$. During type construction, the system will try to prove $x = 0 \Rightarrow x \in$ DOMAIN f, and this will fail, hence the translation will fall back to the untyped encoding (which will in turn fail to prove the formula, as it should). In many practical examples, the domain condition $x \in$ DOMAIN f can be established during type construction, leading to shorter and simpler proof obligations.

One of the contributions of this chapter is the novel use of dependent and refinement types for TLA$^+$ formulas. Since TLA$^+$ is based on (untyped) Zermelo-Fraenkel set theory, we believe that this approach is more widely applicable for theorem proving in set-theoretic languages. A type system with refinement types is very expressive and actually quite close to set theory itself, giving rise to proof obligations that are undecidable. Specifically, equality between two refinement types $\{x : \tau \mid \phi_1\}$ and $\{x : \tau \mid \phi_2\}$ reduces to prove $\phi_1 \Leftrightarrow \phi_2$, and subtyping between those types reduces to prove $\phi_1 \Rightarrow \phi_2$, in a context where x has type τ.

The type synthesis process is illustrated in Figure 5.1. As a new component of the translation of the previous chapter, it acts as an interface between the Boolification and preprocessing steps. The type synthesis procedure maps Boolified TLA$^+$ expressions to typed TLA$^+$ expressions. As is standard in type inference for programming languages, we divide our algorithm into a constraint generation phase followed by a constraint solving phase. Given a Boolified TLA$^+$ formula ϕ^b, the type synthesis algorithm first generates a constraint $\mathcal{C}$ that replicates the type derivation of the formula. Constraint generation rules are derived directly from typing rules defined for each type system. Constraints for $\mathcal{T}_1$ are formed by type equality conditions thus solving them reduces basically to first-order unification of elementary types, which is decidable. Constraints for $\mathcal{T}_2$ are formed by equality and subtyping conditions. Our constraint solving algorithm with subtyping discharges the typing proof obligations to SMT solvers, which may succeed or not. In the case constraint solving fails, we fall back to the untyped encoding (restricted to the corresponding part of the proof obligation), which is comparable to dynamic type checking.

A TLA$^+$ fragment Before proceeding to the next section, and in order to simplify the exposition and analysis of the type systems, we define a fragment of the TLA$^+$ language defined in Section 2.1, which will be used in the rest of this chapter. That being said, the actual implementation of the type systems handles the TLA$^+$ language completely except for CHOOSE expressions.

The fragment considered here contains the most relevant expressions of set theory, functions and arithmetic. Without loss of generality, we restrict it to unary operators and set enumeration with two elements. In order to adhere to a presentation closer to standard set theory, we also assume that the formulas are already Boolified, according to Section 4.2. The fragment also includes the three primitive function expressions and basic arithmetic expressions.

We describe the fragment by the following grammar, which is a reorganization of the elements of the original grammar given in Section 2.1. As a first step towards classifying the elements of the language, and in contrast to standard TLA$^+$, we distinguish different syntactic categories of expressions: terms t, sets s, expressions e, formulas ϕ, functions f, and numbers n.

$$
\begin{array}{ll}
\text{(terms)} & t ::= v \mid w(e) \mid f[e] \\
\text{(sets)} & s ::= t \mid \{\} \mid \{e, e\} \mid \textsc{subset}\ s \mid \textsc{union}\ s \mid \{v \in s : \phi\} \\
 & \qquad \mid \textsc{domain}\ f \mid \mathit{Int} \\
\text{(functions)} & f ::= t \mid [v \in s \mapsto e] \\
\text{(numbers)} & n ::= t \mid 0 \mid 1 \mid 2 \mid \cdots \mid n + n \\
\text{(expressions)} & e ::= s \mid f \mid n \\
\text{(formulas)} & \phi ::= v^b \mid w^b(e) \mid \textsc{false} \mid \phi \Rightarrow \phi \mid \forall v\colon \phi \mid e = e \mid e \in s \mid n < n
\end{array}
$$

A typed version of the TLA$^+$ language, which we call *typed*-TLA$^+$ or TLA$^{+\tau}$, is basically obtained from the above TLA$^+$ fragment by decorating with types the variable

binders in quantifiers. Its grammar looks as follows:

$$\phi ::= \ldots \mid \forall v^\tau : \phi$$

Typed-TLA$^+$ is parameterized by a language of types τ, which can be mapped to the sorts of many-sorted first-order logic. The relation *has-type* between a TLA$^+$ expression e and a type τ is written $e : \tau$. As the set theory of TLA$^+$ is defined on top of a first-order logic, the set theory of typed-TLA$^+$ is defined on top of many-sorted first-order logic. Accordingly, the typed-TLA$^+$ expression $\forall x^\tau : \phi$ is true if and only if ϕ is true when the variable x takes all possible values in $\mathcal{D}_\tau$, i.e. the domain of interpretation for the type τ. As a consequence, typed-TLA$^+$ in any type language with more than one type is strictly weaker than TLA$^+$ because it enforces a discipline.

Given a TLA$^+$ proof obligation ϕ, our main goal in this chapter is to obtain an equisatisfiable typed-TLA$^+$ formula ϕ^τ through the type synthesis procedure. Then, the formula ϕ^τ can be encoded into the solver's input language, using the type annotations to improve the encoding. However, our type systems are more expressive than the sorts in our target languages. At the final instance of the translation of a TLA$^+$ proof obligation, the quantified formulas with type annotations are mapped to many-sorted logic, or even to unsorted first-order logic.

Chapter overview In Section 5.2, we give the formal definition of the basic type system $\mathcal{T}_1$, including the key concept of typing hypothesis and typing propositions. The system is described through a set of type inference rules containing constraints on types, which is the style that is standard in the type theoretical research community. Section 5.3 presents the system $\mathcal{T}_2$, an extension of $\mathcal{T}_1$ with dependent and refinement types. Section 5.4 proves that those systems are sound. Section 5.5 gives details of the type synthesis algorithm. Section 5.6 extends the type systems $\mathcal{T}_1$ and $\mathcal{T}_2$ with support for TLA$^+$ tuples and records. Section 5.7 gives an account of related work.

5.2. Elementary types

The types of the system $\mathcal{T}_1$ are composed of elementary types satisfying the requirements of many-sorted first-order logic and basic set theory. Assume an enumerable infinite, and disjoint collection $\mathcal{S}$ of nullary type constructors, representing the atomic elements of the set-theoretic fragment of the language. We define the types τ of the system $\mathcal{T}_1$ by the following grammar:

$$\tau ::= \mathsf{t}_1 \mid \mathsf{t}_2 \mid \ldots \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Set}\,\tau \mid \tau \to \tau \mid \alpha$$

An *atomic type* is one of the symbols $\mathsf{t}_1, \mathsf{t}_2, \ldots$ in $\mathcal{S}$. The types Bool for formulas and Int for integers are distinguished atomic types not included in $\mathcal{S}$. The type constructor Set determines the level of set strata, for instance, for the operators SUBSET and UNION. A type of the form $\tau_1 \to \tau_2$ corresponds to a unary function. Type variables α, representing unknown types, are interpreted over the resulting Herbrand universe induced by the preceding type constructors, that is, the set of all ground, i.e. variable-free, types. We usually note atomic types by the letter β, and ground types by the letter γ. A *ground assignment* σ is a mapping, maybe partial, of type variables to ground types, where $\square$ is the empty assignment.

$$\sigma ::= \square \mid \alpha \mapsto \gamma, \sigma.$$

5.2.1. Typing propositions and typing hypotheses

The *typing proposition* of a type assignment $e : \tau$, noted $[\![e : \tau]\!]$, is a characteristic predicate associated to the type τ that is true for precisely those expressions e that the type τ intends to represent. In other words, the TLA$^+$ formula $[\![e : \tau]\!]$ states that the expression e is one of the elements characterized by the type τ. For instance, having an integer type is characterized by being a member of the set of integers, and the proposition associated to types of the form Set τ is derived from the axiom of power set (2.2). Typing propositions allow us to syntactically translate a type assignment $x : \tau$, where τ is ground, to a TLA$^+$ formula $[\![x : \tau]\!]$.

Definition 4 (Typing propositions)**.** *Given a* TLA$^+$ *expression* e *and a type* τ*, the* TLA$^+$ *formula* $[\![e : \tau]\!]$ *is defined as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
[\![e : \mathsf{t}]\!] & \triangleq & t(e) & [\![e : \mathsf{Bool}]\!] & \triangleq & e \in \textsc{boolean} \\
[\![e : \mathsf{Set}\,\tau]\!] & \triangleq & \forall x \in e : [\![x : \tau]\!] & [\![e : \mathsf{Int}]\!] & \triangleq & e \in \mathit{Int}
\end{array}$$

$$\begin{array}{rcl}
[\![e : \tau_1 \to \tau_2]\!] & \triangleq & \land\ e = [x \in \textsc{domain}\ e \mapsto e[x]] \\
& & \land\ \forall x : x \in \textsc{domain}\ e \Leftrightarrow [\![x : \tau_1]\!] \\
& & \land\ \forall x : x \in \textsc{domain}\ e \Rightarrow [\![e[x] : \tau_2]\!]
\end{array}$$

For each atomic type $\mathsf{t} \in \mathcal{S}$*, we introduce a new unary symbol* t *to the set* $\mathcal{P}$ *of predicate symbols. To ensure that types are pairwise disjoint, we introduce the set of axioms*

$$\forall x, y : [\![x : \beta_1]\!] \land [\![y : \beta_2]\!] \Rightarrow x \neq y,$$

for each pair of atomic types $\beta_1, \beta_2 \in \mathcal{S} \cup \{\mathsf{Int}, \mathsf{Bool}\}$.

We say that two ground types γ_1 and γ_2 are equal, noted $\gamma_1 \cong \gamma_2$, if and only if they characterize the same elements. We will come back to this definition later. For the moment, we formally define it, and assert the simple fact that the typing propositions for any two different ground types γ_1 and γ_2 are disjoint.

Definition 5 (Equality of ground types)**.** $\gamma_1 \cong \gamma_2$ *iff* $\forall x : [\![x : \gamma_1]\!] \Leftrightarrow [\![x : \gamma_2]\!]$ *is valid.*

Lemma 10. $\exists x : [\![x : \gamma_1]\!] \wedge [\![x : \gamma_2]\!]$ *implies* $\gamma_1 \cong \gamma_2$.

Proof. By induction on types and the fact that atomic types are disjoint. □

The traditional method to encode a multi-sorted language into a single-sorted one is by relativizing quantifiers (Section 3.1.3). Relativization replaces a sort annotation x^σ of a quantified MS-FOL formula, where σ is an atomic sort, by a fresh, distinguished predicate $P_\sigma(x)$ as a hypothesis. Here, we update the definition of relativization to our type language by using typing propositions instead of simple unary predicate symbols.

Definition 6 (Relativization)**.** *Given a typed-*TLA$^+$ *expression* e*, the relativized expression* $\mathcal{R}(e)$ *is obtained by recursively replacing each type annotation* x^τ *by a new hypothesis* $[\![x : \tau]\!]$*. The relevant transformation rule is* $\mathcal{R}(\forall x^\tau : \phi) \triangleq \forall x : [\![x : \tau]\!] \Rightarrow \mathcal{R}(\phi)$.

Suppose we want to annotate with types the following TLA$^+$ proof obligation:

$$\phi \triangleq \forall x, y : \textsc{union}\ \{x, y\} = \textsc{union}\ \{y, x\}.$$

First, the UNION operator requires its argument to be a set of sets. The stratification of sets, using the type constructor Set, supports the primary idea that a set must have a different type from its elements. Therefore, x and y should have a Set type. Secondly, the typed version of ϕ will be strictly weaker than ϕ, but it is enough to prove that weaker formula if we know from the context the types of x and y, and that they are compatible. Otherwise, if the elements of x and y reside in different universes, their values are necessarily different.

As a consequence, our type disciplines will restrict the sets of the form $\{x, y\}$, and similar expressions that relate more than one sub-expression such as $x = y$ or $x \cup y$, to have elements of the same type. For some atomic type $\mathsf{t} \in \mathcal{S}$, a valid type annotation for ϕ is

$$\phi^\tau \triangleq \forall x^{\mathsf{Set\,t}}, y^{\mathsf{Set\,t}} : \textsc{union}\ \{x, y\} = \textsc{union}\ \{y, x\}$$

The relativization of this decorated formula is

$$\mathcal{R}(\phi^\tau) = \forall x, y : (\forall z \in x : t(z)) \wedge (\forall z \in y : t(z)) \Rightarrow \textsc{union}\ \{x, y\} = \textsc{union}\ \{y, x\}$$

which just expresses in TLA$^+$ the semantics of ϕ^τ. In the original and relativized formulas ϕ and $\mathcal{R}(\phi^\tau)$, the variables are evaluated in the unique domain $\mathcal{D}$. In the annotated formula ϕ^τ, the values of the variables x and y are evaluated in $\mathcal{D}_{\mathsf{Set\,t}}$, which is a fragment of the universe $\mathcal{D}$. The extra hypothesis $\forall z \in x : t(z)$ in the relativized formula just expresses that the elements of x satisfy the predicate t. The same can be said about y. Since all values in the domain $\mathcal{D}$ denote sets, we give a name to this kind of types.

Definition 7 (Safe types)**.** *A type* τ *is* safe *iff* τ *is either an atomic type* β *in* $\mathcal{S}$ *except* Bool *and* Int*, or if it is of the form* Set τ'*, where* τ' *is safe.*

Since typing predicates t_i are uninterpreted and independent from the rest of the formula, safe types cannot introduce any unsoundness to an untyped formula, in the sense of the following lemma.

Lemma 11. *For any safe type* τ, $\vdash \forall x : \phi$ *if and only if* $\vdash \mathcal{R}(\forall x^{\tau} : \phi)$.

Proof. ($\Leftarrow$) This is essentially the relativization lemma for MS-FOL (Lemma 1), with the addition of the Set type. ($\Rightarrow$) The variable x is interpreted in all the values of the domain $\mathcal{D}$, but it is restricted by the formula $[\![x : \tau_s]\!]$. In the typed formula, x is evaluated in the sub-domain $\mathcal{D}_{\tau_s}$, which corresponds to the same values represented by $[\![x : \tau_s]\!]$. □

As in standard relativization that ensures soundness in the translation from MS-FOL to FOL, the following lemma relates validity in typed-TLA$^+$ and TLA$^+$, restricted to ground types.

Lemma 12 (Relativization is sound)**.** $\vdash \forall x^{\gamma} : \phi$ *implies* $\vdash \mathcal{R}(\forall x^{\gamma} : \phi)$.

Proof. The proof follows the proof for many-sorted logic [End01, Man05] by induction on ϕ, extended for the Set constructor, by Lemma 11, and the definition of TLA$^+$ functions, through the axiom (2.10). □

In this chapter, our objective is to go in the opposite direction, that is, from an unsorted universe to a many-sorted universe. Ideally, we should obtain the necessary type information from typing propositions occurring as hypotheses. Typing propositions may appear in a proof obligation in many different, though equivalent, forms. For example, the typing proposition $[\![S : \mathsf{Set\ Int}]\!]$ is equal to $\forall z \in S : z \in \mathit{Int}$, but it may appear, for instance, as the equivalent formula $S \in \textsc{subset}\ \mathit{Int}$. Then, it is not always possible to easily identify them. We have to acquiesce to procure the type information from propositions that appear in the unsorted language in the form of *typing hypotheses*.

Definition 8 (Typing hypothesis)**.** *A* typing hypothesis $\mathcal{H}(x)$ *for a variable* x *is a premise of the form* $x \in e$ *or* $x = e$*, for any expression* e *where* x *is not free in* e.

A typing hypothesis for a variable x is an *upper bound* to the values of x. Hence, it characterizes those values of x which can be embodied by a type. Type-correctness invariants (cf. Section 2.2.2) are thus natural candidates for typing hypotheses. The types that can be inferred from an untyped formula are almost directly taken from their typing hypotheses.

Suppose we want to annotate the invalid formula $\forall x : x + 0 = x$. It is incorrect to say that x is an integer: that would make the formula $\mathcal{R}(\forall x^{\mathsf{Int}} : x + 0 = x)$ valid. However, the formula $\forall x : x \in \mathit{Int} \Rightarrow x + 0 = x$ contains a (typing) hypothesis from which we can soundly infer the type Int for x to obtain the annotated formula

$$\forall x^{\mathsf{Int}} : x \in \mathit{Int} \Rightarrow x + 0 = x.$$

Note that the information that x is an integer is redundant. As proved later by the soundness theorem for type synthesis, the relativization of this formula and the original one are equisatisfiable if the types are correctly constructed. With this in mind, we define the type systems for TLA$^+$.

5.2.2. Type system

A *type system* is defined by a collection of inference rules. Given a TLA$^+$ expression e and an expected type τ for that expression, the problem consists in generating a valid tree derivation of inference rules in order to decide if τ is a type of e. First, we declare some conventional auxiliary definitions [Car97, PR05].

A *typing context* is defined by the grammar $\Gamma ::= \emptyset \mid \Gamma, x : \tau$, where $x \in \mathcal{V} \cup \mathcal{O}$, that is, x is a variable symbol or an operator symbol. Thus, a typing context can be seen as a partial function $\Gamma : \mathcal{V} \cup \mathcal{O} \to \tau$. Moreover, in a context $\Gamma, x : \tau$, the variable x is not allowed to belong to the domain of Γ. The domain dom of a context is defined by

$$dom(\emptyset) \triangleq \{\} \qquad \text{and} \qquad dom(\Gamma, x : \tau) \triangleq \{x\} \cup dom(\Gamma).$$

A type assignment σ applied to a typing context is recursively applied to every type, by the rules: $\sigma\,\emptyset = \emptyset$ and $\sigma\,(\Gamma, x : \tau) = \sigma\,\Gamma, x : \sigma\tau$.

A triple $\Gamma \vdash e : \tau$ is called a *judgement*, and asserts that the TLA$^+$ expression e has type τ in the typing context Γ. A *type inference rule* is of the form

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma_n \vdash e_n : \tau_n \qquad \mathcal{C}_1 \quad \cdots \quad \mathcal{C}_m}{\Gamma \vdash e : \tau}\ \mathrm{R}$$

That is, the rule named R has a judgement as a conclusion, and its premises consists in a finite collection of judgements, where e_i are typically the sub-expressions of e, and a finite collection of *constraints* $\mathcal{C}_i$ on types. If the premises are valid, the conclusion must hold. A judgement $\Gamma \vdash e : \tau$ is valid by the rule R if and only if $\Gamma_i \vdash e_i : \tau_i$ and the constraints $\mathcal{C}_i$ are also valid. Therefore, in order to prove that a judgement is valid, one exhibits a tree-like *derivation* of type inference rules.

A pair $\langle \Gamma, \tau \rangle$ is a *typing* of an expression e iff $FV(e) \subseteq dom(\Gamma)$ and the pre-judgement $\Gamma \vdash e : \tau$ is valid. Likewise, the typing of a formula is just Γ, assuming its type is Bool. A formula ϕ is *typable* iff it admits a typing. Given an untyped formula $\phi \triangleq \forall x : \psi$ such that $\Gamma \vdash \psi :$ Bool is valid and $FV(\psi) \subseteq dom(\Gamma)$, then the corresponding *annotated* (i.e. sorted) formula is $\phi^\tau \triangleq \forall x^{\Gamma(x)} : \psi$.

The definition of the type inference rules for the systems $\mathcal{T}_1$ and $\mathcal{T}_2$ is inspired from standard type systems usually studied by the type theoretical research community, such as those found in functional programming languages derived from the simple-typed λ-calculus [Mil78]. During a type derivation, the type inference rules for those

systems introduce constraints with many fresh type variables, which are unified throughout to obtain a most general type. Here, we combine in a single system of inference rules the type synthesis process, which consists in unifying the type variables introduced in the typing hypotheses, and the type checking process, which only checks that the type constraints introduced in the rest of the formula hold. After a valid type inference tree is derived, the type variables must be instantiated with ground types, subject to the generated constraints.

Equivalence constraints The core of the type inference system $\mathcal{T}_1$ lies in two equivalence constraints on types: unifying $\equiv$ and non-unifying $\cong$. An *atomic type constraint* $\mathcal{C}_A$ is defined by the grammar:

$$\mathcal{C}_A ::= \tau \equiv \tau \mid \tau \cong \tau.$$

Both propositions are equivalence relations on types, with the difference that the first one allows type variables to be unified, while the latter is the equality between ground types defined before. For example, the unification of the constraint $\mathsf{Set}\,\alpha_1 \to \mathsf{Int} \equiv \mathsf{Set}\,\mathsf{Bool} \to \alpha_2$ yields the ground assignment $\sigma \triangleq \alpha_1 \mapsto \mathsf{Bool}, \alpha_2 \mapsto \mathsf{Int}$. Then, $\sigma(\mathsf{Set}\,\alpha_1 \to \sigma\mathsf{Int}) \cong \sigma(\mathsf{Set}\,\mathsf{Bool} \to \alpha_2)$ is valid.

A constraint $\tau_1 \equiv \tau_2$ is *satisfied* by a type assignment σ, noted $\sigma \models \tau_1 \equiv \tau_2$, if and only if there exists a type assignment σ that makes $\sigma\tau_1 \cong \sigma\tau_2$ valid. A constraint $\gamma_1 \cong \gamma_2$ is *valid*, where γ_1 and γ_2 are ground types, if and only if it can be proved that the TLA$^+$ formula $\forall x : [\![x : \tau_1]\!] \Leftrightarrow [\![x : \tau_2]\!]$ is valid. We can write these definitions as the semantic rules:

$$\frac{\sigma\tau_1 \cong \sigma\tau_2}{\sigma \models \tau_1 \equiv \tau_2}\,\textsc{Eq} \qquad \frac{\vdash \forall x : [\![x : \gamma_1]\!] \Leftrightarrow [\![x : \gamma_2]\!]}{\gamma_1 \cong \gamma_2}\,\textsc{Eq'}$$

In this way, type equality in Eq' reduces to prove this formula valid, for instance, by an external SMT solver through our untyped encoding. However, generating a verification condition each time a type equality needs to be checked is too expensive. From the rules Eq and Eq', and the fact that types are disjoint (Lemma 10), it is trivial to derive the following equivalent, purely syntactic rules, which we use in $\mathcal{T}_1$ instead of Eq and Eq':

$$\frac{\beta \in \mathcal{S}}{\sigma \models \beta \equiv \beta}\,\mathcal{C}\text{-}\equiv\text{-}\beta \qquad \frac{\sigma \models \tau_1 \equiv \tau_2}{\sigma \models \mathsf{Set}\,\tau_1 \equiv \mathsf{Set}\,\tau_2}\,\mathcal{C}\text{-}\equiv\text{-}\textsc{Set} \qquad \frac{\sigma \models \tau_1 \equiv \tau_1' \quad \sigma \models \tau_2 \equiv \tau_2'}{\sigma \models \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}\,\mathcal{C}\text{-}\equiv\text{-}\textsc{Arrow}$$

$$\frac{\beta \in \mathcal{S}}{\beta \cong \beta}\,\mathcal{C}\text{-}\cong\text{-}\beta \qquad \frac{\tau_1 \cong \tau_2}{\mathsf{Set}\,\tau_1 \cong \mathsf{Set}\,\tau_2}\,\mathcal{C}\text{-}\cong\text{-}\textsc{Set} \qquad \frac{\tau_1 \cong \tau_1' \qquad \tau_2 \cong \tau_2'}{\tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'}\,\mathcal{C}\text{-}\cong\text{-}\textsc{Arrow}$$

Type inference rules The typing rules for Boolean TLA$^+$ expressions are given in Figure 5.2. The types for variables are taken directly from the typing context Γ

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \text{T-Var} \qquad \frac{\Gamma(p) \equiv \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1' \qquad \tau_1' \cong \tau_1}{\Gamma \vdash p(e) : \tau_2}\ \text{T-Op}$$

$$\frac{}{\Gamma \vdash x^b : \mathsf{Bool}}\ \text{T-Var}^b \qquad \frac{\Gamma(p) \equiv \tau \to \mathsf{Bool} \qquad \Gamma \vdash e : \tau' \qquad \tau' \cong \tau}{\Gamma \vdash p(e)^b : \mathsf{Bool}}\ \text{T-Op}^b$$

$$\frac{}{\Gamma \vdash \textsc{false} : \mathsf{Bool}}\ \text{T-False} \qquad \frac{\Gamma \vdash \phi_1 : \mathsf{Bool} \qquad \Gamma \vdash \phi_2 : \mathsf{Bool}}{\Gamma \vdash \phi_1 \Rightarrow \phi_2 : \mathsf{Bool}}\ \text{T-Implies}$$

$$\frac{\Gamma, x : \tau \vdash \phi : \mathsf{Bool}}{\Gamma \vdash \forall x : \phi : \mathsf{Bool}}\ \text{T-Quant} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash \phi : \mathsf{Bool} \quad {\scriptstyle x \notin FV(e)}}{\Gamma \vdash \forall x : x = e \Rightarrow \phi : \mathsf{Bool}}\ \text{Th}_1\text{-Eq}$$

$$\frac{\Gamma \vdash e : \mathsf{Set}\,\tau \quad \Gamma, x : \tau \vdash \phi : \mathsf{Bool} \quad {\scriptstyle x \notin FV(e)}}{\Gamma \vdash \forall x : x \in e \Rightarrow \phi : \mathsf{Bool}}\ \text{Th}_1\text{-Mem}$$

Figure 5.2.: $\mathcal{T}_1$-Typing rules for Boolean expressions

(rule T-Var). The type for operators are also obtained from the typing context by unification (rule T-Op), that is, the operator symbol p is always expected to have a functional type (consistent with its arity). As expected, once a formula has been Boolified, the rules for FALSE and $\Rightarrow$ are trivial. Rule T-Quant evaluates the body of $\forall x : \phi$ in the typing context Γ extended with the assignment $x : \tau$, for some type τ, with the implicit assumption $x \notin dom(\Gamma)$.

We obtain the typing hypotheses by decomposing the assumptions found in a formula by elementary heuristics. The rules Th$_1$-Eq and Th$_1$-Mem in Figure 5.2, which are applied with higher priority than rule T-Quant, encapsulate this requirement in a simplified presentation. In our implementation, proof obligations are preprocessed to obtain typing hypotheses occurring in disguised shapes. However, typing hypotheses may not be completely captured by merely syntactic analysis. In our type systems, the typing rules are syntax-directed, including the rules for plain quantifiers and quantifiers with typing hypotheses. This means that, for any expression ϕ, at most one typing rule may be applied. Therefore, the derivation tree is unique and fully determined by the shape of the expression, in a given typing context. This can be easily proved by induction on the structure of ϕ.

If we just want to type-check a typed-TLA$^+$ formula, we use the same system of rules, except that the typing rules Th$_1$-Mem, Th$_1$-Eq and T-Quant for quantifiers would be no longer needed; they would be replaced by the following rule:

$$\frac{\Gamma, x : \tau \vdash \phi : \mathsf{Bool}}{\Gamma \vdash \forall x^\tau : \phi : \mathsf{Bool}}\ \text{T-Check}$$

$$\frac{}{\Gamma \vdash \{\} : \mathsf{Set}\,\alpha}\ \mathrm{T_1\text{-}Empty} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \cong \tau_2}{\Gamma \vdash \{e_1, e_2\} : \mathsf{Set}\,\tau_1}\ \mathrm{T_1\text{-}Pair}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\,\tau \qquad \Gamma, x : \tau \vdash \phi : \mathsf{Bool}}{\Gamma \vdash \{x \in S : \phi\} : \mathsf{Set}\,\tau}\ \mathrm{T_1\text{-}SetComp}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\,\tau}{\Gamma \vdash \textsc{subset}\ S : \mathsf{Set}\,\mathsf{Set}\,\tau}\ \mathrm{T_1\text{-}Power} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \cong \tau_2}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ \mathrm{T_1\text{-}Eq}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\,\mathsf{Set}\,\tau}{\Gamma \vdash \textsc{union}\ S : \mathsf{Set}\,\tau}\ \mathrm{T_1\text{-}Union} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \mathsf{Set}\,\tau_2 \qquad \tau_1 \cong \tau_2}{\Gamma \vdash e_1 \in e_2 : \mathsf{Bool}}\ \mathrm{T_1\text{-}Mem}$$

$$\frac{\Gamma \vdash f : \tau_1 \qquad \Gamma \vdash e : \tau_2 \qquad \mathsf{dom}(\tau_1) \cong \tau_2}{\Gamma \vdash f[e] : \mathsf{cod}(\tau_1)}\ \mathrm{T_1\text{-}App}$$

$$\frac{\Gamma \vdash f : \tau}{\Gamma \vdash \textsc{domain}\ f : \mathsf{Set}\,(\mathsf{dom}(\tau))}\ \mathrm{T_1\text{-}Dom} \qquad \frac{\Gamma \vdash S : \mathsf{Set}\,\tau_1 \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash [x \in S \mapsto e] : \tau_1 \to \tau_2}\ \mathrm{T_1\text{-}Fun}$$

$$\frac{}{\Gamma \vdash \mathit{Int} : \mathsf{Set}\,\mathsf{Int}}\ \mathrm{T_1\text{-}Int} \qquad \frac{\Gamma \vdash e_i : \tau_i \qquad \tau_i \cong \mathsf{Int} \qquad {\scriptstyle i \in \{1,2\}}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}\ \mathrm{T_1\text{-}Plus}$$

$$\frac{{\scriptstyle n \in \{0,1,2,\ldots\}}}{\Gamma \vdash n : \mathsf{Int}}\ \mathrm{T_1\text{-}Num} \qquad \frac{\Gamma \vdash e_i : \tau_i \qquad \tau_i \cong \mathsf{Int} \qquad {\scriptstyle i \in \{1,2\}}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}\ \mathrm{T_1\text{-}Less}$$

Figure 5.3.: $\mathcal{T}_1$-typing rules for set, function and arithmetic expressions

This means that, during type-checking, there are no special derivations from typing hypotheses, and type annotations in quantifiers are passed directly to the body's context. All the type inference rules for Boolean expressions are the same for both type systems, except for some minor differences (namely, equality checking is replaced by subtype checking).

The remaining inference rules of the type system $\mathcal{T}_1$ are shown in Figure 5.3. The type of the empty set is compatible with any Set type (rule T$_1$-Empty). The subexpressions $x \in S$ in the rules T$_1$-SetComp and T$_1$-Fun are typing hypotheses and are therefore treated as such. This means that the types of the bound variable and its domain are compatible, therefore only one type variable is used for both. In contrast, the types of the arguments of equality (rule T$_1$-Eq) and pairs (T$_1$-Pair) must not be unified, so we just check that they are equal.

When evaluating an expression f that is supposed to be a function, as in the case of

rules T_1-App and T_1-Dom, we cannot evaluate the type of f by imposing a functional type on it. We expect to derive the desired functional type from the context. In order to extract the domain and codomain from a functional type, as needed by these two rules, we use two new special type operators:

$$\tau ::= \ldots \mid \mathsf{dom}(\tau) \mid \mathsf{cod}(\tau)$$

These type modifiers are defined by the properties

$$\mathsf{dom}(\tau_1 \to \tau_2) = \tau_1 \qquad \text{and} \qquad \mathsf{cod}(\tau_1 \to \tau_2) = \tau_2.$$

That is, the types $\mathsf{dom}(\tau)$ and $\mathsf{cod}(\tau)$ represent the domain and codomain of some type τ, when τ is a functional type. By applying their properties as rewriting rules, the types dom and cod can be eliminated when they are applied to the expected functional type.

Literal integers (T_1-Num) and the set of integers (T_1-Int) have a constant type. The rules T_1-Plus and T_1-Less require their arguments to be integers through the condition $\tau_i \cong \mathsf{Int}$.

Derivation example We show a type derivation for the universal closure of the above toy example of formula 5.1. The construction of the derivation starts by applying the rule Th_1-Eq with an empty context, because this rule has more priority than the more general rule for quantifiers T-Quant. The typing rules are systematically applied with fresh type variables at each step. We proceed to evaluate the premises in a depth-first order from left to right (this is irrelevant for the final result, but it shows the order in which the numbering for the type variable identifiers are generated). Because of lack of space, the derivation is divided into sub-steps (1), (2), etc.

$$\dfrac{\dfrac{\overset{(1)}{}}{\emptyset \vdash [x \in \{1,2,3\} \mapsto x * x] : \tau_1}\,\mathrm{T}_1\text{-Fun} \quad \dfrac{\overset{(2)}{}}{f : \tau_1 \vdash f[0] < f[0] + 1 : \mathsf{Bool}}\,\mathrm{T}_1\text{-Less}}{\emptyset \vdash \forall f : f = [x \in \{1,2,3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1 : \mathsf{Bool}}\,\mathrm{Th}_1\text{-Eq}$$

Type derivation (1):

$$\dfrac{\dfrac{\overset{(3)}{}}{\emptyset \vdash \{1,2,3\} : \mathsf{Set}\,\tau_2}\,\mathrm{T}_1\text{-Enum} \quad \dfrac{\dfrac{(x : \tau_2)(x) \equiv \tau_7}{x : \tau_2 \vdash x : \tau_7}\,\mathrm{T}_1\text{-Var} \quad \tau_7 \cong \mathsf{Int} \quad (\times 2)}{x : \tau_2 \vdash x * x : \tau_3 \equiv \mathsf{Int}}\,\text{T-Mult}}{\emptyset \vdash [x \in \{1,2,3\} \mapsto x * x] : \tau_1 \equiv \tau_2 \to \tau_3}\,\mathrm{T}_1\text{-Fun}$$

Type derivation (3):

$$\dfrac{\dfrac{\mathsf{Int} \equiv \tau_4}{\varnothing \vdash 1 : \tau_4} \quad \dfrac{\mathsf{Int} \equiv \tau_5}{\varnothing \vdash 2 : \tau_5} \quad \dfrac{\mathsf{Int} \equiv \tau_6}{\varnothing \vdash 3 : \tau_6} \quad \tau_4 \cong \tau_5 \cong \tau_6}{\varnothing \vdash \{1,2,3\} : \mathsf{Set}\,\tau_2 \equiv \mathsf{Set}\,\tau_4}\ \mathrm{T}_1\text{-Enum}$$

In the derivations (1) and (3), the returning types τ_1 and $\mathsf{Set}\,\tau_2$ are the types propagated from the previous derivations in the lower rules of the derivation tree. We have equated them to the returning types that they should be unified with, as specified by the rules T_1-Fun and T_1-Enum, respectively. The constraints $\tau_1 \equiv \tau_2 \to \tau_3$ and $\mathsf{Set}\,\tau_2 \equiv \mathsf{Set}\,\tau_4$ are will appear explicitly in the constraint generation rules of Section 5.5.1.

Type derivation (2):

$$\dfrac{\dfrac{(4)}{f : \tau_1 \vdash f[0] : \tau_8}\ \mathrm{T}_1\text{-App} \quad \dfrac{\dfrac{(4')}{f : \tau_1 \vdash f[0] : \tau_{10}} \quad \dfrac{\mathsf{Int} \equiv \tau_{11}}{f : \tau_1 \vdash 1 : \tau_{11}}\ \mathrm{T}_1\text{-Num}}{f : \tau_1 \vdash f[0] + 1 : \tau_9}\ \mathrm{T}_1\text{-Plus}\ \begin{array}{l}\tau_{10} \cong \mathsf{Int} \\ \tau_{11} \cong \mathsf{Int}\end{array} \quad \begin{array}{l}\tau_8 \cong \mathsf{Int} \\ \tau_9 \cong \mathsf{Int}\end{array}}{f : \tau_1 \vdash f[0] < f[0] + 1 : \mathsf{Bool}}\ \mathrm{T}_1\text{-Less}$$

Type derivation (4):

$$\dfrac{\dfrac{(f : \tau_1)(f) \equiv \tau_{12}}{f : \tau_1 \vdash f : \tau_{12}}\ \text{T-Var} \quad \dfrac{\mathsf{Int} \equiv \tau_{13}}{f : \tau_1 \vdash 0 : \tau_{13}}\ \text{T-Num} \quad \tau_{13} \cong \mathsf{dom}(\tau_{12})}{f : \tau_1 \vdash f[0] : \tau_8 \equiv \mathsf{cod}(\tau_{12})}\ \mathrm{T}_1\text{-App}$$

The derivation (4′) is the same as (4) except that τ_8 replaces τ_3, and that it generates fresh variables with different names that we do not show here.

When the derivation is finished, we gather the list of generated constraints:

$$\begin{array}{llll}
\mathsf{Int} \equiv \tau_4 & \mathsf{Int} \equiv \tau_5 & \mathsf{Int} \equiv \tau_6 & \tau_4 \cong \tau_5 \cong \tau_6 \\
\mathsf{Set}\,\tau_2 \equiv \mathsf{Set}\,\tau_4 & \tau_2 \equiv \tau_7 & \tau_7 \equiv \mathsf{Int} & \tau_3 \equiv \mathsf{Int} \\
\tau_1 \equiv \tau_2 \to \tau_3 & \tau_1 \equiv \tau_{12} & \mathsf{Int} \equiv \tau_{13} & \tau_{13} \cong \mathsf{dom}(\tau_{12}) \\
\tau_8 \equiv \mathsf{cod}(\tau_{12}) & \mathsf{Int} \equiv \tau_{11} & \tau_{10} \cong \mathsf{Int} & \tau_{11} \cong \mathsf{Int} \\
\tau_8 \cong \mathsf{Int} & \tau_9 \cong \mathsf{Int} & &
\end{array}$$

The constraints are context-independent because each type variable name was generated with a fresh identifier. After resolving the unifying equalities, we obtain the

following assignment σ:

$$\tau_1, \tau_{12} \mapsto \mathsf{Int} \to \mathsf{Int},$$
$$\tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}, \tau_{11}, \tau_{13} \mapsto \mathsf{Int}$$

The non-unifying constraints that remain to check are $\sigma\tau_i \cong \sigma\mathsf{Int}$, where $i \in 7..10$, $\sigma\tau_{13} \cong \sigma\mathsf{dom}(\tau_{12})$, and $\sigma\tau_8 \cong \sigma\mathsf{cod}(\tau_{12})$. They are trivially valid using the equivalences $\mathsf{dom}(\mathsf{Int} \to \mathsf{Int}) = \mathsf{Int}$ and $\mathsf{cod}(\mathsf{Int} \to \mathsf{Int}) = \mathsf{Int}$. Therefore, the whole derivation is valid. Note that we can link the variables τ_i to the point in the formula's syntactic tree where they were generated. Then, it is possible to know the resulting type of every sub-expression. For instance, the variable f is associated with the type variable τ_1, which has assigned the type $\mathsf{Int} \to \mathsf{Int}$.

5.3. Dependent and refinement types

We extend the type signature of the previous type system $\mathcal{T}_1$ by upgrading functional types to dependent types and by adding refinement types:

$$\tau ::= \ldots \mid (v : \tau) \to \tau \mid \{x : \tau \mid \phi\}$$

A *refinement* type $\{x : \tau \mid \phi\}$ [FP91] represents the set of values of type τ that satisfy the refinement predicate ϕ, where the variable x is free in ϕ:

$$[\![e : \{x : \tau \mid \phi\}]\!] \triangleq [\![e : \tau]\!] \wedge \phi[x \leftarrow e]$$

Refinement types suit perfectly the characterization of set comprehension objects of the form $\{x \in S : \phi\}$.

Property 13 (Refinement linearizability)**.** *A refinement type based on a refinement type is a refinement type with the base of the second one:*

$$\{x : \{y : \tau \mid \phi_1\} \mid \phi_2\} = \{x : \tau \mid \phi_1[y \leftarrow x] \wedge \phi_2\}$$

Proof. By expanding the definition of type equivalence:

$$\begin{aligned}
\forall z : [\![z : \{x : \{y : \tau \mid \phi_1\} \mid \phi_2\}]\!] &\Leftrightarrow [\![z : \{x : \tau \mid \phi_1[y \leftarrow x] \wedge \phi_2\}]\!] && \text{iff} \\
\forall z : [\![z : \{y : \tau \mid \phi_1\}]\!] \wedge \phi_2[x \leftarrow z] &\Leftrightarrow [\![z : \tau]\!] \wedge (\phi_1[y \leftarrow x] \wedge \phi_2)[x \leftarrow z] && \text{iff} \\
\forall z : [\![z : \tau]\!] \wedge \phi_1[y \leftarrow z] \wedge \phi_2[x \leftarrow z] &\Leftrightarrow [\![z : \tau]\!] \wedge \phi_1[y \leftarrow z] \wedge \phi_2[x \leftarrow z] && \square
\end{aligned}$$

Property 13 is applied whenever possible to obtain normal forms of refinement types. A refinement type $\{x : \tau \mid \phi\}$ is in *normal form* iff the base type τ is an atomic type or a Set of an atomic type. In the following, we will refer to their normal forms when we talk about refinement types. However, a refinement type whose base is a

functional or a Set type cannot always be reduced to a refinement in normal form. The reason is that refinement, functional and Set types correspond to different axioms in TLA$^+$ set theory, which are independent of each other. Another property is that any type τ is equivalent to the trivial type $\{x : \tau \mid \text{TRUE}\}$.

A *dependent* type $(x : \tau_1) \to \tau_2$ [AC01] is a functional type where τ_1 represents the domain of the function and the term x may occur in the codomain type τ_2. The variable x of type τ_1 is bound in type τ_2. If x does not occur in τ_2, we can omit it from the syntax to obtain the simpler function type $\tau_1 \to \tau_2$ of the previous type system. The updated typing proposition is:

$$\begin{aligned}[\![e : (x : \tau_1) \to \tau_2]\!] \triangleq\ & \wedge\ e = [x \in \text{DOMAIN } e \mapsto e[x]] \\ & \wedge\ \forall z :\ z \in \text{DOMAIN } e \Leftrightarrow [\![z : \tau_1]\!] \\ & \wedge\ \forall x :\ [\![x : \tau_1]\!] \Rightarrow [\![e[x] : \tau_2]\!]\end{aligned}$$

Note that, in the case τ_2 is or contains a refinement type, the refinement predicate may contain free occurrences of x. In the type $(x : \tau_1) \to \tau_2$, x would be bound by the domain $(x : \tau_1)$, while in the last conjunct of the typing proposition, it would be bound by the quantifier.

Equality constraints

The free variables of a refinement type $\{x : \mathsf{Int} \mid \phi\}$ are the free variables of the predicate ϕ minus the variable x. The free variables may not be necessarily bound by the domain of a functional type. As a result, atomic type constraints in $\mathcal{T}_2$ have to be interpreted in a typing context Γ, which binds both types in the equations:

$$\mathcal{C}_A ::= \Gamma \vdash \tau \equiv \tau \mid \Gamma \vdash \tau \cong \tau.$$

When Γ can be assumed from the context, we just write $\tau_1 \equiv \tau_2$ or $\tau_1 \cong \tau_2$.

The semantic rules for $\equiv$ and $\cong$ are updated accordingly:

$$\frac{\sigma\Gamma \vdash \sigma\tau_1 \cong \sigma\tau_2}{\sigma \models \Gamma \vdash \tau_1 \equiv \tau_2}\ \text{EQ} \qquad \frac{\Gamma \vdash \forall x :\ [\![x : \gamma_1]\!] \Leftrightarrow [\![x : \gamma_2]\!]}{\Gamma \vdash \gamma_1 \cong \gamma_2}\ \text{EQ}'$$

A constraint $\Gamma \vdash \tau_1 \equiv \tau_2$ is *satisfiable*, noted $\sigma \models \Gamma \vdash \tau_1 \equiv \tau_2$, if and only if there exists a ground assignment σ that makes $\sigma\Gamma \vdash \sigma\tau_1 \cong \sigma\tau_2$ valid. Given two ground types γ_1 and γ_2, a constraint $\Gamma \vdash \gamma_1 \cong \gamma_2$ is *valid* if and only if it can be proved that the TLA$^+$ formula $\forall x :\ [\![x : \gamma_1]\!] \Leftrightarrow [\![x : \gamma_2]\!]$ is valid in the typing context Γ.

The operator $\mathcal{F}$ helps us define what does it mean for a TLA$^+$ formula to be valid in a typing context.

Definition 9 (Γ-relativization)**.** *The operator $\mathcal{F}$ is recursively defined by*

$$\mathcal{F}(\varnothing \vdash \phi) \triangleq \phi \qquad \textit{and} \qquad \mathcal{F}(\Gamma, x : \tau \vdash \phi) \triangleq \mathcal{F}(\Gamma \vdash \forall x :\ [\![x : \tau]\!] \Rightarrow \phi).$$

Thus, a formula ϕ is valid in a typing context Γ, written $\Gamma \vdash \phi$, iff the relativization of the context Γ with the formula ϕ, that is, the TLA$^+$ formula $\mathcal{F}(\Gamma \vdash \phi)$, is valid.

As we did before for the system $\mathcal{T}_1$, we partition the rules Eq and Eq' in the following rules:

$$\frac{\beta \in \mathcal{S}}{\sigma \models \Gamma \vdash \beta \equiv \beta}\,\mathcal{C}\text{-}\equiv\text{-}\beta \qquad \frac{\sigma \models \Gamma \vdash \tau_1 \equiv \tau_1' \qquad \sigma \models \Gamma, x : \tau_1 \vdash \tau_2 \equiv \tau_2'}{\sigma \models \Gamma \vdash (x : \tau_1) \to \tau_2 \equiv (x : \tau_1') \to \tau_2'}\,\mathcal{C}\text{-}\equiv\text{-}\textsc{Arrow}$$

$$\frac{\sigma \models \Gamma \vdash \tau_1 \equiv \tau_2}{\sigma \models \Gamma \vdash \mathsf{Set}\,\tau_1 \equiv \mathsf{Set}\,\tau_2}\,\mathcal{C}\text{-}\equiv\text{-}\textsc{Set} \qquad \frac{\sigma\Gamma \vdash \{x : \sigma\tau_1 \mid \phi_1\} \cong \{x : \sigma\tau_2 \mid \phi_2\}}{\sigma \models \Gamma \vdash \{x : \tau_1 \mid \phi_1\} \equiv \{x : \tau_2 \mid \phi_2\}}\,\mathcal{C}\text{-}\equiv\text{-}\textsc{Ref}$$

$$\frac{\beta \in \mathcal{S}}{\Gamma \vdash \beta \cong \beta}\,\mathcal{C}\text{-}\cong\text{-}\beta \qquad \frac{\Gamma \vdash \tau_1 \cong \tau_1' \qquad \Gamma, x : \tau_1 \vdash \tau_2 \cong \tau_2'}{\Gamma \vdash (x : \tau_1) \to \tau_2 \cong (x : \tau_1') \to \tau_2'}\,\mathcal{C}\text{-}\cong\text{-}\textsc{Arrow}$$

$$\frac{\Gamma \vdash \tau_1 \cong \tau_2}{\Gamma \vdash \mathsf{Set}\,\tau_1 \cong \mathsf{Set}\,\tau_2}\,\mathcal{C}\text{-}\cong\text{-}\textsc{Set} \qquad \frac{\Gamma \vdash \tau_1 \cong \tau_2 \qquad \Gamma, x : \tau_1 \vdash \phi_1 \Leftrightarrow \phi_2}{\Gamma \vdash \{x : \tau_1 \mid \phi_1\} \cong \{x : \tau_2 \mid \phi_2\}}\,\mathcal{C}\text{-}\cong\text{-}\textsc{Ref}$$

With these rules, only equality of ground refinement types (rule $\mathcal{C}$-$\cong$-Ref) yields type-correctness conditions that have to be proved for the derivation to be valid. Two refinement types in normal form are equal iff their base type are equal and their predicates are equivalent. Unifying equality of refinement types is lifted to equality checking on ground types. The rest of the equality rules are just updated from $\mathcal{T}_1$ to include typing contexts. Since this is a first-order problem, equality of refinement types is undecidable, and deciding the equality of two types in general becomes undecidable in this type system.

Subtyping constraints

The type inference rules of $\mathcal{T}_2$ rely on the previous type equality relations $\equiv$ and $\cong$ plus two additional subtype relationships: unifying $<:$, and its corresponding non-unifiable version $\prec:$ that just checks that the subtype relationship between two types holds.

$$\mathcal{C}_A ::= \ldots \mid \Gamma \vdash \tau <: \tau \mid \Gamma \vdash \tau \prec: \tau.$$

The subtyping relations are pre-orders on types, that is, they are reflexive and transitive.

A constraint $\Gamma \vdash \tau_1 <: \tau_2$ is *satisfied* by an assignment σ, noted $\sigma \models \Gamma \vdash \tau_1 <: \tau_2$, if and only if there exists a ground assignment σ that makes $\sigma\Gamma \vdash \sigma\tau_1 \prec: \sigma\tau_2$ valid. The general definition of subtyping validity for ground types is similar to equality: for any two ground types γ_1 and γ_2, $\Gamma \vdash \gamma_1 \prec: \gamma_2$ is valid if and only if it can be proved that $\mathcal{F}(\Gamma \vdash \forall x : [\![x : \gamma_1]\!] \Rightarrow [\![x : \gamma_2]\!])$ is valid.

$$\frac{\sigma\Gamma \vdash \sigma\tau_1 \prec: \sigma\tau_2}{\sigma \models \Gamma \vdash \tau_1 <: \tau_2}\,\textsc{Sub} \qquad \frac{\Gamma \vdash \forall x : [\![x : \gamma_1]\!] \Rightarrow [\![x : \gamma_2]\!]}{\Gamma \vdash \gamma_1 \prec: \gamma_2}\,\textsc{Sub}'$$

$$\frac{\Gamma(p) \equiv \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1' \qquad \Gamma \vdash \tau_1' \prec: \tau_1}{\Gamma \vdash p(e) : \tau_2}\ \text{T}_2\text{-Op}$$

$$\frac{\Gamma(p) \equiv \tau_1 \to \mathsf{Bool} \qquad \Gamma \vdash e : \tau_1' \qquad \Gamma \vdash \tau_1' \prec: \tau_1}{\Gamma \vdash p(e)^b : \mathsf{Bool}}\ \text{T}_2\text{-Op}^b$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash \phi : \mathsf{Bool} \qquad {\scriptstyle x \notin FV(e)}}{\Gamma \vdash \forall x \colon x = e \Rightarrow \phi : \mathsf{Bool}}\ \text{TH}_2\text{-Eq}$$

$$\frac{\Gamma \vdash e : \mathsf{Set}\,\tau_1 \qquad \Gamma, x : \tau_2 \vdash \phi : \mathsf{Bool} \qquad \Gamma \vdash \tau_1 <: \tau_2 \qquad {\scriptstyle x \notin FV(e)}}{\Gamma \vdash \forall x \colon x \in e \Rightarrow \phi : \mathsf{Bool}}\ \text{TH}_2\text{-Mem}$$

Figure 5.4.: $\mathcal{T}_2$-typing rules for Boolean expressions and typing hypotheses

The partitioned semantics of subtyping constraints yields the following rules:

$$\frac{{\scriptstyle \beta \in \mathcal{S}}}{\sigma \models \Gamma \vdash \beta <: \beta}\ \mathcal{C}\text{-}<:\text{-}\beta \qquad \frac{\sigma \models \Gamma \vdash \tau_1' <: \tau_1 \qquad \sigma \models \Gamma, x : \tau_1' \vdash \tau_2 <: \tau_2'}{\sigma \models \Gamma \vdash (x : \tau_1) \to \tau_2 <: (x : \tau_1') \to \tau_2'}\ \mathcal{C}\text{-}<:\text{-Arrow}$$

$$\frac{\sigma \models \Gamma \vdash \tau_1 <: \tau_2}{\sigma \models \Gamma \vdash \mathsf{Set}\,\tau_1 <: \mathsf{Set}\,\tau_2}\ \mathcal{C}\text{-}<:\text{-Set} \qquad \frac{\sigma\Gamma \vdash \{x : \sigma\tau_1 \mid \phi_1\} \prec: \{x : \sigma\tau_2 \mid \phi_2\}}{\sigma \models \Gamma \vdash \{x : \tau_1 \mid \phi_1\} <: \{x : \tau_2 \mid \phi_2\}}\ \mathcal{C}\text{-}<:\text{-Ref}$$

$$\frac{{\scriptstyle \beta \in \mathcal{S}}}{\Gamma \vdash \beta \prec: \beta}\ \mathcal{C}\text{-}\prec:\text{-}\beta \qquad \frac{\Gamma \vdash \tau_1' \prec: \tau_1 \qquad \Gamma, x : \tau_1' \vdash \tau_2 \prec: \tau_2'}{\Gamma \vdash (x : \tau_1) \to \tau_2 \prec: (x : \tau_1') \to \tau_2'}\ \mathcal{C}\text{-}\prec:\text{-Arrow}$$

$$\frac{\Gamma \vdash \tau_1 \prec: \tau_2}{\Gamma \vdash \mathsf{Set}\,\tau_1 \prec: \mathsf{Set}\,\tau_2}\ \mathcal{C}\text{-}\prec:\text{-Set} \qquad \frac{\Gamma \vdash \tau_1 \cong \tau_2 \qquad \Gamma, x : \tau_1 \vdash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{x : \tau_1 \mid \phi_1\} \prec: \{x : \tau_2 \mid \phi_2\}}\ \mathcal{C}\text{-}\prec:\text{-Ref}$$

Rule $\mathcal{C}$-$\prec$:-Ref says that a type $\{x : \tau_1 \mid \phi_1\}$ is a subtype of $\{x : \tau_2 \mid \phi_2\}$ in a context Γ if and only if the normal forms of the base types τ_1 and τ_2 are equal, and $\phi_1 \Rightarrow \phi_2$ in the context $\Gamma, x : \tau_1$ is valid. Note that dependent functions are contra-variant on their arguments while they are covariant on their result (rules $\mathcal{C}$-$<$:-Arrow and $\mathcal{C}$-$\prec$:-Arrow). This has the effect of shrinking their domain while expanding their codomain.

Type inference rules

The expressiveness and flexibility of refinement types allow us to use them in rather subtle ways. Any TLA$^+$ formula ϕ can be typed $\{x : \mathsf{Bool} \mid x \Leftrightarrow \phi\}$, and any TLA$^+$ expression e can be typed $\{x : \tau \mid x = e\}$, for some type τ. We chose to type non-Boolean expressions with the most precise refinement possible. But in order to keep

$$\frac{}{\Gamma \vdash \{\} : \mathsf{Set}\,\{x : \alpha \mid \textsc{false}\}}\ \mathrm{T}_2\text{-}\textsc{Empty} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_1, e_2\} : \mathsf{Set}\,(\tau_1 \uplus \tau_2)}\ \mathrm{T}_2\text{-}\textsc{Pair}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\,\tau_1 \qquad \Gamma, x : \tau_2 \vdash \phi : \mathsf{Bool} \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash \{x \in S : \phi\} : \mathsf{Set}\,\{x : \tau_1 \mid \phi\}}\ \mathrm{T}_2\text{-}\textsc{SetComp}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash \tau_1 \prec: \tau_1 \uplus \tau_2 \\ \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_2 \prec: \tau_1 \uplus \tau_2\end{array}}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ \mathrm{T}_2\text{-}\textsc{Eq} \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e_2 : \mathsf{Set}\,\tau_2\end{array} \qquad \Gamma \vdash \tau_1 \prec: \tau_2}{\Gamma \vdash e_1 \in e_2 : \mathsf{Bool}}\ \mathrm{T}_2\text{-}\textsc{Mem}$$

$$\frac{\begin{array}{l}\Gamma \vdash f : \tau_1 \\ \Gamma \vdash e : \tau_2\end{array} \qquad \Gamma \vdash \tau_2 \prec: \mathsf{dom}(\tau_1)}{\Gamma \vdash f[e] : \mathsf{cod}(\tau_1 \cdot [e])}\ \mathrm{T}_2\text{-}\textsc{App} \qquad \frac{\Gamma \vdash f : \tau_1}{\Gamma \vdash \textsc{domain}\ f : \mathsf{Set}\,(\mathsf{dom}(\tau_1))}\ \mathrm{T}_2\text{-}\textsc{Dom}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\,\tau_1 \qquad \Gamma, x : \tau_2 \vdash e : \tau_3 \qquad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash [x \in S \mapsto e] : (x : \tau_2) \to \tau_3}\ \mathrm{T}_2\text{-}\textsc{Fun}$$

$$\frac{n \in Int}{\Gamma \vdash n : \{x : \mathsf{Int} \mid x = n\}}\ \mathrm{T}_2\text{-}\textsc{Num} \qquad \frac{\Gamma \vdash e_i : \tau_i \qquad \Gamma \vdash \tau_i \prec: \mathsf{Int} \qquad i \in \{1,2\}}{\Gamma \vdash e_1 + e_2 : \{x : \mathsf{Int} \mid x = e_1 + e_2\}}\ \mathrm{T}_2\text{-}\textsc{Plus}$$

$$\frac{\Gamma \vdash e_i : \tau_i \qquad \Gamma \vdash \tau_i \prec: \mathsf{Int} \qquad i \in \{1,2\}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}\ \mathrm{T}_2\text{-}\textsc{Less}$$

Figure 5.5.: $\mathcal{T}_2$-typing rules for set, function and arithmetic expressions

the Boolean reasoning at the Boolean level instead of at the type level, we type Boolean expressions just as Bool, without any refinement. Consequently, the typing rules for logical expressions are almost the same as in $\mathcal{T}_1$ (Figure 5.2). The exceptions are in the rules for operators and for typing hypotheses with set membership, where equality checking is replaced by subtype checking, as shown in Figure 5.4.

We introduce an additional union type operator $\uplus$ that combines two types, as defined by the following properties:

$$\begin{array}{rcll}
\{x : \tau_1 \mid \phi_1\} \uplus \{x : \tau_2 \mid \phi_2\} & = & \{x : \tau_1 \mid \phi_1 \vee \phi_2\} & \text{if } \tau_1 \cong \tau_2 \\
((x : \tau_1) \to \tau_1') \uplus ((x : \tau_2) \to \tau_2') & = & (x : \tau_1) \to (\tau_2 \uplus \tau_2') & \text{if } \tau_1 \cong \tau_2 \\
\mathsf{Set}\,\tau_1 \uplus \mathsf{Set}\,\tau_2 & = & \mathsf{Set}\,(\tau_1 \uplus \tau_2) & \\
\beta_1 \uplus \beta_2 & = & \beta_1 & \text{if } \beta_1 \cong \beta_2
\end{array}$$

In particular, refinement types can be combined if they have the same base type, and functions can be combined if they have the same domain. For any other pair

of types τ_1 and τ_2, $\tau_1 \uplus \tau_2$ is undefined. Now we can simply assign a type of the form $\mathsf{Set}\,(\tau_1 \uplus \tau_2)$ to a pair $\{e_1, e_2\}$, where $e_1 : \tau_1$ and $e_2 : \tau_2$ (rule T_2-Pair).

Another consequence of the precision provided by refinement types is that they force us to use a weak form of type equality. In the case of $x = y$, if we require the types of x and y to be exactly equal, we would be ruling out many typable expressions. Instead, the rules for equality (TH_2-Eq and T_2-Eq) requires them to have a least common super-type, represented by the union of the types of x and y. Suppose we want to type the expression $3 = 4$. It is false but still typable because the types $\{x : \mathsf{Int} \mid x = 3\}$ and $\{x : \mathsf{Int} \mid x = 4\}$, which have the same base type Int, are both subtypes of $\{x : \mathsf{Int} \mid x = 3\} \uplus \{x : \mathsf{Int} \mid x = 4\} = \{x : \mathsf{Int} \mid x = 3 \vee x = 4\}$. (In fact, the $\mathcal{T}_2$-types of any two expressions e_1 and e_2 carry all the information required to decide the validity of $e_1 = e_2$; so we could use it to rewrite the equality to FALSE, but we do not exploit this possibility.)

In order to extract the domain and codomain of a functional type we use the same type operators as in $\mathcal{T}_1$, with their properties updated for dependent types:

$$\mathsf{dom}((x : \tau_1) \to \tau_2) = \tau_1 \qquad \mathsf{cod}((x : \tau_1) \to \tau_2) = \tau_2$$

Observe in rule T_2-Fun that the type τ_3 may have a free variable x, which is bound either by the context $\Gamma, x : \tau_2$ or by the function domain $x : \tau_2$.

The rule T_2-App assigns the type $\mathsf{cod}(\tau_1[e])$ for a function application expression. The type $\tau_1[e]$ represents an explicit substitution applied to τ_1, the type of f. This notion of substitution is similar to the explicit substitutions for λ-calculus [ACCL90]. Assuming τ_1 is a functional type of the form $(x : \alpha_1) \to \alpha_2$, as expected by the typing rule, the type $\tau_1[e]$ would replace x by the TLA$^+$ expression e in α_2. The substitution is meant to be delayed until α_2 is instantiated to a ground type and x can be determined. At that point, the substitution $[x \leftarrow e]$ can be applied to refinement predicates as a usual TLA$^+$ variable substitution (cf. Section 2.1).

We append explicit substitution modifiers θ to type variables in the type grammar:

$$\tau ::= \ldots \mid \alpha \cdot \theta \qquad\qquad \theta ::= \square \mid [e], \theta \mid [x \leftarrow e], \theta$$

where $\square$ is the empty substitution, x is a variable symbol in $\mathcal{V}$, and e is a TLA$^+$ expression. Instead of $\alpha \cdot \square$, we write simply α. A substitution θ is a sequence of atomic substitutions. There are two kind of atomic substitutions. In the case of $[e]$, the name of the variable to replace is not known until it is applied to a functional

type.[2] The other is the usual $[x \leftarrow e]$. Their properties are:

$$\begin{array}{rcll} \{x' : \tau \mid \phi\} \cdot [x \leftarrow e], \theta & \triangleq & \{x' : \tau \mid \phi[x \leftarrow e]\} \cdot \theta & \text{if } x \neq x' \\ ((x : \tau_1) \to \tau_2) \cdot [e], \theta & \triangleq & (x : \tau_1) \to (\tau_2 \cdot [x \leftarrow e], \theta) & \text{if } x \notin dom(\theta) \\ (\mathsf{Set}\,\tau) \cdot \theta & \triangleq & \mathsf{Set}\,(\tau \cdot \theta) & \\ \beta \cdot \theta & \triangleq & \beta & \end{array}$$

where the domain of a substitution is defined by:

$$dom(\square) \triangleq \{\}, \quad dom([x \leftarrow e], \theta) \triangleq \{x\} \cup dom(\theta), \quad \text{and} \quad dom([e], \theta) \triangleq dom(\theta).$$

The relevant cases are for refinement and functional types. The first property effectively applies the substitution to a refinement predicate. The second property changes the nameless $[e]$ to $[x \leftarrow e]$, when applied to the codomain, taking the name x from the function's argument. If $[e]$ is not first applied to a functional type as expected, the substitution fails. Additionally, $\tau \cdot \square \triangleq \tau$ and $(\tau \cdot \theta_1) \cdot \theta_2 \triangleq \tau \cdot (\theta_1, \theta_2)$. By systematically applying all these properties, the right ground types can be simplified and reduced to types without substitutions.

The rules for arithmetic operators (T_2-Plus and T_2-Less) require their arguments to be integers through the condition $e_i \prec: \mathsf{Int}$. Literal integers (T_2-Num) and addition (T_2-Plus) carry the exact value they represent encoded in their refinement type. Note that the arithmetic constants and operators can have constant types:

$$\begin{array}{rl} n & : \{z : \mathsf{Int} \mid z = n\} \\ Int & : \mathsf{Set}\,\mathsf{Int} \\ + & : (x : \mathsf{Int}) \to (y : \mathsf{Int}) \to \{z : \mathsf{Int} \mid z = x + y\} \\ \div & : (x : \mathsf{Int}) \to (y : \{z : \mathsf{Int} \mid 0 < z\}) \to \{z : \mathsf{Int} \mid z = x \div y\} \\ .. & : (x : \mathsf{Int}) \to (y : \mathsf{Int}) \to \mathsf{Set}\,\{z : \mathsf{Int} \mid x \leq z \wedge z \leq y\} \end{array}$$

They can be treated as any other operator with a fixed type, but the dedicated inference rules presented in Figure 5.5 generate slightly shorter derivations.

5.4. Soundness

Type annotations, as well as the typing hypotheses, restrict the domain of interpretation of the quantified variables. Suppose the formula ϕ is not valid. Then, there exists some valuation in the universe $\mathcal{D}$ which makes the formula false. Still, there may exist some other values in $\mathcal{D}$ that makes ϕ true. Let us call A the set of all

[2] Our implementation internally represents TLA$^+$ expressions using De Bruijn indices [ACCL90]. Therefore, substitutions of the form $[x \leftarrow e]$ are not needed, and implementing explicit substitutions $[e]$ is greatly simplified.

values that make ϕ true. We want to show that the type system does not generate annotations for ϕ, resulting in ϕ', such that those annotations restrict or confine the domain of evaluation of the variables to the set A which would make ϕ' valid.

For example, consider $\forall x : x < x + 1$. Since the semantics of TLA$^+$ does not define the meaning of $<$ when $x \notin Int$, the formula is not known to be true for some valuations of x, precisely when $x \notin Int$. However, if we annotate x incorrectly with an integer type τ_{Int} (that is, either Int in $\mathcal{T}_1$, or $\{x : \mathsf{Int} \mid \phi\}$ in $\mathcal{T}_2$, even when ϕ is equivalent to FALSE), then $\forall x^{\tau_{Int}} : x < x + 1$ would become valid, because x would be evaluated precisely in those values that make $x < x + 1$ true. In essence, we need to prove that type assignments only follow from typing hypotheses.

The following theorem proves that the type system $\mathcal{T}_1$ is sound. The proof for $\mathcal{T}_2$ is practically the same, since they share the typing rules for Boolean expressions and typing hypotheses.

Theorem 14 (Soundness)**.** *If $x : \tau$ is a typing of ϕ, then $\vdash \forall x : \phi$ iff $\vdash \forall x^\tau : \phi$.*

Proof. $\Rightarrow$) If ϕ is true in all models of the untyped universe, then in a sorted universe that restricts the domain of interpretation, ϕ is also trivially true.

$\Leftarrow$) Let $\phi_1 \triangleq \forall x^\tau : \phi$. Assuming $\vdash \phi_1$, we want to prove $\vdash \forall x : \phi$.

PROOF We know that:

$\langle 1 \rangle 1$. $x : \tau \vdash \phi : \mathsf{Bool}$ is valid (i.e. there is a type derivation), by the hypothesis.

$\langle 1 \rangle 2$. Let $\phi_2 \triangleq \mathcal{R}(\phi_1) = \forall x : [\![x : \tau]\!] \Rightarrow \phi$. Then $\vdash \phi_2$ holds, from assumption $\vdash \phi_1$, by Lemma 12.

We need to show that $[\![x : \tau]\!]$, derived from $x : \tau$, does not constrain the domain of evaluation of x in ϕ. We proceed by a case analysis on the shape of ϕ.

$\langle 1 \rangle 3$. CASE 1. There is no typing hypothesis for the variable x in ϕ.

PROOF

$\langle 2 \rangle 1$. The type derivation on ϕ yields the judgement $x : \tau_x \vdash \phi : \mathsf{Bool}$, by step $\langle 1 \rangle 1$. After unification, type τ_x will be equal to τ. The first applied rule is T-QUANT, the only possible one, since there are no typing hypotheses.

$\langle 2 \rangle 2$. The type τ_x can only be unified to a safe type τ_s.

PROOF The TH (typing hypothesis) rules, where unification of types happens, do not apply, meaning that τ_x cannot be unified with any non-safe type such as Bool, Int or functions. The only applicable rules that may promote τ_x are the rules $\mathrm{T}_{1,2}$-MEM, $\mathrm{T}_{1,2}$-SETCOMP, $\mathrm{T}_{1,2}$-PAIR, $\mathrm{T}_{1,2}$-POWER or $\mathrm{T}_{1,2}$-UNION, but these result in a safe Set type. For example, rule $\mathrm{T}_{1,2}$-PLUS requires establishing that τ_x is an integer, which is impossible.

$\langle 2 \rangle 3$. Finally, since τ_s is safe, it does not compromise the validity of ϕ_2 when $x : \tau_s$ is relativized to $[\![x : \tau_s]\!]$, by Lemma 11.

⟨1⟩4. CASE 2. If ϕ is of the form $\mathcal{H}(x) \Rightarrow \phi_1$, then $\vdash \forall x : \mathcal{H}(x) \Rightarrow \phi_1$.

PROOF

⟨2⟩1. Suffices to prove that $\mathcal{H}(x) \Rightarrow [\![x : \tau]\!]$.

⟨2⟩2. Suppose that $\mathcal{H}(x)$ is of the form $x \in s$. The first rule applied in the type derivation is necessarily TH-MEM, yielding

$$\boxed{1} \quad \vdash s : \mathsf{Set}\,\tau_x \qquad\qquad \boxed{2} \quad x : \tau_x \vdash \phi_1 : \mathsf{Bool}$$

Here, we see that the fresh type variable τ_x is the same in both sides of the derivation, which results in the unification of the types of x and s. The TH rules are the only ones that share type variables in their different premises.

We apply induction on $FV(\mathcal{H}(x))$. For simplicity, we consider that $\mathcal{H}(x)$ does not include quantified formulas.

⟨3⟩1. (Base case) There are no free variables, meaning that the type of x does not depend on the type of any other variable. Therefore, it is trivially a constant type or an atomic type t. For instance, if s is Int, the goal is to show that $x \in Int \Rightarrow [\![x : \tau_x]\!]$. So τ_x is unified with Int and $[\![x : \mathsf{Int}]\!] = x \in Int = \mathcal{H}(x)$.

⟨3⟩2. (Inductive step) We proceed by a case analysis on the shape of s, which has to be necessarily a set, otherwise it would not match with $\mathsf{Set}\,\tau_x$ in $\boxed{1}$.

⟨4⟩1. CASE $s \triangleq$ SUBSET t. The goal is to show that $x \in$ SUBSET $t \Rightarrow [\![x : \tau_x]\!]$. Given that $t : \alpha_t$, then τ_x is unified with $\mathsf{Set}\,\alpha_t$. Then

$$[\![x : \mathsf{Set}\,\alpha_t]\!] = \forall z \in x : [\![z : \alpha_t]\!],$$

by the inductive hypothesis $z \in t \Rightarrow [\![z : \alpha_t]\!]$.

⟨4⟩2. The other cases are proved in a similar way.

⟨2⟩3. The case where $\mathcal{H}(x)$ is of the form $x = e$ is similar to the step ⟨2⟩2.

⟨2⟩4. QED, by ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3.

⟨1⟩5. QED, by steps ⟨1⟩2, ⟨1⟩4 and ⟨1⟩5. □

5.5. Type synthesis

A type derivation through inference rules exhibits a TLA$^+$ expression with each of its sub-expressions associated with a type, subject to certain type constraints. In order to construct a tree derivation, the expected type of an expression and the types obtained from the premises are forced to match. As it is already standard for many variants of simple-typed λ-calculus [Pot96, OSW97, PR05], the algorithmic perspective of such construction is decomposed into a constraint generation phase followed by a constraint solving phase. It essentially amounts to generate and solve

a single large constraint that encompasses all atomic type constraints appearing in the derivation. This constraint, built according to the type inference rules, reproduces the structure of the type derivation using extended constraint constructors. Our algorithm for type synthesis takes as input a TLA$^+$ expression and, in case all constraints are satisfied, the output is an equivalid annotated version of the original formula.

The constraint solving phase synthesizes types mainly from constraints derived from typing hypotheses, by recording them in a type variable mapping. Then, the constraints ground by that type assignment have to be proved valid. The type synthesis algorithm is parameterized by a type system. Constraint solving in $\mathcal{T}_1$ reduces simply to first-order unification of type equality relationships. Constraint solving in $\mathcal{T}_2$ in addition requires unification of subtype constraints. The atomic constraints for $\mathcal{T}_2$ subsume the atomic constraints for $\mathcal{T}_1$. Therefore, we consider only one constraint language —and, consequently, one constraint solving algorithm— for both type systems.

Constraints Typically, constraint languages for type reconstruction in programming languages include constraints for equality and subtyping [KF07, Pot96, PR05], analogous to our relations $\equiv$ and $<:$. In our constraint language, we additionally include non-unifiable relationships between types. The complete definition of the constraint language is:

$$\begin{aligned} \mathcal{C}_A &::= \Gamma \vdash \tau \equiv \tau \mid \Gamma \vdash \tau \cong \tau \mid \Gamma \vdash \tau <: \tau \mid \Gamma \vdash \tau \prec: \tau \\ \mathcal{C} &::= \mathcal{C}_A \mid \mathcal{C} \wedge \mathcal{C} \mid \exists \alpha. \mathcal{C} \end{aligned}$$

In addition to atomic constraints $\mathcal{C}_A$, which includes the equivalence and subtyping relations, a constraint $\mathcal{C}$ is either a conjunction of constraints, or the existential quantification of type variables. The two new constraints allow to replicate the tree-structure of a type derivation in a single constraint expression. Just for presentational purposes, sometimes we write $\exists \alpha_a. (\ldots (\exists \alpha_b. \mathcal{C}))$ as $\exists \alpha_{a,\ldots,b}. \mathcal{C}$, and a concatenation of constraint conjunctions $\mathcal{C}_1 \wedge (\mathcal{C}_2 \wedge (\ldots \wedge \mathcal{C}_n))$ as a multi-line list of constraints, as in TLA$^+$.

A constraint $\mathcal{C}$ is *satisfiable*, noted $\sigma \models \mathcal{C}$, iff there exists a ground assignment σ that satisfies $\mathcal{C}$. Non-atomic constraints are interpreted by the following rules:

$$\frac{\sigma \models \mathcal{C}_1 \qquad \sigma \models \mathcal{C}_2}{\sigma \models \mathcal{C}_1 \wedge \mathcal{C}_2}\, \mathcal{C}\text{-}\wedge \qquad \frac{\sigma, \alpha \mapsto \gamma \models \mathcal{C}}{\sigma \models \exists \alpha. \mathcal{C}}\, \mathcal{C}\text{-}\exists$$

where the mapping $\sigma, \alpha \mapsto \gamma$ updates σ with a new assignment where $\alpha \notin dom(\sigma)$ and γ is a ground type. Atomic constraint judgements are interpreted by the above rules $\mathcal{C}$-$\cong$-*, $\mathcal{C}$-$\equiv$-*, $\mathcal{C}$-$<:$-*, and $\mathcal{C}$-$\prec:$-*, where * stands for β, Set, Arrow, and Ref.

5.5.1. Constraint generation

We define a constraint generation (CG) operator that, given a typing context Γ, a TLA$^+$ expression e, and an expected type τ, with $FV(e) \subseteq dom(\Gamma)$, returns a constraint $\langle\langle \Gamma \vdash e : \tau \rangle\rangle$. The operator is recursively defined over the structure of e. Initially, e is a proof obligation, the expected type is Bool, and the typing context maps the free variables of e to fresh type variables. The defining rules are essentially derived from their corresponding type inference rules for each expression. The resulting constraint has a linear size with respect to the size of the original formula.

The complete set of CG rules for $\mathcal{T}_1$ and $\mathcal{T}_2$, including the rules for tuples, records and IF-THEN-ELSE expressions, can be found in the appendix B. We show through an example how CG rules can be obtained from the inference rules. The CG rule for set comprehension, named CG$_2$-SETCOMP, is the following:

$$\begin{aligned}\langle\langle \Gamma \vdash \{x \in S : \phi\} : \tau \rangle\rangle \;\triangleq\; \exists \alpha_1 \alpha_2.\; & \wedge \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1 \rangle\rangle \\ & \wedge \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool} \rangle\rangle \\ & \wedge \Gamma \vdash \alpha_2 <: \alpha_1 \\ & \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}\end{aligned}$$

The constraint is the conjunction of the premises of the inference rule T$_2$-SETCOMP plus an extra atomic constraint matching the expected type with the prospective type assembled from the premises. That is, the type τ passed as the third argument is unified in the last constraint with the type assigned to $\{x \in S : \phi\}$ in the conclusion of the inference rule. Every free type that appears in the typing rule premises (τ_1 and τ_2) is replaced by a fresh type variable (α_1 and α_2) and existentially bound.

The following theorem, which is applicable to both type systems, asserts soundness and completeness of the CG rules with respect to the type inference rules, when the judgements are ground by a type assignment σ. The correspondence with the inference system is straightforward since the CG rules are defined in a systematic way and almost verbatim from the inference rules.

Theorem 15 (CG soundness and completeness)**.** *Assuming $FV(e) = dom(\Gamma)$, then $\sigma \models \langle\langle \Gamma \vdash e : \tau \rangle\rangle$ if and only if $\sigma\Gamma \vdash e : \sigma\tau$, for some ground assignment σ.*

In other words, if the constraint $\langle\langle\ \Gamma \vdash e : \tau\ \rangle\rangle$ is satisfied by σ, then the generated constraint contains only valid equality and subtype relationships, and the expression e is typable. Specifically, $\langle \sigma\Gamma, \sigma\tau \rangle$ is a typing of e. Conversely, the CG rules applied to a typable expression, generate a satisfiable constraint.

Proof. By structural induction on e, using in every case the corresponding type inference rules, the CG rules, and constraint satisfiability. We prove a representative case: when e is $\{x \in S : \phi\}$ in the system $\mathcal{T}_2$. All other cases are simpler than this one.

PROOF

$\langle 1\rangle 1$. ($\Rightarrow$) Assume $\sigma\Gamma \vdash \{x \in S : \phi\} : \sigma\tau$.

$\langle 2\rangle 1$. We may assume, without lost of generality, that $x \notin dom(\Gamma)$. Since only rule the inverse of the rule T_2-SETCOMP is applicable, there must exist τ_1, τ_2, such that:

$$\boxed{1}\ \sigma\Gamma \vdash S : \sigma(\mathsf{Set}\,\tau_1) \qquad \boxed{2}\ \sigma\Gamma, x : \tau_2 \vdash \phi : \mathsf{Bool}$$
$$\boxed{3}\ \sigma \models \Gamma \vdash \tau_2 <: \tau_1 \qquad \boxed{4}\ \sigma \models \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \tau_1 \mid \phi\}$$

$\langle 2\rangle 2$. Let $\alpha_1, \alpha_2 \notin dom(\sigma)$ and let $\sigma' \triangleq \sigma, \alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2$. The following holds:

$\langle 3\rangle 1$. $\sigma' \models \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle$. This follows from $\boxed{1}$, by inductive hypothesis.

$\langle 3\rangle 2$. $\sigma' \models \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle$. From $\boxed{2}$, by inductive hypothesis.

$\langle 3\rangle 3$. $\sigma' \models \Gamma \vdash \alpha_2 <: \alpha_1$. From $\boxed{3}$, by rule SUB.

$\langle 3\rangle 4$. $\sigma' \models \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$. From $\boxed{4}$, by rule EQ.

$\langle 2\rangle 3$. The assertions in $\langle 2\rangle 2$ can be conjoined by rule $\mathcal{C}$-$\wedge$ into:

$$\sigma' \models \wedge\ \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle\ \wedge\ \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle \\ \wedge\ \Gamma \vdash \alpha_2 <: \alpha_1\ \wedge\ \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$$

$\langle 2\rangle 4$. From $\langle 2\rangle 3$, by the rule $\mathcal{C}$-$\exists$:

$$\sigma \models \exists \alpha_1, \alpha_2.\ \wedge\ \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle\ \wedge\ \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle \\ \wedge\ \Gamma \vdash \alpha_2 <: \alpha_1\ \wedge\ \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$$

$\langle 2\rangle 5$. QED. By CG_2-SETCOMP, $\langle 2\rangle 4$ is equal to the goal.

$\langle 1\rangle 2$. ($\Leftarrow$) Assume $\sigma \models \langle\langle \Gamma \vdash \{x \in S : \phi\} : \tau\rangle\rangle$.

$\langle 2\rangle 1$. By CG_2-SETCOMP, assuming $x \notin dom(\Gamma)$, the following is valid:

$$\sigma \models \exists \alpha_1, \alpha_2.\ \wedge\ \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle\ \wedge\ \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle \\ \wedge\ \Gamma \vdash \alpha_2 <: \alpha_1\ \wedge\ \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$$

$\langle 2\rangle 2$. We may assume, without lost of generality, that $\alpha_1, \alpha_2 \notin dom(\sigma)$. By rule $\mathcal{C}$-$\exists$, there must exists τ_1 and τ_2, where $\sigma' \triangleq \sigma, \alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2$, such that:

$$\sigma' \models \wedge\ \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle\ \wedge\ \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle \\ \wedge\ \Gamma \vdash \alpha_2 <: \alpha_1\ \wedge\ \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$$

$\langle 2\rangle 3$. From $\langle 2\rangle 2$, the following judgements are valid, by the rule $\mathcal{C}$-$\wedge$:

$$\boxed{1}\ \sigma' \models \langle\langle \Gamma \vdash S : \mathsf{Set}\,\alpha_1\rangle\rangle \qquad \boxed{2}\ \sigma' \models \langle\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool}\rangle\rangle$$
$$\boxed{3}\ \sigma' \models \Gamma \vdash \alpha_2 <: \alpha_1 \qquad \boxed{4}\ \sigma' \models \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}$$

$\langle 2 \rangle 4$. QED. Finally, to show $\sigma\Gamma \vdash \{x \in S : \phi\} : \sigma\tau$, it suffices to prove that the following four steps are valid. From $\langle 2 \rangle 3$, by the rule T_2-SetComp:

$\langle 3 \rangle 1$. $\sigma\Gamma \vdash S : \sigma(\mathsf{Set}\,\tau_1)$. This follows from (1), by the inductive hypothesis and by simplification of substitutions.

$\langle 3 \rangle 2$. $\sigma\Gamma, x : \tau_2 \vdash \phi : \mathsf{Bool}$. From (2), by the inductive hypothesis and by simplification of substitutions.

$\langle 3 \rangle 3$. $\sigma \models \tau_2 <: \tau_1$. From (3), by rule Sub.

$\langle 3 \rangle 4$. $\sigma\tau \cong \sigma\mathsf{Set}\,\{x : \tau_1 \mid \phi\}$. From (4), by rule Eq. □

Running example 1/5 We use the toy formula 5.1 for a running example of the constraint generation and the constraint solving algorithms. In Figure 5.6, we show the constraint $\mathcal{C}_0$ that is generated from this formula in the type system $\mathcal{T}_2$. Since it has no free variables, the typing context passed to the CG procedure is initially empty. The expected type of any proof obligation is Bool. The empty contexts in atomic constraints are omitted. For instance, $\emptyset \vdash \tau_1 \equiv \tau_2$ is just written as $\tau_1 \equiv \tau_2$.

5.5.2. Constraint solving

The constraint solving algorithm takes as input the constraint $\mathcal{C}$ generated in the previous section and proceeds in four main steps. We outline the algorithm as follows, with descriptions of the relevant parts given afterwards.

1. Structural type assignment. The first step is to find an assignment for the type variables by the application of two similar unification algorithms. First, for $\equiv$-equality constraints, and subsequently, for $<:$-subtyping constraints. Other than being simple first-order unification algorithms, they unify types up to refinement predicates between subtype relations. That is, the structural shape of the types is constructed except for the refinement predicates, which are deferred to the subsequent step by inserting placeholders formulas in their place. The result is a pair $\langle \sigma, \mathcal{C} \rangle$, where σ is a (partial) type assignment for the type variables of $\mathcal{C}$, and $\mathcal{C}$ is the reduced constraint containing no unifying relations. Both σ and $\mathcal{C}$ may contain placeholders.
2. Placeholder solution. Placeholders represent unknown TLA$^+$ formulas in implications resulting from subtype unification. Our approach to find concrete refinement predicates is based on the algorithm by Knowles and Flanagan [KF07], which, in turn, is based on the intuition that implications can be analyzed as dataflow graphs. Once a solution is found for every introduced placeholder, the types in σ and $\mathcal{C}$ can be completed. Obviously, this step is only required for the system $\mathcal{T}_2$.

$$
\begin{aligned}
\mathcal{C}_0 \triangleq\ & \langle\langle \varnothing \vdash \forall f : f = [x \in \{1,2,3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1 : \mathsf{Bool} \rangle\rangle \\
=\ & \wedge\ \mathsf{Bool} \equiv \mathsf{Bool} \\
& \wedge\ \exists \alpha_1 . \wedge\ \exists \alpha_2 . \wedge\ \alpha_2 \equiv \alpha_1 \\
& \qquad \wedge\ \exists \alpha_{3,4,8} . \wedge\ \exists \alpha_{5,6,7} . \wedge\ \alpha_5 \equiv \{z : \mathsf{Int} \mid z = 1\} \\
& \qquad\qquad \wedge\ \alpha_6 \equiv \{z : \mathsf{Int} \mid z = 2\} \\
& \qquad\qquad \wedge\ \alpha_7 \equiv \{z : \mathsf{Int} \mid z = 3\} \\
& \qquad\qquad \wedge\ \mathsf{Set}\, \alpha_4 \equiv \mathsf{Set}\, (\alpha_5 \uplus \alpha_6 \uplus \alpha_7) \\
& \qquad\qquad \wedge\ \alpha_4 <: \alpha_3 \\
& \qquad\qquad \wedge\ \exists \alpha_{9,10} . \wedge\ \alpha_9 \equiv \alpha_3 \\
& \qquad\qquad\qquad \wedge\ \alpha_{10} \equiv \alpha_3 \\
& \qquad\qquad\qquad \wedge\ x : \alpha_3 \vdash \alpha_9 \prec : \mathsf{Int} \\
& \qquad\qquad\qquad \wedge\ x : \alpha_3 \vdash \alpha_{10} \prec : \mathsf{Int} \\
& \qquad\qquad\qquad \wedge\ x : \alpha_3 \vdash \alpha_8 \equiv \{z : \mathsf{Int} \mid z = x * x\} \\
& \qquad\qquad \wedge\ \alpha_2 \equiv (x : \alpha_4) \to \alpha_8 \\
& \qquad \wedge\ \mathsf{Bool} \equiv \mathsf{Bool} \\
& \quad \wedge\ \exists \alpha_{11,12} . \wedge\ \exists \alpha_{13,14} . \wedge\ f : \alpha_1 \vdash \alpha_{13} \equiv \Gamma(f) \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{14} \equiv \{z : \mathsf{Int} \mid z = 0\} \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{14} \prec : \mathsf{dom}(\alpha_{13}) \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{11} \equiv \mathsf{cod}(\alpha_{13} \cdot [x \leftarrow 0]) \\
& \qquad \wedge\ \exists \alpha_{15,16} . \wedge\ \exists \alpha_{17,18} . \wedge\ f : \alpha_1 \vdash \alpha_{17} \equiv \Gamma(f) \\
& \qquad\qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{18} \equiv \{z : \mathsf{Int} \mid z = 0\} \\
& \qquad\qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{18} \prec : \mathsf{dom}(\alpha_{17}) \\
& \qquad\qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{15} \equiv \mathsf{cod}(\alpha_{17} \cdot [x \leftarrow 0]) \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{16} \equiv \{z : \mathsf{Int} \mid z = 1\} \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{15} \prec : \mathsf{Int} \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{16} \prec : \mathsf{Int} \\
& \qquad\qquad \wedge\ f : \alpha_1 \vdash \alpha_{12} \equiv \{z : \mathsf{Int} \mid z = f[0] + 1\} \\
& \qquad \wedge\ f : \alpha_1 \vdash \alpha_{11} \prec : \mathsf{Int} \\
& \qquad \wedge\ f : \alpha_1 \vdash \alpha_{12} \prec : \mathsf{Int} \\
& \qquad \wedge\ \mathsf{Bool} \equiv \mathsf{Bool}
\end{aligned}
$$

Figure 5.6.: Constraint generation example in $\mathcal{T}_2$

3. Type-correctness conditions (TCC). The type assignment we were searching for is in the mapping σ, given that two conditions are met:
   a) The unification algorithms apply simplification rules on atomic constraints. In particular, transformations derived from the rules $\mathcal{C}$-$\cong$-Ref and $\mathcal{C}$-$\prec$:-Ref, generate TCCs, that have to be discharged to, and proved by, an SMT solver.
   b) The constraint $\mathcal{C}$ may still contain residual equality and subtyping constraints that have to be solved.
4. Solve the residual constraint. The final step is to solve the residual atomic constraints $\cong$ and $\prec$: in $\mathcal{C}$. We replace $\cong$ by $\equiv$, $\prec$: by $<$:, and the constraint solving algorithm is again executed from the first step on the resulting constraint. The new run may generate new placeholders to solve and new TCCs to prove. If new the conditions are satisfied and the final constraint is reduced to the trivially true constraint, then the algorithm finishes successfully, and the assignment σ, which was recorded in step 3, is correct.

Running example 2/5 At the same time that the constraint $\mathcal{C}_0$ is generated in Figure 5.6, a typed-TLA$^+$ formula ϕ^α is constructed, which is an annotated version of the formula original example formula:

$$\phi^\alpha \triangleq \forall f^{\alpha_1} : f = [x \in \{1,2,3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1$$

This formula is the same as the original one, except that the variable binder in the quantifier is decorated with the type variable α_1, which functions as a placeholder for the final type of the variable, if any. The type α_1 was generated by the CG rule for typing hypotheses corresponding to f. After unification, α_1 should be equal to the final type of f, if the other constraints related to f are valid.

Simplification of atomic constraints

The unification algorithms that we describe below traverse the whole constraint searching non-deterministically for type variables to instantiate. At the same time, they systematically apply whenever possible all rules and properties that simplify atomic constraints and the types occurring in them. Simplification rules for atomic constraints are $\mathcal{C}$-$\equiv$-*, $\mathcal{C}$-$\cong$-*, $\mathcal{C}$-$<$:-*, and $\mathcal{C}$-$\prec$:-*, with * standing for β, Set, Arrow, or Ref. Additionally, we use as rewriting rules the properties for types and type operators, namely, the Property 13 for refinement types, the equations defining the modifiers dom, cod, $\uplus$, delayed substitution, etc. Any relation between incompatible types would make the algorithm abort with a "type error".

Two particular cases are the transformations derived from $\mathcal{C}$-$\cong$-REF and $\mathcal{C}$-$\prec$:-REF, which generate type-correctness conditions. The transformation rules are the following, where τ_1 and τ_2 are ground types, and the refinement types are in normal form:

$$\begin{aligned}
&\Gamma \vdash \{x : \tau_1 \mid \phi_1\} \cong \{y : \tau_2 \mid \phi_2\} \implies_C \Gamma \vdash \tau_1 \cong \tau_2 \\
&\quad \& \; \mathcal{T}cc := \mathcal{T}cc \cup \{\Gamma, x : \tau_1 \vdash \phi_1 \Leftrightarrow (\phi_2[y \leftarrow x])\} \\
&\Gamma \vdash \{x : \tau_1 \mid \phi_1\} \prec: \{y : \tau_2 \mid \phi_2\} \implies_C \Gamma \vdash \tau_1 \cong \tau_2 \\
&\quad \& \; \mathcal{T}cc := \mathcal{T}cc \cup \{\Gamma, x : \tau_1 \vdash \phi_1 \Rightarrow (\phi_2[y \leftarrow x])\}
\end{aligned}$$

These rules reduce to checking that the base types are equal, while the corresponding type-correctness conditions are recorded in $\mathcal{T}cc$. The set $\mathcal{T}cc$, initially empty, collects all TCCs that have to be verified later by an SMT solver.

Type equivalence unification

We describe the unification algorithm for equality constraints as a rule-based state transition system $\Longrightarrow_{Eq}$ operating on states of the form $\langle \sigma \,;\, \mathcal{C} \rangle$, where σ is a type variable assignment and $\mathcal{C}$ is a constraint. Initially, σ is the empty assignment $\Box$ and $\mathcal{C}$ is the constraint to solve. By non-deterministically applying $\Longrightarrow_{Eq}$, a final state $\langle \sigma_f \,;\, \mathcal{C}_f \rangle$ is reached, where the mapping σ_f is the final type assignment, and $\mathcal{C}_f$ contains no $\equiv$-constraints. Incompatible $\equiv$-constraints make the algorithm to abort with a type error. Otherwise, when the transformation rules cannot further be applied, the algorithm terminates successfully.

A single rule (with its symmetric counterpart) instantiates the type variables introduced during constraint generation in the constraints belonging to the variable's scope. The transformation rule is:

$$\langle \sigma \,;\, \exists \alpha . \mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta \equiv \tau \rangle \implies_{Eq} \langle \sigma' \,;\, (\mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta \equiv \tau)[\alpha \leftarrow \tau] \rangle$$

where $\sigma' \triangleq \text{if } FV(\tau) = \{\} \text{ then } \sigma, \alpha \mapsto \tau \text{ else } \sigma$, that is, the assignment is recorded only if τ is ground.[3] Non-ground types correspond to intermediate type variables, which we can ignore in order to avoid carrying their typing contexts into σ. The free variables of a type τ, noted $FV(\tau)$, are the free variables of its refinement predicates. By assumption of the CG rules, for a type τ occurring in a context Γ, $FV(\tau) \subseteq dom(\Gamma)$ holds.

In the constraint $\mathcal{C}_0$ in Figure 5.6, there are two cases of refinement types with free variables. One is the atomic constraint $x : \alpha_3 \vdash \alpha_8 \equiv \{z : \mathsf{Int} \mid z = x * x\}$, where x is

[3] In our TLA$^+$ fragment, recursive operators are not allowed. If that was not the case, the type variable α may occur in the type τ. Then, we would have to check for those occurrences, in order to avoid circular substitutions.

free in refinement type and belongs to the domain of the context. When α_8 is replaced by $\{z : \mathsf{Int} \mid z = x * x\}$ in the constraint $\alpha_2 \equiv (x : \alpha_4) \to \alpha_8$, the variable x is still bound. The second case is the constraint $f : \alpha_1 \vdash \alpha_{12} \equiv \{z : \mathsf{Int} \mid z = f[0] + 1\}$, when α_{12} is substituted in $f : \alpha_1 \vdash \alpha_{12} \prec: \mathsf{Int}$.

The explicit substitution θ is not captured in the ground assignment σ, but the constraint containing it has to be satisfied by a subsequent constraint simplification. Consider the case of unification applied to a constraint such as $\Gamma \vdash \alpha \cdot \theta \equiv \{x : \tau \mid \phi\}$ where the substitution θ is not empty. After an application of the rule $\Longrightarrow_{Eq}$, this constraint results in $\Gamma \vdash \{x : \tau \mid \phi\} \cdot \theta \equiv \{x : \tau \mid \phi\}$. Assuming τ is ground, by simplifying $\equiv$ to $\cong$, the simplification rules yield a new type-correctness condition $\Gamma, x : \tau \vdash \phi\theta \Leftrightarrow \phi$.

Running example 3/5 Applying the equality unification procedure to $\langle \Box \,;\, \mathcal{C}_0 \rangle$ results in the final state $\langle \sigma_1 \,;\, \mathcal{C}_1 \rangle$, where the type variable assignment is:

$$\begin{aligned}
\sigma_1 \triangleq\ & \alpha_1, \alpha_2, \alpha_{13}, \alpha_{17} \mapsto (x : \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\}) \to \{z : \mathsf{Int} \mid z = x * x\},\\
& \alpha_4 \mapsto \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\},\\
& \alpha_5, \alpha_{16} \mapsto \{z : \mathsf{Int} \mid z = 1\},\\
& \alpha_6 \mapsto \{z : \mathsf{Int} \mid z = 2\},\\
& \alpha_7 \mapsto \{z : \mathsf{Int} \mid z = 3\},\\
& \alpha_9, \alpha_{10} \mapsto \alpha_3,\\
& \alpha_{11}, \alpha_{15} \mapsto \{z : \mathsf{Int} \mid z = 0 * 0\},\\
& \alpha_{14}, \alpha_{18} \mapsto \{z : \mathsf{Int} \mid z = 0\}
\end{aligned}$$

and the resulting constraint (ignoring repeated sub-constraints) is:

$$\begin{aligned}
\mathcal{C}_1 \triangleq \exists \alpha_3.\ \wedge\ & \vdash \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\} <: \alpha_3\\
\wedge\ & \Gamma_{f,x} \vdash \alpha_3 \prec: \{z : \mathsf{Int} \mid \textsc{true}\}\\
\wedge\ & \Gamma_f \vdash \{z : \mathsf{Int} \mid z = 0\} \prec: \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\}\\
\wedge\ & \Gamma_f \vdash \{z : \mathsf{Int} \mid z = 0 * 0\} \prec: \{z : \mathsf{Int} \mid \textsc{true}\}\\
\wedge\ & \Gamma_f \vdash \{z : \mathsf{Int} \mid z = 1\} \prec: \{z : \mathsf{Int} \mid \textsc{true}\}\\
\wedge\ & \Gamma_f \vdash \{z : \mathsf{Int} \mid z = f[0] + 1\} \prec: \{z : \mathsf{Int} \mid \textsc{true}\}
\end{aligned}$$

where $\Gamma_f \triangleq f : (x : \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\}) \to \{z : \mathsf{Int} \mid z = x * x\}$ and $\Gamma_{f,x} \triangleq \Gamma_f, x : \alpha_3$. At this point, all unifying equalities containing type variables have been resolved. Observe that the last three constraints are trivially true because the refinement types have the same base type and the right-hand side refinement predicates are TRUE. However, the third conjunct of $\mathcal{C}_1$ yields a non-trivial TCC:

$$\mathcal{T}cc = \{\Gamma_f, z : \mathsf{Int} \vdash z = 0 \Rightarrow z = 1 \vee z = 2 \vee z = 3\}$$

An assignment for α_3 is still missing (together with α_{10} and α_{11} that belong to the same equivalence class). The next step is to solve the unifying subtype relations $<:$.

Subtype unification

Consider the unification process applied to a constraint like $\Gamma \vdash \alpha \equiv \{x : \mathsf{Int} \mid \phi\}$. Finding the shape of the type variable α results in α having τ as a base type and being refined by the predicate ϕ. That is not the case for a subtyping constraint between α and $\{x : \tau \mid \phi\}$, or between $\{x : \tau \mid \phi\}$ and α. In the first case, α could be trivially instantiated to $\{x : \tau \mid \textsc{false}\}$ for the constraint to hold, that is, for the formula $\mathcal{F}(x : \tau \vdash \textsc{false} \Rightarrow \phi)$ to be valid. In the latter case, α could be trivially instantiated to $\{x : \tau \mid \textsc{true}\}$. For the moment, we reconstruct the shape of the variable only with the base type of the refinement. The reconstruction of the refinement predicate is deferred by inserting in its place a *placeholder* symbol, representing a TLA$^+$ formula. We call PH to the set of placeholder identifiers. A placeholder is of the form $\boxed{i} \cdot \theta$, with $i \in PH$. It has a (possibly empty) delayed substitution θ, stemming from the type variable replaced by the refinement type.

Unification in constraints of the form $\Gamma \vdash \tau_1 <: \tau_2$ is performed by the transformation rules $\Longrightarrow_{Sub}$. They operate on states of the form $\langle \sigma \,;\, \mathcal{C} \rangle$, in a similar way as the rules $\Longrightarrow_{Eq}$ for equality unification.

$$\begin{array}{l}
\langle \sigma \,;\, \exists \alpha . \mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta <: \{x : \tau \mid \phi\} \rangle \Longrightarrow_{Sub} \qquad \text{(fresh } i \in PH\text{)} \\
\quad \langle \sigma, \alpha \mapsto \{x : \tau \mid \boxed{i}\} \,;\, (\mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta <: \{x : \tau \mid \phi\})[\alpha \leftarrow \{x : \tau \mid \boxed{i}\}] \rangle \\
\langle \sigma \,;\, \exists \alpha . \mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta <: \tau \rangle \Longrightarrow_{Sub} \qquad \text{if } \tau \text{ is not of the form } \{x : \tau' \mid \phi'\} \\
\quad \langle \sigma, \alpha \mapsto \tau \,;\, (\mathcal{C} \wedge \Gamma \vdash \alpha \cdot \theta <: \tau)[\alpha \leftarrow \tau] \rangle
\end{array}$$

The second transformation unifies general subtyping relationships between a type variable and another type, when this type is not a refinement type. It operates analogously to equality unification. The first case, involving refinement types, has to be treated differently. Subtype unification of a type variable $\alpha \cdot \theta$ with $\{x : \tau \mid \phi\}$ introduces a placeholder symbol $\boxed{i}$ to defer the reconstruction of the refinement predicate. A TLA$^+$ formula ϕ^i is a solution for $\boxed{i}$ if and only if $\mathcal{F}(\Gamma, x : \tau \vdash \phi^i \theta \Rightarrow \phi)$ is valid.

Suptyping constraints now may include refinement types with placeholders instead of predicates. Therefore, we add new simplification rules for this kind of atomic constraints.

$$\begin{array}{rl}
\Gamma \vdash \{x : \tau_1 \mid \boxed{i} \cdot \theta\} \prec: \{y : \tau_2 \mid \phi\} \Longrightarrow_C & \Gamma \vdash \tau_1 \cong \tau_2 \\
& \quad \& \; \mathcal{I}mp := \mathcal{I}mp \cup \{\Gamma, x : \tau_1 \vdash \boxed{i} \cdot \theta \Rightarrow \phi[y \leftarrow x]\} \\
\Gamma \vdash \{x : \tau_1 \mid \phi\} \prec: \{y : \tau_2 \mid \boxed{i} \cdot \theta\} \Longrightarrow_C & \Gamma \vdash \tau_1 \cong \tau_2 \\
& \quad \& \; \mathcal{I}mp := \mathcal{I}mp \cup \{\Gamma, y : \tau_1 \vdash \phi[x \leftarrow y] \Rightarrow \boxed{i} \cdot \theta\} \\
\Gamma \vdash \{x : \tau_1 \mid \boxed{i} \cdot \theta_1\} \prec: \{y : \tau_2 \mid \boxed{j} \cdot \theta_2\} \Longrightarrow_C & \Gamma \vdash \tau_1 \cong \tau_2 \\
& \quad \& \; \mathcal{I}mp := \mathcal{I}mp \cup \{\Gamma, x : \tau_1 \vdash \boxed{i} \cdot \theta_1 \Rightarrow \boxed{j} \cdot (\theta_2, [y \leftarrow x])\}
\end{array}$$

As the above simplification rules, they reduce to check that the types have the same base type, and in addition they generate type-correctness conditions. We collect

these verification conditions, which include unsolved placeholders, in the set $\mathcal{I}mp$ of placeholder implication formulas.

After subtyping unification, unifying constraints may appear in the constraint $\mathcal{C}$ only in the form $\Gamma \vdash \alpha_1 \equiv \alpha_2$ or $\Gamma \vdash \alpha_1 <: \alpha_2$. The variables α_1 and α_2 can be set to a concrete atomic type t, making the equality and subtype relations valid by reflexivity. At this point, the only atomic constraints remaining in $\mathcal{C}$ are of the form $\Gamma \vdash \tau_1 \cong \tau_2$ and $\Gamma \vdash \tau_1 \prec: \tau_2$.

Running example 4/5 Subtype unification on $\langle \sigma_1, \mathcal{C}_1 \rangle$ results in $\langle \sigma_2, \mathcal{C}_2 \rangle$. The updated type assignment is $\sigma_2 \stackrel{\Delta}{=} \sigma_1, \alpha_3 : \{z : \mathsf{Int} \mid \boxed{1}\}$, and the resulting constraint is:

$$\mathcal{C}_3 \stackrel{\Delta}{=} \Gamma_f, x : \{z : \mathsf{Int} \mid \boxed{1}\} \vdash \{z : \mathsf{Int} \mid \boxed{1}\} \prec: \{z : \mathsf{Int} \mid \textsc{true}\}$$

Through the simplification rules, the intermediate constraint

$$\vdash \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\} \prec: \{z : \mathsf{Int} \mid \boxed{1}\}$$

yields a placeholder implication: $\mathcal{I}mp = \{z : \mathsf{Int} \vdash z = 1 \vee z = 2 \vee z = 3 \Rightarrow \boxed{1}\}$.

Finding placeholder formulas

The next step is to compute solutions for the placeholders. Placeholders can appear in the implication set $\mathcal{I}mp$ in the forms $\Gamma \vdash \boxed{i} \cdot \theta \Rightarrow \phi$, $\Gamma \vdash \boxed{i} \cdot \theta_1 \Rightarrow \boxed{j} \cdot \theta_2$ and $\Gamma \vdash \phi \Rightarrow \boxed{i} \cdot \theta$. Before applying the proper solving procedure, some preprocessing of the implications is required. Essentially, we need to obtain formulas where the typing contexts bind only the placeholders' free variables, and where the placeholders occur only with empty delayed substitutions.

Free variable elimination The placeholders are introduced from constraints such as $\Gamma \vdash \alpha \cdot \theta <: \{x : \tau \mid \phi\}$. Every placeholder $\boxed{i} \cdot \theta$ has an associated typing context Γ^i defined as $\Gamma^i \stackrel{\Delta}{=} \Gamma, x : \tau$. If a solution for $\boxed{i}$ exists, it has to be valid in its context Γ^i. More precisely, $\boxed{i}$'s context would be one that binds the free variables of the placeholder. The set of free variables of a placeholder is defined as:

$$FV(\boxed{i} \cdot \theta) \stackrel{\Delta}{=} (dom(\Gamma^i) \setminus dom(\theta)) \cup FV(range(\theta)).$$

Through a series of transformations applied to the implication formulas in $\mathcal{I}mp$, we can remove from the contexts the variables not relevant for the placeholder's solution. To define the transformations, we prove the following equivalences between Γ-relativized formulas.

Lemma 16. *Let* $FV_{i+j} \triangleq FV(\boxed{i} \cdot \theta_i) \cup FV(\boxed{j} \cdot \theta_j)$. *The following equivalences hold:*

1. $\mathcal{F}(\Gamma, x:\tau \vdash \phi \Rightarrow \boxed{j} \cdot \theta)$ *iff* $\mathcal{F}(\Gamma \vdash (\exists x: [\![x:\tau]\!] \wedge \phi) \Rightarrow \boxed{j} \cdot \theta)$ *if* $x \notin FV(\boxed{j} \cdot \theta)$
2. $\mathcal{F}(\Gamma, x:\tau \vdash \boxed{i} \cdot \theta \Rightarrow \phi)$ *iff* $\mathcal{F}(\Gamma \vdash \boxed{i} \cdot \theta \Rightarrow (\forall x: [\![x:\tau]\!] \Rightarrow \phi))$ *if* $x \notin FV(\boxed{i} \cdot \theta)$
3. $\mathcal{F}(\Gamma, x:\tau \vdash \boxed{i} \cdot \theta_i \Rightarrow \boxed{j} \cdot \theta_j)$ *iff* $\mathcal{F}(\Gamma \vdash \boxed{i} \cdot \theta_i \Rightarrow \boxed{j} \cdot \theta_j)$ *if* $x \notin FV_{i+j}$

Proof. In the line 1, by the tautology $(\forall x: \phi_1 \Rightarrow \phi_2) \Leftrightarrow ((\exists x: \phi_1) \Rightarrow \phi_2)$, when $x \notin FV(\phi_2)$. In line 2, by $(\forall x: \phi_1 \Rightarrow \phi_2) \Leftrightarrow (\phi_1 \Rightarrow \forall x: \phi_2)$, when $x \notin FV(\phi_1)$. Line 3 just handles irrelevant assignments. □

After systematically applying these transformations to the formulas in $\mathcal{I}mp$, we know that, if the implication contains a placeholder $\boxed{i} \cdot \theta$, then the context of the formula is Γ^i and $FV(\boxed{i} \cdot \theta) \subseteq dom(\Gamma^i)$ holds.

Delayed substitution elimination We require the following auxiliary definitions, which define the semantic content of a substitution:

$$eq(\Box) \triangleq \textsc{true} \qquad eq([x^\tau \leftarrow e], \theta) \triangleq x = e \wedge eq(\theta) \qquad eq([e], \theta) \triangleq eq(\theta)$$

The following lemmas define the transformations that we apply to the implications in $\mathcal{I}mp$ in order to eliminate the substitutions:

Lemma 17. *The following equivalences hold:*

1. $\mathcal{F}(\Gamma^i \vdash \phi \Rightarrow \boxed{i} \cdot \theta)$ *iff* $\mathcal{F}(\Gamma^i \vdash eq(\theta) \wedge \phi \Rightarrow \boxed{i})$
2. $\mathcal{F}(\Gamma^i \vdash \boxed{i} \cdot \theta \Rightarrow \phi)$ *iff* $\mathcal{F}(\Gamma^i \vdash \boxed{i} \Rightarrow (\neg eq(\theta) \vee \phi))$
3. $\mathcal{F}(\Gamma^j \vdash \boxed{i} \cdot \theta \Rightarrow \boxed{j})$ *iff* $\wedge\, \mathcal{F}(\Gamma^j \vdash \neg eq(\theta) \Rightarrow \boxed{j})$ $\wedge\, \mathcal{F}(\Gamma^j \vdash \boxed{i} \Rightarrow \boxed{j})$
4. $\mathcal{F}(\Gamma^j \vdash \boxed{i} \Rightarrow \boxed{j} \cdot \theta)$ *iff* $\wedge\, \mathcal{F}(\Gamma^j \vdash eq(\theta) \Rightarrow \boxed{i})$ $\wedge\, \mathcal{F}(\Gamma^j \vdash \boxed{i} \Rightarrow \boxed{j})$ *if* $dom(\theta) \not\subseteq FV(\boxed{i})$

Proof. In line 1, the substitutions θ are just replaced by equalities in the antecedent. A solution for $\boxed{i}$ is $eq(\theta) \wedge \phi$. Because $dom(\theta) \not\subseteq FV(\phi)$, then $(eq(\theta) \wedge \phi) \cdot \theta = \phi \cdot \theta = \phi$. Thus, the equivalence holds. Similarly, in line 2 we can replace $\boxed{i}$ by $\neg eq(\theta) \vee \phi$. In line 3, replace $\boxed{j}$ by $\neg eq(\theta) \vee \boxed{i}$. In line 4, replace $\boxed{j}$ by $eq(\theta) \vee \boxed{i}$. Note also that in lines 3 and 4, $\Gamma^i \subseteq \Gamma^j$, by the assumptions. □

After a systematic application of these transformations, the implication formulas in the set $\mathcal{I}mp$ are in one of the following forms: $\Gamma^i \vdash \boxed{i} \Rightarrow \phi$, $\Gamma^j \vdash \boxed{i} \Rightarrow \boxed{j}$ or $\Gamma^i \vdash \phi \Rightarrow \boxed{i}$.

Algorithm for placeholder solution

The algorithm takes the implication formulas in $\mathcal{I}mp$ and generates a graph, whose edges represents the implications. Therefore, the graph has two kind of nodes: fixed TLA$^+$ formulas ϕ and placeholders $\boxed{i}$, where $i \in PH$. An edge has one of these forms: $\phi \longrightarrow \boxed{i}$, $\boxed{i} \longrightarrow \boxed{j}$, or $\boxed{i} \longrightarrow \phi$, where ϕ is a TLA$^+$ formula. In particular, since we do not handle recursive definitions in our TLA$^+$ fragment,[4] the (directed) graph contains no loops and there is at most one path between any two different nodes, greatly simplifying the procedure. Additionally, each placeholder node $\boxed{i}$ has a value, noted v_i, where a value is either undefined, noted $\bullet$, or a TLA$^+$ formula ϕ. We define $\bullet \wedge \phi = \phi$ and $\bullet \vee \phi = \phi$.

We represent the graph as set G of implications. The goal is to find potential solutions for the placeholder's values, i.e. TLA$^+$ formulas, that may satisfy all implications in G. At the end of the procedure, we obtain a placeholder solution that is used to replace the placeholders in the constraint and the type assignment. The resulting placeholder-free constraint still needs to be proven valid. The procedure analyzes each edge of the graph to pass information between the two nodes. At the end, the set G is empty and the solution is in the node's values. For example, suppose $G = \{\phi_1 \Rightarrow \boxed{i}, \boxed{i} \Rightarrow \phi_2\}$ holds. Unless $\Gamma_i \vdash \phi_1 \Rightarrow \phi_2$ holds, there is no valid solution for $\boxed{i}$. Otherwise, $v_i = \phi_1$ or $v_i = \phi_2$ are valid solutions, but we choose the strongest formula of the two, that is ϕ_1. After replacing the placeholder in the constraints by their solutions, $\Gamma^i \vdash \phi_1 \Rightarrow \phi_2$ will have to be added to $\mathcal{T}cc$ and proved as any other type-correctness condition.

Some auxiliary definitions are required. For a given node $\boxed{i}$, the set $i \leftarrow_f$ denotes the collection of formula nodes pointing to $\boxed{i}$. Conversely, the set $i \rightarrow_f$ denotes those formula nodes to which node $\boxed{i}$ points to. Similarly for the set of placeholder nodes $i \leftarrow_p$ and $i \rightarrow_p$.

$$i \leftarrow_f \triangleq \{\phi : \phi \longrightarrow \boxed{i} \in G\} \qquad i \leftarrow_p \triangleq \{\boxed{j} : \boxed{j} \longrightarrow \boxed{i} \in G\}$$
$$i \rightarrow_f \triangleq \{\phi : \boxed{i} \longrightarrow \phi \in G\} \qquad i \rightarrow_p \triangleq \{\boxed{j} : \boxed{i} \longrightarrow \boxed{j} \in G\}$$

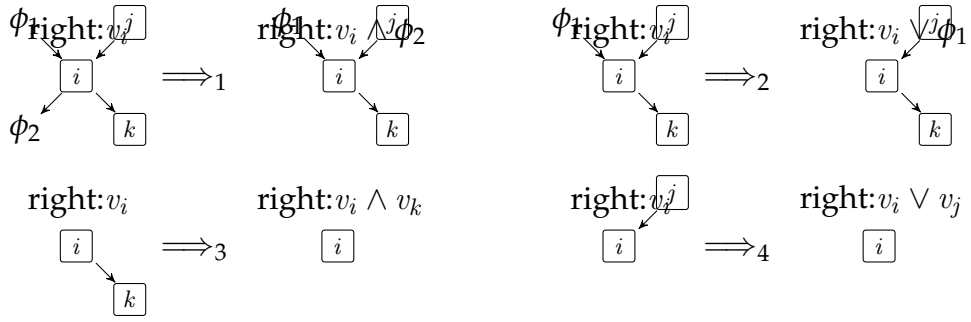
Initially, the values of all placeholder nodes are undefined, that is, $v_i = \bullet$ for all identifiers $i \in PH$. The procedure performs four steps. First, it executes repeatedly all possible instances of step 1, then all possible instances of step 2, until no progress is possible. Then, it executes steps 3 and 4 repeatedly until no progress is possible.

[4] Although TLA$^+$ supports the definition of recursive operators, our fragment does not currently handle them. In the presence of recursive functions, a fixed-point computation would be expected instead of our rudimentary procedure.

The steps are the following:

1. Pick $\boxed{i}$ s.t. $i{\rightarrow_f} \neq \{\}$. Then $v_i := v_i \wedge \bigwedge i{\rightarrow_f}$; $G := G \setminus i{\rightarrow_f}$.
2. Pick $\boxed{i}$ s.t. $i{\leftarrow_f} \neq \{\}$. Then $v_i := v_i \vee \bigvee i{\leftarrow_f}$; $G := G \setminus i{\leftarrow_f}$.
3. Pick $\boxed{i}$ s.t. $i{\rightarrow_p} \neq \{\}$ and $i{\leftarrow_p} = \{\}$. Then $v_i := v_i \wedge \bigwedge i{\rightarrow_p}$; $G := G \setminus i{\rightarrow_p}$.
4. Pick $\boxed{i}$ s.t. $i{\leftarrow_p} \neq \{\}$ and $i{\rightarrow_p} = \{\}$. Then $v_i := v_i \vee \bigvee i{\leftarrow_p}$; $G := G \setminus i{\leftarrow_p}$.

The following diagrams represent the execution of each of the four steps. Each diagram focuses on the node $\boxed{i}$ and its environment, in the more general scenario. To the right of node $\boxed{i}$ appears its value. Without loss of generality, we simplified the diagrams to one node of each kind, that is, either a formula or a placeholder, pointing in or from node $\boxed{i}$.



The first two steps take the values from formula nodes, and pass them to a placeholder node. Every step eliminates from the graph the evaluated edges. Step 1 assigns to node $\boxed{i}$ the conjunction of the formula nodes pointing to it. Step 2 assigns to node $\boxed{i}$ the disjunction of the formula nodes that $\boxed{i}$ points to. At this point, no edges to and from any formula belong to G. Steps 3 and 4 resolve the remaining implications between placeholder nodes in a similar way: they find placeholder nodes that have only incoming or outgoing placeholder nodes. Because there are no multiple paths between nodes, the four transformations cover all possible cases to reach $G = \{\}$.

Running example 5/5 In our example, the solution for the placeholder $\boxed{1}$ is trivially $z = 1 \vee z = 2 \vee z = 3$ in the context $z : \mathsf{Int}$. After replacing the placeholder by its solution, the type assignment is updated to $\sigma_3 \triangleq \sigma_1, x : \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\}$, and the constraint to

$$\mathcal{C}_4 \triangleq \Gamma_f, \Gamma_x \vdash \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\} \prec: \{z : \mathsf{Int} \mid \textsc{true}\}$$

where Γ_x is $x : \{z : \mathsf{Int} \mid z = 1 \vee z = 2 \vee z = 3\}$. The assignment σ_3 would be final if all the type-correctness conditions in the set $\mathcal{T}cc$ can be proved and the remaining constraint $\mathcal{C}_4$ can be reduced to true.

The only condition to prove in $\mathcal{T}cc$ is invalid. Indeed, it corresponds to checking the domain condition for the function application $f[0]$. Therefore, the typing for the proof obligation of our running example fails, as expected. (Actually, that condition could have been discharged to the solver before, which would have answered that was invalid, avoiding the successive steps of constraint solving.)

5.6. Tuples and records

Tuples in $\mathcal{T}_1$ When analyzing an expression such as $f[3]$, it makes sense to think that the sub-expression f should be a function (in the sense of $IsAFcn(f)$) for the whole expression to have the expected meaning, that is, the result of f applied to 3, if $3 \in \text{DOMAIN}\, f$. It also makes sense to think that the expression f could be a tuple, because the function application operator $\_[\_]$ as used for tuple projection, and the argument is an integer. TLA$^+$ tuples are functions, so it is not accurate to say that the operator $\_[\_]$ is "overloaded" for functions and tuples. Distinguishing between arbitrary functions and those functions used as tuples could give useful type information for improving the translation. However, it is not possible to discern just by a syntactic analysis to which of this two kind of expressions the first argument of $\_[\_]$ belongs to.

As a consequence, we are forced to assign a functional type to tuples if we want to be consistent with the TLA$^+$ syntax. (Remember the type inference rule T_1-App for the function application operator.) The type of a tuple has to be necessarily compatible with the type of a TLA$^+$ function, that is, a functional type $\tau_1 \to \tau_2$. If this was not the case, we could have introduced a dedicated type for tuples as we do below for records. In order to match the type operators dom and cod, it has to have a domain and a codomain.

The typing rules for tuples are the presented in Figure 5.7. Since the domain of a tuple of arity n is the interval $1\,..\,n$, its type is over-approximated to Int. Thus, the task to check the domain conditions are left to the solvers. Functional types impose another restriction: the elements of a tuple should be of the same type. The empty tuple is the only function whose domain is the empty set, and it is treated as a special case, in a similar way to the empty set.

Records in $\mathcal{T}_1$ TLA$^+$ records are functions whose domain is a set of strings, and whose return values are of any type. In our typing discipline, functions are constrained to have the same type for every element in its codomain. By giving a functional type to a record, we would be discarding many records that have different kind of values in each field. What we do instead is to introduce a new dedicated

$$\frac{}{\Gamma \vdash \langle\rangle : \alpha \to \alpha}\ \text{T}_1\text{-EmptyTuple} \qquad \frac{\Gamma \vdash e_i : \tau_i \qquad \tau_1 \cong \ldots \cong \tau_n \qquad {}^{i \in 1..n}}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \mathsf{Int} \to \tau_1}\ \text{T}_1\text{-Tuple}$$

$$\frac{\Gamma \vdash S_i : \mathsf{Set}\,\tau_i \qquad \tau_1 \cong \ldots \cong \tau_n \qquad {}^{i \in 1..n}}{\Gamma \vdash S_1 \times \ldots \times S_n : \mathsf{Set}\,(\mathsf{Int} \to \tau_i)}\ \text{T}_1\text{-Product}$$

$$\frac{\Gamma \vdash e_i : \tau_i \qquad {}^{i \in 1..n}}{\Gamma \vdash [h_i \mapsto e_i]_{1..n} : \mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n}}\ \text{T}_1\text{-Rec} \qquad \frac{\Gamma \vdash r : \tau}{\Gamma \vdash r.h : \mathsf{dot}(\tau, h)}\ \text{T}_1\text{-Dot}$$

$$\frac{\Gamma \vdash S_i : \mathsf{Set}\,\tau_i \qquad {}^{i \in 1..n}}{\Gamma \vdash [h_i : S_i]_{1..n} : \mathsf{Set}\,(\mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n})}\ \text{T}_1\text{-RecSet}$$

Figure 5.7.: $\mathcal{T}_1$-typing rules for tuples and records

type Map that mimics records by mapping strings to some other type. The disadvantage in this case is that records appearing in a proof obligation explicitly as functions will not be captured by this type discipline.

$$\tau ::= \ldots \mid \mathsf{Map}\,[s \mapsto \tau, \ldots, s \mapsto \tau] \mid \mathsf{dot}(\tau, s) \mid \mathsf{Str}$$

The type Map takes as argument a list of pairs of the form $h \mapsto \tau$ where the string h represents a field and τ its associated type. Its typing proposition is derived from the extensionality property for records (4.21):

$$\begin{aligned}[\![r : \mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}]\!] \triangleq\ & \wedge\ r = [x \in \textsc{domain}\ r \mapsto r[x]] \\ & \wedge\ \textsc{domain}\ r = \{h_1, \ldots, h_n\} \\ & \wedge\ [\![r.h_1 : \tau_1]\!] \wedge \ldots \wedge [\![r.h_n : \tau_n]\!]\end{aligned}$$

Additionally, we introduce an atomic type Str in $\mathcal{S}$ representing strings. As we did for the encoding of strings, we ignore the fact that strings are actually a sequence of characters.

We use the following abbreviations for records, record sets, Map types, and enumeration types:

$$\begin{aligned}[h_i \mapsto e_i]_{1..n} &\triangleq [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \\ [h_i : S_i]_{1..n} &\triangleq [h_1 : S_1, \ldots, h_n : S_n] \\ \mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n} &\triangleq \mathsf{Map}\,[h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n] \\ \{e_1, \ldots, e_n\}_\tau &\triangleq \{x : \tau \mid x = e_1 \vee \ldots \vee x = e_n\}\end{aligned}$$

The Map type enforces the expression r on $r.h$ to have a record type. The record selection operator $r.h$ has a special type $\mathsf{dot}(\tau, h)$, representing an explicit record

$$\frac{}{\Gamma \vdash \langle\rangle : (i : \{x : \alpha \mid \textsc{false}\}) \to \{x : \alpha \mid \textsc{false}\}}\ \text{T}_2\text{-EmptyTuple}$$

$$\frac{\Gamma \vdash e_i : \alpha_i \quad {\scriptstyle i \in 1..n}}{\Gamma \vdash \langle e_1, \ldots, e_n\rangle : (i : \{1, \ldots, n\}_{\mathsf{Int}}) \to \{x : \alpha_1 \mid i = 1 \wedge x = e_1\} \uplus \ldots \uplus \{x : \alpha_n \mid i = n \wedge x = e_n\}}\ \text{T}_2\text{-Tuple}$$

$$\frac{\Gamma \vdash S_i : \mathsf{Set}\,\alpha_i \quad {\scriptstyle i \in 1..n}}{\Gamma \vdash S_1 \times \ldots \times S_n : \mathsf{Set}\,((i : \{1, \ldots, n\}_{\mathsf{Int}}) \to \{x : \alpha_1 \mid i = 1 \wedge x = e_1\} \uplus \ldots \uplus \{x : \alpha_n \mid i = n \wedge x = e_n\})}\ \text{T}_2\text{-Product}$$

Figure 5.8.: $\mathcal{T}_2$-typing rules for tuples

projection. The reason for introducing this special type is that our constraint language cannot express pattern matching on a record type, which has an indefinite number of arguments. They have the the following properties, which are applied whenever possible to simplify types:

$$\begin{aligned} \mathsf{dot}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n}], h) &= \tau_i \quad \text{when } h_i = h \\ \mathsf{dom}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n}) &= \mathsf{Str} \\ \mathsf{cod}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{1..n}) &= \tau_1 \end{aligned}$$

Two Maps are equal iff if (i) they have the same domain and (ii) for each field h_i of the first map coinciding with a field h_j of the second map, the corresponding types τ_i and τ'_j are equal.

$$\frac{\{h_1,\ldots,h_m\}=\{h'_1,\ldots,h'_n\} \qquad \Gamma \vdash \tau_i \cong \tau'_j \ \ (\text{if } h_i=h'_j \text{ for } i\in 1..n \wedge j\in 1..m)}{\Gamma \vdash \mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n} \cong \mathsf{Map}\,[h'_i \mapsto \tau'_i]_{i:1\,..\,m}}\ \text{Eq-Map}$$

Tuples and records in $\mathcal{T}_2$ With the help of dependent and refinement types, we can precisely represent the domain and codomain of a tuple. The typing rules are shown in Figure 5.8, and they are basically an upgrade of the rules for $\mathcal{T}_1$.

The typing rules for records in $\mathcal{T}_2$ are the same as in $\mathcal{T}_1$, except that the domain and codomain properties of a Map have to be updated:

$$\begin{aligned} \mathsf{dom}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) &= \{h_1, \ldots, h_n\}_{\mathsf{Str}} \\ \mathsf{cod}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) &= \tau_1 \uplus \ldots \uplus \tau_n \end{aligned}$$

Similarly to the above rule for Map-equality, we define the rule for Map-subtype.

$$\frac{\{h_1,\ldots,h_m\}=\{h'_1,\ldots,h'_n\} \qquad \Gamma \vdash \tau_i \prec: \tau'_j \;\; (\text{if } h_i=h'_j \text{ for } i\in 1..n \wedge j\in 1..m)}{\Gamma \vdash \mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n} \prec: \mathsf{Map}\,[h'_i \mapsto \tau'_i]_{i:1\,..\,m}}\ \textsc{Sub-Map}$$

5.7. Related work

Type systems and how they are implemented varies a lot from one specification language to another [Wie06]. In formalisms based in high-order logic or type theory (e.g. in HOL Light, Isabelle/HOL, PVS, Coq, Nuprl, or Twelf), types are part of the foundation of the language. Then there are the languages that have "soft" types as a layer on top of an untyped foundation (e.g. ACL2, B method, Z, or Mizar). We can see that, in practice, types are ubiquitous for almost all formal languages.[5] The exceptions are set theory-based Isabelle/ZF, Metamath, and TLA$^+$.

Soft type systems In the Z [Spi92] and B [Abr10] formalisms, terms and functions have monomorphic types that have a natural correspondence to many-sorted logic. B's type system is studied by Turner [Tur01] on the MacLane set theory fragment. Likewise, Mizar [lM93] implements a type system with dependent types on top of its set-theoretical foundation. Wiedijk [Wie07] describes Mizar's type system as a collection of type inference rules where types are interpreted by relativizing quantifiers and encoding other type-features of the language in unsorted first-order logic.

These kind of type systems have type annotations that can be encoded in the language's foundational logic. Wiedijk [Wie07] calls them *soft type systems*. This term, in turn, originates from the programming languages community [CF91] as an attempt to combine the advantages of untyped and typed programming languages. In this context, it means that expressions are statically typed as much as possible and, for what cannot be statically typed, run-time checks are inserted. Our approach for TLA$^+$ is similar to the first definition in the sense that we add types annotations on top of the language, and to the second in that type "errors" are encoded as first-order dynamic checks in the translation. The difference with Z, B or Mizar is that types are not part of the language's syntax; the implementation of our type systems run under the hood of the ATP/SMT backend.

Refinement types Freeman and Pfenning [FP91] originally defined refinement types to be applied to object-oriented features of programming languages. They define

[5] Type systems for untyped programming languages, such as scripting languages, have been studied [THF10a, THF10b], but we do not cover these cases in this short survey.

them in the same way as we do, that is, as subsets of pre-existing inductive definitions, but allowing base types to be refined by Boolean expressions. Dependent ML [XP99] implements this general form of refinement type, with refinement predicates restricted so as to make type checking decidable. In programming languages, refinement types extends the benefits of static type-checking while providing useful information for compilers. Hybrid type checking [Fla06] reintroduce the original concept of soft typing [CF91] but for refinement types. In this context, Knowles [KF07] separates type reconstruction for refinement types from type-checking, making type reconstruction decidable, while deferring (undecidable) type-checking for a subsequent phase. SMT solvers are employed to check the validity of refinements but not for subtyping, which is checked by traditional syntactic techniques.

The PVS [ORSSC99] system is a LCF-style interactive theorem prover founded on high-order logic and augmented with dependent types and predicate subtypes [ROS98], the name used in PVS for refinement types. Predicates subtypes are part of PVS's foundational logic. The term *type-correctness condition* for subtype constraints originates in PVS. Unlike our soft-typing-with-SMT-solvers approach, if the type-correctness conditions cannot be proved by automatic procedures, they need to be proved interactively, while conventional type-checking is performed algorithmically. Refinement types are known also by subset types in Coq [Soz07].

Lamport and Paulson [LP99] analyze a series of problems for implementing predicate subtypes for specification languages like in PVS. First, they consider type inference over complete specifications, which can be difficult to achieve when subformulas are structured in many definitions. Instead, we apply the type synthesis algorithm to one formula (proof obligation) at a time. Secondly, they claim that predicate subtypes would lead the users to encode program invariants inside the refinement predicates. In our approach, types are transparent for the user, and if type construction is not possible, we fall back to the untyped encoding, maintaining the flexibility of untyped formalisms. Only recently the SMT technology has made possibly to resolve subtyping constraints efficiently and without user intervention. For instance, Bierman et al. [BGHL10] study a first-order functional programming language that combines refinement types with type-tests (a Boolean expression testing whether a value belongs to a type), relying on SMT solvers for subtyping.

Constraint solving Many type systems extend the standard Hindley-Milner type inference system with constraints, e.g. [OSW97, PR05]. Pottier [Pot96] and Odersky et al. [OSW97] have developed Hindley-Milner systems parameterized by a subtyping constraint system. Our constraint solving algorithm and parts of the notation was inspired from Pottier [PR05], as a non-deterministic system of constraint rewriting rules and first-order unification. Our algorithm for the placeholder solution was inspired from Knowles and Flanagan [KF07]. Constraint-based inference for type systems with subtyping is an extensive research topic [JK91] and we do not

fully review it here. Many optimizations to subtyping constraints can be performed, see [Pot96].

Other related work The SPASS [Wei99] theorem prover for untyped first-order logic implements special inference rules in its calculus that infer sort information in order to detect ill-sorted clauses and to simplify the sort information contained in the clauses. Sorts are considered monadic predicates without any further structure.

Claessen et al. [CLS11] propose an encoding of unsorted into many-sorted first-order logic (and vice-versa) using sort-monotonicity. A sort is *monotone* if its domain for any model can be made larger without affecting satisfiability. For the unsorted-to-sorted case, an algorithm computes the maximal typing of each variable by assigning to every variable a unique sort and putting them in equivalence classes when they should be equal to each other. If the resulting sorts are monotone, the original and the sorted formula are equisatisfiable. Otherwise, for non-monotone sorts, the algorithm constraints the sorted formula by adding an injection from smaller to bigger sorts.

Power types for programming languages, introduced by Cardelli [Car88], and developed by Aspinall [Asp00] are comparable to our Set type constructor. The idea is that $\mathsf{Power}(\tau)$ is a type whose elements are subtypes of the type τ. Subtyping is defined as type-inhabitation of power types: $\tau_1 <: \tau_2 \stackrel{\Delta}{=} \tau_1 : \mathsf{Power}(\tau_2)$.

Chapter 6.

Evaluation

> Ce n'est pas une démonstration proprement dite, [...] c'est une vérification. [...] La vérification diffère precisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile.
>
> Henri Poincaré, 1902, talking about a proof of $2 + 2 = 4$

In this chapter we perform an empirical evaluation of the methods presented in Chapters 4 and 5. We implemented a new generic back-end in TLAPS for provers supporting the input formats TPTP/FOF and SMT-LIB/AUFLIA. To validate our approach, we reproved several test cases that had been proved interactively using the previously available TLAPS back-end provers, namely Zenon, Isabelle/TLA$^+$ and a decision procedure for Presburger arithmetic. We will refer to the combination of those three backends as ZIP for short.

For each benchmark, we compare two dimensions of an interactive proof: size and time. We define the *size* of an interactive proof as the number of non-trivial proof obligations generated by the Proof Manager.[1] The proof size then corresponds to the number of leaf steps that are passed to the back-end provers as proof obligations. This number is proportional to the number of interactive steps and therefore represents the user effort for making TLAPS check the proof. The *time* is the number of seconds required by the Proof Manager to verify those proofs on a standard laptop.[2]

[1] Trivial proof obligations correspond to facts manually added to the proof steps. The Proof Manager needs to check in the corresponding proof context that those facts are actually given hypotheses.

[2] The experiments presented here were carried on with a 2.2GHz Intel Core i7, with 8GB RAM.

TLAPS uses short timeouts for automatic backends, hence running times are not very significant for comparison.

In the following, we present results for four case studies. In the result tables, each line corresponds to an interactive proof of a given size. In the columns we show the running times for the ATP/SMT backend using a given automated prover, where each prover is executed on all generated proof obligations. For our tests we have used the state-of-the-art ATP systems E [Sch13] and SPASS [Wei99], and the SMT solvers CVC4 [BCD+11] and Z3 [dB08].[3] In turn, for each prover we present three different times corresponding to the untyped encoding (the columns labeled u), the encoding using the type system $\mathcal{T}_1$ (labeled t_1), and the encoding using the type system $\mathcal{T}_2$ (labeled t_2). In the last two cases the time required for type construction is included in the general time. The backends were executed with a timeout of 300 seconds. In the tables, an entry with the symbol "-" means that the solver has reached the timeout without finding the proof for at least one of the proof obligations.

Peterson In Chapter 2, we used Peterson's algorithm [Pet81] as a running example. We compare in the following table the times for two interactive proofs that the algorithm implements mutual exclusion.

	ZIP	E			SPASS			CVC4			Z3		
size		u	t_1	t_2	u	t_1	t_2	u	t_1	t_2	u	t_1	t_2
3	-	63.32	63.87	49.64	7.55	7.59	3.16	0.41	0.45	0.46	0.34	0.38	0.40
10	5.69	4.22	4.16	1.65	44.64	43.99	1.28	0.78	0.91	0.96	0.80	0.91	0.97

The proof of size 10 is the same proof shown in Figure 2.4. The older back-end provers can check this proof in less than 6 seconds. The inductive step (the step labeled $\langle 1 \rangle 2$ in the figure) is the larger one. The new back-ends can prove it in a single step instead of 8, reducing to total proof size to 3.

The running times for the SMT solvers are very low, as expected. The comparatively higher times of E and SPASS are caused by one sub-step ($\langle 2 \rangle 1$), whose corresponding proof obligation does not involve anything more than reasoning about the structure of the specification at a high level, something that the tableau calculus of Zenon is good at. When the operator definitions in this step are expanded the proof obligation becomes a large formula. With some tweaking to the configuration of E and SPASS, this proof step could probably be discharged much faster.

We observe also that the type system $\mathcal{T}_2$ helps to reduce the verification time in the ATP cases. Since these proofs do not require any arithmetic reasoning, the type

[3] We have used the E prover version 1.7 executed with the parameters `--tstp-format --auto`; SPASS 3.5 with `-TPTP -Auto`; CVC4 1.3 with `--lang=smt2 --pre-skolem-quant`; and Z3 4.3.0 with `-smt2 AUTO_CONFIG=false PULL_NESTED_QUANTIFIERS=true MBQI=true CNF_MODE=1`. All versions are publicly available.

system $\mathcal{T}_1$ does not improve the performance with respect to the untyped encoding. The reason is that the type system $\mathcal{T}_2$ causes a simpler encoding of functions by not including domain checkings.

Bakery The well-known Bakery algorithm [Lam74] again solves the problem of mutual exclusion. Moreover, the specification of this algorithm is parameterized by a number of n processes, and data structures are represented by functions ranging over the processes. In this simplified version of the algorithm called Atomic Bakery, reading and writing of a process' number is considered an atomic operation. The specification of the algorithm uses some simple arithmetic, basically to increment the "ticket numbers" given at the "bakery", and to compare those numbers. For this reason, the ATP systems, which cannot handle all the proof obligations, are not included in the results.

	ZIP	CVC4			Z3			CVC4+Z3		
size		u	t_1	t_2	u	t_1	t_2	u	t_1	t_2
16	-	-	9.59	6.57	-	13.09	7.15	-	6.54	5.75
223	52.74									

The original proof for mutual exclusion generates 223 non-trivial proof obligations, taking the older backends almost one minute to prove them. The reduced interactive proof produces much better running times. In the third column of the table we show the results for a combination of both SMT solvers, which invokes at each obligation the faster solver. The untyped encoding does not scale for this example, although some obligations that do not involve arithmetic can be proved, but it times out for the complete proof. Again, $\mathcal{T}_2$ is noticeably better than $\mathcal{T}_1$ because of the simplifications to the domain checkings.

Memoir The Memoir [PLD+11] security architecture is specified at three abstraction levels, where the level-1 specification refines the higher-level specification. We consider three test cases: the inductive proof of the type-correctness invariant at the level-1 specification, the proof that the level-1 initial state refines the high-level initial state, and the proof that one of the level-1 actions implements the high-level specification under a defined refinement.[4] Respectively, we call T, I and A the benchmarks for these proofs.

[4]Specifically, the theorems are $LL1TypeInvariant \land [LL1Next]_{LL1Vars} \Rightarrow LL1TypeInvariant'$, $LL1Init \land LL1Refinement \land LL1TypeInvariant \land CorrectnessInvariants \Rightarrow HLInit$, and $LL1PerformOperation \Rightarrow HLAdvanceService$.

		ZIP	E			SPASS			CVC4			Z3		
	size		u	t_1	t_2	u	t_1	t_2	u	t_1	t_2	u	t_1	t_2
T	1	-	-	-	-	39.72	42.76	2.03	-	-	-	1.99	2.34	1.53
T	12	-	-	-	3.63	9.43	10.32	3.43	3.11	4.53	3.46	3.21	4.49	3.51
T	424	7.31												
I	2	-	1.52	2.11	2.14	1.51	2.06	2.13	-	-	-	-	-	-
I	8	-	3.95	5.54	5.90	4.11	5.53	5.80	3.84	5.65	5.79	9.35	10.68	10.23
I	61	8.20												
A	6	-	7.78	11.19	7.93	12.86	16.65	7.80	-	-	-	-	-	-
A	27	-	9.96	15.15	14.42	9.99	16.27	14.32	11.31	17.64	14.36	11.46	17.78	14.30
A	126	19.10												

The type-invariant proof can be reduced from the original 424 proof obligations to just one, which can be proved in about two seconds with Z3 or with SPASS in the $\mathcal{T}_2$-encoding. For the interactive proof of size 12, all the new backends can discharge the obligations before the timeout (but E can do so only when $\mathcal{T}_2$ is used).

The Memoir specification does not make any use of arithmetic or any sorted theory. Therefore, the FOF and SMT-LIB files resulting from the untyped encoding and the type system $\mathcal{T}_1$ are practically the same. The overhead times of the latter are just due to proof construction, which affects similarly the type system $\mathcal{T}_2$. However, most data structures in the specification are represented with records. The type system $\mathcal{T}_2$ allows optimizations in the encoding by removing the domain checking for records, thus reducing the proof search time in many cases with respect to $\mathcal{T}_1$.

In general, we observed that the ATP systems perform comparatively better than the SMT solvers, particularly in the two benchmarks for the refinement proofs. A possible reason is that these proof obligations are large first-order formulas, with variables and operators ranging over one sort, something that ATP systems excel at.

Finite sets The following table shows results for a collection of theorems about the cardinality of finite sets. For instance, the benchmark called CardinalityOne corresponds to the proofs shown in Section 1.1. These theorems are straightforward but include set of sets and many universally and existentially quantified formulas that make the SMT solvers lag behind. Moreover, they are mixed with some elementary linear arithmetic. Therefore, the ATP systems cannot participate in these benchmarks. For each entry, we compare the original proof with a shorter one checked by a combination of Zenon with CVC4 and Z3.

	ZIP		Zenon+SMT			
	size		size	u	t_1	t_2
CardinalityZero	11	5.42	5	0.48	0.48	0.48
CardinalityPlusOne	39	5.35	3	0.49	0.48	0.52
CardinalityOne	6	5.36	1	0.35	0.35	0.35
CardinalityOneConverse	9	0.63	2	0.35	0.36	0.36
FiniteSubset	62	7.16	19	-	5.78	5.77
PigeonHole	42	7.07	20	7.01	7.24	7.22
CardinalityMinusOne	11	5.44	5	0.75	0.75	0.73

Zenon is the only prover that can handle some of the proof obligations in one step. Like in the sub-step $\langle 2 \rangle 1$ in Peterson's algorithm proof, the harder steps are about large, high-level structural formulas that make the SMT solvers get lost in the proof search generating quantifier instances. On the other side, the SMT solvers permit to prove non-trivial arithmetic formulas, thus reducing the proof size of many sub-steps. For example, Z3 is the only one that can prove CardinalityOne in one step.

The shorter interactive proofs are almost instantly checked. Except for FiniteSubset where the untyped encoding fails for one the proof obligations, we practically do not observe differences in the Zenon+SMT times. Also, these benchmarks almost do not involve functions, therefore we do not get much profit from the system $\mathcal{T}_2$ with respect to $\mathcal{T}_1$, and $\mathcal{T}_1$ does not improve the untyped encoding.

In these benchmarks we clearly see one of the problems of our encoding, which is how the user-defined operators are declared in SMT-LIB. For instance, $Cardinality$ is defined axiomatically in a TLA$^+$ standard module, as shown in page 1.1. The expression $Cardinality(S)$ denotes an integer value provided that S is a finite set. Therefore, we cannot simply declare $Cardinality$ with the SMT sort $\mathsf{U} \to \mathsf{Int}$, even if this would probably improve the backend performance. At the end of the next chapter we provide two possible solutions for the encoding of user-defined operators.

Discussion

In all our case studies we have observed that the use of the new backend leads to significant reductions in proof sizes and running times compared to the original interactive proofs. In particular, the "shallow" proofs of the first three case studies required only minimal interaction. We have also used the new ATP/SMT backend with good success on several examples not shown here.

The success of the ATP/SMT backend for these and similar benchmarks are mostly due to the fact that they can handle obligations that mix set theory, functions, and arithmetic. The original Isabelle and Zenon backends have very limited support for

arithmetic reasoning, while SimpleArithmetic handles only pure arithmetic formulas, requiring the user to decompose proof obligations until they fall within the respective fragments. Another important aspect of the new backend is that proof obligations are massaged and simplified by the preprocessing techniques before passing them to the solvers. From our experiments, we believe that this has a significant impact on the solvers' success.

We were pleasantly surprised to see that the four automated provers under evaluation offer complementary strengths. Both SMT solvers offer similar results, with Z3 being better at reasoning about arithmetic. In a few cases, CVC4 is faster or even proves obligations on which Z3 fails. In many proof obligations restricted to the equational first-order domain, the ATP systems outperform the SMT solvers, Zenon and Isabelle/TLA$^+$. Even the SMT solvers are in general faster than Zenon for formulas that Zenon could handle, except for the structural proof cases mentioned above.

The ATP/SMT backend equipped with one of the two type systems clearly benefits from the obtained type information to improve the encoding. In all the tests, we observed that if the untyped encoding succeeds, the typed encodings succeeds as well. Type information is necessary for most formulas involving non-trivial arithmetic. When type information is not needed, the type construction algorithm incurs in a slightly noticeable time overhead for large formulas. Our prototypical implementation of this algorithm may surely benefit from optimizations. In all our tests, the theorems were known a priori to be valid. Type synthesis succeeded for all cases, so no dynamic domain checkings were needed in the encoding. The generated type-verification conditions were all discharged as trivial.

Finally, we also observed that, for some proof obligations, the chances of the ATP systems and SMT solvers for finding a proof highly depends on the order in which the formulas are presented to the solvers, as it is expected. For instance, we found that in general it is better in the FOF and SMT-LIB files to place first the standard axioms, and additionally in the case of SMT-LIB, the conclusion before the hypotheses.

Chapter 7.

Conclusions

> A proof is a story; a computer-assisted proof is a story that's too long to be told in full, so you have to settle for the executive summary and a huge automated appendix.
>
> Ian Stewart, 2013

What we have presented in the preceding chapters is a sound and effective way of discharging TLA$^+$ proof obligations to automated theorem provers based on unsorted and many-sorted first-order logic. A new implemented ATP/SMT back-end verifier integrates external automated provers as oracles to the TLA$^+$ Proof System TLAPS.

The first main component of the backend is a generic translation framework that allows to plug to TLAPS any ATP system or SMT solver that supports the de facto standard formats TPTP/FOF or SMT-LIB constrained to the logic AUFLIA.[1]The second component consists of two type systems for TLA$^+$ and a type synthesis algorithm that, given a TLA$^+$ proof obligation, returns a typed-TLA$^+$ formula. The type information in the second formula is used to improve the encoding. The translation and the type synthesis algorithm provides the formal basis for implementing the ATP and SMT-based back-end prover.

[1]A translation to TPTP-TFF0 [SSCB12], which is very similar to SMT-LIB, could be easily accommodated within the same translation framework. Except for the final part of the encoding that has to take into account how to encode sorted theories, the rest is treated by the same translation algorithm.

Translation

Embedding set-theoretic concepts into first-order logic is often unnatural. In order to encode second-order expressions, such as those related to the axioms of set comprehension and replacement, the first part of the translation consists of a preprocessing and optimization phase relying on term-rewriting techniques coupled with an abstraction method to efficiently handle non-basic expressions. We derive the collection of rewriting rules from the TLA$^+$ semantics and proved them in Isabelle/TLA$^+$ to ensure their soundness. The rewriting rules, however, introduce many additional quantified formulas, which are difficult for the solvers to handle.

For non-*basic* expressions, that is, those expressions that cannot be reduced to first-order logic through rewriting, the abstraction method replaces them by a fresh Skolem function, and then adds a new hypothesis defining the new function symbol. Consequently, this method produces equi-satisfiable formulas but not equivalent formulas. The mechanism that combines term-rewriting with abstraction also enables the backends to successfully handle CHOOSE expressions (Hilbert's choice) and explicit TLA$^+$ function expressions corresponding to λ-abstractions. TLA$^+$ functions are total, but they have a domain on which the function's arguments are defined. In our encoding we treat functions as if they were partial functions, with undefined values encoded through an uninterpreted function.

The second part of the translation is a sound encoding of first-order TLA$^+$ formulas to unsorted and many-sorted first-order logic. At this point, the proof obligations were already massaged and reduced to first-order formulas in the previous phase. To address the problem of embedding an unsorted language such as TLA$^+$ into a sorted one, we have introduced a method to virtually delegate type inference to the solvers. When sorted theories are supported by the target language, the translation encodes sorted expressions such as arithmetic ones through an homomorphic embedding: an unsorted expression is "lifted" to a sorted counterpart through an injective function. The Boolification process makes possible the distinction between Boolean and non-Boolean expressions.

The soundness of the encoding is immediate: all the axioms about sets, functions, records, tuples, etc. are theorems in the background theory of TLA$^+$ that exist in the Isabelle encoding. The "lifting" axioms for the encoding of arithmetic assert that TLA$^+$ arithmetic coincides with SMT arithmetic over integers. Excluding the standard axiom of set extensionality, the translation is complete, meaning that a valid TLA$^+$ proof obligation, it produces a valid translation. We include only instances of extensionality for specific sets, function domains, and functions. In order to reason about set equality in its most general form, the user would have to manually add the axiom of extensionality to the proof. We have demonstrated that the intermediate steps of the preprocessing and the proper encoding of basic formulas are sound and that they terminate.

The resulting translation can handle a useful fragment of the TLA$^+$ language, including set theory, functions, linear arithmetic expressions, tuples, records, and the CHOOSE operator. Some restrictions apply on the well-formed expressions supported by the translation. For instance, after Boolification, the translation would reject silly expressions like $x \wedge 42$. A set like $\{x, y\}$ can be translated only when its components can be mapped to the same SMT sort. Similarly, functions, including tuples and records, are constrained to range over a unique sort in the target language. For instance, a function like $[x \in Int \mapsto \text{IF } c \text{ THEN } x \text{ ELSE FALSE}]$ would be rejected.

Our original goal was to attempt the integration of SMT solvers only. After a successful experimental implementation, we observed that we could additionally integrate ATP systems practically for free. But for that, sorted theories have to be ignored, that is, we have to leave the arithmetic operators unspecified in the translation. Being able to discharge proof obligations to ATP systems represents a convenient alternative to Zenon, the other purely first-order backend in TLAPS.

Type synthesis

Beyond the recurring debates about using typed versus untyped languages for formalizing mathematics or software systems [LP99], we observed that types, regarded just as a classification of the elements of a language, arise quite naturally in untyped set theory. Motivated by the integration of automatic provers based on multi-sorted logic, we defined two type systems for TLA$^+$. One of the advantages of the SMT approach to automated theorem proving is that the sorts of MS-FOL partition the universe of discourse, unlike in unsorted FOL. The type systems for TLA$^+$ aim at synthesizing the missing type information from untyped formulas to better classify the elements of a proof obligation. In a given proof obligation, types are synthesized from typing hypotheses, which often occur in the form of type-correctness invariants. When type synthesis succeeds, we obtain type annotations on top of an untyped specification language, getting the best of both the typed and untyped approaches.

The first type system for TLA$^+$, called $\mathcal{T}_1$, is based on elementary types, which are on a par with the sorts of many-sorted first-order logic. Even if this type system over-approximates the values of TLA$^+$ expressions, it results in a decidable algorithm for constructing types. A second, more sophisticated type system called $\mathcal{T}_2$, expands the previous one with dependent and refinement types, making possible to capture with precision the values and semantics of sets and functions. With this type system, domain checkings are performed during type construction, so they are no longer needed in the encoding. However, the type synthesis algorithm for this type system becomes undecidable. Formulas for which type inference fails are still translated according to the "untyped" encoding, and may thus be proved by the first-order solvers. We have proved that the type systems are sound, meaning that

the synthesized type annotations do not change the validity of the original TLA$^+$ expressions.

Introducing types to set theory is like shifting from one foundational paradigm to the other, in the sense that, inevitably, type systems constrain the fragment of accepted set-theoretic expressions. Nevertheless, we can profit from that and use the type systems for the detection of: (i) malformed or inconsistent expressions, that is, expressions that do not have a defined semantic value, e.g. $3 + \text{TRUE}$, (ii) occasionally useful expressions that, though semantically correct, cannot be translated to the target languages, e.g. $\{3, \text{TRUE}\}$, and (iii) expressions that do not have the expected meaning, e.g. $f[x]$ when $x \notin \text{DOMAIN } f$ (only in $\mathcal{T}_2$).

Empirical results

We consider that our initial goal of improving the automation capabilities of TLAPS has been successfully achieved. Encouraging results show that ATP systems and SMT solvers boost the interactive proving performance for the verification of both "shallow" TLA$^+$ proof obligations, and more involved formulas including, for instance, linear arithmetic expressions in the case of SMT solvers. Both the size of the interactive proof and the time required to find automatic proofs can be remarkably reduced with the new back-end prover. We consider the reduction in proof size to be more important, as it reflects the number of user interactions. Moreover, the automated provers under evaluation, either in the ATP or SMT flavors, complement each other very well.

Our experience with the implementation of the type construction algorithm on top of the ATP/SMT backend has been quite positive: types are successfully inferred for the vast majority of proof obligations that we have seen in practice. The decidable type system $\mathcal{T}_1$ can only add benefits to the untyped encoding, except at the expense of a slight time-overhead required for type construction. But when type synthesis is applicable, the "untyped" encoding can be greatly simplified by generating fewer quantifiers and simpler formulas, increasing the number of proof obligations that both the ATP and SMT-based backend can handle without human interaction. Since the type system $\mathcal{T}_2$ is a refinement of $\mathcal{T}_1$, $\mathcal{T}_2$ never fails when $\mathcal{T}_1$ succeeded, increasing even more the number of handled proof obligations. The improvements introduced by $\mathcal{T}_2$ are particularly noticeable in specifications that contain a significant number of function applications. These expressions occur frequently in TLA$^+$ specifications given that functions, tuples, and records are fundamental data structures.

Future Work

The translation

There are many ways in which the translation can be improved. First, the handled TLA$^+$ fragment can be extended to support real arithmetic and sequences. Nowadays, SMT solvers are providing increasing support for real arithmetic. It should not be difficult to extend the integer encoding to the domain of the reals. Sequences are frequently used in specifications of algorithms to represent many different data structures. They are not built-in expressions in the language, but they are defined axiomatically as functions in a TLA$^+$ module, just as the *Cardinality* operator that we mentioned before is defined for finite sets. Rewriting rules for sequences, or a more direct embedding, would be tied to the way they are defined in the *Sequences* module. The idea of improved typing hypotheses that we develop below may be a solution for a better encoding of expressions of these characteristics.

Another way to optimize the translation is to simplify the encoding of simple sets in the style of the Rodin plugins [KV12]: if no set of sets occur in the proof obligation, sets can be encoded simply through their characteristic predicates. An interesting theoretical question is whether the encoding techniques we have described for set theory, or even for full TLA$^+$, such as the rewriting and abstraction mechanism or the type system with refinements, can be integrated as preprocessing for superposition-based systems or inside the SMT architecture.

Proof certification

Perhaps the most important work left to be done is to complete the integration by making it one that we can fully trust. In this work we have taken the faithful approach to integration by treating the external provers as sound oracles, temporarily including them as part of TLAPS's trusted base. But not only the external tools may contain errors. The translation is complex and includes many transformation steps. A simple typo in a translation rule or in the implementation code may introduce a soundness bug. From our experience with the implementation of the backend, we know that the clockwork mechanisms of the integration are delicate piece of software which can be easily corrupted.

In future work, we intend to reconstruct the proof objects (along the lines presented in [AFG$^+$11] and [BBP13]) that many ATP systems and SMT solvers can produce, within Isabelle/TLA$^+$, the trusted kernel of TLAPS that faithfully encodes TLA$^+$. This would allow us to check the results of these solvers, as well as of the translation, in order to further raise our confidence in the ATP/SMT backend, just as currently TLAPS can direct Isabelle/TLA$^+$ to check proofs produced by Zenon. In order to reconstruct the ATP and SMT proofs in Isabelle/TLA$^+$ we would have to take into account not only the proofs generated by the solvers, but also all the steps performed

during the translation, such as rewriting, abstraction, and any other transformation. In Isabelle/HOL [BBP13], direct reconstruction of proofs produced by SMT solvers has been investigated and implemented for Z3. In Coq, proof certification was implemented [AFG+11] for veriT [BDODF09] restricted to the quantifier-free fragment.

Type systems and type synthesis

We believe that refinement types have a lot of potential for future improvements of the encoding. For instance, some forms of equality can be deduced from the types of their sub-expressions. It would also be interesting to study the applicability of this type system to proofs of mathematical theorems in ZF set theory. There is also a lot of room for improvement and optimizations in the constraint solving phase of type construction. Some work has been done for simplifying subtyping constraints [Pot96, TS96, SAN+02].

One advantage of doing type inference with constraints is that it is possible to know exactly which variables, operators, or any sub-expression cannot be typed in a proof obligation. Type variables are generated for each sub-expression of a TLA$^+$ formula, so it is possible to know to which point of the syntactic tree the type variables belong. We could restrict the use of the untyped encoding only to these parts of the proof obligation, and at the same time produce useful type checking warnings and error messages [HHS02, SSW06].

Typing hypotheses

A more sophisticated mechanism to obtain typing hypotheses would make type construction applicable to more expressions. For instance, the following lemma is usually required in proofs about the cardinality of finite sets.

> LEMMA $CardinalityInNat \triangleq \forall S : IsFiniteSet(S) \Rightarrow Cardinality \in Nat$
> BY $CardinalityAxiom$

The lemma asserts that the unary $Cardinality$ operator presented in page 19 denotes a natural number provided that its argument is a finite set. It can be trivially proved by expanding the definition of $CardinalityAxiom$. The reason for hiding behind a lemma the fact that $Cardinality$ is a natural number is in order to not expand $CardinalityAxiom$ in other theorems.

An idea for a possible solution is to obtain better type information about the operator from a typing hypothesis like the lemma $CardinalityInNat$. Using refinement types, we could assign to $Cardinality$ a type like $(S : \{x : \alpha \mid IsFiniteSet(x)\}) \to \{x : \mathsf{Int} \mid 0 \leq x\}$. The condition $IsFiniteSet(x)$ would by applied as part of subtype constraint solving during the type construction algorithm, and $Cardinality$ would be declared with the SMT sort $\mathsf{U} \to \mathsf{Int}$, instead of $\mathsf{U} \to \mathsf{U}$.

Alternatively, we can declare $Cardinality$ directly with the SMT sort $\mathsf{U} \to \mathsf{Int}$, and assert $\forall x \colon 0 \leq Cardinality(x)$. Then every expression in the proof obligation of the form $P(Cardinality(S))$ occurring negatively, where P is a predicate, should be replaced by $IsFiniteSet(S) \Rightarrow P(Cardinality(S))$.

These techniques would be also applicable to other user-defined operators, such as TLA$^+$ sequences. For instance, the standard operator Len represents the length of a sequence only when its argument is a member of $Seq(S)$, which is the set of sequences whose elements belong to the set S.

The type of CHOOSE

Assigning a type to the expression CHOOSE $x \colon P(x)$ is not a trivial task. Depending on the formula P, the variable x can denote any kind of value, from an arbitrary integer, a set, or a function to a Boolean expression. Even in the case of a bounded expression like CHOOSE $x \in S \colon P(x)$, or similarly, if we extract from P some typing hypothesis for x, the type for x cannot fully be determined from S or from the typing hypothesis. The semantics of CHOOSE say that x denotes a value satisfying P if the condition $\exists x \colon P(x)$ is valid in the context of the CHOOSE expression. Therefore, the type of x is conditioned to proving that the predicate P is inhabited. Moreover, the expression CHOOSE $x \colon$ FALSE in all its equivalent forms denotes a specific value that may deserve a specific type.

For all these reasons, the only possible solution in the type system $\mathcal{T}_1$ may be to assign the most general type to any CHOOSE expression, that is, a new type that is super-type of any other type. In the type system $\mathcal{T}_2$, we may assign to CHOOSE $x \colon P(x)$ a type of the form $\{x \colon \alpha \mid P(x)\}$, while verifying separately that there is a witness for P, that is, $\exists y \colon P(y)$ holds. If that is not the case, the type would be the special type for CHOOSE $x \colon$ FALSE. The problem is that this two cases cannot be expressed in our constraint language without expanding it. All these potential solutions have to be further studied.

Appendix A.

Rewriting rules

This appendix lists the collection of rewriting rules applied during the pre-processing phase of the translation (Section 4.4.1). This list is not comprehensive; some trivial rules are omitted. The expression $[h_i \mapsto e_i]_{i:1\,..\,n}$ abbreviates $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ and $[h_i : e_i]_{i:1\,..\,n}$ abbreviates $[h_1 : e_1, \ldots, h_n : e_n]$.

First-order logic

$$x \in \textsc{boolean} \overset{x:\mathsf{Bool}}{\longrightarrow} \textsc{true}$$

$$\begin{aligned}
\forall x : x \in \{e_1, \ldots, e_n\} \Rightarrow p(x) &\longrightarrow p(e_1) \wedge \ldots \wedge p(e_n) && (x \notin FV_{1..n}) \\
\exists x : x \in \{e_1, \ldots, e_n\} \wedge p(x) &\longrightarrow p(e_1) \vee \ldots \vee p(e_n) && (x \notin FV_{1..n}) \\
\forall x \in \{y \in S : q(y)\} : p(x) &\longrightarrow \forall x \in S : q(x) \Rightarrow p(x) \\
\exists x \in \{y \in S : q(y)\} : p(x) &\longrightarrow \exists x \in S : q(x) \wedge p(x)
\end{aligned}$$

where $FV_{1..n} = FV(e_1) \cup \ldots \cup FV(e_n))$.

Set theory

$$\begin{aligned}
x \in \{\} &\longrightarrow \textsc{false} & x \notin S &\longrightarrow \neg(x \in S) \\
x \in \{e_1, \ldots, e_n\} &\longrightarrow x = e_1 \vee \ldots \vee x = e_n & S \subseteq T &\longrightarrow \forall x : x \in S \Rightarrow x \in T \\
x \in \{y \in S : p(y)\} &\longrightarrow x \in S \wedge p(x) & x \in e_1 \cup e_2 &\longrightarrow x \in e_1 \vee x \in e_2 \\
S \in \textsc{subset}\ T &\longrightarrow \forall x : x \in S \Rightarrow x \in T & x \in e_1 \cap e_2 &\longrightarrow x \in e_1 \wedge x \in e_2 \\
x \in \textsc{union}\ S &\longrightarrow \exists T : T \in S \wedge x \in T & x \in e_1 \setminus e_2 &\longrightarrow x \in e_1 \wedge \neg(x \in e_2)
\end{aligned}$$

Instances of extensionality:

$$
\begin{aligned}
S = \{\} &\longrightarrow \forall x : \neg(x \in S) \\
S = \{e_1, \ldots, e_n\} &\longrightarrow \forall x : x \in S \Leftrightarrow x = e_1 \vee \ldots \vee x = e_n \\
S = \textsc{subset}\ T &\longrightarrow \forall x : x \in S \Leftrightarrow (\forall y : y \in x \Rightarrow y \in T) \\
S = \textsc{union}\ T &\longrightarrow \forall x : x \in S \Leftrightarrow (\exists y : y \in T \wedge x \in y) \\
S = \{x \in T : p(x)\} &\longrightarrow \forall x : x \in S \Leftrightarrow x \in T \wedge p(x) \\
S = \{e(y) : y \in T\} &\longrightarrow \forall x : x \in S \Leftrightarrow (\exists y : y \in T \wedge x = e(y)) \\
S = T \cup U &\longrightarrow \forall x : x \in S \Leftrightarrow x \in T \vee x \in U \\
S = T \cap U &\longrightarrow \forall x : x \in S \Leftrightarrow x \in T \wedge x \in U \\
S = T \setminus U &\longrightarrow \forall x : x \in S \Leftrightarrow x \in T \wedge \neg(x \in U) \\
\forall x : x \in S \Leftrightarrow x \in T &\longrightarrow S = T
\end{aligned}
$$

Functions

$$
\begin{aligned}
[x \in S \mapsto e(x)][a] &\longrightarrow \textsc{if}\ a \in S \\
&\quad\ \textsc{then}\ e(a)\ \textsc{else}\ \ \omega([x \in S \mapsto e(x)], a) \\
[f\ \textsc{except}\ ![x] = y][a] &\longrightarrow \textsc{if}\ a \in \textsc{domain}\ f \\
&\quad\ \textsc{then}\ (\textsc{if}\ x = a\ \textsc{then}\ y\ \textsc{else}\ \alpha(f, a)) \\
&\quad\ \textsc{else}\ \ \omega([f\ \textsc{except}\ ![x] = y], a) \\
\textsc{domain}\ [x \in S \mapsto e] &\longrightarrow S \\
\textsc{domain}\ [f\ \textsc{except}\ ![x] = y] &\longrightarrow \textsc{domain}\ f \\
f \in [S \to T] &\longrightarrow \wedge\ isAFcn(f) \\
&\quad\ \wedge\ \textsc{domain}\ f = S \\
&\quad\ \wedge\ \forall x \in S : \alpha(f, x) \in T \\
[g\ \textsc{except}\ [a] = b] \in [S \to T] &\longrightarrow \wedge\ isAFcn(g) \\
&\quad\ \wedge\ \textsc{domain}\ g = S \\
&\quad\ \wedge\ a \in S \\
&\quad\ \wedge\ b \in T \\
&\quad\ \wedge\ \forall x \in S \setminus \{a\} : \alpha(f, x) \in T \\
[x \in S' \mapsto e(x)] \in [S \to T] &\longrightarrow \wedge\ S' = S \\
&\quad\ \wedge\ \forall x \in S : e(x) \in T \\
isAFcn([x \in S \mapsto e]) &\longrightarrow \textsc{true} \\
isAFcn([f\ \textsc{except}\ ![x] = y]) &\longrightarrow \textsc{true}
\end{aligned}
$$

Instances of extensionality:

$$
\begin{aligned}
f = [x \in S \mapsto e(x)] &\longrightarrow \land\ isAFcn(f) \\
&\quad\ \land\ \textsc{domain}\ f = S \\
&\quad\ \land\ \forall x \in S : \alpha(f, x) = e(x) \\
f = [x \in S \mapsto e(x)] &\overset{e(x):\mathsf{Bool}}{\longrightarrow} \land\ isAFcn(f) \\
&\quad\ \land\ \textsc{domain}\ f = S \\
&\quad\ \land\ \forall x \in S : \alpha(f, x)^b \Leftrightarrow e(x) \\
g = [f\ \textsc{except}\ ![a] = b] &\longrightarrow \land\ isAFcn(g) \\
&\quad\ \land\ \textsc{domain}\ f = \textsc{domain}\ g \\
&\quad\ \land\ a \in \textsc{domain}\ g \Rightarrow \alpha(g, a) = b \\
&\quad\ \land\ \forall x \in \textsc{domain}\ f \setminus \{a\} : \alpha(f, x) = \alpha(g, x) \\
g = [f\ \textsc{except}\ ![a] = b] &\overset{b:\mathsf{Bool}}{\longrightarrow} \land\ isAFcn(g) \\
&\quad\ \land\ \textsc{domain}\ f = \textsc{domain}\ g \\
&\quad\ \land\ a \in \textsc{domain}\ g \Rightarrow \alpha(g, a)^b \Leftrightarrow b \\
&\quad\ \land\ \forall x \in \textsc{domain}\ f \setminus \{a\} : \alpha(g, x) = \alpha(f, x) \\
[x \in S \mapsto e(x)] = [x \in T \mapsto d(x)] &\longrightarrow S = T \land \forall x \in S : e(x) = d(x)
\end{aligned}
$$

Arithmetic

$$
\begin{aligned}
x \in e_1 \mathinner{..} e_2 &\longrightarrow x \in Int \land e_1 \leq x \land x \leq e_2 \qquad x + 0 \overset{x:\mathsf{Int}}{\longrightarrow} x \qquad x < x \overset{x:\mathsf{Int}}{\longrightarrow} \textsc{false} \\
x \in Int &\overset{x:\mathsf{Int}}{\longrightarrow} \textsc{true} \qquad\qquad\quad x - 0 \overset{x:\mathsf{Int}}{\longrightarrow} x \qquad x \leq x \overset{x:\mathsf{Int}}{\longrightarrow} \textsc{true}
\end{aligned}
$$

IF

$$
\begin{aligned}
\textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u &\overset{t,u:\mathsf{Bool}}{\longrightarrow} c \Rightarrow t \land \neg c \Rightarrow u \\
x \otimes \textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ f &\longrightarrow \textsc{if}\ c\ \textsc{then}\ x \otimes t\ \textsc{else}\ x \otimes f \\
f[\textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u] &\longrightarrow \textsc{if}\ c\ \textsc{then}\ f[t]\ \textsc{else}\ f[u] \\
O_1(\textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u) &\longrightarrow \textsc{if}\ c\ \textsc{then}\ O_1(t)\ \textsc{else}\ O_1(u)
\end{aligned}
$$

where x is a term, $\otimes$ is an infix binary TLA$^+$ operator such as $=$, $\in$, $\Rightarrow$, $\land$, $\Leftrightarrow$, $+$, or $<$, and O_1 is a prefix unary TLA$^+$ operator such as $\neg$, DOMAIN, SUBSET or UNION.

Tuples and records

$$\begin{aligned}
\langle e_1, \ldots, e_n \rangle[i] &\longrightarrow e_i \qquad \text{when } i \in 1 \,..\, n \\
t \in S_1 \times \ldots \times S_n &\longrightarrow \wedge\ isAFcn(t) \\
&\quad\ \wedge\ \textsc{domain}\ t = 1 \,..\, n \\
&\quad\ \wedge\ \alpha(t, 1) \in S_1 \wedge \ldots \wedge \alpha(t, n) \in S_n
\end{aligned}$$

$$\begin{aligned}
[h_i \mapsto e_i]_{i:1..n}.h_j &\longrightarrow e_j \qquad \text{when } j \in 1 \,..\, n \\
[r\ \textsc{except}\ !.h_1 = e].h_2 &\longrightarrow \textsc{if}\ \text{“}\mathsf{h_1}\text{”} = \text{“}\mathsf{h_2}\text{”}\ \textsc{then}\ e\ \textsc{else}\ r.h_2 \\
r.h &\longrightarrow r[\text{“}\mathsf{h}\text{”}] \\
r \in [h_i : S_i]_{i:1..n} &\longrightarrow \wedge\ isAFcn(r) \\
&\quad\ \wedge\ \textsc{domain}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
&\quad\ \wedge\ \alpha(r, \text{“}\mathsf{h_1}\text{”}) \in S_1 \wedge \ldots \wedge \alpha(r, \text{“}\mathsf{h_n}\text{”}) \in S_n \\
[h_i \mapsto e_i]_{i:1..n} \in [f_j : S_j]_{j:1..m} &\longrightarrow \wedge\ \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} = \{\text{“}\mathsf{f_1}\text{”}, \ldots, \text{“}\mathsf{f_m}\text{”}\} \\
&\quad\ \wedge\ \bigwedge e_i \in S_j \qquad \text{when } h_i = f_j, i \in 1 \,..\, n, j \in 1 \,..\, m
\end{aligned}$$

$$\begin{aligned}
\textsc{domain}\ \langle\rangle &\longrightarrow \{\} & \textsc{domain}\ [h_i \mapsto e_i]_{i:1..n} &\longrightarrow \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
\textsc{domain}\ \langle e_1, \ldots, e_n \rangle &\longrightarrow 1 \,..\, n & \textsc{domain}\ [r\ \textsc{except}\ !.h = e] &\longrightarrow \textsc{domain}\ r
\end{aligned}$$

Instances of extensionality:

$$
\begin{aligned}
t = \langle e_1, \ldots, e_n \rangle \longrightarrow\ & \wedge\ isAFcn(t) \\
& \wedge\ \textsc{domain}\ t = 1..n \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(t,i)^b \Leftrightarrow e_i \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(t,i) = e_i \\
T = S_1 \times \ldots \times S_n \longrightarrow\ & \forall x :\ x \in T \Leftrightarrow \wedge\ isAFcn(x) \\
& \qquad\qquad \wedge\ \textsc{domain}\ x = 1..n \\
& \qquad\qquad \wedge\ \alpha(x,1) \in S_1 \wedge \ldots \wedge \alpha(x,n) \in S_n \\
r = [h_i \mapsto e_i]_{i:1..n} \longrightarrow\ & \wedge\ isAFcn(r) \\
& \wedge\ \textsc{domain}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
& \wedge\ \text{“}\mathsf{h_1}\text{”} \in \textsc{domain}\ r \wedge \ldots \wedge \text{“}\mathsf{h_n}\text{”} \in \textsc{domain}\ r \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(r, \text{“}\mathsf{h_i}\text{”})^b \Leftrightarrow e_i \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(r, \text{“}\mathsf{h_i}\text{”}) = e_i \\
x = [y\ \textsc{except}\ !.h = e] \longrightarrow\ & \wedge\ isAFcn(x) \\
& \wedge\ \textsc{domain}\ x = \textsc{domain}\ y \\
& \wedge\ \text{“}\mathsf{h}\text{”} \in \textsc{domain}\ y \Rightarrow \alpha(x, \text{“}\mathsf{h}\text{”}) = e \\
& \wedge\ \forall k \in \textsc{domain}\ y \setminus \{\text{“}\mathsf{h}\text{”}\} :\ \alpha(x,k) = \alpha(y,k) \\
R = [h_i : S_i]_{i:1..n} \longrightarrow\ & \forall r :\ r \in R \Leftrightarrow \\
& \qquad \wedge\ isAFcn(r) \\
& \qquad \wedge\ \textsc{domain}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
& \qquad \wedge\ \text{“}\mathsf{h_1}\text{”} \in \textsc{domain}\ r \wedge \ldots \wedge \text{“}\mathsf{h_n}\text{”} \in \textsc{domain}\ r \\
& \qquad \wedge\ \alpha(r, \text{“}\mathsf{h_1}\text{”}) \in S_1 \wedge \cdots \wedge \alpha(r, \text{“}\mathsf{h_n}\text{”}) \in S_n
\end{aligned}
$$

Appendix B.

Types and constraint generation rules

This appendix lists the type grammar, the typing propositions, type properties, and the constraint generation rules for the systems $\mathcal{T}_1$ and $\mathcal{T}_2$.

Abbreviations

$$
\begin{aligned}
\exists \alpha_{1\,..\,n}.\, c &\triangleq \exists \alpha_1, \ldots, \alpha_n.\, c \\
\alpha_1 \cong \ldots \cong \alpha_n &\triangleq \alpha_1 \cong \alpha_2 \wedge \alpha_2 \cong \alpha_3 \wedge \ldots \wedge \alpha_{n-1} \cong \alpha_n \\
\langle\!\langle \Gamma \vdash e_i : \tau_i \rangle\!\rangle_{i:1\,..\,n} &\triangleq \langle\!\langle \Gamma \vdash e_1 : \tau_1 \rangle\!\rangle \wedge \ldots \wedge \langle\!\langle \Gamma \vdash e_n : \tau_n \rangle\!\rangle \\
\{e_1, \ldots, e_n\}_\tau &\triangleq \{x : \tau \mid x = e_1 \vee \ldots \vee x = e_n\} \\
[h_i \mapsto e_i]_{i:1\,..\,n} &\triangleq [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \\
[h_i : e_i]_{i:1\,..\,n} &\triangleq [h_1 : e_1, \ldots, h_n : e_n] \\
\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n} &\triangleq \mathsf{Map}\,[h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]
\end{aligned}
$$

Elementary types

Type grammar

$$
\begin{aligned}
\tau ::= \;& \mathsf{t}_1 \mid \mathsf{t}_2 \mid \ldots \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Str} \mid \mathsf{Set}\,\tau \mid \alpha \\
& \mid \tau \to \tau \mid \mathsf{dom}(\tau) \mid \mathsf{cod}(\tau) \\
& \mid \mathsf{Map}\,[s \mapsto \tau, \ldots, s \mapsto \tau] \mid \mathsf{dot}(\tau, s)
\end{aligned}
$$

Typing propositions

$$
\begin{aligned}
[\![e : \mathsf{t}]\!] &\triangleq t(e) \\
[\![e : \mathsf{Bool}]\!] &\triangleq e \in \textsc{boolean} \\
[\![e : \mathsf{Int}]\!] &\triangleq e \in \mathit{Int} \\
[\![e : \mathsf{Set}\,\tau]\!] &\triangleq \forall x \in e : [\![x : \tau]\!] \\
[\![e : \tau_1 \to \tau_2]\!] &\triangleq \wedge\ e = [x \in \textsc{domain}\ e \mapsto e[x]] \\
&\quad \wedge\ \forall x : x \in \textsc{domain}\ e \Leftrightarrow [\![x : \tau_1]\!] \\
&\quad \wedge\ \forall x : [\![x : \tau_1]\!] \Rightarrow [\![e[x] : \tau_2]\!] \\
[\![e : \mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}]\!] &\triangleq \wedge\ e = [x \in \textsc{domain}\ e \mapsto e[x]] \\
&\quad \wedge\ \textsc{domain}\ e = \{h_1, \ldots, h_n\} \\
&\quad \wedge\ [\![r.h_1 : \tau_1]\!] \wedge \ldots \wedge [\![r.h_n : \tau_n]\!]
\end{aligned}
$$

Type properties

$$
\begin{aligned}
\mathsf{dom}(\tau_1 \to \tau_2) &= \tau_1 \\
\mathsf{cod}(\tau_1 \to \tau_2) &= \tau_2 \\
\mathsf{dom}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) &= \mathsf{Str} \\
\mathsf{dot}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}, s) &= \tau_i \qquad \text{if } h_i = s
\end{aligned}
$$

First-order logic

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle &\triangleq \tau \equiv \Gamma(x) \\
\langle\!\langle \Gamma \vdash x^b : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash w(e_1, \ldots, e_n) : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1\,..\,n+1} \alpha'_{1\,..\,n}.\ \wedge\ \Gamma(w) \equiv \alpha_1 \to \ldots \to \alpha_n \to \alpha_{n+1} \\
&\qquad \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge\ \alpha_1 \cong \alpha'_1 \wedge \ldots \wedge \alpha_n \cong \alpha'_n \\
&\qquad \wedge\ \tau \equiv \alpha_{n+1} \\
\langle\!\langle \Gamma \vdash w(e_1, \ldots, e_n)^b : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1\,..\,n} \alpha'_{1\,..\,n}.\ \wedge\ \Gamma(w) \equiv \alpha_1 \to \ldots \to \alpha_n \to \mathsf{Bool} \\
&\qquad \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge\ \alpha_1 \cong \alpha'_1 \wedge \ldots \wedge \alpha_n \cong \alpha'_n \\
&\qquad \wedge\ \tau \equiv \mathsf{Bool}
\end{aligned}
$$

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash \textsc{true} : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash \textsc{false} : \tau \rangle\!\rangle \\
&\triangleq \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash e_1 \wedge e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e_1 \vee e_2 : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash e_1 \Rightarrow e_2 : \tau \rangle\!\rangle \\
&\triangleq \tau \equiv \mathsf{Bool} \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Bool} \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon \phi : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha.\, \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon x = e \Rightarrow \phi : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha.\, \langle\!\langle \Gamma \vdash e : \alpha \rangle\!\rangle \wedge \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon x \in e \Rightarrow \phi : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha.\, \langle\!\langle \Gamma \vdash e : \mathsf{Set}\,\alpha \rangle\!\rangle \wedge \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash e_1 = e_2 : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2 \alpha_3.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 \simeq \alpha_2
\end{aligned}
$$

Sets

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash e_1 \in e_2 : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 \simeq \alpha_2 \\
\langle\!\langle \Gamma \vdash \{\} : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \tau \equiv \mathsf{Set}\,\alpha \\
\langle\!\langle \Gamma \vdash \{e_1, \ldots, e_n\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \ldots \alpha_n.\, \wedge \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad\qquad \wedge \alpha_1 \simeq \ldots \simeq \alpha_n \\
&\qquad\qquad \wedge \tau \equiv \mathsf{Set}\,\alpha_1 \\
\langle\!\langle \Gamma \vdash \{x \in s : \phi\} : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
&\qquad\qquad \wedge \tau \equiv \mathsf{Set}\,\alpha \\
\langle\!\langle \Gamma \vdash \{e : x \in s\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma, x : \alpha_1 \vdash e : \alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \tau \equiv \mathsf{Set}\,\alpha_2 \\
\langle\!\langle \Gamma \vdash \textsc{subset}\ s : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha \rangle\!\rangle \wedge \tau \equiv \mathsf{Set}\,\mathsf{Set}\,\alpha \\
\langle\!\langle \Gamma \vdash \textsc{union}\ s : \tau \rangle\!\rangle &\triangleq \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\tau \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash e_1 \subseteq e_2 : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 \cong \alpha_2 \\
\langle\!\langle \Gamma \vdash e_1 \cup e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e_1 \cap e_2 : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash e_1 \setminus e_2 : \tau \rangle\!\rangle \\
&\triangleq \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 \cong \alpha_2 \\
&\qquad\qquad \wedge \tau \equiv \mathsf{Set}\,\alpha_1
\end{aligned}
$$

Functions

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash f[e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash f : \alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e : \alpha_2 \rangle\!\rangle \\
&\qquad \wedge \alpha_2 \cong \mathsf{dom}(\alpha_1) \\
&\qquad \wedge \tau \equiv \mathsf{cod}(\alpha_1) \\
\langle\!\langle \Gamma \vdash [x \in s \mapsto e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma, x : \alpha_1 \vdash e : \alpha_2 \rangle\!\rangle \\
&\qquad \wedge \tau \equiv \alpha_1 \to \alpha_2 \\
\langle\!\langle \Gamma \vdash \textsc{domain}\ f : \tau \rangle\!\rangle &\triangleq \exists \alpha.\ \langle\!\langle \Gamma \vdash f : \alpha \rangle\!\rangle \wedge \tau \equiv \mathsf{Set}\,(\mathsf{dom}(\alpha)) \\
\langle\!\langle \Gamma \vdash [s \to t] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash t : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\qquad \wedge \tau \equiv \mathsf{Set}\,(\alpha_1 \to \alpha_2) \\
\langle\!\langle \Gamma \vdash [f\ \textsc{except}\ ![a] = b] : \tau \rangle\!\rangle &\triangleq \exists \alpha_f.\ \wedge \langle\!\langle \Gamma \vdash f : \alpha_f \rangle\!\rangle \\
&\qquad \wedge \exists \alpha_a \alpha_b.\ \wedge \langle\!\langle \Gamma \vdash a : \alpha_a \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash b : \alpha_b \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_a \to \alpha_b \cong \alpha_f \\
&\qquad \wedge \tau \equiv \alpha_f
\end{aligned}
$$

Arithmetic

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash n : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Int} \qquad \text{for } n \in \mathit{Int} \\
\langle\!\langle \Gamma \vdash \mathit{Nat} : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Set}\,\mathsf{Int} \\
\langle\!\langle \Gamma \vdash \mathit{Int} : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Set}\,\mathsf{Int} \\
\langle\!\langle \Gamma \vdash -e : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Int} \wedge \exists \alpha.\ \langle\!\langle \Gamma \vdash e : \alpha \rangle\!\rangle \wedge \alpha \simeq \mathsf{Int} \\
\langle\!\langle \Gamma \vdash x + y : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash x - y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x * y : \tau \rangle\!\rangle \\
&\triangleq \tau \equiv \mathsf{Int} \wedge \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \alpha_1 \simeq \mathsf{Int} \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \alpha_2 \simeq \mathsf{Int} \\
\langle\!\langle \Gamma \vdash x < y : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash x \leq y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x > y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x \geq y : \tau \rangle\!\rangle \\
&\triangleq \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \alpha_1 \simeq \mathsf{Int} \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \alpha_2 \simeq \mathsf{Int} \\
\langle\!\langle \Gamma \vdash x \mathinner{..} y : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Set}\,\mathsf{Int} \wedge \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \alpha_1 \simeq \mathsf{Int} \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \alpha_2 \simeq \mathsf{Int}
\end{aligned}
$$

Tuples and records

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash \langle e_1, \ldots, e_n \rangle : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\ \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
&\quad\ \wedge\ \alpha_1 \cong \ldots \cong \alpha_n \\
&\quad\ \wedge\ \tau \equiv \mathsf{Int} \to \alpha_1 \\
\langle\!\langle \Gamma \vdash S_1 \times \ldots \times S_n : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\ \wedge\ \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\,\alpha_i \rangle\!\rangle_{i:1..n} \\
&\quad\ \wedge\ \alpha_1 \cong \ldots \cong \alpha_n \\
&\quad\ \wedge\ \tau \equiv \mathsf{Set}\,(\mathsf{Int} \to \alpha_i) \\
\langle\!\langle \Gamma \vdash [h_i \mapsto e_i]_{i:1..n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\ \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1..n} \wedge \tau \equiv \mathsf{Map}\,[h_i \mapsto \alpha_i]_{i:1..n} \\
\langle\!\langle \Gamma \vdash r.h : \tau \rangle\!\rangle_1 &\triangleq \exists \alpha.\ \langle\!\langle \Gamma \vdash r : \alpha \rangle\!\rangle \wedge \tau \equiv \mathsf{dot}(\alpha, h) \\
\langle\!\langle \Gamma \vdash [h_i : S_i]_{i:1..n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\ \wedge\ \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\,\alpha_i \rangle\!\rangle_{i:1..n} \\
&\quad\ \wedge\ \tau \equiv \mathsf{Set}\,(\mathsf{Map}\,[h_i \mapsto \alpha_i]_{i:1..n})
\end{aligned}
$$

Miscellaneous constructs

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash \text{“}abc\text{”} : \tau \rangle\!\rangle &\triangleq \tau \equiv \mathsf{Str} \\
\langle\!\langle \Gamma \vdash \textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u : \tau \rangle\!\rangle &\triangleq \exists \alpha_1\alpha_2\alpha_3\alpha_4.\ \wedge\ \langle\!\langle \Gamma \vdash c : \alpha_1 \rangle\!\rangle \wedge \alpha_1 \cong \mathsf{Bool} \\
&\qquad\qquad\ \wedge\ \langle\!\langle \Gamma \vdash t : \alpha_2 \rangle\!\rangle \\
&\qquad\qquad\ \wedge\ \langle\!\langle \Gamma \vdash u : \alpha_3 \rangle\!\rangle \\
&\qquad\qquad\ \wedge\ \alpha_2 \cong \alpha_3 \\
&\qquad\qquad\ \wedge\ \tau \equiv \alpha_3 \\
\langle\!\langle \Gamma \vdash \textsc{case}\ c_1 \to e_1\ : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}, \alpha'_{1..n}.\ \wedge\ \langle\!\langle \Gamma \vdash c_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
\square\ \ldots &\qquad\qquad\ \wedge\ \alpha_1 \cong \mathsf{Bool} \wedge \ldots \wedge \alpha_n \cong \mathsf{Bool} \\
\square\ c_n \to e_n &\qquad\qquad\ \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1..n} \\
&\qquad\qquad\ \wedge\ \alpha'_1 \cong \ldots \cong \alpha'_n \\
&\qquad\qquad\ \wedge\ \tau \equiv \alpha'_1 \\
\langle\!\langle \Gamma \vdash \textsc{case}\ c_1 \to e_1\ : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}, \alpha'_{1..n+1}.\ \wedge\ \langle\!\langle \Gamma \vdash c_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
\square\ \ldots &\qquad\qquad\ \wedge\ \alpha_1 \cong \mathsf{Bool} \wedge \ldots \wedge \alpha_n \cong \mathsf{Bool} \\
\square\ c_n \to e_n &\qquad\qquad\ \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1..n+1} \\
\square\ \textsc{other}\ \to e_{n+1} &\qquad\qquad\ \wedge\ \alpha'_1 \cong \ldots \cong \alpha'_{n+1} \\
&\qquad\qquad\ \wedge\ \tau \equiv \alpha'_1
\end{aligned}
$$

Dependent and refinement types

Type grammar

$$\begin{aligned}\tau ::= \;& \beta \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Str} \mid \alpha \cdot \theta \mid \mathsf{Set}\,\tau \\ \mid\; & (x:\tau) \to \tau \mid \mathsf{dom}(\tau) \mid \mathsf{cod}(\tau) \\ \mid\; & \mathsf{Map}\,[s \mapsto \tau, \ldots, s \mapsto \tau] \mid \mathsf{dot}(\tau, s) \\ \mid\; & \{x{:}\tau \mid \phi\} \mid \mathsf{base}(\tau) \mid \tau \uplus \tau \mid \tau \oplus \tau\end{aligned}$$

$$\theta ::= \; \square \mid [e], \theta \mid [x \leftarrow e], \theta$$

Additional typing propositions

$$\begin{aligned}[\![e : \{x:\tau \mid \phi\}]\!] &\triangleq [\![e:\tau]\!] \wedge \phi[x \leftarrow e] \\ [\![e : (x:\tau_1) \to \tau_2]\!] &\triangleq \wedge f = [x \in \textsc{domain}\ e \mapsto e[x]] \\ &\quad \wedge \forall z:\, z \in \textsc{domain}\ e \Leftrightarrow [\![z:\tau_1]\!] \\ &\quad \wedge \forall z:\, [\![z:\tau_1]\!] \Rightarrow (\forall x:\, [\![x:\tau_1]\!] \Rightarrow [\![e[z]:\tau_2]\!])\end{aligned}$$

Type properties

$$\begin{aligned}\{x:\{y:\tau \mid \phi_1\} \mid \phi_2\} &= \{x:\tau \mid \phi_1[y \leftarrow x] \wedge \phi_2\} \\ \{x:\tau \mid \textsc{true}\} &= \tau \\ \mathsf{dom}((x:\tau_1) \to \tau_2) &= \tau_1 \\ \mathsf{dom}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) &= \{h_1, \ldots, h_n\}_{\mathsf{Str}} \\ \mathsf{cod}((x:\tau_1) \to \tau_2) &= \tau_2\end{aligned}$$

$$\begin{aligned}\mathsf{base}(\{x:\tau \mid \phi\}) &= \tau \\ \mathsf{base}(\tau) &= \tau \qquad \text{if } \tau \in \{\mathsf{t}_i, \mathsf{Bool}, \mathsf{Int}\} \\ \mathsf{base}(\mathsf{Set}\,(\tau)) &= \mathsf{Set}\,(\mathsf{base}(\tau)) \\ \mathsf{base}((x:\tau_1) \to \tau_2) &= (x:\mathsf{base}(\tau_1)) \to \mathsf{base}(\tau_2) \\ \mathsf{base}(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) &= \mathsf{Map}\,[h_i \mapsto \mathsf{base}(\tau_i)]_{i:1\,..\,n} \\ \mathsf{base}(\theta \cdot \tau) &= \theta \cdot (\mathsf{base}(\tau))\end{aligned}$$

$$\begin{aligned}\{x:\tau_1 \mid \phi_1\} \uplus \{x:\tau_2 \mid \phi_2\} &= \{x:\tau_1 \mid \phi_1 \vee \phi_2\} && \text{if } \tau_1 = \tau_2 \\ \mathsf{Set}\,\tau_1 \uplus \mathsf{Set}\,\tau_2 &= \mathsf{Set}\,(\tau_1 \uplus \tau_2) && \\ ((x:\tau_1) \to \tau_2) \uplus ((x:\tau_1') \to \tau_2') &= (x:\tau_1) \to (\tau_2 \uplus \tau_2') && \text{if } \tau_1 = \tau_1' \\ \tau_1 \uplus \tau_2 &= \tau_1 && \text{if } \tau_1 = \tau_2 \\ (\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) \uplus (\mathsf{Map}\,[h_i' \mapsto \tau_i']_{i:1\,..\,m}) &= \mathsf{Map}\,[h_i \mapsto \tau_i \uplus \tau_i']_{i:1\,..\,n} && \text{if } n = m \text{ and} \\ & && \{h_i, \ldots\} = \{h_i', \ldots\}\end{aligned}$$

$$
\begin{array}{rcll}
\{x:\tau_1 \mid \phi_1\} \uplus \{x:\tau_2 \mid \phi_2\} & = & \{x:\tau_1 \mid \phi_1 \wedge \phi_2\} & \text{if } \tau_1 = \tau_2 \\
\mathsf{Set}\,\tau_1 \uplus \mathsf{Set}\,\tau_2 & = & \mathsf{Set}\,(\tau_1 \uplus \tau_2) & \\
((x:\tau_1) \to \tau_2) \uplus ((x:\tau_1') \to \tau_2') & = & (x:\tau_1) \to (\tau_2 \uplus \tau_2') & \text{if } \tau_1 = \tau_1' \\
\tau_1 \uplus \tau_2 & = & \tau_1 & \text{if } \tau_1 = \tau_2 \\
(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) \uplus (\mathsf{Map}\,[h_i' \mapsto \tau_i']_{i:1\,..\,m}) & = & \mathsf{Map}\,[h_i \mapsto \tau_i \uplus \tau_i']_{i:1\,..\,n} & \text{if } n = m \text{ and } \\
 & & & \{h_i, \ldots\} = \{h_i', \ldots\}
\end{array}
$$

$$
\begin{array}{rcll}
\{x':\tau \mid \phi\} \cdot [x \leftarrow e], \theta & \triangleq & \{x':\tau \mid \phi[x \leftarrow e]\} \cdot \theta & \text{if } x \neq x' \\
((x:\tau_1) \to \tau_2) \cdot [e], \theta & \triangleq & (x:\tau_1 \cdot [e], \theta) \to (\tau_2 \cdot [x \leftarrow e], \theta) & \text{if } x \notin dom(\theta) \\
(\mathsf{Set}\,\tau) \cdot \theta & \triangleq & \mathsf{Set}\,(\tau \cdot \theta) & \\
\beta \cdot \theta & \triangleq & \beta & \\
(\mathsf{Map}\,[h_i \mapsto \tau_i]_{i:1\,..\,n}) \cdot \theta & \triangleq & \mathsf{Map}\,[h_i \mapsto \tau_i \cdot \theta]_{i:1\,..\,n} & \\
\tau \cdot \Box & \triangleq & \tau & \\
(\tau \cdot \theta_1) \cdot \theta_2 & \triangleq & \tau \cdot (\theta_1, \theta_2) &
\end{array}
$$

First-order logic

$$
\begin{array}{rcl}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle & \triangleq & \emptyset \vdash \tau \equiv \Gamma(x) \\
\langle\!\langle \Gamma \vdash x^b : \tau \rangle\!\rangle & \triangleq & \emptyset \vdash \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash w(e_1, \ldots, e_n) : \tau \rangle\!\rangle & \triangleq & \exists \alpha_{1\,..\,(n+1)} \alpha'_{1\,..\,n}. \\
 & & \quad \wedge\ \emptyset \vdash \Gamma(w) \equiv \alpha_1 \to \ldots \to \alpha_n \to \alpha_{n+1} \\
 & & \quad \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha_i' \rangle\!\rangle_{i:1\,..\,n} \\
 & & \quad \wedge\ \Gamma \vdash \alpha_1' \prec: \alpha_1 \wedge \ldots \wedge \Gamma \vdash \alpha_n' \prec: \alpha_n \\
 & & \quad \wedge\ \emptyset \vdash \tau \equiv \alpha_{n+1} \\
\langle\!\langle \Gamma \vdash w(e_1, \ldots, e_n)^b : \tau \rangle\!\rangle & \triangleq & \exists \alpha_{1\,..\,n} \alpha'_{1\,..\,n}. \\
 & & \quad \wedge\ \emptyset \vdash \Gamma(w) \equiv \alpha_1 \to \ldots \to \alpha_n \to \mathsf{Bool} \\
 & & \quad \wedge\ \langle\!\langle \Gamma \vdash e_i : \alpha_i' \rangle\!\rangle_{i:1\,..\,n} \\
 & & \quad \wedge\ \Gamma \vdash \alpha_1' \prec: \alpha_1 \wedge \ldots \wedge \Gamma \vdash \alpha_n' \prec: \alpha_n \\
 & & \quad \wedge\ \emptyset \vdash \tau \equiv \mathsf{Bool}
\end{array}
$$

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash \textsc{true} : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash \textsc{false} : \tau \rangle\!\rangle \\
&\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash e_1 \wedge e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e_1 \vee e_2 : \tau \rangle\!\rangle \;=\; \langle\!\langle \Gamma \vdash e_1 \Rightarrow e_2 : \tau \rangle\!\rangle \\
&\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Bool} \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon \phi : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \wedge \exists \alpha.\, \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon x = e \Rightarrow \phi : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \wedge \exists \alpha.\, \wedge \langle\!\langle \Gamma \vdash e : \alpha \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \colon x \in e \Rightarrow \phi : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e : \mathsf{Set}\, \alpha_1 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_2 <: \alpha_1 \\
\langle\!\langle \Gamma \vdash e_1 = e_2 : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Bool} \wedge \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma \vdash e_2 : \alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 \prec: \alpha_1 \uplus \alpha_2 \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_2 \prec: \alpha_1 \uplus \alpha_2
\end{aligned}
$$

Sets

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash e_1 \in e_2 : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \\
&\qquad \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\, \alpha_2 \rangle\!\rangle \\
&\qquad \wedge \Gamma \vdash \alpha_1 \prec: \alpha_2 \\
&\qquad \wedge \emptyset \vdash \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash \{\} : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \emptyset \vdash \tau \equiv \mathsf{Set}\, \{x : \alpha \mid \textsc{false}\} \\
\langle\!\langle \Gamma \vdash \{e_1, \ldots, e_n\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1\,..\,n}.\, \wedge \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\, (\alpha_1 \uplus \ldots \uplus \alpha_n) \\
\langle\!\langle \Gamma \vdash \{x \in s : \phi\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\, \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\, \alpha_2 \rangle\!\rangle \\
&\qquad \wedge \langle\!\langle \Gamma, x : \alpha_1 \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\
&\qquad \wedge \Gamma \vdash \alpha_1 <: \alpha_2 \\
&\qquad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\, \{x : \alpha_1 \mid \phi\} \\
\langle\!\langle \Gamma \vdash \{e : x \in s\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \alpha_3.\, \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\, \alpha_2 \rangle\!\rangle \\
&\qquad\qquad \wedge \langle\!\langle \Gamma, x : \alpha_1 \vdash \phi : \alpha_3 \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \alpha_1 <: \alpha_2 \\
&\qquad\qquad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\, \alpha_3 \\
\langle\!\langle \Gamma \vdash \textsc{subset}\ s : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\, \alpha \rangle\!\rangle \\
&\qquad\qquad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\, \mathsf{Set}\, \alpha \\
\langle\!\langle \Gamma \vdash \textsc{union}\ s : \tau \rangle\!\rangle &\triangleq \langle\!\langle \Gamma \vdash s : \mathsf{Set}\, \tau \rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash e_1 \subseteq e_2 : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \alpha_3.\ \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \alpha_1 \prec: \alpha_1 \uplus \alpha_2 \\
&\quad \wedge \Gamma \vdash \alpha_2 \prec: \alpha_1 \uplus \alpha_2 \\
&\quad \wedge \varnothing \vdash \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash e_1 \cup e_2 : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \alpha_3.\ \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,(\alpha_1 \uplus \alpha_2) \\
\langle\!\langle \Gamma \vdash e_1 \cap e_2 : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \alpha_3.\ \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,(\alpha_1 \uplus\!\!\uplus \alpha_2) \\
\langle\!\langle \Gamma \vdash e_1 \setminus e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash \{x \in e_1 \cup e_2 : x \notin e_2\} : \tau \rangle\!\rangle \\
&\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,\{x : \alpha_1 \uplus \alpha_2 \mid x \notin e_2\}
\end{aligned}
$$

Functions

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash f[e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash f : \alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash e : \alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \alpha_2 \prec: \mathsf{dom}(\alpha_1) \\
&\quad \wedge \Gamma \vdash \tau \equiv \mathsf{cod}(\alpha_1 \cdot [e]) \\
\langle\!\langle \Gamma \vdash [x \in s \mapsto e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \alpha_3.\ \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma, x : \alpha_2 \vdash e : \alpha_3 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \alpha_2 <: \alpha_1 \\
&\quad \wedge \Gamma \vdash \tau \equiv (x : \alpha_2) \to \alpha_3 \\
\langle\!\langle \Gamma \vdash \textsc{domain}\ f : \tau \rangle\!\rangle &\triangleq \exists \alpha.\ \langle\!\langle \Gamma \vdash f : \alpha \rangle\!\rangle \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,(\mathsf{dom}(\alpha)) \\
\langle\!\langle \Gamma \vdash [s \to t] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\ \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash t : \mathsf{Set}\,\alpha_2 \rangle\!\rangle \\
&\quad \wedge \Gamma \vdash \tau \equiv \mathsf{Set}\,((\_ : \alpha_1) \to \mathsf{base}(\alpha_2)) \\
\langle\!\langle \Gamma \vdash [f\ \textsc{except}\ ![a] = b] : \tau \rangle\!\rangle &\triangleq \\
&\exists \alpha_f.\ \wedge \langle\!\langle \Gamma \vdash f : \alpha_f \rangle\!\rangle \\
&\quad \wedge \exists \alpha_a \alpha_b.\ \wedge \langle\!\langle \Gamma \vdash a : \alpha_a \rangle\!\rangle \\
&\qquad \wedge \langle\!\langle \Gamma \vdash b : \alpha_b \rangle\!\rangle \\
&\qquad \wedge \Gamma \vdash (x : \alpha_a) \to \alpha_b \prec: \alpha_f \\
&\quad \wedge \Gamma \vdash \tau \equiv (x : \mathsf{dom}(\alpha_f)) \to \{z : \mathsf{cod}(\alpha_f) \mid x \neq a\} \uplus \\
&\qquad\qquad \{z : \mathsf{base}(\mathsf{cod}(\alpha_f)) \mid x = a \wedge z = b\}
\end{aligned}
$$

Arithmetic

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash n : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \{x : \mathsf{Int} \mid x = n\} \qquad \text{for } n \in \mathit{Int} \\
\langle\!\langle \Gamma \vdash \mathit{Nat} : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Set}\,\{x : \mathsf{Int} \mid 0 \leq x\} \\
\langle\!\langle \Gamma \vdash \mathit{Int} : \tau \rangle\!\rangle &\triangleq \emptyset \vdash \tau \equiv \mathsf{Set}\,\{x : \mathsf{Int} \mid \textsc{true}\} \\
\langle\!\langle \Gamma \vdash -e : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \wedge\, \langle\!\langle \Gamma \vdash e : \alpha \rangle\!\rangle \wedge \Gamma \vdash \alpha \prec: \mathsf{Int} \\
&\qquad \wedge\, \Gamma \vdash \tau \equiv \{x : \mathsf{Int} \mid x = -e\} \\
\langle\!\langle \Gamma \vdash x + y : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\, \wedge\, \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \Gamma \vdash \alpha_1 \prec: \mathsf{Int} \\
&\qquad \wedge\, \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \Gamma \vdash \alpha_2 \prec: \mathsf{Int} \\
&\qquad \wedge\, \Gamma \vdash \tau \equiv \{z : \mathsf{Int} \mid z = x + y\} \\
\langle\!\langle \Gamma \vdash x < y : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash x \leq y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x > y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x \geq y : \tau \rangle\!\rangle \\
&\triangleq \exists \alpha_1 \alpha_2.\, \wedge\, \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \Gamma \vdash \alpha_1 \prec: \mathsf{Int} \\
&\qquad \wedge\, \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \Gamma \vdash \alpha_2 \prec: \mathsf{Int} \\
&\qquad \wedge\, \emptyset \vdash \tau \equiv \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash x \mathinner{..} y : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2.\, \wedge\, \langle\!\langle \Gamma \vdash x : \alpha_1 \rangle\!\rangle \wedge \Gamma \vdash \alpha_1 \prec: \mathsf{Int} \\
&\qquad \wedge\, \langle\!\langle \Gamma \vdash y : \alpha_2 \rangle\!\rangle \wedge \Gamma \vdash \alpha_2 \prec: \mathsf{Int} \\
&\qquad \wedge\, \Gamma \vdash \tau \equiv \mathsf{Set}\,\{z : \mathsf{Int} \mid x \leq z \wedge z \leq y\}
\end{aligned}
$$

Tuples and records

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash \langle\rangle : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \tau \equiv (i : \{x : \alpha \mid \textsc{false}\}) \to \{x : \alpha \mid \textsc{false}\} \\
\langle\!\langle \Gamma \vdash \langle e_1, \ldots, e_n \rangle : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1\,..\,n}.\, \wedge\, \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge\, \tau \equiv (i : \{1, \ldots, n\}_{\mathsf{Int}}) \to \\
&\qquad\qquad \{x : \alpha_1 \mid i = 1 \wedge x = e_1\} \\
&\qquad\qquad \uplus \ldots \\
&\qquad\qquad \uplus \{x : \alpha_n \mid i = n \wedge x = e_n\} \\
\langle\!\langle \Gamma \vdash S_1 \times \ldots \times S_n : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1\,..\,n}.\, \wedge\, \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\,\alpha_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge\, \tau \equiv \mathsf{Set}\,((i : \{1, \ldots, n\}_{\mathsf{Int}}) \to \\
&\qquad\qquad \{x : \alpha_1 \mid i = 1 \wedge x = e_1\} \\
&\qquad\qquad \uplus \ldots \\
&\qquad\qquad \uplus \{x : \alpha_n \mid i = n \wedge x = e_n\})
\end{aligned}
$$

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash [h_i \mapsto e_i]_{i:1\,..\,n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\, \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \wedge \tau \equiv \mathsf{Map}\,[h_i \mapsto \alpha_i]_{i:1\,..\,n} \\
\langle\!\langle \Gamma \vdash r.h : \tau \rangle\!\rangle &\triangleq \exists \alpha.\, \langle\!\langle \Gamma \vdash r : \alpha \rangle\!\rangle \wedge \tau \equiv \mathsf{dot}(\alpha, h) \\
\langle\!\langle \Gamma \vdash [h_i : S_i]_{i:1\,..\,n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n}.\, \wedge\, \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\,\alpha_i \rangle\!\rangle_{i:1\,..\,n} \\
&\qquad \wedge\, \tau \equiv \mathsf{Set}\,(\mathsf{Map}\,[h_i \mapsto \alpha_i]_{i:1\,..\,n})
\end{aligned}
$$

Miscellaneous constructs

$$\langle\!\langle \Gamma \vdash \text{“abc”} : \tau \rangle\!\rangle \triangleq \varnothing \vdash \tau \equiv \{x : \mathsf{Str} \mid x = \text{“abc”}\}$$

$$\begin{aligned}\langle\!\langle \Gamma \vdash \textsc{if}\ c\ \textsc{then}\ t\ \textsc{else}\ u : \tau \rangle\!\rangle \triangleq \exists \alpha_1 \alpha_2 \alpha_3.\ &\wedge \langle\!\langle \Gamma \vdash c : \alpha_1 \rangle\!\rangle \wedge \alpha_1 \cong \mathsf{Bool} \\ &\wedge \langle\!\langle \Gamma \vdash t : \alpha_2 \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash u : \alpha_3 \rangle\!\rangle \\ &\wedge \Gamma \vdash \tau \equiv \{x : \alpha_2 \mid c\} \uplus \{x : \alpha_3 \mid \neg c\}\end{aligned}$$

$$\begin{aligned}\langle\!\langle \Gamma \vdash \textsc{case}\ c_1 \to e_1\ \square \ldots\ \square\ c_n \to e_n : \tau \rangle\!\rangle \triangleq \exists \alpha_{1\,..\,n}, \alpha'_{1\,..\,n}.\ &\wedge \langle\!\langle \Gamma \vdash c_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \\ &\wedge \alpha_1 \cong \mathsf{Bool} \wedge \ldots \wedge \alpha_n \cong \mathsf{Bool} \\ &\wedge \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1\,..\,n} \\ &\wedge \tau \equiv \{x : \alpha'_1 \mid c_1\} \uplus \ldots \uplus \{x : \alpha'_n \mid c_n\}\end{aligned}$$

$$\begin{aligned}\langle\!\langle \Gamma \vdash \textsc{case}\ c_1 \to e_1\ \square \ldots\ \square\ c_n \to e_n\ \square\ \textsc{other} \to e_{n+1} : \tau \rangle\!\rangle \triangleq \exists \alpha_{1\,..\,n}, \alpha'_{1\,..\,n}.\ &\wedge \langle\!\langle \Gamma \vdash c_i : \alpha_i \rangle\!\rangle_{i:1\,..\,n} \\ &\wedge \alpha_1 \cong \mathsf{Bool} \wedge \ldots \wedge \alpha_n \cong \mathsf{Bool} \\ &\wedge \langle\!\langle \Gamma \vdash e_i : \alpha'_i \rangle\!\rangle_{i:1\,..\,n} \\ &\wedge \Gamma \vdash \tau \equiv \{x : \alpha'_1 \mid c_1\} \uplus \ldots \uplus \{x : \alpha'_n \mid c_n\} \\ &\qquad \uplus \{x : \alpha'_{n+1} \mid \neg c_1 \wedge \ldots \wedge \neg c_n\}\end{aligned}$$

Bibliography

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[AC01] David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit Substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.

[AF06] Nicolas Ayache and Jean-Christophe Filliâtre. Combining the Coq proof assistant with first-order decision procedures. http://www.lri.fr/ filliatr/publis/coq-dp.ps, 2006.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *1st Intl. Conf. Certified Programs and Proofs (CPP 2011)*, volume 7086 of *LNCS*, pages 135–150, Kenting, Taiwan, 2011. Springer.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In Matt Kaufmann and LawrenceC. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer Berlin Heidelberg, 2010.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AM96] Martín Abadi and Stephan Merz. On TLA as a logic. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, pages 235–271, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[AS14] Yamine Aït Ameur and Klaus-Dieter Schewe, editors. *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*. Springer, 2014.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, February 1975.

[Asp00] David Aspinall. Subtyping with power types. In *Computer Science Logic*, pages 156–171. Springer, 2000.

[Avi03] Jeremy Avigad. Eliminating definitions and Skolem functions in first-order logic. *ACM Trans. Comput. Logic*, 4(3):402–415, July 2003.

[AW13] Noran Azmy and Christoph Weidenbach. Computing tiny clause normal forms. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 109–125, Berlin, Heidelberg, 2013. Springer-Verlag.

[AZ13] Jeremy Avigad and Richard Zach. The epsilon calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2013 edition, 2013.

[BBN11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in Isabelle/HOL. In *Frontiers of Combining Systems*, pages 12–27. Springer, 2011.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In *Certified Programs and Proofs*, pages 151–166. Springer, 2011.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *14th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007)*, volume 4790 of *LNCS*, pages 151–165, Yerevan, Armenia, 2007. Springer.

[BDODF09] Thomas Bouton, Diego Caminha B De Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.

[BDP89] Leo Bachmair, Nachum Dershowitz, and David A Plaisted. Completion without failure. *Resolution of equations in algebraic structures*, 2:1–30, 1989.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.

[BGHL10] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 105–116, New York, NY, USA, 2010. ACM.

[BK09] Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. *Handbook of Satisfiability*, 185:339–401, 2009.

[BK11] Jasmin Christian Blanchette and Alexander Krauss. Monotonicity inference for higher-order formulas. *Journal of Automated Reasoning*, 47(4):369–398, 2011.

[BL03] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. In FrankS. de Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 2003.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[BSST09] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of satisfiability*, 185:825–885, 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). `www.SMT-LIB.org`, 2010.

[Bur11] Guillaume Burel. Experimenting with deduction modulo. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 162–176, Berlin, Heidelberg, 2011. Springer-Verlag.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79. ACM, 1988.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[CCKL08] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantic combination of congruence closure with solvable theories. *Electron. Notes Theor. Comput. Sci.*, 198(2):51–69, May 2008.

[CDL$^+$] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. In Dimitra Giannakopoulou and Dominique Méry, editors, *18th International Symposium On Formal Methods - FM 2012*, Paris, France.

[CDLM08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA$^+$ Proof System. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Knowledge Exchange: Automated Provers and Proof Assistants (LPAR Workshops)*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA$^+$Proof System. In Jürgen Giesl and Reiner Hähnle, editors, *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148, Edinburgh, UK, 2010. Springer. `http://tla.msr-inria.inria.fr/tlaps/`.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. *ACM SIGPLAN Notices*, 26(6):278–292, 1991.

[CI14] Sylvain Conchon and Mohamed Iguernelala. Tuning the Alt-Ergo SMT solver for B proof obligations. In Ameur and Schewe [AS14], pages 294–297.

[CLS11] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. Sort it out with monotonicity. In Nikolaj Bjorner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 207–221. Springer Berlin Heidelberg, 2011.

[Coh87] Anthony G Cohn. A more expressive formulation of many sorted logic. *Journal of automated reasoning*, 3(2):113–200, 1987.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[dB08] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *14th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.

[DCE73] George B Dantzig and B Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[DCKP12] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.

[Dd06] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, 2006.

[DDG+04] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, New York, NY, USA, 2004. ACM.

[DDG+13] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon Modulo: When Achilles outruns the tortoise using deduction modulo. In Ken McMillan, Aart Middeldorp, and Voronkov Andrei, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *Lecture Notes in Computer Science (LNCS)/Advanced Research in Computing and Software Science (ARCoSS)*, pages 274–290, Stellenbosch (South Africa), December 2013. Springer.

[DDMM14] David Delahaye, Catherine Dubois, Claude Marché, and David Mentré. The BWare project: Building a proof platform for the automated verification of B proof obligations. In Yamine Ait Ameur and Klaus-Dieter

Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 290–293. Springer Berlin Heidelberg, June 2014.

[Déh10] David Déharbe. Automatic verification for a class of proof obligations with SMT-solvers. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z (ASM 2010)*, volume 5977 of *LNCS*, pages 217–230, Orford, Canada, 2010. Springer.

[DFG+00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 59–73, London, UK, UK, 2000. Springer-Verlag.

[DFGV12] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *LNCS*, pages 194–207, Pisa, Italy, 2012. Springer.

[DKL+14] Damien Doligez, Jael Kriener, Leslie Lamport, Tomer Libal, Stephan Merz, et al. Coalescing: Syntactic abstraction for reasoning in first-order modal logics. In *Automated Reasoning in Quantified Non-Classical Logics*, 2014.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DLP+11] John R. Douceur, Jacob R. Lorch, Bryan Parno, James Mickens, and Jonathan M. McCune. Memoir—Formal Specs and Correctness Proofs. Technical Report MSR-TR-2011-19, Microsoft Research, 2011.

[dMB08] Leonardo de Moura and Nikolaj Bjorner. Proofs and refutations, and Z3. In *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[End01] H.B. Enderton. *A Mathematical Introduction to Logic, ch. 4.3 (Many-Sorted Logic)*. Harcourt/Academic Press, 2nd edition, 2001.

[Fla06] Cormac Flanagan. Hybrid type checking. In *ACM Sigplan Notices*, volume 41, pages 245–256. ACM, 2006.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[FM03] Michael J. Fischer and Michael Merritt. Appraising two decades of distributed computing theory research. *Distrib. Comput.*, 16(2-3):239–247, September 2003.

[FMM+06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg, 2006.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conf. on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[FRZ04] Pascal Fontaine, Silvio Ranise, and Calogero G Zarba. Combining lists with non-stably infinite theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 51–66. Springer, 2004.

[GBT09] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122, February 2009.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer aided verification*, pages 175–188. Springer, 2004.

[GM09] Yeting Ge and Leonardo Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[HHM+11] Thomas C Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the kepler conjecture. In *The Kepler Conjecture*, pages 341–376. Springer, 2011.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical report, 2002.

[HL12] Dominik Hansen and Michael Leuschel. Translating TLA$^+$ to B for Validation with ProB. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 24–38. Springer Berlin Heidelberg, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Hol97] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[Hol14] Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, 2014.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, October 1980.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[JBDD12] Mélanie Jacquel, Karim Berkani, David Delahaye, and Catherine Dubois. Tableaux modulo theories using superdeduction: An application to the verification of b proof rules with the zenon automated theorem prover. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, IJCAR'12, pages 332–338, Berlin, Heidelberg, 2012. Springer-Verlag.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321, 1991.

[JP09] Paul B. Jackson and Grant O. Passmore. Proving SPARK Verification Conditions with SMT solvers. December 2009.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263 – 297. Pergamon, 1970.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[KF07] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 505–519, Berlin, Heidelberg, 2007. Springer-Verlag.

[KMM00] Matt Kaufmann, J Strother Moore, and Panagiotis Manolios. *Computer-aided reasoning: an approach*. Kluwer Academic Publishers, 2000.

[KRW09] Daniel Kröning, Philipp Rümmer, and Georg Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22*, 2009.

[KSJ09] Hyondeuk Kim, Fabio Somenzi, and Hoonsang Jin. Efficient term-ite conversion for satisfiability modulo theories. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 195–208, Berlin, Heidelberg, 2009. Springer-Verlag.

[KU13] Cezary Kaliszyk and Josef Urban. Automated reasoning service for HOL Light. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *Proceedings of the 6th Conference on Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 120–135. Springer-Verlag, 2013.

[Kul09] Oliver Kullmann. Fundaments of branching heuristics. *Handbook of Satisfiability*, 185:205–244, 2009.

[Kun80] Kenneth Kunen. Set theory, volume 102 of Studies in Logic and the Foundations of Mathematics, 1980.

[KV12] Matthias Konrad and Laurent Voisin. Translation from set-theory to predicate calculus. Technical report, ETH Zurich, 2012.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification*, pages 1–35. Springer, 2013.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–454, 1974.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[Lam83] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[Lam94a] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[Lam94b] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[Lam95] Leslie Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.

[Lam02] Leslie Lamport. *Specifying Systems: The* TLA$^+$ *Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.

[Lam06] Leslie Lamport. Checking a multithreaded algorithm with+CAL. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin Heidelberg, 2006.

[Lam08] Leslie Lamport. The +CAL algorithm language. *URL http://research. microsoft. com/users/lamport/tla/pluscal. html. The page can also be found by searching the Web for the*, 2008.

[Lam11] Leslie Lamport. Byzantizing paxos by refinement. Available at http://research.microsoft.com/en-us/um/people/lamport/pubs/web-byzpaxos.pdf, 2011.

[Lan86] S.M. Lane. *Mathematics, form and function*. Springer-Verlag, 1986.

[lM93] Micha l Muzalewski. An outline of PC Mizar. *Fondation Philippe le Hodey, Brussels*, 1993.

[LMTY02] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with tla+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 45–48, New York, NY, USA, 2002. ACM.

[LMW11] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards verification of the Pastry protocol using TLA$^+$. In *Formal Techniques for Distributed Systems*, pages 244–258. Springer, 2011.

[LMW$^+$12] Tianxiang Lu, Stephan Merz, Christoph Weidenbach, et al. Formal verification of Pastry using TLA$^+$. In *International Workshop on the TLA+ Method and Tools*, 2012.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, May 1999.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Man05] M. Manzano. *Extensions of First-Order Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2nd edition, 2005.

[Mcc62] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[Meg07] Norman D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Publishing, Morrisville, North Carolina, 2007. `http://us.metamath.org/downloads/metamath.pdf`.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MMFA12] David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from atelier b using multiple automated provers. In *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 238–251, Berlin, Heidelberg, 2012. Springer-Verlag.

[MP06] Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In *ESCoR (CEUR Workshop Proceedings*, pages 70–80, 2006.

[MP08] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, January 2008.

[MS99] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence*, pages 62–74. Springer, 1999.

[MSLM09] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. *Handbook of satisfiability*, 185:131–153, 2009.

[MSS99] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[MV12a] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA$^+$ proof obligations with SMT solvers. In Nikolaj Bjorner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2012.

[MV12b] Stephan Merz and Hernán Vanzetto. Harnessing SMT Solvers for TLA$^+$ Proofs. *ECEASST*, 53, 2012.

[MV14] Stephan Merz and Hernán Vanzetto. Refinement Types for TLA$^+$. In JuliaM. Badger and KristinYvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 143–157. Springer International Publishing, 2014.

[New14] Chris Newcombe. Why Amazon chose TLA$^+$. In Ameur and Schewe [AS14], pages 25–39.

[Nip89] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12(2):123–149, 1989.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NO80] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Term Rewriting and Applications*, pages 453–468. Springer, 2005.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[NR95] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computation*, 19(4):321–351, 1995.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier and MIT Press, 2001.

[ORSSC99] Sam Owre, John M. Rushby, Natarajan Shankar, and David W. J. Stringer-Calvert. Pvs: An experience report. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 338–345, London, UK, UK, 1999. Springer-Verlag.

[OSW97] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997.

[Pau93] Lawrence C. Paulson. Set theory for verification: I. from foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.

[Pau99] Lawrence C Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999.

[PB10] Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *PAAR@ IJCAR*, pages 1–10, 2010.

[PD09] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers with restarts. In *Principles and Practice of Constraint Programming-CP 2009*, pages 654–668. Springer, 2009.

[PdMB10] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjorner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, 2010.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PL12] Daniel Plagge and Michael Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 372–386. Springer Berlin Heidelberg, 2012.

[PLD+11] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2011.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[Pot96] Francois Pottier. Simplifying subtyping constraints. In *In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, 1996.

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[RH06] Erik Reeber and Jr. Hunt, WarrenA. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, volume

4130 of *Lecture Notes in Computer Science*, pages 453–467. Springer Berlin Heidelberg, 2006.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, sep 1998.

[RTG+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *Automated Deduction–CADE-24*, pages 377–391. Springer, 2013.

[RW83] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In JörgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 298–313. Springer Berlin Heidelberg, 1983.

[SAN+02] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 203–216, New York, NY, USA, 2002. ACM.

[Sch13] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[Seb07] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[SMLN+03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003.

[Soz07] Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, pages 237–252. Springer, 2007.

[Spi92] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[SSCB12] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In Nikolaj Bjorner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 406–419. Springer, 2012.

[SSW06] Peter J Stuckey, Martin Sulzmann, and Jeremy Wazny. Type processing by constraint reasoning. In *Programming Languages and Systems*, pages 1–25. Springer, 2006.

[Sut00] Geoff Sutcliffe. System description: SystemOnTPTP. In *Automated Deduction-CADE-17*, pages 406–410. Springer, 2000.

[Sut09] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reason.*, 43(4):337–362, December 2009.

[Sut10] Geoff Sutcliffe. The TPTP World - infrastructure for automated reasoning. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 1–12, Berlin, Heidelberg, 2010. Springer-Verlag.

[THF10a] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ACM Sigplan Notices*, volume 45, pages 117–128. ACM, 2010.

[THF10b] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis*, pages 349–365. Springer, 1996.

[Tse68] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[Tur01] Raymond Turner. Type inference for set theory. *Theor. Comput. Sci.*, 266(1-2):951–974, September 2001.

[TYBK02] Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE International Workshop on Microprocessor Test and Verification (MTV '02)*. Institute of Electrical and Electronics Engineers, Inc., June 2002.

[Urb03] Josef Urban. Translating Mizar for first-order theorem provers. In Andrea Asperti, Bruno Buchberger, and JamesHarold Davenport, editors, *Mathematical Knowledge Management*, volume 2594 of *Lecture Notes in Computer Science*, pages 203–215. Springer Berlin Heidelberg, 2003.

[Urb08] Josef Urban. Automated reasoning for Mizar: Artificial intelligence through knowledge exchange. In *LPAR Workshops*, volume 418. Citeseer, 2008.

[Wei99] Christoph Weidenbach. Spass: Combining superposition, sorts and splitting. *Handbook of automated reasoning*, 2:1965–2013, 1999.

[Wen07] Makarius Wenzel. Isabelle/Isar —a generic framework for human-readable proof documents. *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[Wie06] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[Wie07] Freek Wiedijk. Mizar's soft type system. In *Theorem Proving in Higher Order Logics*, pages 383–399. Springer, 2007.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38, Montreal, Canada, 2008. Springer.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Andrew W. Appel and Alex Aiken, editors, *POPL*, pages 214–227. ACM, 1999.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[Zav12] Pamela Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

[Zav14] Pamela Zave. How to make Chord correct. unpublished, febraury 2014.

Abstract

This thesis presents effective techniques for discharging TLA$^+$ proof obligations to automated theorem provers based on unsorted and many-sorted first-order logic. TLA$^+$ is a formal language for specifying and verifying concurrent and distributed systems. Its non-temporal fragment is based on a variant of Zermelo-Fraenkel set theory for specifying the data structures. The TLA$^+$ Proof System TLAPS is an interactive proof environment in which users can deductively verify safety properties of TLA$^+$ specifications. While TLAPS is a proof assistant that relies on users for guiding the proof effort, it generates proof obligations and passes them to backend verifiers to achieve a satisfactory level of automation.

We developed a new back-end prover that soundly integrates into TLAPS external automated provers, specifically, ATP systems and SMT solvers. Two main components provide the formal basis for implementing this new backend. The first is a generic translation framework that allows to plug to TLAPS any automated prover supporting the standard input formats TPTP/FOF or SMT-LIB/AUFLIA. In order to encode higher-order expressions, such as sets by comprehension or total functions with domains, the translation to first-order logic relies on term-rewriting techniques coupled with an abstraction method. Sorted theories such as linear integer arithmetic are homomorphically embedded into many-sorted logic. The second component is a type synthesis algorithm for (untyped) TLA$^+$ formulas. The algorithm, which is based on constraint solving, implements one type system for elementary types, similar to those of many-sorted logic, and an expansion with dependent and refinement types. The obtained type information is then implicitly exploited to improve the translation. Empirical evaluation validates our approach: the ATP/SMT backend significantly boosts the proof development in TLAPS.

Keywords: formal verification, theorem proving, set theory, type systems

---

Resumé

Cette thèse présente des techniques efficaces pour déleguer des obligations de preuves TLA$^+$ dans des démonstrateurs automatiques basées sur la logique du premier ordre non-sortée et multi-sortée. TLA$^+$ est un langage formel pour la spécification et vérification des systèmes concurrents et distribués. Sa partie non-temporelle basée sur une variante de la théorie des ensembles Zermelo-Fraenkel permet de définir des structures de données. Le système de preuves TLAPS pour TLA$^+$ est un environnement de preuve interactif dans lequel les utilisateurs peuvent vérifier de manière deductive des propriétés de sûreté sur des spécifications TLA$^+$. TLAPS est un assistant de preuve qui repose sur les utilisateurs pour guider l'effort de preuve, il permet de génèrer des obligations de preuve puis les transmet aux vérificateurs d'arrière-plan pour atteindre un niveau satisfaisant d'automatisation.

Nous avons développé un nouveau démonstrateur d'arrière-plan qui intègre correctement dans TLAPS des vérificateurs externes automatisés, en particulier, des systèmes ATP et solveurs SMT. Deux principales composantes constituent ainsi la base formelle pour la mise en oeuvre de ce nouveau vérificateur. Le premier est un cadre de traduction générique qui permet de raccorder à TLAPS tout démonstrateur automatisé supportant les formats standards TPTP/ FOF ou SMT-LIB/AUFLIA. Afin de coder les expressions d'ordre supérieur, tels que les ensembles par compréhension ou des fonctions totales avec des domaines, la traduction de la logique du premier ordre repose sur des techniques de réécriture couplées à une méthode par abstraction. Les théories sortées telles que l'arithmétique linéaire sont intégrés par injection dans la logique multi-sortée. La deuxième composante est un algorithme pour la synthèse des types dans les formules (non-typées) TLA$^+$. L'algorithme, qui est basé sur la résolution des contraintes, met en oeuvre un système de type avec types élémentaires, similaires à ceux de la logique multi-sortée, et une extension avec des types dépendants et par raffinement. Les informations de type obtenues sont ensuite implicitement exploitées afin d'améliorer la traduction. Cette approche a pu être validé empiriquement permettant de démontrer que les vérificateurs ATP/SMT augmentent de manière significative le développement des preuves dans TLAPS.

Mots-clés: verification formelle, démonstration automatique des théorèmes, théorie des ensembles, systèmes de types.