Encodages de la théorie des ensembles de TLA+ pour la preuve automatique

Antoine Defourné

UNIVERSITÉ DE LORRAINE

IAEM
Loria

Thèse

Présentée et soutenue publiquement pour l'obtention du titre de

DOCTEUR DE l'UNIVERSITE DE LORRAINE

Mention : INFORMATIQUE

par **Antoine DEFOURNÉ**

Sous la direction de Stephan MERZ

Encoding TLA$^+$'s Set Theory for Automated Theorem Provers

7 novembre 2023

Membres du jury :

Directeur(s) de thèse :	**M. Stephan MERZ** **M. Pascal FONTAINE**	**Directeur de recherche, INRIA, Nancy** **Professeur à l'Université de Liège**
Président de jury :	**Mme Catherine DUBOIS**	**Professeure à l'ENSIIE, Évry-Courcouronnes**
Rapporteurs :	**M. David DELAHAYE** **Mme Aurélie HURAULT**	**Professeur à l'Université de Montpellier** **Maîtresse de conférences à l'ENSEEIHT, Toulouse**
Examinateurs :	**M. Jasmin BLANCHETTE** **M. Guillaume BONFANTE** **Mme Chantal KELLER**	**Professeur à l'Université de Munich** **Maître de conférences à l'Université de Lorraine** **Maîtresse de conférences à l'Université Paris-Saclay**

Remerciements

En préambule de ce long document, je tiens à remercier les personnes qui m'ont aidée à chaque étape de mon parcours. Je remercie pour commencer Aurélie Hurault, David Delahaye, Catherine Dubois, Chantal Keller et Guillaume Bonfante, pour avoir accepté d'évaluer mes travaux et ce manuscrit. Mes encadrants, Stephan Merz, Pascal Fontaine et Jasmin Blanchette, ont également lu et commenté cette thèse de multiples fois. Je veux surtout les remercier pour leurs bons conseils, leur patience à mon égard, et leur soutien dans mes moments de doute parfois terribles.

Je veux ensuite remercier les personnes qui ont partagé mon quotidien au LORIA, pour avoir su créer un environment bienveillant me permettant d'évoluer en toute confiance. Merci à Daniel et Hans-Jörg pour m'avoir accueillie mon premier jour à Nancy. Merci à Margaux et Pierre, pour leur accueil également. Merci à notre assistante d'équipe Sophie, pour son aide enthousiaste avec toutes les corvées administratives. Merci à Benjamin, Thomas et Dylan, pour m'avoir laissé la meilleure place du bureau. Merci à Marie, Sophie, Engel, Horatiu, pour les échanges à l'heure du repas, en séminaire, et aux soirées jeux. Merci à Alessio pour ses questions, qui ont ravivé mon intérêt pour TLA$^+$ alors que ma thèse s'achevait. Merci à Claire et Athénaïs, pour leur soutien moral plus que bienvenu dans la dernière ligne droite.

Les dernières personnes que je veux remercier m'ont vue grandir et m'ont soutenue dans le moment le plus décisif de ma vie. Merci à ma famille, et tout particulièrement ma mère, mon père, ma sœur et mon frère, pour votre aide, votre soutien, et votre amour. Enfin, merci à Lucie. Je te remercie en dernier parce que tu es la personne la plus importante dans ma vie désormais. Merci d'avoir bien voulu la partager.

Résumé

Cette thèse porte sur TLA$^+$, un langage de spécification fondé sur la logique temporelle et la théorie des ensembles non typée. TLA$^+$ est principalement utilisé dans l'industrie pour vérifier des systèmes concurrents et distribués. On s'intéresse en particulier à la preuve interactive de théorèmes TLA$^+$, qui peut être réalisée avec l'outil TLAPS. Cet assistant de preuve emploie une batterie de prouveurs automatiques externes afin de vérifier les obligations de preuve qui correspondent aux étapes de raisonnement de l'utilisateur. Parmi les prouveurs disponibles, on trouve LS4 pour la logique temporelle, Isabelle/TLA$^+$, Zenon, et des solveurs SMT dont CVC4, veriT et Z3.

Divers encodages de TLA$^+$ sont implémentés pour que TLAPS traduise les obligations vers les logiques d'entrée de chaque prouveur. L'encodage SMT, en particulier, est fondé sur un puissant système de réécriture, complété par un mécanisme d'inférence de types, permettant à TLAPS de simplifier les obligations de façon considérable avant l'envoi vers SMT. Il est naturel de chercher à simplifier les obligations : la théorie des ensembles de TLA$^+$ est bien plus expressive que la logique de SMT, qui comprend la logique du premier ordre et certaines théories, par exemple l'arithmétique des entiers. Il est possible d'encoder directement TLA$^+$ en fournissant à SMT tous les axiomes qui spécifient les ensembles ; mais cette approche a auparavant été rejetée, car les axiomes sont des formules avec quantificateurs, et SMT ne peut gérer ces quantificateurs efficacement.

Néanmoins, il est important de s'assurer que les encodages de TLAPS n'introduisent pas d'erreur dans la traduction, autrement il serait possible de prouver de faux résultats avec l'outil. L'encodage SMT actuel est trop complexe pour être vérifié, nous avons donc entrepris d'implémenter l'encodage direct et naïf de TLA$^+$, dans l'optique de l'optimiser ensuite. Ce travail a abouti à deux contributions. La première est l'implémentation d'un encodage vers la logique d'ordre supérieure et l'ajout d'un nouveau prouveur à TLAPS : Zipperposition. Il est intéressant de disposer d'un tel prouveur, car une petite partie des obligations en TLA$^+$ impliquent des étapes d'ordre supérieur. Cela se produit particulièrement souvent autour des preuves par induction. La seconde contribution est une refonte de l'encodage SMT, cette fois basée sur l'encodage direct mais optimisée avec des heuristiques pour instancier les axiomes de la théorie des ensembles. Il s'agit de la technique des *triggers*, populaire parmi les utilisateurs de SMT, et qui nous permet en quelque sorte d'implémenter des procédures d'instanciation selon nos besoins. L'intérêt majeur des *triggers* pour notre approche est qu'ils s'ajoutent à la traduction directe sans en modifier le sens, donc sans compromettre la sûreté de l'encodage. De façon assez surprenante, notre encodage mène à des résultats aussi bons que le précédent.

Abstract

This thesis is about TLA$^+$, a specification language based on temporal logic and untyped set theory. TLA$^+$ is mainly used in the industry for the verification of concurrent and distributed systems. We focus on interactive theorem proving in the context of TLA$^+$, which is supported by the proof assistant TLAPS. This system manages the proof obligations that correspond to each step of the user's reasoning, and calls different backend automated provers to check these obligations. Currently the provers available are LS4 for the temporal fragment of the logic, Isabelle/TLA$^+$, Zenon, and the SMT solvers CVC4, veriT and Z3.

All TLA$^+$ obligations are encoded into the respective logics of the backend solvers by TLAPS. The SMT encoding, notably, combines a powerful preprocessing phase with type inference, simplifying obligations in an attempt to remove the TLA$^+$ primitive symbols with no counterpart in SMT's language. It is natural to perform these simplifications, as TLA$^+$'s set theory is much more expressive than what SMT is designed for, which is first-order logic with theories such as integer arithmetic. In principle, one could encode TLA$^+$ directly by inserting the necessary axioms of set theory into the SMT problem. This approach was discarded in favor of preprocessing because SMT solvers do not cope well with quantified formulas, and the axiomatic theory of TLA$^+$ involves many of them.

Regardless, we must ensure that TLAPS does not encode invalid TLA$^+$ obligations as valid problems, otherwise one would be able to prove faulty statements unknowingly. Unfortunately, the current version of the SMT encoding is too complex to be verified, which is why we decided to implement the naive, direct encoding of TLA$^+$, as a basis for future optimizations. This work has lead to two important contributions. The first is an encoding into higher-order logic, completed by the addition of a new backend for TLAPS: Zipperposition. A higher-order backend is a worthy addition to the system, because specific situations, in particular around induction, require higher-order reasoning to be carried out. The second contribution is a new version of the SMT encoding. This version extends the direct encoding with heuristics for instantiating the axioms of set theory. More precisely, we annotated our axioms with *triggers*. Triggers are a popular feature of SMT solvers that allow users to guide the instantiation module of the solvers for custom axiomatizations. They allow us to extend the encoding without compromising its soundness, as they do not alter the semantics of formulas. Surprisingly, our encoding achieves performances comparable to the previous version, despite its simpler design.

Résumé étendu

Introduction

Cette thèse porte sur le langage de spécification TLA$^+$ et son assistant de preuve TLAPS, qui utilise des prouveurs externes pour automatiser la vérification de théorèmes mathématiques. Notre objectif est d'améliorer l'automatisation apportée par TLAPS en intervenant au niveau des encodages de TLA$^+$ pour les prouveurs.

Preuve interactive et Preuve automatique

Nos travaux s'inscrivent dans le champ des méthodes formelles, dont l'objectif général est la vérification des programmes informatiques. Tout développeur veut pouvoir s'assurer qu'il ou elle écrit du code rigoureux. Dans certains contextes, il est crucial de pouvoir garantir qu'un système ne tombera pas en panne à cause d'un bug informatique [52, 54, 58, 73]. Les méthodes formelles recouvrent plusieurs approches visant à certifier que des programmes satisfont leurs spécifications. Il ne s'agit pas de tester les programmes, mais d'apporter des preuves mathématiques qu'ils ne contiennent pas de bugs.

Le champ qui nous concerne est la preuve interactive, plus particulièrement dans le contexte du langage TLA$^+$. Ce langage de spécification permet à l'utilisateur de modéliser des systèmes par des formules mathématiques [48, 51, 85]. TLA$^+$ a de nombreux points communs avec d'autres langages de spécification comme la Notation Z ou la Méthode B [1,2,74]. Comme ces derniers, TLA$^+$ propose la théorie des ensembles comme base théorique, et y ajoute des notions de logique temporelle pour modéliser l'évolution des systèmes au cours du temps.

Une fois un système modélisé en TLA$^+$, il est possible d'exprimer ses propriétés de correction par une formule. Supposons par exemple qu'on a défini une formule temporelle *Spec* pour représenter un système. Si n est une variable du système, alors la formule

$$Spec \Rightarrow \Box(n \in Nat)$$

exprime le fait que n est un entier naturel à tout moment de l'exécution du système. Pour certifier que notre système vérifie la propriété, il suffit de démontrer cette formule comme n'importe quel théorème mathématique. TLA$^+$ propose une syntaxe pour écrire des preuves, qui peuvent ensuite être prises en charge par l'outil TLAPS.

TLAPS est un logiciel faisant partie de la famille des prouveurs interactifs, aussi appelés assistants de preuve. Ces outils associent des langages mathématiques expressifs à des procédures de vérification : l'utilisateur développe sa théorie mathématique à partir de ses propres définitions, choisit des théorèmes pour lesquels il écrit des preuves.Le rôle du système est de vérifier que chaque étape du raisonnement de l'utilisateur est logiquement valide. Comme exemples de prouveurs interactifs populaires, on peut citer Coq, qui est basé sur la théorie

des types dépendants, et Isabelle/HOL, qui est basé sur la logique d'ordre supérieur (aussi appelée théorie des types simples) [17, 60].

Le fonctionnement de TLAPS repose essentiellement sur l'emploi de prouveurs automatique externes pour assurer la vérification [19]. Un prouveur automatique est un outil prenant en entrée des problèmes mathématiques afin de décider de leur validité—quand cela est possible, car le problème général est indécidable pour des logiques suffisamment expressives. Tous les prouveurs ne prennent pas en charge les mêmes logiques, et tous n'implémentent pas les mêmes procédures pour aboutir à une réponse. Cette thèse concerne particulièrement ce qu'on appelle les solveurs SMT (*Satisfiability Modulo Theory*), qui prennent en charge diverses théories de la logique du premier ordre, avec ou sans quantificateurs. La plupart des solveurs SMT acceptent le langage SMT-LIB en entrée [6, 29, 32].

Étant donnée une spécification TLA$^+$ avec des preuves, l'assistant TLAPS identifie les *obligations de preuve*, qui correspondent à peu près aux étapes du raisonnement de l'utilisateur, et traite ces obligations pour les passer aux prouveurs externes disponibles. Ce traitement consiste essentiellement en un *encodage* des obligations pour chaque prouveur invoqué. Un encodage est une traduction d'une logique vers une autre. Concevoir et implémenter un encodage peut s'avérer difficile, car certains aspects de la logique d'entrée peuvent ne pas avoir d'équivalent dans la logique de sortie. En outre, pour faire confiance à TLAPS, il faut pouvoir faire confiance aux prouveurs externes d'une part, et aux encodages d'autre part.

Encodage de TLA$^+$ pour SMT

Les prouveurs de TLAPS ont des forces et des faiblesses complémentaires, et tous ne prennent pas nécessairement en charge la totalité du langage. Les différents prouveurs sont LS4—seul prouveur à supporter la logique temporelle du langage—Isabelle/TLA$^+$, Zenon [11], et quelques solveurs SMT : CVC4 [5], veriT [12] et Z3 [25]. La logique de SMT est la logique du premier ordre sortée. Différentes théories incluant des symboles interprétés sont également disponibles, par exemple l'arithmétique entière. Les solveurs SMT sont réputés pour leur efficacité sur des problèmes de très grande taille. Le défi pour un encodage de TLA$^+$ vers SMT-LIB est de traduire les expressions impliquant des symboles primitifs du langage, car la grande majorité n'ont pas leur équivalent en SMT. Même pour les symboles qui ont un équivalent en SMT, en particulier les opérateurs arithmétiques, la traduction directe n'est pas toujours possible puisque TLA$^+$ autorise des combinaisons de symboles non conventionnelles.

Pour donner un bref exemple, on considère la formule suivante :

$$\forall x : x + 0 = x$$

Cette formule est exprimable en TLA$^+$, mais elle n'est pas valide, car il manque l'hypothèse de type $x \in \mathit{Int}$. Sans cette hypothèse, l'expression $x + 0$ est dite *sous-spécifiée*, et il n'est pas possible de la simplifier en x. Considérons maintenant la formule suivante en logique sortée :

$$\forall x^{\mathsf{int}} : x + 0 = x$$

L'annotation de sorte int indique que la variable x est interprétée dans le domaine des entiers relatifs. Pour cette raison, la dernière formule est valide, car l'argument x de l'addition est toujours un entier dans le contexte de l'expression.

Ce simple exemple illustre la difficulté d'encoder la logique non typée de TLA$^+$ dans une logique typée : la traduction requiert d'assigner des types aux symboles et variables

quantifiées (car il faut au moins que les expressions encodées respectent les règles de typage), mais toutes les assignations ne sont pas possibles *a priori*. On dit qu'un encodage est *correct* si la validité d'une obligation encodée implique la valididé de l'obligation source. La correction est essentielle puisqu'un encodage incorrect autoriserait la dérivation de faux théorèmes. La traduction naïve de l'exemple précédent est donc incorrecte.

Une traduction correcte possible consisterait à redéfinir l'addition de TLA$^+$ dans le problème généré pour SMT. Concrètement, on peut créer une sorte ι afin d'accueillir les expressions de TLA$^+$, et déclarer les symboles dont on a besoin comme symboles *non interprétés*. En l'occurrence, si on représente l'addition par une fonction plus binaire sur ι, et 0 par une constante zero de type ι, la formule peut être traduite

$$\forall x^{\iota} : \mathsf{plus}(x, \mathsf{zero}) = x$$

En parallèle, les interprétations de plus et zero seront fournies par un ensemble d'*axiomes* que nous ajoutons au problème encodé. Les solveurs SMT pourront alors raisonner sur ces nouveaux symboles à partir des axiomes, qui incluent principalement la théorie des ensembles, les fonctions, et l'arithmétique.

Cet encodage hypothétique, que l'on nomme *direct*, a le mérite d'être simple à implémenter et prouver correct. Mais on peut *a priori* penser qu'il ne se montrera pas très efficace en pratique. En effet, la présence de quantificateurs nuit grandement aux performances des solveurs SMT, qui ont d'abord été conçu pour des problèmes sans quantificateurs. Or, l'encodage direct implique d'ajouter de nombreux axiomes qui, pour la grande majorité, sont des formules quantifiées. Les solveurs SMT devront déterminer quelles sont les instances nécessaires des axiomes pour résoudre les problèmes encodés ; la difficulté de cette tâche est exacerbée par le fait que les obligations encodées ne contiennent pas d'information de typage : toutes les variables quantifiées se voient assigner la sorte générique ι.

Nos travaux poursuivent les efforts de Vanzetto, qui a conçu et implémenté l'encodage SMT actuel de TLAPS [53]. Partant de cette constatation à propos de l'encodage direct, Vanzetto a proposé une solution qui consiste à prétraiter les obligations afin de les simplifier. L'objectif concret de cette simplification est d'éliminer le plus d'occurrences possible de symboles primitifs de TLA$^+$ afin de se ramener à la logique de SMT. L'implémentation s'articule autour d'un module de réécriture permettant de transformer les obligations. Optionnellement, il est possible de faire une inférence de types pour affiner la réécriture. Par exemple, la règle suivante pourrait être appliquée pour simplifier des expressions arithmétiques :

$$x + 0 \longrightarrow x \qquad \text{si } x \in \mathit{Int}$$

La règle ne s'applique pas à l'exemple précédent, mais elle peut permettre de simplifier la formule

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

Vérifier que la condition $x \in \mathit{Int}$ est satisfaite dans le contexte de l'expression $x + 0$ relève de l'inférence de types. Une fois la règle appliquée, le but à prouver est simplement $x = x$. En admettant qu'il n'est pas possible de simplifier la formule davantage, l'encodage direct est appliqué, seulement il n'est plus nécessaire de fournir autant d'axiomes à SMT, et le problème est plus simple à résoudre.

Contributions

L'encodage vers SMT de TLAPS est particulièrement efficace. De nombreuses obligations de preuve importantes en taille ne peuvent être résolues que par l'un des solveurs SMT intégrés à TLAPS. Toutefois, l'encodage tire son efficacité de techniques particulièrement sophistiquées, qui sont difficiles à mettre en place et à vérifier. Cela pose un problème pour la sûreté de TLAPS, car si un encodage prend en charge une partie du raisonnement—comme celui qui sous-tend la simplification des formules—alors il incombe aux développeurs de TLAPS de prouver que leur implémentation est correcte.

Cette thèse améliore l'encodage SMT du point de vue de la sûreté. En effet, notre approche consiste à partir d'une implémentation de l'encodage direct pour ensuite chercher à l'optimiser de manière prudente. Il était initialement prévu de réimplémenter la réécriture et l'inférence de types, mais une autre solution, plus satisfante en termes de correction, a été trouvée. Concrètement, on peut résumer cette thèse en deux contribution, chacune ayant abouti en une publication [27, 28] :

L'encodage direct, réduit au strict nécessaire, ne vise pas la logique de SMT mais la logique d'ordre supérieur. On propose un encodage direct réduit à deux étapes élémentaires, faciles à prouver correctes. L'encodage a été implementé afin d'intégrer un nouveau prouveur à TLAPS : Zipperposition [20, 82, 83]. L'intérêt d'un prouveur d'ordre supérieur est aussi d'améliorer l'automatisation de preuves qui étaient mal prises en charge auparavant.

L'encodage SMT étend cet encodage direct de manière minimale. Afin d'optimiser la recherche de preuves, on fournit des *heuristiques d'instantiation* pour nos axiomes à SMT. L'intérêt de cette technique est qu'elle ne nécessite pas de modifier les expressions par de la réécriture. La correction de l'encodage n'est donc pas compromise, et on a pu constater par l'évaluation que nos heuristiques permettent d'atteindre le même degré d'efficacité qu'auparavant.

Développons maintenant ces deux contributions dans l'ordre.

Encodage direct de TLA$^+$ vers HOL

Ce premier encodage est qualifié de "direct" car il maintient la structure des expressions aussi fidèlement que possible. Notre objectif global est d'étendre l'encodage direct pour les solveurs SMT. Pour des raisons qui seront clarifiées dans cette section, il est utile de définir dans un premier temps un encodage vers la logique d'ordre supérieur (HOL).

Formalisation de TLA$^+$

Avant de détailler l'encodage de TLA$^+$, il faut clarifier la définition formelle du langage, c'est-à-dire donner sa syntaxe et sa sémantique. On propose une définition qui se rapproche autant que possible d'une formalisation standard de la logique du premier ordre. Le mérite de cette approche est qu'elle met en relief ce qui distingue TLA$^+$ de cette logique, indiquant ainsi quels aspects du langage sont les plus difficiles à traduire.

La différence entre TLA$^+$ et la logique du premier ordre tient en trois points. Le premier est la présence de symboles interprétés, principalement pour la théorie des ensembles, les fonctions

et l'arithmétique. Ces symboles sont en réalité sous-spécifiés, car le langage non typé autorise des combinaisons inhabituelles, comme $x + 0$ dans le cas où x n'est pas un nombre. Il est en fait assez simple de formaliser cet aspect de TLA$^+$: au lieu de définir une interprétation pour chaque symbole, on donne la collection d'axiomes que chaque interprétation est censée satisfaire. Dire que TLA$^+$ est sous-spécifié revient à dire que cette collection d'axiomes—cette théorie—n'a pas d'interprétation canonique, qu'aucune interprétation n'est plus légitime qu'une autre.

Le principe de la sous-spécification s'applique en quelque sorte aux connecteurs Booléens, mais nous choisissons de formaliser ceux-ci autrement. La principale raison est qu'il serait difficile en pratique d'encoder les connecteurs Booléens sous la forme de symboles non interprétés et de fournir des axiomes aux prouveurs ; on préférerait directement utiliser les connecteurs équivalents dans la logique de sortie. L'examen de la documentation officielle sur TLA$^+$ révèle qu'il est possible de donner une interprétation totale pour chaque connecteur. Considérons par exemple l'implication, notée $\Rightarrow$. En logique traditionnelle, l'implication permet de combiner des formules, mais en TLA$^+$ les arguments de $\Rightarrow$ ne sont pas des formules *a priori*. Par exemple, l'expression $1 + 1 \Rightarrow 2$ est légitime, mais le sens qu'il faut lui donner n'est pas clair. La bonne façon d'interpréter $\Rightarrow$ consiste à comparer la valeur de ses arguments avec la valeur de vérité "vrai" pour les évaluer en tant que formules. Notre exemple est logiquement équivalent à l'expression $(1 + 1 = \text{TRUE}) \Rightarrow (2 = \text{TRUE})$, qui est valide.

Le dernier point à souligner est la présence de symboles d'ordre supérieur, spécifiquement du second ordre. Parmi les symboles primitifs de TLA$^+$, on trouve notamment les expressions $\{x \in S : e\}$ pour la compréhension ensembliste et $[x \in S \mapsto e]$ pour les fonctions. Ces deux symboles lient la variable x dans l'expression e, on considère donc qu'elles sont paramétrées par la fonction assignant e à tout x, représentée par la lambda-expression $\lambda x : e$. Cet aspect ne s'arrête pas aux symboles primitifs, puisque depuis la seconde version du langage l'utilisateur peut définir ses propres symboles du second ordre, qui acceptent comme arguments des symboles du premier ordre ou des lambda-expressions.

Il est facile d'étendre la logique du premier ordre pour inclure les symboles du second ordre dont TLA$^+$ a besoin. Cette modification ne fait pas de TLA$^+$ une logique d'ordre supérieur, car les quantificateurs universel $\forall$ et existentiel $\exists$ ne permettent pas de quantifier sur des symboles de fonctions. Cependant, comme TLA$^+$ autorise le paramétrage de théorèmes par des symboles de fonction, certaines situations relèvent techniquement de l'ordre supérieur. Typiquement, une preuve par récurrence exigera d'invoquer le principe selon lequel, *pour tout prédicat* P, il suffit de prouver $P(0)$ et $\forall n \in Nat : P(n) \Rightarrow P(n + 1)$ pour dériver $\forall n \in Nat : P(n)$. L'une des étapes de la preuve consistera donc à unifier le but courant avec $P(n)$ pour déterminer la valeur de P, qui est une variable du second ordre.

Pour cette raison, il est intéressant de proposer un encodage vers la logique d'ordre supérieur. Parmi les prouveurs disponibles dans TLAPS, seul Isabelle/TLA$^+$ est capable de traiter les obligations d'ordre supérieur. Ces obligations ne sont pas particulièrement courantes dans la pratique de TLAPS, mais elles sont inévitables dans certains contextes, et l'expérience montre que Isabelle/TLA$^+$ est souvent mis en difficulté lorsque le raisonnement à l'ordre supérieur devient trop complexe.

Définition de l'encodage direct

L'encodage direct vers HOL se résume à deux étapes essentielles : une passe sur les expressions pour corriger les ambiguïtés entre termes et formules, puis l'ajout d'axiomes au problème pour

encoder la sémantique des symboles de TLA$^+$.

La première étape vise à retrouver la sémantique habituelle pour les connecteurs Booléens. Dans la logique de TLA$^+$, il n'y pas de distinction entre termes et formules, mais dans la logique de sortie, les termes sont identifiés par la sorte ι, et les formules sont identifiées par la sorte o. Il s'agit d'insérer des conversions appropriées entre termes et formules de façon à obtenir des expressions bien typées. Ces conversions impliquent une injection cast_o de type $o \to \iota$. Convertir une formule en terme se fait en la passant en argument à cast_o. L'autre direction est moins intuitive, mais nous l'avons déjà illustrée avec l'exemple de $1 + 1 \Rightarrow 2$, dans laquelle deux termes apparaissent là où des formules sont attendues. Notre encodage réécrira cette expression en $(1 + 1 = \mathsf{cast}_o(\text{TRUE})) \Rightarrow (2 = \mathsf{cast}_o(\text{TRUE}))$.

La seconde étape insère des déclarations explicites pour chaque symbole primitif de TLA$^+$ dans l'obligation. Chaque symbole se voit assigné un ou plusieurs axiomes (ou schémas d'axiomes) qui spécifient sa sémantique. Ces axiomes sont insérés dans le contexte de l'obligation avec la déclaration ; si un axiome mentionne un nouveau symbole, la procédure est répétée de façon récursive. Typiquement, chaque symbole est spécifié par un à trois axiomes. Nos axiomes sont formulés de sorte à réduire le nombre de dépendances entre symboles. Par exemple, le symbole $\cap$, pour l'intersection entre deux ensembles, est spécifié par l'unique axiome

$$\forall a, b, x : x \in (a \cap b) \Leftrightarrow x \in a \wedge x \in b$$

Il serait également possible de définir le symbole par la définition $a \cap b \triangleq \{x \in a : x \in b\}$, mais cela ferait dépendre $\cap$ de la notation de compréhension.

Les axiomes que nous utilisons ne proviennent pas de la documentation officielle, mais on peut les inférer depuis celle-ci assez facilement. Pour résumer la théorie de TLA$^+$, on peut se concentrer sur trois piliers : les ensembles, les fonctions et l'arithmétique. Les axiomes de théorie des ensembles sont standards et correspondent à ceux de la théorie ZFC. Les axiomes des fonctions spécifient le constructeur $[x \in S \mapsto e]$ et les expressions DOMAIN f et $f[x]$ pour récupérer le domaine d'une fonction et son image par x, respectivement. On trouve également le constructeur $[S \to T]$ pour les ensembles de fonctions. Une présentation complète de TLA$^+$ devrait inclure les n-uplets, les enregistrements et les séquences, mais ces trois types de structures sont définis à partir des fonctions, et leurs axiomes sont directement conséquences de la théorie des fonctions.

À ce stade de nos travaux, nous choisissons d'ignorer l'arithmétique, pour plusieurs raisons. L'arithmétique entière est une théorie que de nombreux prouveurs prennent en charge de façon native. Plutôt que de fournir aux prouveurs nos propres axiomes pour l'arithmétique, il est naturel de chercher à exploiter les procédures natives de ces prouveurs lorsqu'elles sont disponibles. Pour notre encodage HOL, le prouveur que nous avons choisi d'intégrer à TLAPS est Zipperposition, mais la version que nous utilisons ne prend pas en charge l'arithmétique. Puisque nous envisageons Zipperposition pour une utilisation spécialisée—les obligations d'ordre supérieur—nous traitons le cas de l'arithmétique comme secondaire, et nous l'ignorons. Il y a des axiomes que nous ajoutons néanmoins, car leur omission rendrait non prouvables trop de théorèmes : chaque symbole arithmétique de TLA$^+$ a un axiome qui donne son type, par exemple $+$ a l'axiome

$$\forall x, y \in \mathit{Int} : (x + y) \in \mathit{Int}$$

Nous incluons au moins ces axiomes, car ils sont assez simples et nécessaires pour beaucoup de vérifications de routine.

Intégration de Zipperposition

Zipperposition est un prouveur par superposition pour la logique du premier ordre. Il inclut une extension à l'ordre supérieur, extension qui a permis à l'outil de gagner la compétition CASC dans la section THF. Le dialecte THF appartient au standard TPTP pour exprimer des problèmes mathématiques, et c'est vers ce langage que notre encodage direct traduit les obligations de TLA$^+$.

Comme dit précédemment, l'intérêt d'un prouveur comme Zipperposition pour TLAPS est principalement de renforcer l'automatisation pour les obligations d'ordre supérieur, qui autrement sont prises en charge par Isabelle seulement. Nous avons donc comparé les performances de Zipperposition et d'Isabelle sur un ensemble assez grand d'obligations, qui englobe notamment la librairie d'exemples TLA$^+$ officielle dans son entièreté. Notre évaluation démontre que les deux prouveurs se complètent assez bien, puisque certaines obligations sont résolues par Zipperposition uniquement, et d'autres par Isabelle uniquement. La principale faiblesse de Zipperposition est qu'il ne peut pas raisonner dans l'arithmétique entière, qui est souvent nécessaire. En outre, notre expérience avec Zipperposition suggère que le prouveur serait capable de résoudre des obligations plus difficiles quand du raisonnement à l'ordre supérieur est requis.

Encodage de TLA$^+$ vers SMT-LIB

L'encodage vers SMT-LIB est conçu comme une extension de l'encodage direct. L'extension se résume à trois modifications, dont seule la première est absolument nécessaire, puisqu'il s'agit d'éliminer l'ordre supérieur des obligations. Par exemple, les expressions $\{x \in S : e\}$ et $[x \in S \mapsto e]$ doivent être exprimées dans la logique du premier ordre. Il existe des procédures standards pour implémenter cette traduction, et notre approche s'inspire des méthodes de défonctionnalisation qu'utilisent les compilateurs de langages fonctionnels. Intuitivement, la méthode consiste à créer un symbole spécialisé pour chaque mention d'un symbole d'ordre supérieur présent dans l'obligation. Si par exemple une obligation mentionne l'ensemble $\{n \in Nat : n < 100\}$, il suffit de créer un symbole spécialisé pour l'argument $\lambda n : n < 100$. Ce symbole, que l'on peut noter $\mathsf{setst}^\bullet$, doit être spécifié par l'axiome

$$\forall a, x : x \in \mathsf{setst}^\bullet(a) \Leftrightarrow x < 100$$

L'expression d'origine peut être traduite $\mathsf{setst}^\bullet(Nat)$.

Intégration de l'arithmétique SMT

La deuxième modification est essentielle pour supporter l'arithmétique entière de manière satisfaisante. Les solveurs SMT sont dotés de techniques spéciales pour raisonner sur l'arithmétique entière, et on souhaite pouvoir mettre ces techniques à profit. Cela implique d'encoder des expressions vers la sorte int de SMT, ce qui n'est pas simple *a priori* puisque TLA$^+$ est non typé. Notre solution, qui était déjà utilisée par l'encodage précédent pour la version sans inférence de types, consiste à fournir quelques axiomes spéciaux permettant aux solveurs SMT de basculer le raisonnement dans le domaine des entiers.

Ces axiomes reposent sur un injecteur $\mathsf{cast}_{\mathsf{int}}$ de type $\mathsf{int} \to \iota$. Si $x \in Int$ est vrai, SMT pourra déduire l'existence d'un n tel que $x = \mathsf{cast}_{\mathsf{int}}(n)$. Pour encoder ce principe, il suffit

d'ajouter cet axiome au problème final :

$$\forall x^{\iota} : x \in Int \Leftrightarrow \left[\exists n^{\mathsf{int}} : x = \mathsf{cast}_{\mathsf{int}}(n)\right]$$

En plus de cette axiome, chaque symbole arithmétique de TLA$^+$ avec un symbole SMT équivalent se voit assigner un axiome pour lier les deux versions. Par exemple, nous lions le $+$ de TLA$^+$ avec le $+_{\mathsf{int}}$ interprété de SMT par l'axiome

$$\forall m^{\mathsf{int}}, n^{\mathsf{int}} : \mathsf{cast}_{\mathsf{int}}(m) + \mathsf{cast}_{\mathsf{int}}(n) = \mathsf{cast}_{\mathsf{int}}(m +_{\mathsf{int}} n)$$

Heuristiques d'instantiation pour SMT

La troisième modification concerne la collection d'axiomes entière. L'encodage SMT précédent tirait son efficacité du fait qu'il simplifie grandement les expressions de TLA$^+$. Notre encodage innove sur ce point en se contentant de fournir des heuristiques pour guider les procédures d'instantiation de SMT. Cette technique ne nécessite pas de simplifier les expressions, il n'y a donc aucun risque de rendre l'encodage incorrect.

La technique particulière que nous utilisons est celle des *triggers* de SMT [24, 55]. Dans le cas de problèmes avec quantificateurs, les solveurs SMT doivent générer des instances pour les formules universellement quantifiées afin de progresser. Il est difficile en pratique de déterminer quelles sont les instances importantes, et les solveurs peuvent implémenter diverses approches pour répondre à ce problème. Un *trigger* est une annotation sur une séquence de quantificateurs indiquant aux solveurs comment les instancier.

Considérons par exemple l'axiome suivant, qui spécifie l'opérateur d'intersection entre deux ensembles :

$$\begin{aligned} \forall a, b, x : \ & \{x \in (a \cap b)\} \\ & \{a \cap b, x \in a\} \\ & \{a \cap b, x \in b\} \\ & x \in (a \cap b) \Leftrightarrow x \in a \wedge x \in b \end{aligned}$$

Nous indiquons les *triggers* entre accolades ; dans ce cas, trois *triggers* ont été choisis. Considérons le premier : le motif $x \in (a \cap b)$ indique au solveur SMT qu'il peut chercher les termes de cette forme. Lorsqu'un tel terme est détecté, l'assignation correspondante pour les variables x, a et b est utilisée pour générer l'instance de l'axiome. En l'occurrence, il s'agira de la formule qui donne la caractérisation de $x \in (a \cap b)$.

De nombreux solveurs SMT génèrent automatiquement leurs propres *triggers*, mais nous avons trouvé profitable de choisir les nôtres manuellement. Notre stratégie pour sélectionner des *triggers* repose sur l'hypothèse qu'une obligation TLA$^+$ typique ne requiert pas de raisonner sur des ensembles ou fonctions qui ne sont pas mentionnées. On peut donc se contenter de récupérer le maximum d'informations sur les termes connus, et en même temps rejeter les *triggers* susceptibles d'introduire de nouveaux ensembles et fonctions, ce qui complexifierait inutilement les problèmes.

Un autre aspect de notre stratégie consiste à anticiper le plus de situations de preuve possible à l'aide de *triggers*. Dans l'exemple précédent, le premier *trigger* permet essentiellement de simplifier une proposition $x \in (a \cap b)$. Or, il est parfois nécessaire d'inférer une proposition

$x \in (a \cap b)$ sans qu'elle soit explicitement mentionnée—typiquement parce qu'elle intervient à un niveau plus profond du raisonnement. Par exemple, la formule

$$S \cap Int = \emptyset \wedge 1 \in S \Rightarrow \text{FALSE}$$

est bien prouvable, mais le solveur doit en quelque sorte deviner l'étape $1 \in (S \cap Int)$ qui permet au raisonnement d'aboutir. Les deuxième et troisième *triggers* de notre exemple sont adaptés à cette situation, car il suffit de connaître un certain ensemble $a \cap b$ et un certain élément x de a ou b pour générer une instance pertinente.

Nous nous sommes efforcés d'appliquer des principes comme ceux que nous venons de décrire de façon aussi systématique que possible pour l'ensemble de l'axiomatisation. Les axiomes de théorie des ensembles sont traités de façon assez similaire ; certains axiomes doivent être reformulés pour placer tous les quantificateurs au début de la formule, afin qu'ils puissent recevoir une annotation. La théorie des fonctions de TLA$^+$ inclut le constructeur $[x \in S \mapsto e]$ et, étant donné une fonction f, les opérateurs DOMAIN f et $f[x]$ dénotant respectivement le domaine de f et l'application de f à x. Notre stratégie s'adate bien aux axiomes spécifiant ces symboles : nous évitons d'introduire de nouvelles fonctions, mais nous sélectionnons des *triggers* qui permettent de récupérer des informations sur les domaines et valeurs d'application des fonctions connues.

Évaluation de l'encodage SMT

L'évaluation comparée de notre encodage avec l'encodage précédent montre que notre approche ne mène pas à une perte d'efficacité en termes d'obligations résolues. Compte tenu du fait que notre implémentation est plus sûre d'utilisation, on peut affirmer que notre encodage améliore l'état de l'art, puisqu'il est moins risqué de l'utiliser.

Certaines obligations sont résolues par une version de l'encodage mais pas l'autre. Il est toujours difficile d'expliquer la raison pour laquelle l'un des deux encodages pourrait surpasser l'autre dans une situation donnée. L'approche par simplification semble parfois échouer parce qu'elle élimine des termes tout en laissant intact des quantificateurs à instancier avec ces mêmes termes. Notre approche par *triggers* a quelques faiblesses connues, notamment au niveau des axiomes d'extensionnalité, pour lesquels il est difficile de sélectionner des annotations pertinentes. Pour l'axiome d'extensionnalité ensembliste, nous nous contentons de générer des instances pour les égalités entre ensembles dans l'obligation initiale. Pour l'extensionnalité fonctionnelle, nous générons une instance pour toute paire de fonctions, ce qui mène à beaucoup d'instances non pertinentes dans certaines situations impliquant de nombreuses fonctions.

Perspectives

Cette thèse a permis de rendre TLAPS plus sûr d'utilisation, mais il serait possible d'aller encore plus loin dans la certification des preuves. Nous nous sommes efforcés de justifier la correction de nos deux encodages, mais nos preuves pourraient être erronées, ou notre implémentation pourrait contenir des erreurs. Une idée serait de chercher à certifier l'implémentation de l'encodage ; un assistant de preuves comme Coq semble adapté à cette tâche. Il s'agit toutefois d'un projet ambitieux, et une autre solution semble possible. Certains prouveurs automatiques permettent de récupérer une preuve lorsqu'une obligation a été prouvée.

Compte tenu du fait que notre encodage modifie assez peu les expressions, il semble possible de traduire ces preuves dans le langage de TLA$^+$ afin de les faire vérifier par un programme de confiance. C'est ce qui est actuellement fait avec le prouveur Zenon, dont la sortie peut être vérifiée par Isabelle/TLA$^+$. Notre encodage SMT pourrait permettre de reproduire cette fonctionnalité pour les solveurs SMT capables de produire des preuves.

Pour ce qui est d'améliorer l'encodage du point de vue de l'efficacité, la situation est moins claire. Certaines faiblesses de notre axiomatisation à base de *triggers* pourraient être corrigées, notamment au niveau des axiomes d'extensionnalité. L'axiomatisation pourrait aussi être étendue à d'autres fragments de TLA$^+$. Il semble particulièrement facile d'adapter la méthode pour exploiter l'arithmétique entière de SMT à l'arithmétique réelle. Nous avons formulé une stratégie pour sélectionner des *triggers* de manière plus ou moins systématique, la question d'automatiser la génération des *triggers* se pose donc. Il serait notamment intéressant de pouvoir générer des *triggers* pour les instances des schémas d'axiomes que nous générons lors de la réduction au premier ordre.

Contents

Chapter 1

Introduction

This thesis presents translations of TLA$^+$ formulas into the input logics of automated theorem provers. TLA$^+$ is a formal language combining temporal logic and set theory, used in the industry to specify and verify computer systems [48]. We implemented our translations into TLAPS, an interactive proof system that uses backend provers to automatically verify every step of the user's reasoning [19]. Special attention is paid to the soundness of each translation, as any mistake may compromise the integrity of TLAPS, letting users derive invalid statements with the system.

1.1 Context

Formal methods are a rigorous approach to the verification of software [54]. They are applied in diverse industrial fields such as avionics [73], cloud computing [58] and hardware design [33] to specify, check, and sometimes certify various systems. The methods by which programs can be verified are diverse as well and include model checking and theorem proving.

This thesis is about theorem proving in the context of TLA$^+$, a specification language aimed at distributed and concurrent systems [48]. The TLA$^+$ Proof System (TLAPS) is an interactive theorem prover whose purpose is to interpret proofs written in a TLA$^+$ environment, generate proof obligations corresponding to each step of the user's reasoning, and have these obligations proved by specialized automated theorem provers [19]. The logics understood by most provers are very different than TLA$^+$'s, therefore translations (encodings) of TLA$^+$ must be defined for the backends.

Specification with TLA$^+$

As a formal language, TLA$^+$ combines the Temporal Logic of Actions (TLA) with axiomatic set theory, more precisely the ZFC theory, named after the mathematicians Zermelo and Frænkel, and the axiom of choice [47, 48, 51]. It is a purely logical language; one is expected to define different predicates describing the states and transitions of a system in an abstract way. This is illustrated in Figure 1.1, where a simple system is defined from an initial state *Init* and a transition predicate *Next*. The variable *out* is temporal: its value changes over time, and its value in the next state is denoted out'. The temporal formula *Spec* specifies the system as a whole. It features a temporal connective to state that, for every transition of the system, either *Next* holds or *out* does not change.

MODULE *HelloWorld*

VARIABLE *out*

$Init \triangleq out = \text{“”}$

$Next \triangleq out = \text{“”} \land out' = \text{“Hello World!”}$

$Spec \triangleq Init \land \Box[Next]_{out}$

Figure 1.1: TLA$^+$ Specification of the “Hello World!” program

TLA$^+$ is close to the Z notation [74], the B method and its extension Event-B [1,2], which are all specification languages based on set theory. In its classical formulation, set theory extends first-order logic (FOL) with a membership relation $\in$ and several axioms. It is a very expressive theory; in fact, it is a suitable framework for any kind of mathematics, and is largely adopted by the mathematical community. In TLA$^+$, systems and their properties are expressed in the same mathematical language. For instance, the fact that *out* is a string of characters is expressed $out \in \text{STRING}$, and the fact that this property is an invariant of the system is stated

$$Spec \Rightarrow \Box(out \in \text{STRING})$$

It is a valid temporal formula: for every trace satisfying *Spec*, the variable *out* is indeed a string in all states.

Invariants such as this one are commonly referred to as *typing invariants*. It is important to verify them, because TLA$^+$ is an untyped language. This is in line with the choice of ZFC as a foundation, but unconventional in the context of computer science. In programming, statically-typed languages benefit from the fact that many type errors can be caught at compile time. In formalized mathematics, most existing systems are based on a variant of type theory. As examples, we mention Isabelle and the object logic Isabelle/HOL which are based on higher-order logic (HOL) [14, 60, 61], the proof assistant Coq which is based on dependent type theory [17, 18], and PVS which is based on predicate subtyping [67]. The Z Notation and B-Method are based on typed variants of set theory. The Mizar proof assistant takes untyped set theory as a foundation, but its syntax includes type annotations, and expressions must pass type-checking [38, 44, 84].

In a polemic article, Lamport argues in favor of untyped formalisms for specification [49]. Type systems restrict one's ability to specify things easily; rich type systems offer more expressiveness, but these languages are too complicated for non-experts to use. In contrast, it is easy to express typing conditions in untyped set theory. For instance, in ordinary mathematics, one is allowed to write

$$\forall x : x \in Real \land x \neq 0 \Rightarrow x/x = 1$$

This is a valid statement, despite the fact that it indirectly refers to nonsensical expressions, like $0/0$. The legitimacy of these expressions is not an issue in an untyped framework, because the typing condition does not hold when x is 0. Semantically, division is treated as a totally-defined symbol, specified for some input values, unspecified for the rest [39].

Interactive Proof with TLAPS

Once a TLA$^+$ specification is written and its properties expressed, one can attempt to verify it. There are two ways to perform verification in TLA$^+$. The first is model checking. The model checker TLC explores all the possible traces given by a temporal formula in an exhaustive way [85]. However, TLC is limited to specifications involving finite data and fixed parameters. For the general case, one has to write a proof and check it with TLAPS.

TLAPS is an *interactive theorem prover* (ITP), also called *proof assistant*. ITPs combine expressive mathematical languages, interactive environments for users to write proofs, and procedures that formally check every step of the user's reasoning. Proofs often involve many tedious verifications, which is why many ITPs rely on automated procedures to solve easy subgoals. Notably, there is a growing interest in integrating *automated theorem provers* (ATPs) with interactive environments. This integration comes with many challenges, as ATPs tend to specialize on particular domains of mathematical logic, and typically do not support logics as expressive as the ITPs [34]. One particularly successful application of ATPs for interactive proving is the Sledgehammer system for Isabelle/HOL, which uses relevance-filtering and an unsound encoding of HOL to quickly derive proofs for the desired goals [10, 62]. CoqHammer is a similar tool for Coq, although it is more difficult to translate Coq's dependent type theory into the input logics of ATPs [22]. TLAPS is not as complex a system, but it also relies on automated backends to verify proof obligations [19]. Given a proof script written in a dialect of TLA$^+$, TLAPS generates obligations corresponding to the important steps, and translates each obligation into the logics of the available backends.

Let us review the current backend provers of TLAPS. LS4 is a prover for propositional temporal logic [76]. No other backend supports temporal logic, but TLA$^+$ proofs are easily structured in such a way that the temporal part of the reasoning is contained in a single obligation. The Isabelle backend uses a direct translation to the object logic Isabelle/TLA$^+$ and attempts to prove obligations by using one of the basic automation tactic of Isabelle: auto, force and blast. Zenon is a tableaux-based prover for first-order logic [11]. One distinctive feature of the Zenon backend is that it can optionally produce detailed proofs in Isabelle's format. These proofs may then be checked automatically by Isabelle to increase trust in the proof system. Finally, the SMT backend uses an encoding into the SMT-LIB standard format for SMT solvers [6]. The SMT solvers available are CVC4 [5], veriT [12], and Z3 [25].

SMT is currently TLAPS's most efficient backend. SMT solvers are recognized for their ability to handle very large problems, and used by many ITPs to discharge the most tedious obligations [9, 29, 32]. They excel in contexts involving decidable theories of first-order logic, but do not cope well with quantifiers. The SMT encoding of TLAPS is designed with these strengths and limitations of SMT in mind [53, 81]. One can start from the idea of a *direct encoding* of TLA$^+$ for SMT solvers. To implement this encoding, it suffices to declare every primitive of TLA$^+$ as an *uninterpreted symbol* of SMT (a symbol with no meaning attached to it), then insert axioms into the problem and translate expressions faithfully. This hypothetical encoding suffers from at least three issues. SMT solvers are first-order, but TLA$^+$ includes a few second-order constructs, for example $\{x \in S : P(x)\}$. First-order instantiation is a difficult task for SMT solvers, which may struggle to find the relevant axiom instances. Finally, some symbols of TLA$^+$ have interpreted counterparts in SMT (typically, every operator of integer arithmetic), but the direct encoding translates them as uninterpreted symbols.

These problems have been tackled through the implementation of various preprocessing techniques. The core technique is rewriting, which effectively eliminates primitive TLA$^+$

symbols from the source obligation. The following rule eliminates an occurrence of $\cup$:

$$x \in (a \cup b) \longrightarrow x \in a \vee x \in b$$

However, there are many situations where nonprimitive symbols occur but no rule is applicable. Rewriting is then complemented by auxiliary techniques that modify the logical structure of obligations, enabling further rewritings. Optionally, the preprocessing phase may be optimized by type synthesis. This mechanism attempts to infer types for all subexpressions and annotates expressions accordingly. This enables many optimizations of the base rewriting system:

$$x + y \overset{x \in Int,\, y \in Int}{\longrightarrow} x +_{\mathsf{int}} y$$

where $+$ is TLA$^+$'s untyped addition, and $+_{\mathsf{int}}$ is SMT's interpreted addition. The rule above enables a translation of $x + y$ directly as an integer term in SMT's logic.

1.2 Motivation

This thesis continues the work of Vanzetto on the SMT encoding of TLAPS [53, 81]. We will refer to Vanzetto's implementation as the "original encoding", to distinguish it from our own work. In order to achieve good results, the original encoding relies on a complex preprocessing phase. This phase is powerful enough to actually solve some obligations before they are sent to the SMT solvers. As evidence for this, we report that, during our evaluation of the original SMT backend (which is detailed in Chapter 5), we found that for about 11% of all obligations, the goal had been reduced to TRUE during preprocessing.

While this kind of efficiency is desirable, we must also ensure that the encoding is sound. An unsound encoding may produce a valid SMT problem from an invalid TLA$^+$ obligation, letting users derive faulty statements with TLAPS. If an invalid statement is proved using a particular backend, the error may result from a bug in the ATP, or an error in the translation implemented in TLAPS. Some ATPs can output proofs along with their results, so that these proofs may be checked internally by the ITP. We already mentioned Zenon, whose output can be checked by Isabelle. Let us also mention an ongoing effort to reconstruct TLA$^+$ proofs from veriT's output [16, 68, 69].

However, the current SMT encoding carries an important part of the actual solving, and its output can hardly be exploited for debugging purposes, let alone proof reconstruction. Encoded proof obligations have often little in common with the source TLA$^+$ obligations, leaving us with the only option to inspect the source code of TLAPS when a bug is found.

Our primary concern is to increase trust in TLAPS, especially the SMT backend. The direct encoding would be a safe alternative to the current one, since it is very easy to prove sound. But it was never implemented fully, as it was assumed too inefficient. Our starting point is the idea of using the direct encoding as a basis for a more modular design, in which preprocessing could be turned off to obtain stronger soundness guarantees. We may not expect the direct encoding to be as efficient, but it would at least let us verify a portion of the current encoding's positive results.

1.3 Contributions

Two contributions have been made in the form of new translations of TLA$^+$ for ATPs. The first is a *direct encoding* of TLA$^+$ into HOL. The second translation is an *optimized encoding*

for SMT solvers. Many assets from the first encoding are reused to implement this new SMT encoding, which will effectively replace the original one in a future version of TLAPS. Each contribution has lead to a publication in an international conference [27, 28].

The direct encoding targets HOL, so that the second-order constructs and axiom schemas of TLA$^+$ can be easily encoded. To test this encoding, we added support for Zipperposition, a superposition prover that supports higher-order logic [20, 82, 83]. As we were working on this encoding, Zipperposition had just won the CASC competition in the THF division [77, 78]. Besides soundness, a higher-order backend is a worthwhile addition to TLAPS, because specific TLA$^+$ obligations do require higher-order unification to be solved. As example, one may take any proof that involves the principle of natural induction:

$$P(0) \wedge (\forall x \in Nat : P(x) \Rightarrow P(x+1)) \Rightarrow \forall x \in Nat : P(x)$$

The principle is formalized as a lemma of TLAPS's standard library. When it is invoked, the current goal must be unified with the higher-order variable P. Similar situations arise for general well-founded inductions, or whenever a proof involves user-defined recursive operators. Zipperposition is especially useful for these goals, as the only other backend supporting higher-order logic is Isabelle, whose basic automation tactic are often unable to handle mildly complex obligations.

Our original plans for the SMT encoding included a reimplementation of type synthesis and preprocessing. Instead, we found that most obligations could be solved by SMT with heuristics for the instantiation of TLA$^+$'s axioms. These heuristics are implemented by *triggers* indicating to the solver some terms to look for in order to derive relevant instances. Triggers are a popular solution for users of SMT wishing to stabilize custom theories [30, 50, 55]. They are pivotal in our approach. We use triggers to implement heuristics for reasoning in untyped set theory, and the efficacy of these heuristics for TLA$^+$ has been confirmed experimentally. Moreover, triggers extend the direct encoding with little need to change the translation itself, so soundness is not compromised by this technique. Only some adjustments are needed to support second-order constructs, integer arithmetic, and the axiom of set extensionality.

1.4 Outline

In Chapter 2, we present the necessary context about automated theorem proving (with a focus on SMT) and the specification language TLA$^+$. The first section is an overview of SMT's theory with its fundamental results and limits. Some generic definitions from the field of mathematical logic are given, in anticipation of the formal treatment of TLA$^+$'s logic in future chapters. The second section is an overview of TLA$^+$ and its tools. The model checker TLC is briefly introduced, followed by the proof assistant TLAPS. Working TLA$^+$ examples are presented to illustrate the usage of these tools.

In Chapter 3, we develop our formal definition of TLA$^+$'s constant fragment, including its syntax and semantics. The constant fragment includes TLA$^+$'s variant of first-order logic and set theory, but it does not contain the temporal aspects of the logic. In TLA$^+$, silly expressions such as $1 \cup$ "`foo`" have a denotation. We may not know what this expression denotes, but we know, for example, that it is a set, and that it equals "`foo`" $\cup\, 1$. We propose to formalize the primitive symbols of TLA$^+$ as uninterpreted symbols specified only by axioms. TLA$^+$ is thus formalized as a logic $\mathcal{L} + T$, where $\mathcal{L}$ is a generic logic providing the interpretations of

Boolean connectives (including quantifiers), and T a standard theory containing the axioms of set theory, functions, arithmetic, and more.

In Chapter 4, we define a direct encoding of TLA$^+$ into HOL. A higher-order logic is necessary to encode the second-order constructs of TLA$^+$ and axiom schemas. For instance, set comprehension $\{x \in S : e\}$ is viewed as a second-order application $\mathsf{setst}(S, \lambda x : e)$, and the full schema of comprehension from ZF is encoded as a single axiom. The direct encoding consists in two essential steps: first, an elementary pass over expressions to correct ambiguities regarding Boolean and non-Boolean expressions; second, the insertion of relevant axioms into the final problem. The translation itself preserves the structure of the original TLA$^+$ expressions. The direct encoding has been implemented in TLAPS, and the superposition prover Zipperposition added as a new backend. We compare the performances of Zipperposition and Isabelle on a large collection of TLA$^+$ proof obligations; our results demonstrate that the two backends have complementary strengths.

In Chapter 5, we extend the direct encoding to an SMT encoding for TLA$^+$. Most of the direct encoding's structure and implementation is reused for the SMT encoding; parts of the axiomatizations are revised. Axioms are reformulated and annotated with E-matching patterns (triggers), which have been chosen and refined through an empirical process. We illustrate this process through a simple but realistic TLA$^+$ proof obligation, exploring several options for triggers and observing their effects on the SMT procedure. We then derive some general principles underlying our strategy for trigger selection. A few peripheral topics about the axiomatization of TLA$^+$ for SMT are addressed: reduction to first-order logic and axiom schemas; axioms for using SMT's integer arithmetic; axioms for set extensionality. Finally, our version of the SMT encoding is compared with the original one of TLAPS. Despite the simpler design of our encoding, we achieve similar performances.

Chapter 2

Background

This chapter is divided into two sections. We first review the theory of automated theorem proving through the lens of satisfiability modulo theory (SMT). We then give an overview of TLA$^+$ and its main tools TLC and TLAPS. We conclude this chapter by a brief review of the SMT encoding currently implemented in TLAPS. That encoding will be referred to as the "previous" or "original" SMT encoding to distinguish it from our own version.

2.1 Automated Theorem Proving

Automated theorem proving is the design and study of computational procedures (algorithms) for deciding the validity of mathematical formulas. This presupposes two important notions: *validity* and *procedure*. The first is formally defined by mathematical logic [45], and the second by the theory of computability [21].

We assume some informal notion of algorithm and will use pseudo-code for our examples. For any formal problem $P(x)$ with parameter x, we call *decision procedure* an algorithm that takes x as input and answers either "yes" or "no", depending on whether $P(x)$ is true or false. A problem is called *decidable* if there is a decision procedure for it, otherwise it is *undecidable*. Some undecidable problems still admit *semi-decision procedures*, which always answer "yes" in the positive case, but in the negative case may answer "no" or never return an answer. These problems are called *semi-decidable*.

SMT solvers extend decision procedures for propositional logic to handle theories of first-order logic, including the general theory of quantifiers. We present the DPLL procedure for propositional logic, then the SMT approach for theories of first-order logic, focussing on the general problem of first-order instantiation.

In the context of interactive theorem proving, SMT solvers are a popular option to automatically handle large proof obligations [9,29,32]. Another important approach to automated theorem proving is *superposition provers*, which are based on extensions of the resolution inference rule [4, 66]. The treatment of first-order quantifiers is central in resolution-based systems, while SMT's strength is its support for decidable first-order theories.

2.1.1 The DPLL Procedure

For most studies concerning mathematical logic, the starting point is propositional logic (PL). As we present the syntax and semantics of PL, we introduce some general concepts

of mathematical logic. The truth of a propositional formula ϕ depends on the truth of the propositional variables it contains, and the problem of deciding, for a given ϕ, if there exists an assignment of propositional variables making ϕ true is called the SAT problem. The DPLL procedure is an algorithm for deciding the SAT problem [23]. Modern SAT solvers are based on an extension of DPLL called CDCL.

Propositional Logic

We define the generic notion of a *logic* to fix some important terminology. All the logics we will consider (propositional, first-order logic, TLA$^+$) fit into the following definition.

Definition 2.1.1 (Logic)**.** A *logic* is a triple $(L, M, \models)$ where L is a collection of formulas (the language), M a collection of interpretations, and $\models$ a relation on $M \times L$ called the satisfaction relation. For all $\phi \in L$ and $I \in M$, we say that I satisfies ϕ if $I \models \phi$. A formula is satisfiable if it is satisfied by some interpretation, otherwise it is unsatisfiable.

All the languages we will consider can be defined from BNF grammars, which are sets of formation rules describing how complex terms are formed through the combination of simpler ones. The definition of PL's syntax below is based on a BNF grammar.

Definition 2.1.2 (Syntax of Propositional Logic)**.** Propositional logic is parameterized by some collection Σ of propositional variables, called a *propositional signature*. The language of PL is given by the grammar

$$\phi ::= p \mid \text{TRUE} \mid \text{FALSE} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \qquad \text{where } p \in \Sigma$$

The collection of propositional formulas over the signature Σ is noted $P(\Sigma)$.

Grammars are ambiguous when certain terms of the language admit several formations. For instance, $\phi \wedge \psi \vee \rho$ is an ambiguous formula. We use parentheses to remove ambiguities when necessary, for instance we will write $(\phi \wedge \psi) \vee \rho$ or $\phi \wedge (\psi \vee \rho)$. To reduce the number of parentheses, we fix some priority rules between the propositional connectives. The connectives $\wedge$ and $\vee$ have priority over $\Rightarrow$ and $\Leftrightarrow$, so that $\phi \wedge \psi \Rightarrow \rho \vee \chi$ is read $(\phi \wedge \psi) \Rightarrow (\rho \vee \chi)$. The connective $\neg$ has priority over all the other connectives, so $\neg\phi \vee \psi$ is read $(\neg\phi) \vee \psi$ for example.

Definition 2.1.3 (Semantics of Propositional Logic)**.** Let Σ be a propositional signature. A Σ-valuation is a subset $\theta \subseteq \Sigma$. We define the satisfaction relation for all valuations θ and formulas ϕ by

$\theta \models p$	iff $p \in \theta$
$\theta \models \text{TRUE}$	
$\theta \not\models \text{FALSE}$	
$\theta \models \neg\phi$	iff $\theta \not\models \phi$
$\theta \models \phi \wedge \psi$	iff $\theta \models \phi$ and $\theta \models \psi$
$\theta \models \phi \vee \psi$	iff $\theta \models \phi$ or $\theta \models \psi$
$\theta \models \phi \Rightarrow \psi$	iff $\theta \models \phi$ implies $\theta \models \psi$
$\theta \models \phi \Leftrightarrow \psi$	iff $\theta \models \phi$ equals $\theta \models \psi$

Propositional logic over a signature Σ is defined by the language $P(\Sigma)$, the collection of all Σ-valuations, and the satisfaction relation just defined.

There is some additional terminology to consider regarding satisfiability. For theorem proving, we are especially interested in formulas that are true for *all* interpretations, or for classes of interpretations fixed by sets of formulas called theories.

Definition 2.1.4 (Theories)**.** Consider a generic logic with satisfaction relation $\models$. A theory is a collection of formulas T. T is satisfied by an interpretation I if all formulas of T are simultaneously satisfied by I. In that case, we call I a *model* of T, and we note $I \models T$. T is unsatisfiable if it has no model.

Let T be a theory. A formula ϕ is a logical consequence of T if ϕ is satisfied by all models of T. This is noted $T \models \phi$. A *tautology*, or *valid* formula, is a consequence of the empty theory. We note $\models \phi$ when ϕ is valid.

Example 2.1.5. Here are some tautologies of propositional logic (for all ϕ, ψ):

$$\phi \Rightarrow \phi$$
$$\neg\phi \vee \phi$$
$$\neg\neg\phi \Leftrightarrow \phi$$
$$\phi \wedge \psi \Leftrightarrow \psi \wedge \phi$$
$$\phi \vee \psi \Leftrightarrow \psi \vee \phi$$
$$\phi \wedge \psi \Rightarrow \phi$$
$$\phi \Rightarrow \phi \vee \psi$$
$$\neg(\phi \wedge \psi) \Leftrightarrow (\neg\phi \vee \neg\psi)$$
$$\neg(\phi \vee \psi) \Leftrightarrow (\neg\phi \wedge \neg\psi)$$

The satisfiability or validity of a given formula can be established by the method of *truth tables*. Consider the formula $\neg(p \Rightarrow q)$. There are only two propositional variables in that formula, p and q. Clearly, for all valuations θ, we only need to consider whether p and q are satisfied by θ. There are only four possibilities; for each of them, we can simply evaluate whether the formula is satisfied or not by following the recursive definition of $\models$. All four situations are represented by the table below, where 1 indicates satisfaction and 0 indicates non-satisfaction. We also indicate satisfaction for the subformula $p \Rightarrow q$.

p	q	$p \Rightarrow q$	$\neg(p \Rightarrow q)$
0	0	1	**0**
0	1	1	**0**
1	0	0	**1**
1	1	1	**0**

As this table shows, $\neg(p \Rightarrow q)$ is satisfiable, and the only valuation that makes it true is the one that contains p but not q. For a formula to be valid, it must be satisfied in all situations, so $\neg(p \Rightarrow q)$ is not valid.

The method of truth tables demonstrates a naive decision procedure for propositional logic. Formally, the problems "ϕ is satisfiable" and "ϕ is valid" for the parameter $\phi \in P(\Sigma)$ are both decidable. Moreover, there is a close correspondence between the two. Consider the equivalence

$$\models \phi \qquad \text{iff} \qquad \text{for all } \theta,\ \theta \not\models \neg\phi \qquad \text{iff} \qquad \neg\phi \text{ is unsatisfiable}$$

More generally,

$$T \models \phi \qquad \text{iff} \qquad \text{for all } \theta,\ \theta \models T \text{ implies } \theta \not\models \neg\phi \qquad \text{iff} \qquad T \cup \{\neg\phi\} \text{ is unsatisfiable}$$

For theorem proving, we are more interested in deciding validity than satisfiability. But solvers may be more oriented towards the problem of satisfiability. We may still use such solvers for deciding validity by checking that the *negation* of a goal is unsatisfiable, possibly in the context of a theory T.

SAT Solvers

The procedure of decision for PL based on truth tables requires going through all possible valuations for a given formula. If the input formula contains n propositional variables, the number of possibilities is 2^n. Some industrial applications involve problems with millions of parameters, so this naive procedure cannot be used realistically. Modern SAT solvers like Chaff [56], GRASP [72] or MiniSAT [31] are based on a procedure called DPLL [23]. The first step consists in preprocessing formulas into a more restrictive language [3].

Definition 2.1.6 (Conjunctive Normal Forms)**.** Let Σ be a propositional signature. We define literals, clauses and CNF formulas by the grammar

$$\begin{aligned} l &::= p \mid \neg p && \text{(Literals)} \\ C &::= \textsc{false} \mid l \vee C && \text{(Clauses)} \\ N &::= \textsc{true} \mid N \wedge C && \text{(CNF Formulas)} \end{aligned}$$

A formula ϕ is in conjunctive normal form (CNF) if it can be read as a CNF formula.

Two formulas ϕ and ψ are equisatisfiable when ϕ is satisfiable iff ψ is. We admit that for all formulas ϕ, there is a CNF formula $N(\phi)$ such that ϕ and $N(\phi)$ are equisatisfiable.

Example 2.1.7. Consider the formula

$$p \vee q \Rightarrow r$$

A possible CNF formula for it is

$$(\neg p \vee r) \wedge (\neg q \vee r)$$

One can verify that the equivalence of the two formulas is a tautology. Equivalently, one can compare the truth tables for the two formulas side to side and verify that they are satisfied by the same valuations:

p	q	r	$p \vee q$	$p \vee q \Rightarrow r$	$\neg p \vee r$	$\neg q \vee r$	$(\neg p \vee r) \wedge (\neg q \vee r)$
0	0	0	0	**1**	1	1	**1**
0	0	1	0	**1**	1	1	**1**
0	1	0	1	**0**	1	0	**0**
0	1	1	1	**1**	1	1	**1**
1	0	0	1	**0**	0	1	**0**
1	0	1	1	**1**	1	1	**1**
1	1	0	1	**0**	0	0	**0**
1	1	1	1	**1**	1	1	**1**

CNF formulas admit a simpler representation than general formulas. Consider the following logical equivalences for all ϕ, ψ and ρ:

$$\phi \vee \psi \Leftrightarrow \psi \vee \phi$$
$$(\phi \vee \psi) \vee \rho \Leftrightarrow \phi \vee (\psi \vee \rho)$$
$$\phi \vee \phi \Leftrightarrow \phi$$
$$\text{FALSE} \vee \phi \Leftrightarrow \phi$$

These laws express that $\vee$ is commutative, associative, and that FALSE is neutral for $\vee$. They justify the representation of clauses by finite sets of literals: the order of literals does not matter in a clause, nor does the duplication of literals; the clause FALSE can be represented by the empty set. For similar reasons, CNF formulas can be represented by sets of clauses; in that case, the empty set represents the formula TRUE.

A literal l is of the form p or $\neg p$ where p is a propositional variable. Let us write $\bar{l}$ to refer to whichever case is the opposite for l. The DPLL procedure takes a CNF formula as input and incrementally takes decisions for all literals, simplifying the current formula for each decision. That process of simplification is called *propagation*.

Definition 2.1.8 (Propagation)**.** Let N be a CNF formula and l a literal. The *propagation* of l into N is the CNF formula $N[l]$ obtained by removing from N all clauses containing l, and removing $\bar{l}$ from the remaining clauses. Clearly, neither l nor $\bar{l}$ occurs in $N[l]$.

If N is satisfiable, then for all l either $l \wedge N[l]$ or $\bar{l} \wedge N[\bar{l}]$ is satisfiable. A simple procedure is implemented by selecting an l, recording the decision and testing for the satisfiability of $N[l]$; if that formula is unsatisfiable, then we test $N[\bar{l}]$ instead. Unsatisfiability is easily detected by searching for the empty clause in derived formulas. This simple procedure still requires testing for all 2^n possibilities in the unsatisfiable case. DPLL optimizes the search through *unit propagation* to prune the decision tree.

Definition 2.1.9 (DPLL)**.** Figure 2.1 presents the DPLL procedure in pseudo-code. The sub-procedure select returns a literal occurring in N but not in *decisions* or *inferred*.

The DPLL algorithm selects literals to set and records them in a sequence *decisions*. Each decision is propagated through the current formula. If the formula is reduced to the empty CNF formula TRUE, then the decision trail is a valuation that satisfies the original formula; if the current formula contains the empty clause FALSE (this is called a *conflict*), then it is unsatisfiable under the current constraint, so decisions must be undone. In the algorithm above, backtracking to an earlier decision is achieved by using recursion. DPLL optimizes the search by managing another sequence *inferred*. In some situations, the right value for a literal can be deduced from the formula and the current constraint. If deciding l resulted in a conflict, then $\bar{l}$ is necessarily true. If a *unit clause* reduced to the literal l occurs in the formula, then l has to be true. The latter optimization is called *unit propagation*.

Example 2.1.10. Consider the following set of clauses:

$$p \vee q \vee r, \quad \neg p \vee q, \quad \neg p \vee \neg q \vee r, \quad \neg q \vee \neg r$$

```
DPLL_main(decisions, inferred, N):
  // Unit propagation
  for all {l} ∈ N:
    inferred := cons(l, inferred)
    N := N[l]

  if N = ∅ then:
    return SAT(decisions + inferred)
  if ∅ ∈ N:
    return UNSAT
  else:
    l := select(decisions, inferred, N)
    result := DPLL_main(cons(l, decisions), inferred, N[l])
    if result is UNSAT then:
      result := DPLL_main(decisions, cons(l̄, inferred), N[l̄])
    return result

DPLL(φ):
  N := cnf(φ)
  decisions := nil
  inferred := nil
  DPLL_main(decisions, inferred, N)
```

Figure 2.1: DPLL Procedure

We present below a possible run of the DPLL algorithm. Inferred literals either result from unit propagation (UP) or backtracking after a conflict (BT).

$p \vee q \vee r$,	$\neg p \vee q$,	$\neg p \vee \neg q \vee r$,	$\neg q \vee \neg r$	(Init)
	q,	$\neg q \vee r$,	$\neg q \vee \neg r$	(Decide p)
		r,	$\neg r$	(Infer q (UP))
			FALSE	(Infer r (UP))
$p \vee q \vee r$,	$\neg p \vee q$,	$\neg p \vee \neg q \vee r$,	$\neg q \vee \neg r$	(Backtrack)
$q \vee r$,			$\neg q \vee \neg r$	(Infer ¬p (BT))
			$\neg r$	(Decide q)
				(Infer ¬r (UP))

There are no clauses left in the last state so the formula is satisfiable. The solution is given by the set of literals $\{\neg p, q, \neg r\}$. Notice that because q is decided by unit propagation the first time, the second backtracking step reverts to the decision of p.

Modern SAT solvers are based on an extension of DPLL called *conflict-driven clause learning* (CDCL) [43, 72]. Whenever the algorithm encounters a conflict (the empty clause is derived after propagation), the current trail of decisions is analysed to derive a new clause. Suppose the analysis determines that the decisions $l_1, \ldots, l_n$ lead to the conflict. Then the

original formula must entail $\neg(l_1 \wedge \cdots \wedge l_n)$, which corresponds to the clause $\overline{l_1} \vee \cdots \vee \overline{l_n}$. By adding this *learned clause* to the original problem, the risk of replaying the same conflict is removed. This technique can be combined with *backjumping* to backtrack to earlier decisions in the trail.

2.1.2 Satisfiability Modulo Theories

Propositional logic has interesting applications but it is not expressive enough for most mathematical theories. First-order logic (FOL) extends PL in several aspects. The propositional variables are replaced by statements expressing relations between different terms; quantifiers enable the formal expression of universal and existential statements. The satisfaction relation of FOL involves *domains* of objects to interpreted terms, which makes first-order satisfiability inherently more complex than propositional satisfiability. Nevertheless, some theories of FOL are decidable, and validity in pure FOL is semi-decidable. SMT solvers specialize in deciding the satisfiability of formulas for first-order theories of interest.

First-order Logic

Definition 2.1.11 (First-order Terms)**.** A term signature is a collection Σ of symbols with assigned numbers called arities. Let X be some infinite collection of variable symbols. We define the collection of terms $T(\Sigma, X)$ by the grammar

$$t ::= x \mid f(t, \ldots, t)$$

where $x \in X$ and $f \in \Sigma$. An application $f(t_1, \ldots, t_n)$ is well-formed only if the arity of f is n. If c is a symbol with arity 0, the application of c to zero arguments is simply noted c.

The collection of variable symbols X is usually fixed and omitted from all notations, so we usually note $T(\Sigma)$ for a collection of terms.

Definition 2.1.12 (First-order Logic)**.** A first-order signature is an union $\Sigma = \Sigma_F \uplus \Sigma_P$ where Σ_F is a term signature and Σ_P a collection of predicate symbols with arities. The language of first-order logic over Σ is defined by the grammar

$$\begin{aligned}\phi ::= {} & p(t, \ldots, t) \mid t = t \\ & \mid \textsc{true} \mid \textsc{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \\ & \mid \forall x : \phi \mid \exists x : \phi\end{aligned}$$

where $x \in X$, $t \in T(\Sigma_F)$, and $p \in \Sigma_P$. The application $p(t_1, \ldots, t_n)$ is well-formed only if the arity of p is n. An atomic proposition is a formula $p(t_1, \ldots, t_n)$ or $t_1 = t_2$. The collection of atomic propositions is noted $A(\Sigma)$. The collection of formulas is noted $F(\Sigma)$.

Let D be some collection called *domain*. A valuation is a function $\theta : X \to D$. An interpretation I consists of two families $(f^I)_{f \in \Sigma_F}$ and $(p^I)_{p \in \Sigma_P}$ such that $f^I \in D^n \to D$ when the arity of f is n, and $p^I \subseteq D^n$ when the arity of p is n. We define the evaluation of terms as a function $[\![\cdot]\!]^I_\theta$ by

$$\begin{aligned}[\![x]\!]^I_\theta &\triangleq \theta(x) \\ [\![f(t_1, \ldots, t_n)]\!]^I_\theta &\triangleq f^I([\![t_1]\!]^I_\theta, \ldots, [\![t_n]\!]^I_\theta)\end{aligned}$$

For all $t \in T(\Sigma_F)$, the value $[\![t]\!]^I_\theta$ is an element of D. For all valuations θ, variables x and values $v \in D$, let θ^x_v be the valuation that reassigns x to v. We now define the satisfaction relation $\models$ as a ternary relation by

$$\begin{aligned}
&I, \theta \models p(t_1, \ldots, t_n) && \text{iff } ([\![t_1]\!]^I_\theta, \ldots, [\![t_n]\!]^I_\theta) \in p^I \\
&I, \theta \models t_1 = t_2 && \text{iff } [\![t_1]\!]^I_\theta = [\![t_2]\!]^I_\theta \\
&I, \theta \models \forall x : \phi && \text{iff } I, \theta^x_v \models \phi \quad \text{for all } v \in D \\
&I, \theta \models \exists x : \phi && \text{iff } I, \theta^x_v \models \phi \quad \text{for some } v \in D
\end{aligned}$$

The rules involving propositional connectives are omitted as they are analogous to the ones for propositional logic.

We note $Qx : \phi$ for a formula $\forall x : \phi$ or $\exists x : \phi$. In such formulas, the variable x is said to be bound, and ϕ is called the scope of the binder Qx. Any variable in a first-order formula is bound by at most one quantifier, otherwise it is called free. A formula with no free variables is called ground. The interpretation of any formula does not depend on the values $\theta(x)$ such that x is not free. In the case of ground formulas, the interpretation does not matter on θ at all, so we simply note $I \models \phi$ when ϕ is ground and $I, \theta \models \phi$ for some arbitrary θ.

Chains of quantifiers are usually shortened by writing $\forall x, y, z : \phi$ and $\exists x, y, z : \phi$. The scope of a quantifier is assumed to be the formula that extends until the next enclosing parenthesis. For instance, $\forall x : p(x) \Rightarrow q(c)$ is read as $\forall x : (p(x) \Rightarrow q(c))$.

Example 2.1.13. Here are some tautologies of FOL:

$$\begin{aligned}
(\neg \forall x : p(x)) &\Leftrightarrow \exists x : \neg p(x) \\
(\neg \exists x : p(x)) &\Leftrightarrow \forall x : \neg p(x) \\
(p \wedge \forall x : q(x)) &\Leftrightarrow \forall x : p \wedge q(x) \\
((\forall x : p(x)) \Rightarrow q) &\Leftrightarrow \exists x : p(x) \Rightarrow q \\
(\exists y \forall x : p(x, y)) &\Rightarrow \forall x \exists y : p(x, y) \\
(\forall x : p(x, f(x))) &\Rightarrow \forall x \exists y : p(x, y)
\end{aligned}$$

Validity for arbitrary formulas of first-order logic is undecidable [80]. This classical result relies on a reduction of the halting problem (the problem of deciding termination for arbitrary procedures) to the problem of validity for FOL. See for example Paragraph 45 in Kleene [45]. However, some particular theories are known to be decidable. The most popular example is *Presburger arithmetic*, also called linear integer arithmetic.

Example 2.1.14 (Presburger Arithmetic)**.** The signature of linear arithmetic contains the constant symbol Z (zero), the unary symbol S (successor) and the binary symbol $+$ (used in infix notation). It does not include predicates, but comparisons $\leq$ and $<$ can be defined by first-order formulas. The theory is described by the axioms

$$\begin{aligned}
&\forall x : S(x) \neq Z \\
&\forall x, y : S(x) = S(y) \Rightarrow x = y \\
&\forall x : x + Z = x \\
&\forall x, y : x + S(y) = S(x + y) \\
&P(Z) \wedge (\forall x : P(x) \Rightarrow P(S(x))) \Rightarrow \forall x : P(x) \qquad \text{for all formulas } P(x)
\end{aligned}$$

The last line is not a single axiom but an infinite collection of axioms called a schema.

Let T be the collection of axioms for linear arithmetic. Then the problem $T \models \phi$, where the parameter ϕ is a formula in the language of linear arithmetic, is decidable. The proof of this results relies on a quantifier-elimination technique [75].

Other examples of decidable theories include the theory of *real closed fields* or the theory of *fixed-size bitvectors* [8,15]. Expressive theories tend to be undecidable; if one extends linear integer arithmetic with a symbol and axioms for multiplication, one obtains *non-linear integer arithmetic*, which is undecidable. It follows that any extension of linear integer arithmetic in which multiplication can be defined is also undecidable [40].

The generic theory of first-order logic, also called the *empty theory* because no fixed signature or axioms are given, is undecidable. However, under some restrictions, it is decidable [36]. A notable decidable class of FOL is the class of *effectively propositional* formulas (EPR) [26]. While undecidable in general, the empty theory is semi-decidable. This can be established through a classical result by Herbrand [13,41], which is stated below.

A formula is called *quantifier-free* when it does not contain quantifiers. Let ϕ be quantifier-free and let $x_1, \ldots, x_n$ be its free variables. Then the *universal closure* of ϕ is the formula $\forall x_1, \ldots, x_n : \phi$, noted $\forall^\star \phi$. When ϕ is quantifier-free, $\forall^\star \phi$ is said to be *universal* and *prenex*. A *substitution* is a function $\sigma : X \to T(\Sigma)$; a substitution is *ground* if all terms $\sigma(x)$ are ground. The result of applying a substitution σ to ϕ is the formula $\phi\sigma$ (each free variable x is replaced by $\sigma(x)$). Then an *instance* of $\forall^\star \phi$ is a ground formula $\phi\sigma$.

Theorem (Herbrand)**.** *For all ϕ quantifier-free, $\forall^\star \phi$ is unsatisfiable iff there exists a finite, unsatisfiable set of instances $\{\phi\sigma_1, \ldots, \phi\sigma_n\}$.*

Herbrand's Theorem gives the basis for a semi-decision procedure for the empty theory. In order to establish this, we admit two additional results.

Lemma. *Every first-order formula ϕ admits an equisatisfiable prenex formula $\forall^\star \psi$.*

Lemma. *A quantifier-free formula ϕ is an element of $P(A(\Sigma))$. Suppose ϕ is quantifier-free and without equalities. Then ϕ is satisfiable as a first-order formula iff it is satisfiable as a propositional formula.*

The first lemma is proved by applying a series of logical laws (some of them displayed in Example 2.1.13) to move quantifiers up the structure of ϕ, then apply a technique called *Skolemization* to eliminate existential quantifiers [3]. The second lemma involves building a first-order model from a propositional one, called a *Herbrand model*. Its generalization to formulas with equalities is not trivial, but can be done if satisfiability in PL is constrained by axioms for equality [70].

The naive semi-decision procedure for FOL proceeds as follows. The input is a formula ϕ. To decide $\models \phi$, we check for the unsatisfiability of $\neg\phi$. There is a formula $\forall^\star \psi$ that is satisfiable iff $\neg\phi$ is. We enumerate all finite sets of ground instances $\{\psi\sigma_1, \ldots, \psi\sigma_n\}$, testing for the *propositional* satisfiability of every set (using DPLL for example). If a set is satisfiable, we resume with the next set of instances. If a set is unsatisfiable, then it is also unsatisfiable as a set of first-order formulas, and by Herbrand's Theorem $\forall^\star \psi$ is unsatisfiable and thus $\models \phi$. Now, assuming $\models \phi$, then an unsatisfiable set of instances must exist, so the procedure will terminate.

SMT Solvers

SMT solvers were originally created to solve logical constraints expressed in the quantifier-free (QF) fragment of FOL. These constraints may involve interpreted domains and symbols (arithmetic, arrays, fixed size bitvectors, etc.), or uninterpreted symbols over unspecified domains. Two important extensions of this basic model are the possibility to combine theories whose signatures do not intersect, and support for first-order quantifiers. The adequate language for SMT problems is a *multi-sorted first-order logic* (MS-FOL), which extends the syntax of FOL with type annotations and restricts the combination of symbols to prohibit meaningless expressions.

There exist several approaches to implementing SMT. The one we describe here is the *lazy procedure* based on the popular DPLL(T) architecture [35, 59]. As the name suggests, DPLL(T) combines a DPLL procedure with a solver for the parameter theory T, referred to as a T-solver. Besides checking for satisfiability in T, the T-solver must be able to provide *explanations* for unsatisfiable set of literals, in order to cooperate with the propositional solver. Some problems involve several theories with their corresponding solvers; given a set of T_i-solvers such that the signatures of distinct theories T_i do not overlap, there is a procedure for combining all solvers into a single one for the union of all theories [57, 71]. The theory of *equality and uninterpreted functions* (EUF), which is decidable, plays a central role in this framework, as the equality symbol is common to all theories. Procedures for solving problems of EUF are generally based on congruence closure algorithms [70].

Example 2.1.15. This example will illustrate the cooperation between a SAT solver implementing clause learning and a solver for linear integer arithmetic. Consider the formula

$$a - b = 0 \wedge \big(f(a) = f(b) \Rightarrow P(a) \wedge \neg P(b)\big)$$

where a and b are integer constants and f is an uninterpreted function of integers. From the SAT solver's point of view, atoms are abstracted as propositional variables; a possible representation in CNF form is

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge (\neg p_2 \vee \neg p_4)$$

with the definitions

$$p_1 \triangleq a - b = 0 \qquad p_3 \triangleq P(a)$$
$$p_2 \triangleq f(a) = f(b) \qquad p_4 \triangleq P(b)$$

Naive CNF transformations may result in exponential increases of the lengths of the formulas. Modern solvers implement a CNF transformation called the Tseitin transformation, which maintains a linear increase in size [79]. That transformation introduces variables for subformulas; in our example, a variable would be introduced to represent $P(a) \wedge \neg P(b)$.

The propositional formula is satisfied by the valuation $\{p_1, \neg p_2, p_3, p_4\}$. Suppose the SAT solver returns this candidate model to the T-solver, where T combines linear integer arithmetic with EUF. The T-solver will find the subset $\{p_1, \neg p_2\}$ to be unsatisfiable. Indeed, from $a - b = 0$ it follows (by arithmetic) that $a = b$, and then $f(a) = f(b)$ (by congruence). This contradiction is represented by the clause $\neg p_1 \vee p_2$, which is sent back to the SAT solver. The new problem is

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge (\neg p_2 \vee \neg p_4) \wedge (\neg p_1 \vee p_2)$$

The SAT solver will return the model $p_1, p_2, p_3, \neg p_4$ (this is the only solution as it can be found directly by unit propagation). The T-solver will again find a contradiction as $a = b$ entails $\neg(P(a) \wedge \neg P(b))$. The unsatisfiable subset is $\{p_1, p_3, \neg p_4\}$, so the clause $\neg p_1 \vee \neg p_3 \vee p_4$ is added to the propositional problem:

$$p_1 \wedge (\neg p_2 \vee p_3) \wedge (\neg p_2 \vee \neg p_4) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3 \vee p_4)$$

The SAT solver will find that problem to be unsatisfiable, which means the original formula is unsatisfiable.

The cooperation of a SAT solver with a T-solver as we just described defines a *ground* SMT solver. When a problem contains quantifiers $\forall$ or $\exists$, this framework is typically extended with an *instantiation module*. The combination of such a module with a ground SMT solver defines a general SMT solver. Here is a simplified view of this extended architecture: the problem with quantifiers can be represented as a problem $E \uplus Q$ where E is a set of ground formulas, and Q a set of formulas with quantifiers. Usually the elements of Q are assumed to be in universal prenex form $\forall^\star \phi$, although solvers may handle quantifiers differently for efficiency reasons [3]. The ground SMT solver may find E to be unsatisfiable (in which case the original problem is unsatisfiable). If E is satisfiable, then the instantiation module is called. Three outcomes are possible: either the satisfiability of $E \uplus Q$ can be verified (then the answer is "sat"); if it cannot be verified, then a number of *instance lemmas* may be produced and added to the ground problem, restarting the whole process; if the satisfiability of $E \uplus Q$ cannot be verified, and no instance lemmas are produced, the SMT solver will return "unknown" [63].

In the context of automated theorem proving, our goal is generally to check for unsatisfiability. Therefore, we will focus on strategies to generate instance lemmas. The general form of an instance lemma is $(\forall^\star \phi) \Rightarrow \phi\sigma$ where $\forall^\star \phi$ is an element of Q.

Example 2.1.16. The following example is taken from Reynolds et al [63]. We consider the set of formulas

$$\neg P(a) \wedge \neg P(b) \wedge P(c) \wedge \neg R(b) \wedge \big(\forall x : P(x) \vee R(x)\big)$$

where a, b, c are uninterpreted constants and P, R are uninterpreted predicates. As a propositional CNF formula, this problem is represented

$$\neg p_1 \wedge \neg p_2 \wedge p_3 \wedge \neg p_4 \wedge q$$

with the definitions

$$\begin{aligned} p_1 &\triangleq P(a) & \qquad p_4 &\triangleq R(b) \\ p_2 &\triangleq P(b) & \qquad q &\triangleq \forall x : P(x) \vee R(x) \\ p_3 &\triangleq P(c) \end{aligned}$$

The problem is partitioned into the ground problem $E \triangleq \{\neg p_1, \neg p_2, p_3, \neg p_4\}$ and the set $Q \triangleq \{q\}$. E has a trivial (and unique) solution, so the instantiation module is called.

Assuming the only possible terms are a, b and c, there are three possible instance lemmas:

$$\begin{aligned} q &\Rightarrow P(a) \vee R(a) \\ q &\Rightarrow P(b) \vee R(b) \\ q &\Rightarrow P(c) \vee R(c) \end{aligned}$$

The second instance, in particular, reduces to the clause $\neg q \vee p_2 \vee p_4$ by propositional abstraction. Suppose our instantiation strategy selects that lemma. Then the new propositional problem is

$$\neg p_1 \wedge \neg p_2 \wedge p_3 \wedge \neg p_4 \wedge q \wedge (\neg q \vee p_2 \vee p_4)$$

That problem is unsatisfiable, so the solver returns "unsat".

First-order instantiation is a difficult problem, and several strategies for selecting instances exist. Most SMT solvers that support quantifiers implement a combination of them. Some strategies are *complete*, meaning that they will always enable progress towards unsatisfiability if the formula is indeed unsatisfiable. Let us review the four main approaches to instantiation for SMT.

Of all methods the *enumerative approach* is perhaps the simplest to explain [37, 64]. The finite sets of ground terms $\{t_1, \ldots, t_n\}$ can be enumerated. Then the enumeration of all the corresponding substitutions results in a complete instantiation procedure, as a straightforward application of Herbrand's Theorem. For the example above, the enumeration might consider the terms a, b and c for the variable x, in that order. The instance $P(a) \vee R(a)$ does not make the problem unsatisfiable, but $P(b) \vee R(b)$ does. This method can be made more effective by choosing an appropriate ordering of terms, or basing the search on a refined version of Herbrand's Theorem.

If our goal is to prove that a problem is unsatisfiable, then the *conflict-based approach* is a good option [63, 65]. This method looks for instances that contradict the ground problem, such as $P(b) \vee R(b)$ for our example. The other instances are ignored. Successful generation of conflicting instances forces the SAT solver to find a new propositional model, enabling progress. The downside is that finding conflicts is expensive, and may result in too few instances in some cases.

If our goal is to prove that a problem is satisfiable, then *model-based quantifier instantiation* (MBQI) might be a better choice [36, 63]. Consider again the example from above; notice that the instance $P(a) \vee R(a)$, while not contradicting the ground problem E, is not logically entailed by E either. MBQI targets instances that contradict candidate models for E. For instance, a possible model for E might interpret R as the predicate that is always false—that interpretation is possible because all that is known about R is that $\neg R(b)$ is true. But $P(a) \vee R(a)$ contradicts that interpretation (because $\neg P(a)$ holds) so it is generated. MBQI is complete for some fragments of FOL, but it is difficult in general to construct useful models in non trivial satisfiable cases.

The last approach is *E-matching*, which is the most widely used approach for instantiation in SMT [24, 63]. Quantified formulas are annotated with patterns, which are usually noted between curly-braces:

$$\forall x_1, \ldots, x_n : \{p_1, \ldots, p_k\} \ \phi$$

The free variables occurring in the patterns p_i must be exactly the variables $x_1, \ldots, x_n$. Let $t_1, \ldots, t_n$ be terms and σ a substitution such that, for all i, $E \models p_i\sigma = t_i$ in the theory of EUF. Then $(t_1, \ldots, t_n)$ is called a *trigger*, and σ can be used to produce an instance lemma. In general, triggers are searched for in the ground problem E. For instance, assuming the pattern $\{P(x)\}$ is chosen for our example, then the terms $P(a)$, $P(b)$ and $P(c)$ are all triggers, and three instances are generated. The procedure for detecting triggers is implemented in an effective way by modern SMT solvers [24]. However, the problem of selecting patterns for the quantified formulas is not as easy, and SMT solvers rely on heuristics to generate them [50].

Nevertheless, E-matching is a popular technique among users of SMT, who have the option to select their own triggers for custom axiomatizations [55]. A recurrent problem with the E-matching approach is that it tends to generate too many instances, even sometimes resulting in matching loops (new instances triggering more instances in infinite loops).

This concludes our overview of automated reasoning and SMT. The next part of this chapter concerns TLA$^+$, which is a logic based on unsorted set theory. Set theory is a theory of first-order logic, and TLA$^+$ also includes symbols from specific theories like integer arithmetic or functions. The high expressiveness of TLA$^+$ makes it challenging to encode for SMT solvers, which are better suited for quantifier-free problems. Previous attempts to encode TLA$^+$ for SMT relied on preprocessing techniques to eliminate primitive constructs, in particular set-theoretic ones. An important achievement of this thesis is the application of a direct encoding, complemented by axioms with custom E-matching patterns, to handle many proof obligations that were only solved with the help of preprocessing before.

2.2 TLA$^+$

TLA$^+$ is a specification language for modeling computer systems, in particular concurrent and distributed systems [48, 52]. It is based on a variant of Linear Temporal Logic called the Temporal Logic of Actions (TLA), and unsorted set theory. As a formal method, TLA$^+$ is similar to the Z notation or Event-B, which are also based on the language of set theory. The core idea underlying TLA$^+$ is that first-order logic with temporal modalities is a suitable framework for modeling systems, combining specifications, and refining abstract specifications to lower levels; untyped set theory gives the language more expressiveness [47, 49].

The name "TLA$^+$" only refers to the language. Several tools exist to help users with the verification of their TLA$^+$ specifications. The main tools are the model checker TLC, which interprets specifications to directly check that they satisfy desired properties [85], and the interactive proof assistant TLAPS, which uses external solvers to mechanically verify proofs written in TLA$^+$'s syntax [19]. Some other tools of interest are the symbolic model checker Apalache [46], PlusCal for translating algorithms into TLA$^+$ specifications, and the TLA$^+$ Toolbox, an Eclipse-based IDE that integrates TLC, TLAPS, and PlusCal.

In this section, we present TLA$^+$ and its semantics (Section 2.2.1), demonstrate the basic usage of TLC to verify safety and liveness properties (Section 2.2.2), and introduce the proof syntax and TLAPS (Section 2.2.3). Our work focusses on TLAPS, but it is still worth knowing about TLC, which is useful for verifying finite specifications quickly. Users wanting to check proofs with TLAPS are incited to invoke TLC first, to verify that their specifications are correct at least in the finite case. We will finish this section with a look at the current SMT encoding of TLAPS, which is highly optimized and able to cope with very large proof obligations [53, 81].

For a formal presentation of TLA$^+$'s syntax and semantics, we refer the reader to [51]. Our own formalism (Chapter 3) will not include the temporal aspects of TLA$^+$'s logic, for reasons that we will make clearer in the present chapter.

2.2.1 Specifying Systems

The language of TLA$^+$ is based on first-order predicate logic; it includes the usual connectives:

$$\text{TRUE}, \quad \text{FALSE}, \quad \neg A, \quad A \Rightarrow B, \quad A \wedge B, \quad A \vee B, \quad \forall x : P(x), \quad \exists x : P(x)$$

MODULE *SimpleClock*

EXTENDS *Naturals*
VARIABLE *hour*

$$Init \triangleq hour \in 1 \,..\, 12$$

$$Increment \triangleq \begin{array}{l} \land\ hour < 12 \\ \land\ hour' = hour + 1 \end{array}$$

$$Reset \triangleq \begin{array}{l} \land\ hour = 12 \\ \land\ hour' = 1 \end{array}$$

$$Next \triangleq \begin{array}{l} \lor\ Increment \\ \lor\ Reset \end{array}$$

$$Spec \triangleq Init \land \Box[Next]_{hour}$$

Figure 2.2: Specification of a Simple Clock (Specifying Systems)

Two extensions of predicate logic are combined to obtain TLA$^+$. The first extension is set theory, which includes several primitive operators:

$$\in, \quad \{e_1, \ldots, e_n\}, \quad \text{UNION } S, \quad \text{SUBSET } S, \quad \{x \in S : P(x)\}, \quad \{F(x) : x \in S\}, \quad \ldots$$

The operators displayed above are for (in that order) set membership, enumeration set, union of the elements of a set, power set, set comprehension and set replacement. The second extension is TLA. It extends the syntax of FOL with the prime operator $'$, modalities, and quantification over behaviors (flexible variables):

$$x', \quad \Box A, \quad \Diamond A, \quad \forall x : P(x), \quad \exists x : P(x)$$

We introduce the semantics of TLA$^+$ through a practical example: the *SimpleClock* specification, which is inspired by a similar example from TLA$^+$'s reference manual [48].

Specification of a Clock

We specify a clock that tells what *hour* it is. Initially, the hour is some undetermined number between 1 and 12. A clock changes its state by either incrementing the hour value, or resetting it to 1. Which action is taken depends on the current hour (a reset happens iff the current hour is 12). The TLA$^+$ specification for this system consists of a single TLA$^+$ module, which is shown in Figure 2.2.

Let us describe the different parts of this module. The header simply contains the name of the module. It is immediately followed by the names of the modules to import—here, only *Naturals* is imported, because some arithmetical operators are needed. Then we find declarations, in this case only one for the variable *hour*. The Temporal Logic of Action admits two kinds of variables: *flexible* variables take different values across time, while *rigid* variable have a fixed value. The keyword VARIABLE introduces a flexible variable; there are no rigid variables in this specification.

The rest of the specification is just a list of defined expressions, starting with

$$Init \triangleq hour \in 1 \,..\, 12$$

The primitive expression $1\,..\,12$ refers to the set $\{i \in Int : 1 \leq i \wedge i \leq 12\}$. Since *hour* is a flexible variable, the truth of *Init* depends on the current state of execution, and we call *Init* a *state predicate*. A state is simply an assignment of values to the flexible variables of the specification. We represent the state in which *hour* is assigned the value 5 by

$$\boxed{hour = 5}$$

The state predicate *Init* is satisfied by this state. If *hour* is instead assigned the value 0, or 13, or $\sqrt{2}$, then *Init* is not satisfied. TLA$^+$ is not a typed language, so the semantics does not assume distinct domains for interpreting variables. Outside of any context, the value of *hour* could be a natural, a real, a set, a Boolean, or something else entirely.

The definition of *Increment* makes use of TLA$^+$'s notation for multi-line conjunctions; the expressions prefixed by $\wedge$ on the same level of indentation are part of the same conjunction. The definition is equivalent to

$$Increment \triangleq hour < 12 \wedge hour' = hour + 1$$

Multi-line disjunctions as in the definition of *Next* are also allowed. The presence of a primed variable makes *Increment* a *transition predicate*, also called an *action*. Actions express the relationship between the current state and the next state. For all state expressions e, the expression e' denotes the value of e in the next state of the trace; so $hour'$ refers to the next value of *hour*. To evaluate *Increment*, two states are needed. For instance, if $hour = 9$ in the current state and $hour = 10$ in the next state, we represent this situation by

$$\boxed{hour = 9} \longrightarrow \boxed{hour = 10}$$

Increment is of course satisfied by that pair of consecutive states. In the same spirit, the action *Reset* is satisfied by this transition only:

$$\boxed{hour = 12} \longrightarrow \boxed{hour = 1}$$

The action *Next* is simply the disjunction of *Increment* and *Reset*, therefore it is satisfied by either of the two situations above.

Lastly, we define the temporal formula *Spec* specifying the whole system:

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

As a temporal formula, *Spec* is interpreted relative to *behaviors*, which are infinite sequences of states. The predicate *Init* must be true in the first state. The meaning of the second conjunct, $\Box\,[Next]_{hour}$, is that for all future transitions of the system, either $Next$ is true, or $hour' = hour$ is true. The latter case is called a *stuttering step*.

Here is the beginning of a behavior satisfying *Spec*:

$$\boxed{hour = 11} \longrightarrow \boxed{hour = 12} \longrightarrow \boxed{hour = 1} \longrightarrow \boxed{hour = 1} \longrightarrow \boxed{hour = 2} \longrightarrow \cdots$$

Init holds in the first state. *Increment* is true for the first transition, then *Reset* for the next, then there is a stuttering step and the last transition satisfies *Increment*.

The possibility of stuttering steps is essential for combining and refining specifications. Given two systems specified by temporal formulas $Spec_1$ and $Spec_2$, the combination of both systems into a single one is specified by the formula

$$Spec_1 \wedge Spec_2$$

A transition of the whole system is the combination of a transition of $Spec_1$ and a transition of $Spec_2$. Since transitions include possible stuttering steps, the formula above accounts for transitions in which only one of $Spec_1$ or $Spec_2$ progresses—the other one stutters for this transition. In short, stuttering steps model transitions during which the *environment* of the system may progress. This is essential as components in actual systems may not always progress in a synchronized way. A similar principle applies to refinement, where several transitions of the low-level specification may be necessary to model a single transition of the more abstract specification.

Specification of a Sorting Algorithm

The *SimpleClock* specification showcases the most basic features of TLA$^+$, but there are not many meaningful properties to verify about it. To get a fuller picture of the language, we introduce the *AbstractSorting* module (Figure 2.3), which specifies a sorting algorithm that swaps out-of-order values in an array of integers. We call it abstract because it does not specify an explicit strategy for selecting values to swap.

We import operators from the modules *Naturals* and *Functions*. Besides the variable a denoting the array to be sorted, it is parameterized by two constants N and $init$. Constants are rigid variables; their values are fixed for all states. N is the size of the array and $init$ is the initial array.

The specification makes extensive use of TLA$^+$ functions to represent its data. The primitive construct $[x \in S \mapsto e]$ that binds x in the expression e represents a *function* of domain S. The domain of a function f is a set denoted $\text{DOMAIN } f$. The value of f at x is denoted $f[x]$. The class of all functions is defined by the predicate

$$f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

The principle of functional extensionality applies: two functions f and g are equal iff their domains are equal and $f[x] = g[x]$ for all x in their common domain. The set of functions from set S to set T is denoted $[S \to T]$.

The functions of TLA$^+$ are regular objects like sets or integers. In fact, since TLA$^+$ assumes an unsorted universe of set theory, we may say that all functions are sets. However a TLA$^+$ function f is not equal to its graph $\{\langle x, f[x]\rangle : x \in \text{DOMAIN } f\}$ like it would in Event-B. Furthermore, functions should not be confused with *operators*. A function application is written $f[x]$ while an operator application is written $F(x)$, although some operators admit an infix notation, for instance $x \cup y$. All functions have a domain which is a set, while operators may be specified on classes that are not sets. For instance, the operator $\cup$ is specified for all pairs of sets, but there is no function that corresponds to it, because there can be no set of all sets in ZFC.

AbstractSorting involves the particular set

$$ArrayType(n) \triangleq [1 \mathinner{..} n \to Int]$$

MODULE *AbstractSorting*

EXTENDS *Integers*, *Functions*

We specify an abstract sorting algorithm on arrays of integers. The algorithm non-deterministically selects two unordered values to swap at each step of the execution.

CONSTANT N, Size of the array
$init$ The initial array

VARIABLE a The array

Preliminary Definitions and Assumptions

$ArrayType(n) \triangleq [1 \,..\, n \to Int]$

$Swap(f, x, y) \triangleq [f \text{ EXCEPT } ![x] = f[y], ![y] = f[x]]$

ASSUME $NTyping \triangleq N \in Int \land N > 0$
ASSUME $InitTyping \triangleq init \in ArrayType(N)$

Main Specification

$Init \triangleq a = init$

$Next \triangleq \exists\, i, j \in \text{DOMAIN } a : \land\, i < j$
$\land\, a[i] > a[j]$
$\land\, a' = Swap(a, i, j)$

$Spec \triangleq Init \land \Box[Next]_a \land \mathrm{WF}_a(Next)$

Some relevant properties to verify / prove

$TypingInv \triangleq a \in ArrayType(N)$

$g \circ f \triangleq [x \in \text{DOMAIN } f \mapsto g[f[x]]]$

$PermutationInv \triangleq \exists\, p \in Bijection(1 \,..\, N, 1 \,..\, N) : a = init \circ p$

$Sorted(f) \triangleq \forall\, x, y \in \text{DOMAIN } f : x < y \Rightarrow f[x] \leq f[y]$

THEOREM $Spec \Rightarrow \Box TypingInv$

THEOREM $Spec \Rightarrow \Box PermutationInv$

THEOREM $Spec \Rightarrow \Box(Sorted(a) \Rightarrow \neg\text{ENABLED } Next)$

THEOREM $Spec \leadsto Sorted(a)$

Figure 2.3: Abstract Specification of a Sorting Algorithm

to represent arrays as functions of the interval $1\,..\,n$ to integers. The keyword ASSUME introduces two (named) assumptions for specifying type constraints on the constant parameters N and *init*. For swapping two values of a generic function, we introduce the operator *Swap* with the definition

$$Swap(f, x, y) \triangleq [f \text{ EXCEPT } ![x] = f[y]\,, ![y] = f[x]]$$

That definition involves a primitive operator of TLA$^+$ for defining a function differing from another on one or several inputs. If f is a function and $z \in \text{DOMAIN } f$, then we have

$$Swap(f, x, y)[z] = \begin{cases} f[y] & \text{if } z = x \\ f[x] & \text{if } z = y \\ f[z] & \text{otherwise} \end{cases}$$

In short, the values at x and y are swapped.

The rest of the specification follows the pattern of defining the state predicate *Init*, the transition predicate *Next*, and the temporal formula *Spec*. Here, *Spec* contains the additional conjunct $\text{WF}_a(Next)$. The meaning and relevance of this formula will be explained in the next section. The *Init* predicate specifies that the initial array is *init*. The *Next* predicate specifies a transition in which two inputs i and j are selected such that a is unordered at i and j, and a' is obtained by swapping those values. Notice the use of an existential quantifier to specify the non-deterministic selection of i and j.

The middle bar is purely cosmetic; we use it to delimit the actual specification from the relevant properties we want to verify. The keyword THEOREM introduces those properties (these statements have no semantics, they are just here for presentation purposes). Let us explain each of them briefly.

Three properties share the general form $Spec \Rightarrow \Box P$, which is a way of stating that P is an *invariant* of the specification. Invariants are also called *safety properties*. The first invariant is a typing invariant stating that a is an array of size N. TLA$^+$ is an untyped language, which means that operators may be combined in unconventional ways; but the semantics of unconventional expressions is not always specified. It is recommended to always express a typing invariant for a specification's variables and verify it. The next safety property, *PermutationInv*, states that a is a permutation of the initial array. The definition involves the operator *Bijection*, which is defined in the module *Functions*, and the operator $\circ$ for composing functions, which we define ourselves.

The third invariant involves the operator ENABLED. For all actions A, the expression ENABLED A is a state predicate that says a transition specified by A is possible from the current state. Thus the property asserts that if a is sorted, the system cannot progress.

The fourth property is of the form $Spec \leadsto P$. This is a notation for $\Box(Spec \Rightarrow \Diamond P)$, and $\Diamond$ is the modal operator for asserting that a property is true in some future state of the behavior. Therefore, the property is understood as the assertion that a is eventually sorted (together with the previous property, it follows that the specified sorting algorithm is terminating). Properties asserting that something must eventually happen are called *liveness properties*.

2.2.2 Model Checking with TLC

Model checking is the verification of specifications through the construction and evaluation of their models. TLC is called an *explicit-state* model checker, because it is based on explicit

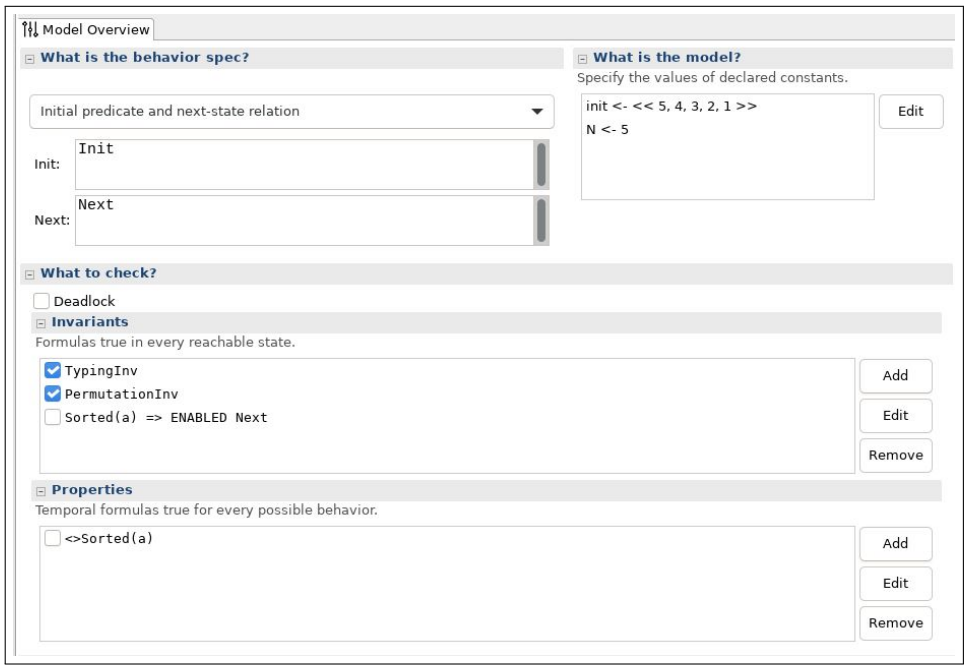


Figure 2.4: TLC Parameters for the Abstract Sorting Specification

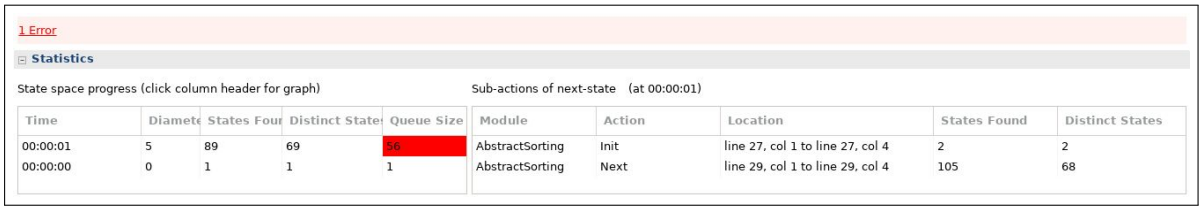
representations of the values of variables of a specification, rather than a symbolic representation. The model checker, which can be invoked from the TLA$^+$ Toolbox, exhausts the space of reachable states to verify that all possible behaviors satisfy the desired properties. TLC is useful for detecting bugs in specifications, but it is limited to specifications involving finite data and fixed parameter values.

Verifying Invariants

A simple configuration for TLC is shown in Figure 2.4. We indicate to TLC that *Init* specifies the initial state and *Next* the transition. We specify the values $N \leftarrow 5$ and $init \leftarrow \langle 5, 4, 3, 2, 1 \rangle$ (this uses TLA$^+$'s notation for tuples, which are the same as functions of a domain $1 \,..\, n$) for the two parameters of the specification. If we want to specify bounds rather than explicit values for the parameters (for instance, to check for all $N \leq 5$), we must write *AbstractSorting* differently—for the sake of simplicity, we will only invoke TLC with two explicit values here.

We indicate to TLC what it should check. The most important parameters are whether we check for *deadlocks* and which *invariants* to verify. If deadlocks are checked, TLC will return an error when a state with no possible transition occurs. If TLC reaches a state violating an invariant, it returns an error. For now, we only verify that *TypingInv* and *PermutationInv* are satisfied in all possible executions; since termination is a desired property in our case, we disable deadlock checking. We wrote one additional invariant and also a temporal formula to verify later. The result of calling TLC is shown in Figure 2.5. The call ended with no errors, so the invariants are indeed satisfied in all reachable states. In total TLC explored 120 distinct states, which corresponds to the number of permutations of a set of five elements.

Figure 2.5: Checking Invariants with TLC



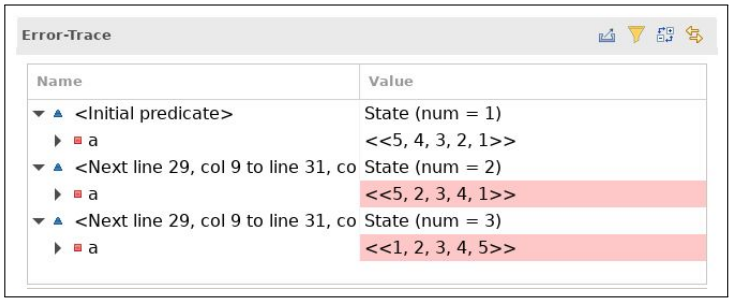


Figure 2.6: Checking Violation of Invariants with TLC

Verifying Liveness Properties

Liveness properties can also be checked by TLC. There is a general method to verify that a state satisfying P is reachable which consists in checking that $\neg P$ is *not* an invariant of the specification. For instance, if we run TLC with the instruction to check that ENABLED *Next* is an invariant of *AbstractSorting*, it will return an error upon finding a terminating state, and show the trace of execution. This is equivalent to selecting "deadlocks" in the configuration window.

Suppose we want to check that a terminating state in which a is sorted is reachable. The negation of this property is expressed by the formula

$$Sorted(a) \Rightarrow \text{ENABLED } Next$$

which is violated exactly when a is sorted and no transition is possible. Running TLC, we get an error, and the trace of execution is displayed in Figure 2.6. This trace is actually a minimal sequence of steps to reach the terminating state in which a is sorted.

This method is not equivalent to the verification of $Spec \rightsquigarrow P$, because by reaching a state violating $\neg P$ we only find *some* path that leads to P. To verify the full liveness property, we may ask TLC to check a temporal formula. For instance, to check that the array is eventually sorted, we can simply input $\Diamond Sorted(a)$ in the last box of TLC's configuration window. This will not work for the present configuration, because the configuration so far uses only the *Init* and *Next* part of the specification and ignores the condition $\text{WF}_a(Next)$. With this configuration, an infinite sequence of stuttering steps in which a is unchanged is

a legitimate behavior, so the property $\Diamond Sorted(a)$ is in fact not true. The *weak fairness* condition $\mathrm{WF}_a(Next)$ prohibits this behavior. Its formal definition is

$$\mathrm{WF}_a(Next) \triangleq (\Diamond \Box \text{ ENABLED } \langle Next \rangle_a) \Rightarrow \Box \Diamond \langle Next \rangle_a$$
$$\langle Next \rangle_a \triangleq Next \wedge a' \neq a$$

The action $\langle Next \rangle_a$ is a *Next* step during which the value of a changes. Therefore $\mathrm{WF}_a(Next)$ states that if a non-stuttering step becomes always possible, then it will necessarily happen. After selecting a different option to provide the full formula *Spec* containing the weak fairness condition, and running TLC, we find that the property $\Diamond Sorted(a)$ is verified as expected.

2.2.3 Interactive Proof with TLAPS

Using TLC, we are only able to verify the *AbstractSorting* specification with bounds on N and *init*. For the general case, and for specifications that involve infinite sets, we must write proofs and check them using TLAPS.

TLA$^+$ Proofs

TLA$^+$ includes a syntax for proofs based on Lamport's hierarchical style [19]. The syntax is detailed in a note introducing the second version of the language.[1] We demonstrate it with the module *AbstractSortingSafety* in Figure 2.7. This module contains proofs that the properties *TypingInv* and *PermutationInv* are inductive invariants of the specification.

A theorem is introduced by the keyword THEOREM. Those statements may be followed by proofs (the proofs are considered omitted otherwise). The syntax of TLA$^+$ proofs is based on a tree structure:

- A leaf is a line of relevant *facts* and *definitions* which are necessary to prove the statement. The facts are introduced by the keyword BY. The definitions are introduced by DEF. If no facts or definitions are invoked, the proof is written OBVIOUS.
- A complex proof is a sequence of *intermediary steps* ending with a *conclusion step* whose statement is QED. Each step is a node in the proof structure and must be justified by its own nested proof.

Each step in a proof is introduced by a level indication and an optional label with the notation $\langle level \rangle$ *label*. The level is a positive number corresponding to the level of nesting in the proof structure. The label is optional and can be any string of ASCII characters, although it is common practice to label steps with natural numbers. The presence of a label has an operational meaning: without a label, the intermediary step is implicitly added as a fact for the subsequent steps at the same level and in deeper levels; if a label is present, the step is not implicitly used as a fact, and it must be explicitly invoked when necessary.

This basic syntax suffices to write a proof that *TypingInv* is an invariant of the specification *Spec*. Let us detail the proof of the theorem named *Typing*. Although the full statement is a temporal formula, it is possible to lay out the proof in such a way that most steps can be solved without temporal reasoning. The first step is

$$Init \Rightarrow TypingInv$$

[1] http://lamport.azurewebsites.net/tla/tla2-guide.pdf

MODULE *AbstractSortingSafety*

EXTENDS *AbstractSorting*, *TLAPS*, *Functions*, *FunctionTheorems*

USE *NTyping*, *InitTyping*

THEOREM $Typing \triangleq Spec \Rightarrow \Box TypingInv$
⟨1⟩1. $Init \Rightarrow TypingInv$
BY DEF *TypingInv*, *Init*
⟨1⟩2. $Next \land TypingInv \Rightarrow TypingInv'$
BY DEF *TypingInv*, *Next*, *ArrayType*, *Swap*
⟨1⟩3. UNCHANGED $a \land TypingInv \Rightarrow TypingInv'$
BY DEF *TypingInv*
⟨1⟩.QED
BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, *PTL* DEF *Spec*

THEOREM $Safety \triangleq Spec \Rightarrow \Box PermutationInv$
⟨1⟩1. $Init \Rightarrow PermutationInv$
⟨2⟩.SUFFICES ASSUME $a = init$
PROVE $\exists\, p \in Bijection(1 \,..\, N, 1 \,..\, N) : a = init \circ p$
BY DEF *Init*, *PermutationInv*, $\circ$
⟨2⟩.$[k \in 1 \,..\, N \mapsto k] \in Bijection(1 \,..\, N, 1 \,..\, N)$
BY *Fun_IsBij*
⟨2⟩.QED
BY DEF *ArrayType*, $\circ$
⟨1⟩2. $Next \land TypingInv \land PermutationInv \Rightarrow PermutationInv'$
⟨2⟩.SUFFICES ASSUME $a \in [1 \,..\, N \to Int]$,
NEW $i \in 1 \,..\, N$, NEW $j \in 1 \,..\, N$, $a' = Swap(a, i, j)$,
NEW $p \in Bijection(1 \,..\, N, 1 \,..\, N)$, $a = init \circ p$
PROVE $\exists\, q \in Bijection(1 \,..\, N, 1 \,..\, N) : a' = init \circ q$
BY DEF *Next*, *TypingInv*, *PermutationInv*, *ArrayType*, $\circ$
⟨2⟩.DEFINE $s \triangleq [k \in 1 \,..\, N \mapsto$ CASE $k = i \to j$
$\Box\ k = j \to i$
$\Box$ OTHER $\to k]$
⟨2⟩.$(p \circ s) \in Bijection(1 \,..\, N, 1 \,..\, N)$
BY *Fun_IsBij*, $s \in Bijection(1 \,..\, N, 1 \,..\, N)$, *Fun_BijTransitive* DEF $\circ$
⟨2⟩.QED
BY DEF *ArrayType*, *Swap*, $\circ$
⟨1⟩3. UNCHANGED $a \land PermutationInv \Rightarrow PermutationInv'$
BY DEF *PermutationInv*
⟨1⟩.QED
BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, *Typing*, *PTL* DEF *Spec*

Figure 2.7: Proofs of Two Safety Properties for the Sorting Algorithm

It is solved simply by expanding the definition of *Init* and *TypingInv*. Actually, the proof also requires the assumption $init \in ArrayType(N)$, which is named *InitTyping* in the specification. With the USE command at the beginning of the module, we make sure that the typing assumptions for *init* and N are always inserted in the proof context.

The second step is the inductive step

$$Next \wedge TypingInv \Rightarrow TypingInv'$$

After expanding the definition of $TypingInv'$, the prime operator is distributed, resulting in the expression $a' \in ArrayType(N)$. As a transition predicate, this formula is interpreted for pairs of states; TLAPS reduces this formula to a regular first-order formula by replacing the primed variable a' by a fresh variable. The proof is essentially the verification of

$$Swap(a, i, j) \in ArrayType(N)$$

under the hypothesis $a \in ArrayType(N)$ and for $i, j \in$ DOMAIN a. This is an elementary verification which only requires expanding the definitions of *Swap* and *ArrayType*.

The last intermediary step is for verifying that *TypingInv* is invariant for stuttering steps:

$$\text{UNCHANGED } a \wedge TypingInv \Rightarrow TypingInv'$$

After this, the three steps $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ and $\langle 1 \rangle 3$ are combined to prove the result. The keyword PTL is not a fact, but a pragma to indicate the backend to invoke. "PTL" stands for Propositional Temporal Logic and refers to the LS4 solver [76]. It is typically invoked for a single step involving elementary temporal logic, like in this proof. The other possible backend calls are Isa (for the Isabelle backend), Z3, CVC4, veriT, SMT (for the default SMT solver which may be configured) and Zenon. There are also some variants to specify a different timeout (ZenonT(30)) or a particular Isabelle method (IsaM("auto"), IsaM("force"), IsaM("blast")).

The proof of *Safety*, which states that *PermutationInv* is an invariant of the specification, is more involved and features some advanced elements of the proof language. The general structure of the proof is the same: we prove that the invariant is implied by *Init*, then that it is inductive for *Next*, for stuttering steps, and we conclude using PTL for the last step. The invariant is proved by constructing a permutation p of $1 \,..\, N$ such that $a = init \circ p$. In the initial state, p is the identity; for the inductive step, p is updated by composing it with s, which is defined as the permutation that swaps i and j.

The statement of the inductive step features the typing invariant as an hypothesis:

$$Next \wedge TypingInv \wedge PermutationInv \Rightarrow PermutationInv'$$

This is justified by the previously proved result *Typing*, which is invoked as a lemma for the last step. In general, if $Spec \Rightarrow \Box P_1$ is proved, then it suffices to prove $Spec \wedge \Box P_1 \Rightarrow \Box P_2$ to deduce $Spec \Rightarrow \Box P_2$.

The first advanced feature of the proof syntax worth noting is the use of the SUFFICES keyword. An intermediary step starting with SUFFICES modifies the current goal of the proof. Formally, assuming the current goal is a formula A, then a step consisting of the statement SUFFICES B corresponds to the obligation $B \Rightarrow A$, and replaces the current goal by B for the subsequent steps. Here, we combine SUFFICES with the use of *sequents* to manage the context of the proof. The general form of a TLA$^+$ sequent is

ASSUME $H_1, H_2, \ldots$
PROVE A

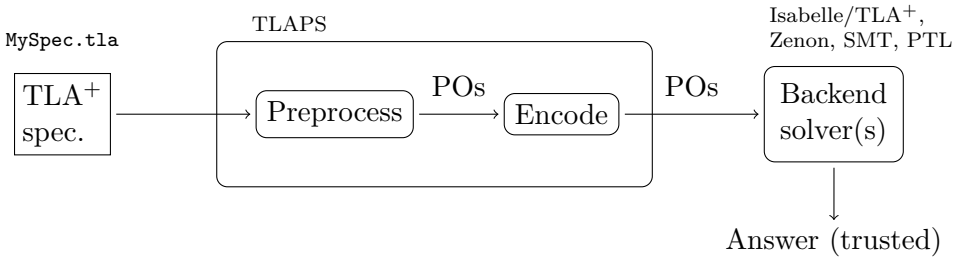


Figure 2.8: Architecture of TLAPS

where A is a formula representing the goal, and each H_i is either a declaration (indicated by the keyword NEW) or a formula representing an hypothesis. A bounded declaration NEW $x \in S$ is just the declaration NEW x followed by the hypothesis $x \in S$. Combining SUFFICES with sequents is the most practical way to reformulate a goal while introducing new symbols and hypotheses.

The keyword DEFINE is used to declare and define a symbol locally. Such steps do not require proofs. Here we use the command to define s as the permutation swapping i and j. By default, local definitions are always expanded by TLAPS, so there is no need to include them after a DEF keyword. In complex proofs, it may become necessary to hide definitions; this can be done with the command HIDE DEF in an intermediary step.

The proof of the intermediary step

$$(p \circ s) \in \mathit{Bijection}(1\,..\,N, 1\,..\,N)$$

involves several facts which are not previous steps of the current proof. The names *Fun_IsBij* and *Fun_BijTransitive* refer to lemmas from the *FunctionTheorems* module. The expression $s \in \mathit{Bijection}(1\,..\,N, 1\,..\,N)$ is an inline, unproved fact for which TLAPS generates an additional proof obligation. The context of that proof obligation is obtained by inserting the previously indicated facts to the current context; here, the lemma *Fun_IsBij*, which states the sufficient conditions for something to be a bijection, is used to prove that s is a permutation of $1\,..\,N$. For the main obligation $(p \circ s) \in \mathit{Bijection}(1\,..\,N, 1\,..\,N)$, the two lemmas and the fact about s are used. The lemma *Fun_BijTransitive* states that the composition of two bijections is a bijection.

TLAPS

The architecture of TLAPS is presented in Figure 2.8. The proof assistant parses a TLA$^+$ specification and generates a number of *proof obligations* (PO) from its proofs. Each obligation is *encoded* for one or several of the available backend solvers: Isabelle (using a custom TLA$^+$ object logic), Zenon, SMT and PTL. The SMT solvers available are Z3, CVC4 and veriT. By default, TLAPS invokes in a portfolio mode Isabelle, Zenon, and Z3. Users may select a particular backend by using a special keyword in place of an invoked fact. If one of the invoked backends answers positively (if the encoded obligation has been solved), then TLAPS considers the PO to be solved. In other words, the backends are treated as *trusted oracles* by TLAPS, although there is some support for proof verification in the case of Zenon, whose output may be translated and checked by Isabelle.

The preprocessing step that precedes the encoding is common to all backends. Preprocessing includes:

- Insertion of all relevant declarations and facts into the PO's context. Explicitly invoked lemmas, proof steps and inline facts following a BY are appended to the current context of the proof for all obligations. Some control is given to the user over which facts are known in the form of the USE and HIDE command, which can be used at the top level of a specification or inside a proof.

- Expansion of all relevant definitions. By default, top definitions are opaque, but local definitions with DEFINE are expanded. Typically, the DEF keyword indicates the additional definitions to use for an obligation. The commands USE DEF and HIDE DEF can be used to override the default behavior of TLAPS for definitions.

- Removal of syntactic sugar. For instance, any expression

  $$[f \text{ EXCEPT } ![d_1] = e_1, \ldots, ![d_n] = e_n]$$

  is rewritten as

  $$[\ldots[f \text{ EXCEPT } ![d_1] = e_1] \ldots \text{ EXCEPT } ![d_n] = e_n]$$

  so that EXCEPT may be treated as a ternary operator. Another example is multi-argument applications; an application

  $$f[x_1, \ldots, x_n]$$

  is rewritten as

  $$f[\langle x_1, \ldots, x_n \rangle]$$

- (For backends other than PTL) Elimination of the prime operator. In the simplest case, all primes are found on flexible variables; in this case it suffices to introduce a fresh variable for any x'. In general a prime may be found on any expression. TLAPS will distribute the prime operator when possible, for instance $(x+y)'$ is rewritten $x'+y'$. For an expression $F(x)'$ where F is *not* expanded, TLAPS will replace the whole expression by a fresh identifier.

After preprocessing, the general form of a TLA$^+$ proof obligation (in the case of a backend other than PTL) is that of a sequent without temporal connectives or primes. One important omission from our examples is the presence of user-defined second-order operators. Here is an example:

$$F(Op(\_), x) \triangleq Op(x)$$

When the definition of F is not expanded, any expression $F(Op, e)$ is only naturally encoded in second-order logic. Currently, these obligations are only supported by the Isabelle backend, since the logics of Zenon and SMT are first-order. However, a large class of specifications do not require second-order user definitions, so the current support of TLAPS for second-order constructs may suffice for those.

The SMT Encoding

The SMT backend of TLAPS uses an optimized encoding of TLA$^+$ that combines soft type inference and rewriting [53,81]. The usefulness of SMT solvers is recognized in the context of interactive theorem proving supported by automation [9,29,32].

The standard input language for SMT solvers is SMT-LIB [6]. As a logic, SMT-LIB is best described as multi-sorted first-order logic (MS-FOL), but the interest of SMT lies in its support for first-order *theories*. While generic first-order logic is undecidable, some theories have decision procedures, like linear arithmetic or bitvectors. Encoding TLA$^+$ in such a way that those procedures can be leveraged is of great interest to the developers of TLAPS, but the fact that TLA$^+$ is untyped makes it more challenging. Nevertheless, the current SMT encoding outputs problems expressed in the SMT logic UFNIA (Uninterpreted Functions and Non-linear Integer Arithmetic).

To understand the SMT encoding, it is useful to consider first a *direct encoding* of TLA$^+$. The direct encoding is summed up in the following key ideas:

- A direct, faithful translation of TLA$^+$ into MS-FOL is implemented by just declaring every TLA$^+$ primitive operator as an uninterpreted symbol in the SMT problem. An uninterpreted sort ι is declared to receive every term. Some transformations are necessary to recover the Boolean sort (this process is called *Boolification* and achieved in a single pass over the expressions).
- The TLA$^+$ primitives, encoded as uninterpreted symbols, are specified by SMT *axioms*. All quantifiers range over the uninterpreted sort ι.
- For arithmetic, we use a different set of axioms to specify a correspondence between TLA$^+$'s integer arithmetic and SMT's builtin arithmetic.

The support for arithmetic involves a cast operator $\mathsf{cast}_{\mathsf{int}} : \mathsf{int} \to \iota$ and a few axioms. One axiom specifies $\mathsf{cast}_{\mathsf{int}}$ as a bijection between the sort int and the class of sets that are members of *Int*:

$$\forall x^{\iota} : x \in \mathit{Int} \Leftrightarrow \exists n^{\mathsf{int}} : x = \mathsf{cast}_{\mathsf{int}}(n)$$

There are axioms to specify a homomorphic relationship between TLA$^+$'s arithmetical operators and their SMT counterparts. For instance:

$$\forall m^{\mathsf{int}}, n^{\mathsf{int}} : \mathsf{cast}_{\mathsf{int}}(m) +_{\iota} \mathsf{cast}_{\mathsf{int}}(n) = \mathsf{cast}_{\mathsf{int}}(m +_{\mathsf{int}} n)$$

Despite the integration of SMT's builtin arithmetic, this direct encoding was considered insufficient, essentially because it results in an SMT problem with many uninterpreted symbols and many axioms with quantifiers over the uninterpreted sort ι. There were attempts to make the axiomatization efficient with SMT triggers, but they did not lead to satisfactory results. Another problem is the encoding of second-order constructs such as explicit functions $[x \in S \mapsto e]$, which is far from trivial. For these reasons, an extension of the direct encoding was implemented. This optimized encoding is based on a *preprocessing phase* and an optional *type synthesis* mechanism.

The preprocessing phase is centered around a rewriting system. A set of rules are provided for eliminating occurrences of TLA$^+$ primitives. For example, an instance of $\cup$ is eliminated by the rule

$$x \in (a \cup b) \longrightarrow x \in a \vee x \in b$$

The more TLA$^+$ primitives we eliminate through rewriting, the better. However, in many situations it is not possible to apply a rewriting rule directly, like in the formula

$$c = (a \cup b) \Rightarrow x \in c$$

Small changes in the structure of expressions can impact the effectiveness of rewriting, which is why rewriting must be complemented by auxiliary techniques. One of these techniques is the *elimination of definitions*, which eliminates hypotheses of the form $x = e$ by substituting e for x in expressions. Another technique is *abstraction*, which substitutes parameterized symbols for complex expressions. For instance, the formula

$$(a \cup b) \in d$$

may be rewritten as

$$k(a, b) \in d$$

where k is a fresh symbol defined by $\forall x^\iota, y^\iota : k(x, y) = (x \cup y)$. That definition would then be rewritten by an application of set extensionality, which is implemented as the rewriting rule

$$x = y \longrightarrow \forall z : z \in x \Leftrightarrow z \in y$$

The second technique is type synthesis, which identifies subexpressions residing in domains of interest (integers, functions, sets, etc.) and assigns them different sorts. Consider the valid TLA$^+$ formula

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

According to TLA$^+$'s semantics, the variable x ranges over the whole universe of set theory. However, it is clear that only the integer values are relevant (the inner formula is trivially true if $x \notin \mathit{Int}$), so type synthesis will annotate $\forall x$ with the sort int. Since 0 is an integer and the addition of two integers is an integer, it follows that $x + 0$ and x are both integers. The rewriting system will also eliminate the hypothesis $x \in \mathit{Int}$ since it is redundant with the sort of x, resulting in the formula

$$\forall x^{\mathsf{int}} : x + 0 = x$$

That formula is *well-typed* in the context of a multi-sorted logic that interprets 0 and + in the domain of integers. However, arbitrary annotations may result in an unsound encoding. Consider the TLA$^+$ formula

$$\forall x : x + 0 = x$$

This is a legitimate TLA$^+$ formula, but it is not valid, because for instance $\text{FALSE} + 0 = \text{FALSE}$ is unprovable. This time, if we annotate $\forall x$ with int, we obtain a formula that is again well-typed, but also valid. Therefore, the typing of x with int is unsound in this case. In general, type synthesis will look for expressions called *typing hypotheses* to justify the introduction of restrictive sorts. The expression $x \in \mathit{Int}$ is an example of a typing hypothesis.

The previous example suggests that type synthesis may need to integrate some reasoning about TLA$^+$'s semantics in order to be practical. The case of TLA$^+$ functions proves this is actually necessary: the natural way to type an expression $f[x]$ involves the verification of $x \in \text{DOMAIN } f$. Since functions can have any set as domain, any reasonable type system for TLA$^+$ must be undecidable.

The SMT backend implements two type systems for TLA$^+$, T_1 and T_2. The encoding that does not use type synthesis is referred to as T_0. Both systems are based on the derivation

of typing judgments $\Gamma \vdash e : \tau$ where e is an expression, τ a type, and Γ a typing context (assignment of types to variables). The essential differences between T_1 and T_2 are seen in their treatment of functional types and applications. The system T_1 features elementary sorts and type constructors including the arrow $\tau_1 \to \tau_2$ for functions. The typing rule for applications is expressed

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Gamma \vdash x : \tau_1}{\Gamma \vdash f[x] : \tau_2}$$

The types of T_1 do not characterize terms finely enough to capture information such as the domains of functions. For instance, the function f defined as $[i \in \{0, 1, 2\} \mapsto i]$ is assigned the type $\mathsf{int} \to \mathsf{int}$. As a result, a naive application of the typing system might result in the typing of $f[3]$ as an int. But it is not the case that $f[3] \in \mathit{Int}$. Therefore, to make T_1 usable, one has to implement a procedure to verify *type checking conditions* like $x \in \text{DOMAIN } f$.

The system T_2 takes a different approach by introducing *dependent* and *refinement types*. These types are expressive enough to characterize the elements of a function's domain. In T_2, we have the typing

$$\vdash [i \in \{0, 1, 2\} \mapsto i] : (i : \{x : \mathsf{int} \mid x = 0 \lor x = 1 \lor x = 2\}) \to \{y : \mathsf{int} \mid y = i\}$$

The expression $f[3]$ will not be well-typed in T_2 because the argument 3 cannot be typed with $\{x : \mathsf{int} \mid x = 0 \lor x = 1 \lor x = 2\}$. The verification of this condition is expressed as part of the typing and verified during the type synthesis mechanism rather than aside from it like in T_1.

Despite its efficiency, the SMT encoding has some issues. The first kind of issue concerns the safety, or soundness of the encoding. For instance, testing the encoding with the option `--debug types2` to enable T_2 (by default, T_1 is enabled), we found that the backend would let us solve the wrong result $[i \in \{0, 1, 2\} \mapsto i][3] \in \mathit{Int}$. This seems to indicate that the type-checking conditions are not actually verified by the implementation. There are no clear indications in the code's documentation for us to know how to fix this problem.

The soundness of the encoding is also compromised by the substantial complexity of the preprocessing phase. Central to preprocessing is the rewriting system, which is directly implemented in TLAPS. One can easily make a mistake when implementing a rewriting rule, resulting in an unsound encoding. These can be difficult to detect, because one usually writes a proof expecting it to be solved by TLAPS. Nevertheless, we found some examples of unsound rules.[2] Consider the following rules for simplifying applications:

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

$$[f \text{ EXCEPT } ![y] = z][x] \quad \longrightarrow \quad \text{IF } x \in \text{DOMAIN } f \text{ THEN IF } x = y \text{ THEN } z \text{ ELSE } \alpha(f, x) \text{ ELSE } \omega([f \text{ EXCEPT } ![y] = z], x)$$

The symbols α and ω are two uninterpreted symbols for representing functional application in the final SMT problem. α assumes the application is specified, ω is for the unspecified

[2]The example we are about to show was found after implementing our own SMT encoding. Comparing the two versions showed a number of proof obligations that were only solved with the original encoding. As it turned out, these obligations were not valid in the first place.

case. The two rules are sound, but the second rule was actually implemented with the term $\omega(f, x)$ for the else case. A consequence is that the wrong formula

$$\forall f, x, y, z : x \neq y \Rightarrow f[x] = [f \text{ EXCEPT } ![y] = z][x]$$

was provable with the SMT backend. The formula is not valid because the assumption $x \in \text{DOMAIN } f$ is missing.

We should point out that the hypothetical direct encoding, which is based on axioms, would suffer from the same problem, because it is easy to make a mistake when coding an axiom in TLAPS's implementation. However, if an axiom is badly implemented, it must be visible in the SMT-LIB problem passed to the solver; one may even narrow the search for the pathological axiom by asking the solver for an unsatisfiable core, if this option is provided. In the case of rewriting, this is not possible, because the point of rewriting is precisely to transform proof obligations to make them simpler.

One aim of the present work is to make the SMT encoding safer. Regarding the state of the original SMT encoding that we just described, we observed that most of the issues stemmed from the complexity of the implementation, in particular the type synthesis algorithm and the preprocessing phase. The direct encoding was considered, but not implemented fully. Our idea then was to revisit the SMT encoding with a more modular design, in which preprocessing (rewriting) could be considered an optimization rather than a necessity. In principle, the direct encoding would not achieve the same performances as the original encoding, but it could be used as a safer version of the SMT encoding.

The next chapter details this direct encoding and presents the arguments for its soundness. In the end, reimplementing type synthesis and preprocessing proved to be unnecessary, as we found a simpler, safer way to optimize the encoding using SMT triggers. The chapter after the next presents that optimization.

Chapter 3

Formal Semantics of TLA$^+$'s Constant Fragment

3.1 Overview

This chapter develops the formal semantics of TLA$^+$'s constant fragment. This fragment excludes all features from temporal logic. The vast majority of TLA$^+$ proof obligations are expressed as constant, state or transition formulas; the state and transition formulas are systematically reduced to constant formulas by TLAPS. For the remaining temporal formulas, users are expected to invoke the PTL backend. For the rest of this manuscript, "TLA$^+$" will implicitly refer to the constant fragment.

All aspects of the logic are covered in the reference book [48] and the note on TLA$^+$ version 2.[1] However our treatment will be more formal. We define TLA$^+$ as a variant of first-order logic (FOL) with the axioms of ZFC set theory. FOL and set theory are both standard frameworks, so it may be helpful to highlight the features that set TLA$^+$ apart. We have identified three:

1. There are second-order operators; those can be passed first-order operators or lambda-expressions as arguments;
2. The unsorted logic permits unusual combinations of the operators; an expression like $1/0$ is not undefined but underspecified;
3. There is no sort for formulas either; an expression like $42 \Rightarrow 42$ is legitimate.

Underspecification is a central feature of TLA$^+$ [39, 49]. To keep the example of $1/0$, in the context of underspecification we can write this expression in our specifications, but we may not be able to say anything meaningful about it. We know division on reals is specified by some laws, for instance

$$\forall x, y \in \mathit{Real} : y \neq 0 \Rightarrow y * (x/y) = x$$

This law will tell us that $0 \neq 0 \Rightarrow 0 * (1/0) = 1$. This is not a very helpful fact, but our only concern for now is to define a precise semantics for untyped operators. As the example of division demonstrates, a natural solution is to avoid interpreting operators and only resort

[1] `http://lamport.azurewebsites.net/tla/tla2-guide.pdf`

to axioms. All interpretations that satisfy the axioms are legitimate. In some interpretations $1/0$ will be a real number, in others it will not.

We formalize TLA$^+$ as a logic $\mathcal{L} + T$, where $\mathcal{L}$ is a variant of FOL and T a theory with all the primitive operators and axioms we need. The logic $\mathcal{L}$ extends FOL with second-order operators and a different semantics for Boolean connectives (Section 3.2). The distinction between term and formula is absent in $\mathcal{L}$, so we use the neutral word "expression" instead. The theory T contains the axioms of ZF, the axiom of foundation, and Hilbert's indefinite choice (also called epsilon-calculus) which trivially implies the axiom of choice (Section 3.3). The theory also includes axioms for functions, arithmetic, and other structures such as tuples and records.

The theorems of TLA$^+$ and the proof obligations treated by TLAPS are expressed in the form of sequents (Section 3.4). Sequents technically make the logic second-order, since they offer a way to quantify over first-order operators. However this is a rather obscure feature of the language and it is rarely used in actual proofs. We will discuss sequents in upcoming chapters to address some fine points about the implementation, but for the most part we will only consider the encoding of expressions and take $\mathcal{L} + T$ as source logic.

3.2 The Logic $\mathcal{L}$

We introduce some general conventions about notation. If C is a collection, we write $x \in C$ to mean that x is an element of C. When we introduce $\in$ as a primitive operator of set theory the symbol will have two different meanings, but it should always be clear in context which meaning is the appropriate one. If C_1 and C_2 are two collections, we note $C_1 \times C_2$ the collection whose elements are pairs of elements $x \in C_1$ and $y \in C_2$, and $C_1 \to C_2$ the collection of functions f mapping elements $x \in C_1$ to elements $f(x) \in C_2$. If f is any function, the collection $\mathrm{Dom}(f)$ denotes its domain. If $x \in \mathrm{Dom}(f)$ and v is some element, we note f^x_v the function that maps x to v and every $y \in \mathrm{Dom}(f)$ other than x to $f(y)$. We generalize this notation to multiple reassignments by defining $f^{x,y}_{v,w}$ as $(f^x_v)^y_w$.

We fix some infinite collection $\mathcal{V}$. The elements of $\mathcal{V}$ are called variables symbols, or simply variables.

Definition 3.2.1 (Expressions)**.** A *shape* is a finite list of natural numbers $\kappa = (n_1, \ldots, n_l)$. If $n_i = 0$ for all i then κ is a first-order shape. If $l = 0$ then κ is the constant shape. A *signature* is a function Σ from *operator symbols* to shapes. We note operators with the letter K.

The syntax of raw Σ-expressions and Σ-arguments is defined by the following grammar:

$$e ::= x \mid K(f, \ldots, f) \mid e = e \mid \textsc{false} \mid e \Rightarrow e \mid \forall x : e \qquad \text{(Expressions)}$$
$$f ::= e \mid K \mid \lambda x, \ldots, x : e \qquad \text{(Arguments)}$$

where $x \in \mathcal{V}$ and $K \in \mathrm{Dom}(\Sigma)$. An argument f is constant if it is an expression, otherwise it is first-order. An operator K occurring as a first-order argument must have a first-order, non constant shape. The list of variables after a λ must be non-empty and without duplicates. An application with zero arguments $K()$ is simply noted K.

We assign an *arity* to every raw Σ-argument f. The arity of an expression e is 0. The arity of a first-order operator K is the length of $\Sigma(K)$. The arity of $\lambda x_1, \ldots, x_n : e$ is n. An application $K(f_1, \ldots, f_p)$ is well-formed if $\Sigma(K) = (n_1, \ldots, n_p)$ and, for all $1 \leq i \leq p$, the raw argument f_i has arity n_i.

We define Σ-expressions (resp. Σ-arguments) as raw Σ-expressions (resp. raw Σ-arguments) in which all applications are well-formed. We may drop the prefix Σ- if it is clear in context which signature is considered.

The logical connectives that are not part of the minimalistic grammar of Definition 3.2.1 are introduced as notations:

$$\begin{aligned}
\text{TRUE} &\triangleq \text{FALSE} \Rightarrow \text{FALSE} \\
\neg e &\triangleq e \Rightarrow \text{FALSE} \\
e_1 \neq e_2 &\triangleq \neg(e_1 = e_2) \\
e_1 \vee e_2 &\triangleq (\neg e_1) \Rightarrow e_2 \\
e_1 \wedge e_2 &\triangleq \neg((\neg e_1) \vee (\neg e_2)) \\
e_1 \Leftrightarrow e_2 &\triangleq (e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1) \\
\exists x : e &\triangleq \neg(\forall x : \neg e)
\end{aligned}$$

Example 3.2.2. Even though the language is unsorted, applications must be well-formed. Let us note $K : \kappa$ to mean that K is assigned the shape κ in the relevant signature. Consider a signature Σ with a constant symbol c, a first-order operator $F : (0)$ and a second-order operator $G : (1, 0)$. Then the following applications are well-formed:

$$\begin{gathered}
F(x) \\
F(F(c)) \\
G(F, c) \\
G(\lambda x : F(x), c) \\
G(\lambda y : G(F, y), z)
\end{gathered}$$

The expression $G(F)$ is not well-formed, because G expects two arguments. F and $\lambda x : x$ are not well-formed expressions, but they are well-formed first-order arguments. G is not a first-order argument, it cannot be passed to another operator like F or $\lambda x : x$ can.

Despite the presence of second-order operators and lambda-expressions, most of the features of higher-order logic are absent. For instance $(\lambda x : x)(c)$ is not an expression. Lambda-expressions can appear only as first-order arguments. There are no partial applications either.

Definition 3.2.3. In an expression $\forall x : e$, the variable x is said to be bound in e. We also say that e is the scope of the quantifier $\forall x$. Similarly, in $\lambda x_1, \ldots, x_n : e$, each x_i is bound in the scope e. A *free variable* in an expression or argument is a variable x that occurs outside the scope of a $\forall x$ or $\lambda x_1, \ldots, x, \ldots, x_n$. The collection $\text{FV}(e)$ (resp. $\text{FV}(f)$) is the collection of the free variables of e (resp. f). An expression or argument is called closed if it does not have any free variables.

We now define the semantics of $\mathcal{L}$, which is very similar to the traditional semantics of FOL despite the differences between those two logics. Expressions are assigned values in some indefinite collection D, which must at least contain two values for "true" and "false". The meaning of operator symbols is given by an interpretation I. The meaning of free variables is given by a valuation θ.

Definition 3.2.4 (Evaluation)**.** A *domain* is a collection D that includes two distinct elements $\top^D$ and $\bot^D$ called the Boolean values. We assign a domain D_n to every arity n and a domain D_κ to every shape κ by the following equations:

$$\begin{aligned} D_0 &\triangleq D \\ D_n &\triangleq D \times \cdots \times D \to D \quad (n \text{ occurrences of } D \text{ on the left of } \to) \\ D_() &\triangleq D \\ D_{(n_1,\ldots,n_p)} &\triangleq D_{n_1} \times \cdots \times D_{n_p} \to D \end{aligned}$$

Let Σ be some signature. A Σ-*interpretation* I consists of a domain D and a family $(K^I)_{K \in \mathrm{Dom}(\Sigma)}$ such that $K^I \in D_{\Sigma(K)}$ for all K. A *valuation* on D is a function $\theta : \mathcal{V} \to D$. For all I and θ, we define recursively the *evaluation* $[\![\cdot]\!]^I_\theta$ of expressions and n-ary arguments as follows:

$$\begin{aligned} [\![x]\!]^I_\theta &\triangleq \theta(x) \\ [\![K(f_1,\ldots,f_n)]\!]^I_\theta &\triangleq K^I([\![f_1]\!]^I_\theta,\ldots,[\![f_n]\!]^I_\theta) \\ [\![e_1 = e_2]\!]^I_\theta &\triangleq \begin{cases} \top^D & \text{if } [\![e_1]\!]^I_\theta = [\![e_2]\!]^I_\theta \\ \bot^D & \text{otherwise} \end{cases} \\ [\![\textsc{false}]\!]^I_\theta &\triangleq \bot^D \\ [\![e_1 \Rightarrow e_2]\!]^I_\theta &\triangleq \begin{cases} \top^D & \text{if } [\![e_1]\!]^I_\theta \neq \top^D \text{ or } [\![e_2]\!]^I_\theta = \top^D \\ \bot^D & \text{otherwise} \end{cases} \\ [\![\forall x : e]\!]^I_\theta &\triangleq \begin{cases} \top^D & \text{if } [\![e]\!]^I_{\theta^x_v} = \top^D \text{ for all } v \text{ in } D \\ \bot^D & \text{otherwise} \end{cases} \end{aligned}$$

For all $v_1,\ldots,v_n \in D$:

$$\begin{aligned} [\![K]\!]^I_\theta(v_1,\ldots,v_n) &\triangleq K^I(v_1,\ldots,v_n) \\ [\![\lambda x_1,\ldots,x_n : e]\!]^I_\theta(v_1,\ldots,v_n) &\triangleq [\![e]\!]^I_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}} \end{aligned}$$

The evaluation of $e_1 \Rightarrow e_2$ does not assume that e_1 or e_2 is evaluated as a Boolean. Similarly for $\forall x : e$. Note that the evaluation of an expression whose top connective is $=$, FALSE, $\Rightarrow$ or $\forall$ will always be a Boolean.

Proposition 3.2.5. *If e is an expression, then $[\![e]\!]^I_\theta$ is well-defined as an element of D. If f is an n-ary argument, then $[\![f]\!]^I_\theta$ is well-defined as an element of D_n.*

Proof. This is proved by induction on the construction of expressions and arguments. The only non-trivial case is the application case. Let $K(f_1,\ldots,f_p)$ and $\Sigma(K) = (n_1,\ldots,n_q)$. Applications are well-formed by assumption, so $p = q$ and the arity of every f_i is n_i. According to the induction hypothesis, $[\![f_i]\!]^I_\theta \in D_{n_i}$ for all i. Since $K^I \in D_{n_1} \times \cdots \times D_{n_p} \to D$ by definition, we have $[\![K(f_1,\ldots,f_p)]\!]^I_\theta \in D$. □

Proposition 3.2.6. *If x is not free in the expression e, then $[\![e]\!]^I_{\theta^x_v}$ equals $[\![e]\!]^I_\theta$ for all θ and v. The same holds analogously for arguments.*

Proof. By induction on the construction of expressions and arguments. In the variable case, the expression must be a variable symbol y such that $y \neq x$; otherwise x would be free in e. Thus $\theta_v^x(y) = \theta(y)$. Binder cases are harder; following the definition, we have

$$[\![\forall z : e]\!]^I_{\theta^x_v} = \begin{cases} \top^D & \text{if } [\![e]\!]^I_{\theta^{x,z}_{v,v'}} = \top^D \text{ for all } v' \\ \bot^D & \text{otherwise} \end{cases}$$

It suffices to verify that $[\![e]\!]^I_{\theta^{x,z}_{v,v'}} = [\![e]\!]^I_{\theta^z_{v'}}$. If $x = z$, the two valuations are equal and the result is immediate. If $x \neq z$, we have

$$[\![e]\!]^I_{\theta^{x,z}_{v,v'}} = [\![e]\!]^I_{\theta^{z,x}_{v',v}} = [\![e]\!]^I_{(\theta^z_{v'})^x_v} = [\![e]\!]^I_{\theta^z_{v'}}$$

where the last equality is justified by the induction hypothesis on e with the valuation $\theta^z_{v'}$. This is possible because x is not free in e. Indeed, if x were free in e, it would be free in $\forall z : e$ since $z \neq x$. □

Proposition 3.2.6 justifies the notation $[\![e]\!]^I$ when e is closed. Since e does not contain any free variables, the values $\theta(x)$ do not matter, and thus θ itself does not matter in the evaluation.

Definition 3.2.7 (Satisfiability)**.** The *satisfaction relation* $\models$ is defined between interpretations and closed expressions by

$$I \models e \text{ iff } [\![e]\!]^I = \top^D$$

Let T be a set of closed expressions. We call I a *model* of T iff every expression of T is satisfied by I. This is written $I \models T$. We note $T \models e$ if e is satisfied by every model of T, and $\models e$ in the special case where T is empty. An expression e such that $\models e$ is said to be valid and is called a tautology.

The logic $\mathcal{L}$ is defined by the language of expressions and the satisfaction relation $\models$.

Example 3.2.8. Consider again the signature Σ of Example 3.2.2. The following expressions are tautologies:

$$c \Rightarrow c$$
$$G(F, c) = G(\lambda x : F(x), c)$$
$$\forall x : x \Leftrightarrow (x = \text{TRUE})$$

The lack of a term/formula distinction sets TLA$^+$ apart from most formalisms. It seems unlikely that users would take advantage of that feature. However, experience shows they do so quite naturally. Consider the following expression, which comes from the specification EWD998 from the library of TLA$^+$ Examples:[2]

$$terminationDetected \in \{\text{FALSE}, termination\}$$

where $\in$ and $\{\cdot, \cdot\}$ are binary operators. *terminationDetected* and *termination* are just constants. With the expected semantics for the pair set, a consequence of the expression above is

$$terminationDetected \Rightarrow termination$$

[2] `https://github.com/tlaplus/Examples`

This is true even if *terminationDetected* and *termination* are not Booleans.

Some consequences of the semantics are stranger, like the tautology $42 \Rightarrow 42$ in a language with numerical constants. Moreover, since the concept of formula does not exist in the logic, any expression can technically play the role of a formula. We have $42 \models 42$.

The semantics of Boolean operators is explained in the reference book [48, Section 16.1.3] and also Section 8.1 of the note on TLA$^+$ version 2. TLA$^+$ actually admits *two* possible interpretations for Boolean connectives, called the *moderate* and the *liberal* interpretation. Our formalism corresponds to the liberal interpretation, which TLAPS officially supports. The advantages of the liberal interpretation is that it is formally very close to FOL, and that it does not impose any constraints on the language of expressions.

3.3 The Theory of TLA$^+$

The logic $\mathcal{L}$ is a generic framework that fixes the interpretation of the Boolean connectives and also equality. We now turn to the second component of TLA$^+$'s logic, which is the theory T. First we provide a formal definition for the notation $\mathcal{L} + T$.

Definition 3.3.1. Two signature Σ_1 and Σ_2 are compatible if their domains do not intersect. We note $\Sigma_1 + \Sigma_2$ the union of two compatible signatures.

Definition 3.3.2. A theory (for $\mathcal{L}$) is a set of closed expressions T over some signature Σ_T. Let T be some theory. We define the logic $\mathcal{L} + T$ as follows:

- A signature of $\mathcal{L} + T$ is a signature of $\mathcal{L}$ compatible with Σ_T;
- A Σ-expression of $\mathcal{L} + T$ is a $(\Sigma_T + \Sigma)$-expression of $\mathcal{L}$;
- A Σ-interpretation is a $(\Sigma_T + \Sigma)$-interpretation that is a model of T;
- The evaluation of expression, arguments, and the satisfaction relation are inherited from $\mathcal{L}$.

By definition, an expression e is a tautology in $\mathcal{L} + T$ iff $T \models e$.

Working in $\mathcal{L} + T$ is essentially the same as working in $\mathcal{L}$, only we assume a collection of *primitive operators* in some signature Σ_T which are *specified* by the set of axioms T.

The rest of this section is an overview of the theory T specifying the primitive operators of TLA$^+$. The presentation is organized in several layers: epsilon-calculus, set theory, functions, arithmetic, and others. We will only present the most important fragments of the theory and not always formally. For the complete list of operators and axioms, we refer the reader to Appendix A.

Some axioms are actually axiom schemas. Those schemas are typically parameterized by one or several natural numbers n, positive numbers p, or operator symbols K. The meaning of declaring an axiom schema in T is that the axioms for all values of n, p or K are declared in T. Thus the collection T must be infinite.

Epsilon Calculus

The epsilon-calculus has a single operator $\mathsf{choose} : (1)$. The expression $\mathsf{choose}(\lambda x : e)$ is usually written CHOOSE $x : e$ in TLA$^+$'s user syntax. This notation is not general, because the argument to choose could be an operator symbol K. However, we have the equality $\mathsf{choose}(K) = \mathsf{choose}(\lambda x : K(x))$, and the second member is represented by CHOOSE $x : K(x)$.

The intuitive semantics of CHOOSE $x : P(x)$ is that it selects a value x such that $P(x)$ is true, if one exists. If no such x exists then the expression is unspecified. This is specified by a schema of axioms, where P is the parameter.

$$\forall x : P(x) \Rightarrow P(\text{CHOOSE } x : P(x))$$

Moreover, a second schema specifies CHOOSE as deterministic: if two predicates P and Q are equivalent then CHOOSE selects the same witness for both of them.

$$(\forall x : P(x) \Leftrightarrow Q(x)) \Rightarrow (\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x))$$

A practical consequence of that last axiom is that we can rewrite expressions below CHOOSE. For instance, we have (CHOOSE $x : x \neq x$) = (CHOOSE $x :$ FALSE).

As a supplement, we define the TLA$^+$ constructs for case expressions and if-then-else expressions as notations. The definition below features TLA$^+$'s indented notation for multi-line disjunctions, which also exists for conjunctions.

$$\begin{aligned}
&\text{CASE } p_1 \to e_1,\ \ldots,\ p_n \to e_n \\
&\quad \triangleq \text{CHOOSE } x :\ \lor\ p_1 \land x = e_1 \\
&\qquad\qquad\qquad\ \lor\ p_2 \land x = e_2 \\
&\qquad\qquad\qquad\ \lor\ \cdots \\
&\qquad\qquad\qquad\ \lor\ p_n \land x = e_n
\end{aligned}$$

$$\begin{aligned}
&\text{CASE } p_1 \to e_1,\ \ldots,\ p_n \to e_n,\ \text{OTHER} \to e \\
&\quad \triangleq \text{CASE } p_1 \to e_1,\ \ldots,\ p_n \to e_n,\ (\neg p_1 \land \cdots \land \neg p_n) \to e
\end{aligned}$$

$$\begin{aligned}
&\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2 \\
&\quad \triangleq \text{CASE } p \to e_1,\ \text{OTHER} \to e_2
\end{aligned}$$

Note that if several conditions p_i are true, then a case expression is ambiguous as it may be equal to any e_i. Users of TLA$^+$ often write case expressions with conditions that obviously exclude each other, so most case expressions are unambiguous in practice.

Set Theory

We declare the binary operator $\mathsf{in} : (0, 0)$. The expressions $\mathsf{in}(x, y)$ and $\neg\mathsf{in}(x, y)$ are respectively written $x \in y$ and $x \notin y$. TLA$^+$'s notations for bounded quantification are defined

$$\begin{aligned}
\forall x \in e_1 : e_2 &\triangleq \forall x : x \in e_1 \Rightarrow e_2 \\
\exists x \in e_1 : e_2 &\triangleq \exists x : x \in e_1 \land e_2 \\
\text{CHOOSE } x \in e_1 : e_2 &\triangleq \text{CHOOSE } x : x \in e_1 \land e_2
\end{aligned}$$

The axiom of set extensionality is specified for every object in the domain, in accordance with the view that every mathematical object is a set:

$$\forall x, y : (\forall z : z \in x \Leftrightarrow z \in y) \Rightarrow x = y$$

The theory admits an operator $\mathsf{subseteq} : (0, 0)$ for the subset relation. The expression $\mathsf{subseteq}(x, y)$ is written $x \subseteq y$ and specified by

$$\forall x, y : x \subseteq y \Leftrightarrow (\forall z : z \in x \Rightarrow z \in y)$$

The rest of the formalization follows the traditional development of ZFC set theory. Here is a summary of the operators of set theory with their shapes and corresponding TLA$^+$ notations:

Operator	Shape	Notation
union	(0)	$\textsc{union}\ e \triangleq \mathsf{union}(e)$
subset	(0)	$\textsc{subset}\ e \triangleq \mathsf{subset}(e)$
enum_n	$(0, \ldots, 0)$	$\{e_1, \ldots, e_n\} \triangleq \mathsf{enum}_n(e_1, \ldots, e_n)$
		$\emptyset \triangleq \mathsf{enum}_0$
setst	$(0, 1)$	$\{x \in e_1 : e_2\} \triangleq \mathsf{setst}(e_1, \lambda x : e_2)$
setof_n	$(0, \ldots, 0, n)$	$\{e_{n+1} : x_1 \in e_1, \ldots, x_n \in e_n\} \triangleq \mathsf{setof}_n(e_1, \ldots, e_n, \lambda x_1, \ldots, x_n : e_{n+1})$
cup	$(0, 0)$	$e_1 \cup e_2 \triangleq \mathsf{cup}(e_1, e_2)$
cap	$(0, 0)$	$e_1 \cap e_2 \triangleq \mathsf{cap}(e_1, e_2)$
diff	$(0, 0)$	$e_1 \setminus e_2 \triangleq \mathsf{diff}(e_1, e_2)$

Note that enum and setof represent two infinite families of operators, indexed by a natural number n. In both cases the shape consists of a list of zeros of length n. For enum we have $n \geq 0$ while for setof we have $n \geq 1$. The ambiguous notation $\{x \in a : x \in b\}$ is interpreted as $\mathsf{setst}(a, \lambda x : x \in b)$.

The operators $\{\cdot\}$ (set enumeration), $\textsc{union}$, $\textsc{subset}$, $\{x \in \cdot : \cdot\}$ (set comprehension) and $\{\cdot : x \in \cdot\}$ (set replacement) are defined by their axioms and axiom schemas:

$$\forall a, b, x : x \in \{a, b\} \Leftrightarrow x = a \vee x = b$$
$$\forall a, x : x \in \textsc{union}\ a \Leftrightarrow (\exists y : y \in a \wedge x \in y)$$
$$\forall a, x : x \in \textsc{subset}\ a \Leftrightarrow x \subseteq a$$
$$\forall a, x : x \in \{y \in a : P(y)\} \Leftrightarrow x \in a \wedge P(x)$$
$$\forall a, x : x \in \{F(y) : y \in a\} \Leftrightarrow (\exists y \in a : x = F(y))$$

P and F are operator parameters. We only show the axioms for enum_2 and setof_1 for the sake of simplicity, but the general schemas for all n are easy to infer.

The operators $\cup$, $\cap$ and $\setminus$ can be specified in two ways. The first solution is to give an axiom that characterizes the elements of each set, for instance:

$$\forall a, b, x : x \in a \cup b \Leftrightarrow x \in a \vee x \in b$$

The second solution is to provide an actual definition for the set:

$$\forall a, b : a \cup b = \textsc{union}\ \{a, b\}$$

Both solutions are easily shown equivalent. We prefer the second solution for defining T, as it makes clear that the additional operators do not make the theory stronger. The following definitions are admitted:

$$a \cup b \triangleq \text{UNION } \{a, b\}$$
$$a \cap b \triangleq \{x \in a : x \in b\}$$
$$a \setminus b \triangleq \{x \in a : x \notin b\}$$

So far our axioms make T correspond to the theory ZF minus the axiom of infinity. The final theory contains that axiom and also the axiom of foundation:

$$\exists x : \emptyset \in x \wedge \forall y : y \in x \Rightarrow y \cup \{y\} \in x \tag{Inf}$$
$$\forall x : x \neq \emptyset \Rightarrow \exists y : y \in x \wedge y \cap x = \emptyset \tag{AF}$$

The resulting theory corresponds to ZFC + AF, where C represents the axiom of choice:

$$\begin{aligned} &\forall x : \emptyset \notin x \wedge (\forall y, z \in x : y \neq z \Rightarrow y \cap z = \emptyset) \Rightarrow \\ &\quad \exists u : \forall v \in x : \exists w : u \cap v = \{w\} \end{aligned} \tag{AC}$$

The axiom (AC) is independent from ZF + AF, but in the context of the epsilon-calculus it can be proved. Given a set x whose elements are non-empty sets all disjoint to each other, we can define

$$u \triangleq \{(\text{CHOOSE } y : y \in v) : v \in x\}$$

Let $v \in x$ and $w \triangleq \text{CHOOSE } y : y \in v$. Since $\emptyset \notin x$ we have $w \in v$, thus $w \in u \cap v$. Now suppose $w' \in u \cap v$. Then there exists a $v' \in x$ such that $w' = \text{CHOOSE } y : y \in v'$. Since $w' \in v$ and all elements of x are disjoint, we have $v' = v$, thus $w' = w$. Therefore $u \cap v = \{w\}$.

Functions

Functions are defined axiomatically in TLA$^+$. The class of functions is identified by a unary operator isafcn, which is not part of the user syntax. Here is a summary of the most important operators:

Operator	Shape	Notation
isafcn	(0)	
fcn	$(0, 1)$	$[x \in e_1 \mapsto e_2] \triangleq \mathsf{fcn}(e_1, \lambda x : e_2)$
domain	(0)	$\text{DOMAIN } e \triangleq \mathsf{domain}(e)$
fcnapp	$(0, 0)$	$e_1[e_2] \triangleq \mathsf{fcnapp}(e_1, e_2)$
arrow	$(0, 0)$	$[e_1 \to e_2] \triangleq \mathsf{arrow}(e_1, e_2)$

Functions are built using the constructor $[x \in e_1 \mapsto e_2]$. The domain of a function f is accessed through DOMAIN. The value at x is $f[x]$. This is specified by the following axioms:

$$\forall a : \mathsf{isafcn}([x \in a \mapsto F(x)])$$
$$\forall a : \text{DOMAIN } [x \in a \mapsto F(x)] = a$$
$$\forall a, x : x \in a \Rightarrow [z \in a \mapsto F(z)][x] = F(x)$$

where F is an operator parameter.

Two functions with the same domain and the same values on that shared domain are specified as equal. This is the principle of functional extensionality, which is formally written

$$\begin{aligned}\forall f, g : \ & \wedge \mathsf{isafcn}(f) \wedge \mathsf{isafcn}(g) \\ & \wedge \text{DOMAIN } f = \text{DOMAIN } g \\ & \wedge (\forall x : x \in \text{DOMAIN } f \Rightarrow f[x] = g[x]) \\ & \Rightarrow f = g\end{aligned}$$

Finally, the set of functions from a to b, written $[a \to b]$, is specified by

$$\begin{aligned}\forall a, b, f : f \in [a \to b] \Leftrightarrow \ & \wedge \mathsf{isafcn}(f) \\ & \wedge \text{DOMAIN } f = a \\ & \wedge (\forall x : x \in a \Rightarrow f[x] \in b)\end{aligned}$$

In TLA$^+$'s reference book, the operator isafcn has an actual definition:

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

The proposition below justifies our approach.

Proposition 3.3.3. *The following TLA$^+$ expressions are equivalent:*

(i) $\mathsf{isafcn}(f)$

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

(iii) $f \in [\text{DOMAIN } f \to \text{Im}(f)]$ *where* $\text{Im}(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

(iv) $\exists a, b : f \in [a \to b]$

Proof. We prove $(i) \Rightarrow (ii)$ using extensionality on f and $[x \in \text{DOMAIN } f \mapsto f[x]]$. Both objects are in the class isafcn, have DOMAIN f for domain, and $f[x]$ for value at $x \in \text{DOMAIN } f$. Therefore, they are equal.

For proving $(ii) \Rightarrow (iii)$, it suffices to prove $[x \in \text{DOMAIN } f \mapsto f[x]] \in [\text{DOMAIN } f \to \text{Im}(f)]$. This follows from the axioms of functions and the axiom of set replacement.

The proof of $(iii) \Rightarrow (iv)$ is trivial. $(iv) \Rightarrow (i)$ follows from the axiom specifying $[a \to b]$. □

The equivalence $(i) \Leftrightarrow (ii)$ demonstrates that our formalization is compatible with the reference book. The proposition above presents two other possible definitions, both of which use the notion of sets of functions. (iii) is only more precise than (iv). Note that if $f \in [a \to b]$, then a is necessarily DOMAIN f, whereas b can be any set that contains $\text{Im}(f)$.

Arithmetic

TLA$^+$ includes the arithmetics of naturals, integers and reals. The three sets *Nat*, *Int* and *Real* are defined with their intended structure in such a way that $Nat \subseteq Int \subseteq Real$. The numerical constants and operations of arithmetic are overloaded for the structures in which they are specified. For instance, 1 is specified as an element of *Nat*, but as an element of *Int* or *Real* it still represents the same number. Likewise, the operation $1 + 1 = 2$ can be carried out in *Nat*,

but + is also specified as addition on *Int* and *Real*, and those three possible interpretations of + coincide.

The construction of the three structures is detailed in the reference book [48, Section 18.4]. We will not reproduce it here as it is both tricky and not important for our purposes. We simply provide the list of operators with some clarifications around their semantics:

- The constants 0, 1, 2, etc. are specified as elements of *Nat*. Non-negative decimal constants like 3.14 are specified as elements of *Real*.
- The operator $\leq$ is specified on all sets. The following definitions hold for all x and y:

$$x < y \triangleq x \leq y \wedge x \neq y$$
$$x \geq y \triangleq y \leq x$$
$$x > y \triangleq y \leq x \wedge x \neq y$$

- The operators $+$, $-$ (unary and binary versions), $*$ and $\hat{}$ are specified on all sets. There are no numerical constants for negative numbers; -1 is defined as $-(1)$.
- The operator / is division on *Real*.
- The operators $\div$ and $\%$ are the Euclidean quotient and remainder. They are defined for all $a \in Int$ and $b \in Nat \setminus \{0\}$ and satisfy the equations

$$a = b * (a \div b) + (a \,\%\, b)$$
$$0 \leq (a \,\%\, b) \wedge (a \,\%\, b) < b$$

The last operator we need to introduce is range : $(0, 0)$. The expression $\mathsf{range}(m, n)$ is noted $m \,..\, n$ in TLA$^+$. It represents the interval of integers between m and n (included). Its definition is

$$m \,..\, n \triangleq \{p \in Int : m \leq p \wedge p \leq n\}$$

Other Constructs

The set BOOLEAN is defined by

$$\text{BOOLEAN} \triangleq \{\text{TRUE}, \text{FALSE}\}$$

Tuples are defined from functions and Cartesian products are sets of tuples:

$$\langle x, y\rangle \triangleq [\, i \in 1 \,..\, 2 \mapsto \text{CASE } i = 1 \text{ THEN } x \quad \text{CASE } i = 2 \text{ THEN } y \,]$$
$$a \times b \triangleq \{\langle x, y\rangle : x \in a, y \in b\}$$

The general constructs $\langle x_1, \ldots, x_n\rangle$ and $a_1 \times \cdots \times a_n$ are defined in the same manner. Note that tuples are defined for all n, but Cartesian products are only defined for $n \geq 2$, because there are no notations for $n = 0$ or $n = 1$.

For every string of ASCII characters, for example "foo", there is a TLA$^+$ constant "`foo`". All those constants are specified as elements of some set STRING. Furthermore, "`foo`" is

specified to be a tuple of length 3 whose components are unspecified objects representing the characters f, o and o, in that order. Thus "`foo`"[2] = "`foo`"[3], and "`foo`"[1] $\neq$ "`foo`"[2].

Records and sets of records are defined in a way analogous to tuples. The domain of a record is a finite set of strings:

$$[foo \mapsto x, bar \mapsto y] \triangleq [\ s \in \{\text{“}\texttt{foo}\text{”}, \text{“}\texttt{bar}\text{”}\} \mapsto \text{CASE } s = \text{“}\texttt{foo}\text{”} \text{ THEN } x \\ \text{CASE } s = \text{“}\texttt{bar}\text{”} \text{ THEN } y\ ]$$

$$[foo : a, bar : b] \triangleq \{[foo \mapsto x, bar \mapsto y] : x \in a, y \in b\}$$

The list of fields can be of any length $n > 0$ but must not contain duplicates.

3.4 Sequents

The logic $\mathcal{L} + T$ is adequate to study TLA$^+$. However, the proof obligations generated by TLA$^+$ are expressed in the form of *sequents*, which are not part of that logic.

Definition 3.4.1 (Sequents)**.** We define *contexts* and *sequents* using the following grammar:

$$\Gamma ::= \cdot \mid \Gamma, \text{NEW } K : \kappa \mid \Gamma, e \mid \Gamma, S \qquad \text{(Contexts)}$$
$$S ::= \Gamma \vdash e \qquad \text{(Sequents)}$$

The expressions e and sequents S that occur in a context are called *hypotheses*. The expression e after $\vdash$ is called the *goal*. The subexpressions NEW $K : \kappa$ are *declarations*, and we impose two constraints on them: there can be at most one declaration of a given K, including in nested sequents; the shape κ must be first-order.

Every context Γ can be naturally treated as a signature by taking $\mathrm{Dom}(\Gamma)$ as the collection of symbols K such that NEW $K : \kappa$ is in Γ and then $\Gamma(K) \triangleq \kappa$. Let Σ be a signature. By definition, Σ and Γ are compatible iff $\mathrm{Dom}(\Sigma)$ and $\mathrm{Dom}(\Gamma)$ are disjoint. In that case $\Sigma + \Gamma$ is defined as a signature.

Given a signature Σ, we define well-formedness for contexts and sequents:

- The empty context $\cdot$ is always well-formed;
- $\Gamma, \text{NEW } K : \kappa$ is well-formed iff Γ is well-formed and $K \notin \mathrm{Dom}(\Gamma)$;
- Γ, e is well-formed iff Γ is and e is a well-formed $(\Sigma + \Gamma)$-expression;
- Γ, S is well-formed iff Γ is well-formed and S is a well-formed sequent in the context of the signature $\Sigma + \Gamma$;
- The sequent $\Gamma \vdash e$ is well-formed iff the context Γ, e is.

In the syntax of TLA$^+$, a declaration NEW $K : (0, 0)$ is written NEW $K(\_, \_)$, and similarly for shapes of any length. If the declared K is constant we just write NEW K. A sequent $\Gamma \vdash e$ is written ASSUME Γ PROVE e. The notation NEW $x \in e$ is short for the declaration NEW x followed by the hypothesis $x \in e$.

Example 3.4.2. Here is an example of a TLA$^+$ sequent:

$$\begin{array}{ll} \text{ASSUME} & \text{NEW } P(\_), \\ & \text{NEW } S, \\ & \text{NEW } x \in S, \\ & P(x) \\ \text{PROVE} & \{y \in S : P(y)\} \neq \emptyset \end{array}$$

Suppose the sequent above corresponds to the statement of a theorem named *MyThm* which has been proved by the user. That result may be invoked in proofs further down the specification. For instance, one may want to prove $\{n \in Nat : n > 100\} \neq \emptyset$ from *MyThm* and the fact $101 > 100$. This will result in an obligation with one nested sequent:

$$\begin{array}{lll} \text{ASSUME} & \text{ASSUME} & \text{NEW } P(\_), \\ & & \text{NEW } S, \\ & & \text{NEW } x \in S, \\ & & P(x) \\ & \text{PROVE} & \{y \in S : P(y)\} \neq \emptyset\,, \\ & 101 > 100 & \\ \text{PROVE} & \{n \in Nat : n > 100\} \neq \emptyset & \end{array}$$

The idea of the proof is to instantiate P with the higher-order argument $\lambda n : n > 100$. This demonstrates how second-order logic becomes necessary for some proof obligations.

Definition 3.4.3 (Satisfaction for Sequents)**.** Let Σ be a signature, I a Σ-interpretation, and Γ a context well-formed under Σ. A Γ-extension of I is a $(\Sigma + \Gamma)$-interpretation I' on the same domain and such that $K^{I'} = K^I$ for all $K \in \mathrm{Dom}(\Sigma)$.

A sequent $\Gamma \vdash e$ is satisfied by I if every Γ-extension of I that is a model of the hypotheses in Γ also satisfies e. This is noted $I, \Gamma \models e$. The sequent is called valid if it is satisfied by all interpretations; we note this $\Gamma \models e$.

Contexts are similar to theories as they contain the declarations of a signature and the expressions of a collection of axioms. We view contexts as a way to encode explicit information about a theory: a backend solver taking the encoded obligations as input does not have access to the theory T, so we must select relevant declarations and axioms to insert in Γ. Since T is infinite and ZF cannot be characterized by a finite number of axioms, encodings of TLA$^+$ into first-order logic are necessarily incomplete.

Chapter 4

A Direct Encoding of TLA$^+$ into Higher-order Logic

4.1 Overview

In the previous chapter, we formalized TLA$^+$ as a logic $\mathcal{L} + T$, where $\mathcal{L}$ is a variant of FOL without formulas and T is a standard theory on top of $\mathcal{L}$. While TLA$^+$ is essentially a first-order logic, it features second-order applications. Moreover, if we consider the language of sequents, we have to account for second-order quantifiers. For these reasons, the most natural encoding of TLA$^+$ is an encoding into HOL. This chapter describes that encoding, which has been implemented in TLAPS in the form of the Zipperposition backend [27].

Figure 4.1 shows a diagram of the different steps of the encoding. The logic $\mathcal{L}^\iota$ is a sorted variant of $\mathcal{L}$ (Section 4.2), and T' is the result of encoding T into $\mathcal{L}^\iota$. The first step of the encoding is a simple pass over expressions to recover the usual semantics for Boolean connectives (Section 4.3). It is described as a sound and complete encoding of $\mathcal{L}$ into $\mathcal{L}^\iota$. The optional rewriting step, which is adapted from the original SMT encoding, attempts to eliminate the primitive TLA$^+$ constructs of set theory, functions, arithmetic (Section 4.4). We provide its definition as a rewriting system and prove its termination and confluence. The axiomatization step inserts explicit declarations and axioms in the obligation's context (Section 4.5). The last step is a direct translation of sequents in the TPTP language, more precisely the THF dialect (Section 4.6). We conclude this chapter by an evaluation of the encoding on a large set of TLA$^+$ specifications (Section 4.7).

Example 4.1.1. To illustrate each step of the encoding, we introduce a running example. It is based on the following TLA$^+$ definition of sets of partial functions:

$$PFunc(A, B) \triangleq \text{UNION } \{[X \to B] : X \in \text{SUBSET } A\}$$

The elements of a set $PFunc(A, B)$ can be characterized without any reference to set replacement or union:

$$\begin{aligned} f \in PFunc(A, B) \Leftrightarrow\ & \wedge f \in [\text{DOMAIN } f \to B] \\ & \wedge (\text{DOMAIN } f) \subseteq A \end{aligned}$$

We will focus on the obligation corresponding to the $\Leftarrow$ direction of that equivalence. Figure 4.2 shows a working snippet of code in TLA$^+$ with the definition and the proved

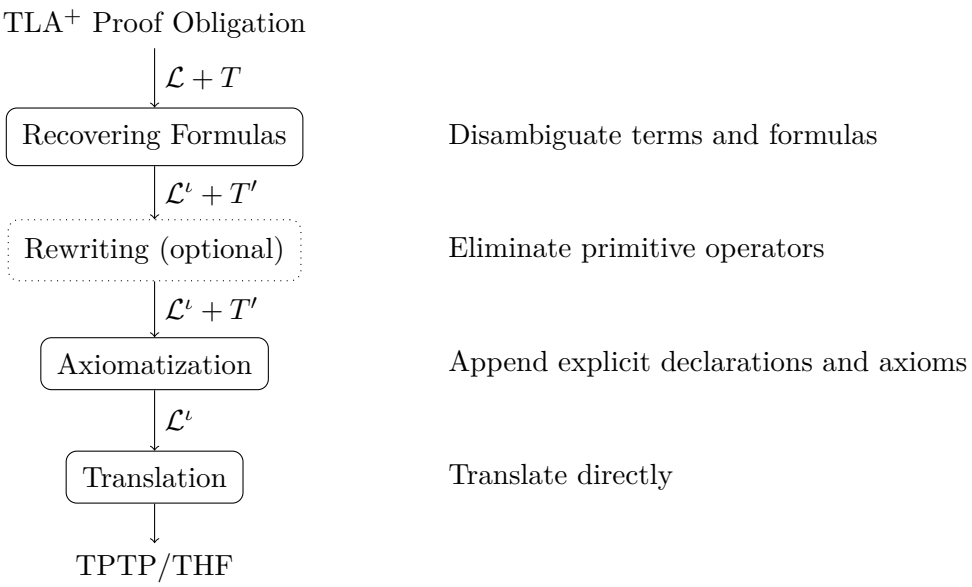


Figure 4.1: Encoding Overview

MODULE *PartialFunctions*

EXTENDS *TLAPS*

$PFunc(A,\, B) \triangleq$ UNION $\{[X \to B] : X \in$ SUBSET $A\}$

THEOREM ASSUME NEW A, NEW B, NEW f,
$f \in [\text{DOMAIN } f \to B]$,
DOMAIN $f \subseteq A$
PROVE $f \in PFunc(A,\, B)$
BY *Zipper* DEF *PFunc*

Figure 4.2: Running Example

result. The keyword *Zipper* in the proof line invokes the Zipperposition backend that is based on our encoding.

TLAPS generates a single proof obligation by expanding the definition of *PFunc* in the original sequent. To clarify the structure of expressions, we will use our notations for TLA$^+$'s primitive constructs. Without the readable TLA$^+$ notations, the proof obligation is written:

$$\begin{array}{ll} \text{ASSUME} & \text{NEW } A,\ \text{NEW } B,\ \text{NEW } f, \\ & \mathsf{in}(f, \mathsf{arrow}(\mathsf{domain}(f), B)), \\ & \mathsf{subseteq}(\mathsf{domain}(f), A) \\ \text{PROVE} & \mathsf{in}(f, \mathsf{union}(\mathsf{setof}_1(\mathsf{subset}(A), \lambda X : \mathsf{arrow}(X, B)))) \end{array}$$

4.2 The Logic $\mathcal{L}^s$

We view $\mathcal{L}^\iota$ as a fragment of a more general logic $\mathcal{L}^s$, which is defined in this section. $\mathcal{L}^s$ is a multi-sorted version of $\mathcal{L}$ enjoying the traditional semantics for Boolean connectives. For the TPTP encoding, we use only one sort other than the Boolean sort, which is noted ι and called the sort of *individuals*. In the next chapter about the SMT encoding, we will add the sort int for integer numbers.

Much of the content of this section is adapted from Section 3.2 on the logic $\mathcal{L}$. The major differences consist in the presence of sort annotations for bound variables, and the restriction of expressions by a type system. Additionally, we introduce *substitutions* and define the substitution of free variables as an operation on terms of $\mathcal{L}^s$.

Syntax

Definition 4.2.1 (Types)**.** Let $\mathcal{S}$ be a non-empty collection and o a symbol not in $\mathcal{S}$. The elements of $\mathcal{S}$ are called *sorts* and o is the *Boolean sort*. Types are defined by the grammar

$$\tau ::= s \mid \tau \times \cdots \times \tau \to s \qquad \text{where } s \in S \cup \{o\}$$

The order of a type is defined by

$$\begin{aligned} \mathrm{ord}(s) &\triangleq 0 \\ \mathrm{ord}(\tau_1 \times \cdots \times \tau_n \to s) &\triangleq \max_{1 \le i \le n} (\mathrm{ord}(\tau_i)) + 1 \qquad \text{when } n > 0 \end{aligned}$$

A sort is also a type of order 0. When $\tau = \tau_1 \times \cdots \times \tau_n \to s$ and $n = 0$, we write $\tau = s$ and we have $\mathrm{ord}(\tau) = 0$. The types of order 1 and 2 are respectively called first-order and second-order types.

For $\mathcal{L}^s$, we will not consider types of order greater than 2. Our syntax for TLA$^+$ was based on two syntactical classes: e for expressions and f for arguments. In $\mathcal{L}^s$, well-formed expressions are assigned a sort; well-formed arguments are assigned a type τ such that $\mathrm{ord}(\tau) \le 1$. The sorts and first-order types characterize arguments, so we call them *argument types*.

Definition 4.2.2 (Expressions)**.** A signature in $\mathcal{L}^s$ is a map Σ from operator symbols to types; if $K \in \mathrm{Dom}(\Sigma)$ then $\Sigma(K)$ is a type. The new syntax of expressions and arguments is

$$\frac{\Gamma(x) = s}{\Gamma \vdash x : s}\ \textsc{Var} \qquad \frac{\Sigma(K) = \tau_1 \times \cdots \times \tau_n \to s \qquad \Gamma \vdash f_i : \tau_i}{\Gamma \vdash K(f_1, \ldots, f_n) : s}\ \textsc{App} \qquad \frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 = e_2 : o}\ \textsc{Eq}$$

$$\frac{}{\Gamma \vdash \textsc{false} : o}\ \textsc{False} \qquad \frac{\Gamma \vdash e_1 : o \qquad \Gamma \vdash e_2 : o}{\Gamma \vdash e_1 \Rightarrow e_2 : o}\ \textsc{Imp} \qquad \frac{\Gamma, x : s \vdash e : o}{\Gamma \vdash (\forall x^s : e) : o}\ \textsc{ForAll}$$

$$\frac{\Sigma(K) = \tau \qquad \mathrm{ord}(\tau) = 1}{\Gamma \vdash K : \tau}\ \textsc{OpArg} \qquad \frac{\Gamma, x_1 : s_1, \ldots, x_n : s_n \vdash e : s}{\Gamma \vdash (\lambda x_1^{s_1}, \ldots, x_n^{s_n} : e) : s_1 \times \cdots \times s_n \to s}\ \textsc{LamArg}$$

Figure 4.3: Typing Rules for $\mathcal{L}^s$

defined by the grammar

$$e ::= x \mid K(f, \ldots, f) \mid e = e \mid \textsc{false} \mid e \Rightarrow e \mid \forall x^s : e \qquad \text{(Expressions)}$$
$$f ::= e \mid K \mid \lambda x^s, \ldots, x^s : e \qquad \text{(Arguments)}$$

A *typing context* is a map Γ from some finite collection of variables to sorts. We note $\Gamma, x : s$ the updated context that maps x to s. We define two kinds of *typing judgments*:

- $\Gamma \vdash e : s$ means the expression e has sort s;
- $\Gamma \vdash f : \tau$ where $\mathrm{ord}(\tau) \leq 1$ means the argument f has type τ.

We note $\vdash e : s$ and $\vdash f : \tau$ when the typing context is empty. The valid typing judgments are defined by the inductive rules of Figure 4.3. An expression or argument is well-formed in some typing context if it has a typing with that context. A *formula* is a well-formed expression whose sort is o. A *predicate* is a well-formed argument whose type ends with o.

Example 4.2.3. We note $K : \tau$ to mean that the operator K has type τ in some signature. Consider a signature Σ with operators $c : s$, $F : s \to s$ and $G : (s \to s) \times s \to s$. Then the term

$$G(F, c) = G(\lambda x^s : F(x), c)$$

is a well-formed formula. This is shown by the following derivation:

$$\dfrac{\dfrac{\dfrac{\Sigma(F) = s \to s}{\vdash F : s \to s}\ \textsc{OpArg} \qquad \dfrac{\Sigma(c) = s}{\vdash c : s}\ \textsc{App}}{\vdash G(F, c) : s}\ \textsc{App} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Sigma(F) : s \to s \qquad \dfrac{}{x : s \vdash x : s}\ \textsc{Var}}{x : s \vdash F(x) : s}\ \textsc{App}}{\vdash (\lambda x^s : F(x)) : s \to s}\ \textsc{LamArg} \qquad \dfrac{\Sigma(c) = s}{\vdash c : s}\ \textsc{App}}{\vdash G(\lambda x^s : F(x), c) : s}\ \textsc{App}}{\vdash G(F, c) = G(\lambda x^s : F(x), c) : o}\ \textsc{Eq}$$

In the example above, the typing $\vdash c : s$ is derived from the rule App, because a constant by itself is read as an application with zero arguments. The same judgment cannot be derived with the rule OpArg because the condition $\mathrm{ord}(\tau) = 1$ does not hold for constants.

Let Γ_1 and Γ_2 be two typing contexts. We write $\Gamma_1 \subseteq \Gamma_2$ when $\mathrm{Dom}(\Gamma_1) \subseteq \mathrm{Dom}(\Gamma_2)$ and $\Gamma_1(x) = \Gamma_2(x)$ for all $x \in \mathrm{Dom}(\Gamma_1)$. Clearly, any valid typing derivation stays valid when the context Γ is extended with fresh variables in every node. This is expressed by the following property, which we admit:

Proposition 4.2.4. *Suppose $\Gamma_1 \subseteq \Gamma_2$. Then $\Gamma_1 \vdash e : s$ implies $\Gamma_2 \vdash e : s$ for all e, and $\Gamma_1 \vdash f : \tau$ implies $\Gamma_2 \vdash f : \tau$ for all f.*

Substitutions

If $F(x, y, z)$ (for example) is a term with free variables x, y, z, and t_1, t_2, t_3 are three other terms, then the term $F(t_1, t_2, t_3)$ is the result of a syntactical operation called *substitution*.

Definition 4.2.5 (Substitution)**.** Let Γ be a typing context. A substitution is a function that assigns terms to variables. A substitution σ is adequate for Γ if $\mathrm{Dom}(\sigma) \subseteq \mathrm{Dom}(\Gamma)$ and $\Gamma \vdash \sigma(x) : \Gamma(x)$ for all x. If the domain of σ is strictly included in Γ, σ may be naturally extended to the whole context by defining $\sigma(x) = x$.

Let e be a term well-formed in Γ. A substitution σ is compatible with e if no bound variable of e occurs free in a term $\sigma(x)$. In that case, we define the application of σ to e, which is a term noted $e\sigma$, by recursion on e:

$$
\begin{aligned}
x\sigma &\triangleq \sigma(x) \\
K(f_1, \ldots, f_n)\sigma &\triangleq K(f_1\sigma, \ldots, f_n\sigma) \\
(e_1 = e_2)\sigma &\triangleq (e_1\sigma) = (e_2\sigma) \\
\textsc{false}\sigma &\triangleq \textsc{false} \\
(e_1 \Rightarrow e_2)\sigma &\triangleq (e_1\sigma) \Rightarrow (e_2\sigma) \\
(\forall x^s : e)\sigma &\triangleq \forall x^s : e\sigma^x
\end{aligned}
$$

$$
\begin{aligned}
K\sigma &\triangleq K \\
(\lambda x_1^{s_1}, \ldots, x_n^{s_n} : e)\sigma &\triangleq \lambda x_1^{s_1}, \ldots, x_n^{s_n} : e\sigma^{x_1,\ldots,x_n}
\end{aligned}
$$

where σ^x is defined as the substitution σ' such that $\sigma'(x) = x$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$. For all n, $\sigma^{x_1,\ldots,x_{n+1}}$ is defined as $(\sigma^{x_1,\ldots,x_n})^{x_{n+1}}$.

We write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the substitution with domain $\{x_1, \ldots, x_n\}$ that maps all x_i to their respective t_i.

To understand why compatibility with e is important in the definition above, consider a typing context $x : s, y : s$ and the expression $\forall x^s : P(x, y)$ where P is some binary predicate. The substitution $\{y \mapsto x\}$ is adequate for the typing context, but if we forget the compatibility condition and apply the substitution naively, we obtain

$$(\forall x^s : P(x, y))\,\{y \mapsto x\} = \forall x^s : P(x, x)$$

The variable y, which is free in the original formula, became bound in the result. We have not defined semantics yet, but we can explain informally why that substitution does not preserve interpretation: if s is interpreted as the domain of natural numbers, and P as the relation $\leq$, then the original formula is false (because $y + 1 \not\leq y$ in particular) while after the substitution it is true (by reflexivity of $\leq$).

This problem, which is called *variable capture*, is avoided by enforcing the compatibility condition between e and σ. The condition can always made to hold by renaming the bound variables of e is necessary. That is why we may accept the following definition as a result:

$$(\forall x^s : P(x, y))\,\{y \mapsto x\} = \forall z^s : P(z, x)$$

The variable z may be chosen among all the variable symbols that are not included in the typing context.

Proposition 4.2.6. *Suppose* $\Gamma \vdash e : s$ *and* σ *is a substitution adequate for* Γ *compatible with* e*. Then* $\Gamma \vdash e\sigma : s$*. The property holds analogously for well-formed arguments.*

Proof. This is proved by induction on the derivation of $\Gamma \vdash e : s$. We treat the variable, application and universal quantification case. All the other cases are trivial or analogous.

If the last rule is VAR then $e = x$, $s = \Gamma(x)$ and $e\sigma = \sigma(x)$. We have $\Gamma \vdash \sigma(x) : \Gamma(x)$ by assumption, which is exactly the same as $\Gamma \vdash e\sigma : s$.

If the last rule is APP then we have $e = K(f_1, \ldots, f_n)$ with $\Gamma \vdash f_i : \tau_i$ for all i and $\Sigma(K) = \tau_1 \times \cdots \times \tau_n \to s$. According to the induction hypothesis, $\Gamma \vdash f_i\sigma : \tau_i$ for all i. Therefore

$$\frac{\Sigma(K) = \tau_1 \times \cdots \times \tau_n \to s \qquad \Gamma \vdash f_i\sigma : \tau_i}{\Gamma \vdash K(f_1\sigma, \ldots, f_n\sigma) : s}\ \textsc{App}$$

But $e\sigma = K(f_1\sigma, \ldots, f_n\sigma)$ so the result is proved.

If the last rule is FORALL then $e = \forall x^s : e'$ with $\Gamma, x : s \vdash e' : o$. Clearly, σ^x is an adequate substitution for $\Gamma, x : s$. Therefore, according to the induction hypothesis, $\Gamma, x : s \vdash e'\sigma^x : o$. Then

$$\frac{\Gamma, x : s \vdash e'\sigma^x : o}{\Gamma \vdash \forall x^s : e'\sigma^x : o}\ \textsc{ForAll}$$

The result is obtained by $e\sigma = (\forall x^s : e'\sigma^x)$. □

Semantics

Compared to $\mathcal{L}$ and its unsorted semantics, $\mathcal{L}^s$ assigns a domain of evaluation for every sort and type. Formulas are always evaluated in the Boolean domain; sort annotations restrict the domain of evaluation of bound variables.

Definition 4.2.7 (Evaluation)**.** Let $\mathcal{S}$ be a collection of sorts. We define D_o as the two-valued collection whose elements are the Boolean values $\top$ and $\bot$. Given a family of collections $(D_s)_{s\in\mathcal{S}}$, we associate a domain to every type in the natural way:

$$D_{\tau_1\times\cdots\times\tau_n\to s} \triangleq D_{\tau_1} \times \cdots \times D_{\tau_n} \to D_s$$

Let Σ be some signature. An interpretation I consists of two families $(D_s)_{s\in\mathcal{S}}$ and $(K^I)_{K\in\mathrm{Dom}(\Sigma)}$ such that $K^I \in D_{\Sigma(K)}$ for all K. Let Γ be a typing context. A valuation θ is a function from $\mathrm{Dom}(\Gamma)$ such that $\theta(x) \in D_{\Gamma(x)}$ for all x. The evaluation of expressions and

arguments is defined by

$$\begin{aligned}
[\![x]\!]^I_\theta &\triangleq \theta(x)\\
[\![K(f_1,\ldots,f_n)]\!]^I_\theta &\triangleq K^I([\![f_1]\!]^I_\theta,\ldots,[\![f_n]\!]^I_\theta)\\
[\![e_1 = e_2]\!]^I_\theta &\triangleq \begin{cases}\top & \text{if } [\![e_1]\!]^I_\theta = [\![e_2]\!]^I_\theta\\ \bot & \text{otherwise}\end{cases}\\
[\![\textsc{false}]\!]^I_\theta &\triangleq \bot\\
[\![e_1 \Rightarrow e_2]\!]^I_\theta &\triangleq \begin{cases}\top & \text{if } [\![e_1]\!]^I_\theta = \bot \text{ or } [\![e_2]\!]^I_\theta = \top\\ \bot & \text{otherwise}\end{cases}\\
[\![\forall x^s : e]\!]^I_\theta &\triangleq \begin{cases}\top & \text{if } [\![e]\!]^I_{\theta^x_v} = \top \text{ for all } v \text{ in } D_s\\ \bot & \text{otherwise}\end{cases}
\end{aligned}$$

For all $v_1 \in D_{s_1}, \ldots, v_n \in D_{s_n}$:

$$\begin{aligned}
[\![K]\!]^I_\theta(v_1,\ldots,v_n) &\triangleq K^I(v_1,\ldots,v_n) \qquad \text{where } \Sigma(k) = s_1 \times \cdots \times s_n \to s\\
[\![\lambda x_1^{s_1},\ldots,x_n^{s_n} : e]\!]^I_\theta(v_1,\ldots,v_n) &\triangleq [\![e]\!]^I_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}
\end{aligned}$$

We admit the following result, which can be proved by induction on the structure of typing derivations:

Proposition 4.2.8. *Let Γ be a typing context, I an interpretation and θ a valuation. We have the following properties:*

- *For all e and s, if $\Gamma \vdash e : s$ then $[\![e]\!]^I_\theta \in D_s$*
- *For all f and τ, if $\Gamma \vdash f : \tau$ then $[\![f]\!]^I_\theta \in D_\tau$*

We proved (Proposition 4.2.6) that substitutions preserve typing. It follows that if $\Gamma \vdash e : s$ and σ is a substitution then $[\![e\sigma]\!]^I_\theta \in D_s$. However, we admit the following stronger property on the evaluation of $e\sigma$:

Proposition 4.2.9. *Let Γ, I and θ as in the previous proposition. Let σ be a substitution adequate for Γ. We define $\sigma\theta$ as a valuation for Γ by*

$$(\sigma\theta)(x) \triangleq [\![\sigma(x)]\!]^I_\theta$$

It follows from the definition of a substitution for Γ and Proposition 4.2.8 that $\sigma\theta$ is well-defined. The following equations hold for all e and f well-typed in Γ:

$$\begin{aligned}
[\![e\sigma]\!]^I_\theta &= [\![e]\!]^I_{\sigma\theta}\\
[\![f\sigma]\!]^I_\theta &= [\![f]\!]^I_{\sigma\theta}
\end{aligned}$$

Typing contexts carry the necessary typing information for the free variables of all expressions. It is clear then that annotations $x : s$ for variables x that do not occur free may be omitted. In particular, well-typed closed expressions can be typed in the empty context.

The letter ϕ denotes well-typed formulas. The satisfaction relation $\models$ for $\mathcal{L}^s$ is defined for closed formulas by

$$I \models \phi \text{ iff } [\![\phi]\!]^I = \top$$

We define the notations $T \models \phi$ and $\models \phi$ as before, where T is a collection of closed formulas.

The language and semantics of TLAPS sequents $\Gamma \vdash e$ are extended too. Let us briefly indicate how the definitions may be adapted. In Γ, the shape specifications that annotate declarations are replaced by type specifications; we write NEW $K : \tau$ for the declaration of K with type τ, where $\text{ord}(\tau) \leq 1$. It follows that the signature induced by a sequent context is a signature of $\mathcal{L}^s$, as it maps operator symbols to *types*. The hypotheses in Γ must be formulas or well-formed nested sequents. The goal must be a formula as well.

4.3 Recovering Formulas

Let $\mathcal{L}^\iota$ be the version of $\mathcal{L}^s$ for which $\mathcal{S}$ is reduced to one sort symbol, noted ι. A *term* is an expression $e : \iota$. We describe a sound and complete encoding of $\mathcal{L}$ into $\mathcal{L}^\iota$. We start with an elementary version of the encoding that systematically assigns types over the sort ι to all operators. For many TLA$^+$ primitive operators, it is desirable to use types with o; for example, $\in$ should be assigned a predicate type. We consider this feature as an extension and introduce it in the last section.

Before we start, let us give the basic intuition behind the encoding. All operators are assigned types corresponding to their original shapes, for instance the shape $(1, 0)$ is mapped to the type $(\iota \to \iota) \times \iota \to \iota$. All variables are annotated with ι. If we attempt to translate expressions from $\mathcal{L}$ to $\mathcal{L}^\iota$ directly in the most natural way, we will encounter two kinds of type errors:

TRUE $\in$ BOOLEAN	TRUE is a formula, but $\in$ expects a term
FALSE $\Rightarrow$ 42	42 is a term, but $\Rightarrow$ expects a formula

It suffices to rewrite the subexpressions causing those errors. The first kind is simpler to address. We add a new primitive operator $\mathsf{cast}_o : o \to \iota$ and rewrite:

$$\mathsf{cast}_o(\text{TRUE}) \in \text{BOOLEAN}$$

The second kind is trickier, as it seems counter-intuitive that any term could be translated as a formula in a sound way. However, we only need to convert terms when a formula is expected. This translation is sound:

$$\text{FALSE} \Rightarrow (42 = \mathsf{cast}_o(\text{TRUE}))$$

That conversion formalizes the intuition that the original TLA$^+$ formula is equivalent to FALSE $\Rightarrow$ (42 = TRUE) under the liberal interpretation of Boolean connectives. It suffices to use cast_o on TRUE to make the expression well-typed.

Example 4.3.1. Here is a simple valid example that illustrates both conversions:

$$\forall x : x = \text{FALSE} \Rightarrow \neg x \qquad \text{is encoded as} \qquad \forall x^\iota : x = \mathsf{cast}_o(\text{FALSE}) \Rightarrow \neg(x = \mathsf{cast}_o(\text{TRUE}))$$

Note that restricting quantification to o would not be sound in general: consider the non valid TLA$^+$ expression $\forall x : x = \text{TRUE} \vee x = \text{FALSE}$, which becomes a valid formula of $\mathcal{L}^\iota$ if we translate it as $\forall x^o : x = \text{TRUE} \vee x = \text{FALSE}$. The encoding we consider here will always annotate quantifiers with ι.

4.3.1 Definition

We define three mappings $\mathcal{B}^\iota$, $\mathcal{B}^o$ and $\mathcal{B}^{op}$. The first two are applied to expressions, the last one is applied to arguments. Before we define the mappings themselves, we define a mapping $\Sigma \mapsto \Sigma^{\mathcal{B}}$ that maps signatures of $\mathcal{L}$ to signatures of $\mathcal{L}^\iota$.

Definition 4.3.2. For all $n > 0$, let $\iota^n \to \iota$ denote the n-ary type $\iota \times \cdots \times \iota \to \iota$. The 0-ary type $\iota^0 \to \iota$ is just the sort ι. For all TLA$^+$ shape $\kappa = (n_1, \ldots, n_k)$, the type $\mathit{type}(\kappa)$ is defined as $\tau_1 \times \cdots \times \tau_k \to \iota$ where τ_i is the n_i-ary type.

Let Σ be a TLA$^+$ signature. We define $\Sigma^{\mathcal{B}}$ as the signature of $\mathcal{L}^\iota$ whose domain is $\mathrm{Dom}(\Sigma)$ plus the fresh symbol cast_o. If $K \in \mathrm{Dom}(\Sigma)$, then $\Sigma^{\mathcal{B}}(K) \triangleq \mathit{type}(\Sigma(K))$. $\Sigma^{\mathcal{B}}(\mathsf{cast}_o) \triangleq o \to \iota$.

For example, the TLA$^+$ operator setst (set comprehension) has the shape $(0, 1)$. Its corresponding type in $\Sigma^{\mathcal{B}}$ is $\iota \times (\iota \to \iota) \to \iota$. Note that for a Σ-interpretation and a $\Sigma^{\mathcal{B}}$-interpretation on the same domain D, if $K \in \mathrm{Dom}(\Sigma)$ then K is interpreted as a function of the collection $D \times (D \to D) \to D$ in both logics.

Definition 4.3.3 (Mappings)**.** The three mappings $\mathcal{B}^\iota$, $\mathcal{B}^o$ and $\mathcal{B}^{op}$ are defined recursively by the following rules:

$$\mathcal{B}^o(e_1 = e_2) \triangleq \mathcal{B}^\iota(e_1) = \mathcal{B}^\iota(e_2)$$
$$\mathcal{B}^o(\textsc{false}) \triangleq \textsc{false}$$
$$\mathcal{B}^o(e_1 \Rightarrow e_2) \triangleq \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2)$$
$$\mathcal{B}^o(\forall x : e) \triangleq \forall x^\iota : \mathcal{B}^o(e)$$
$$\mathcal{B}^o(e) \triangleq \mathcal{B}^\iota(e) = \mathsf{cast}_o(\textsc{true}) \qquad (\star)$$

$$\mathcal{B}^\iota(x) \triangleq x$$
$$\mathcal{B}^\iota(K(f_1, \ldots, f_n)) \triangleq K(\mathcal{B}^{op}(f_1), \ldots, \mathcal{B}^{op}(f_n))$$
$$\mathcal{B}^\iota(e) \triangleq \mathsf{cast}_o(\mathcal{B}^o(e)) \qquad (\star)$$

$$\mathcal{B}^{op}(e) \triangleq \mathcal{B}^\iota(e)$$
$$\mathcal{B}^{op}(K) \triangleq K \qquad (\star\star)$$
$$\mathcal{B}^{op}(\lambda x_1, \ldots, x_n : e) \triangleq \lambda x_1^\iota, \ldots, x_n^\iota : \mathcal{B}^\iota(e)$$

($\star$) We call these rules *conversions*. They are applied with lowest priority.
($\star\star$) Only applies when K is first-order. For constants, the rule just above applies.

The recursive definition above is not inductive, because the two conversion rules are not applied on a subexpression of the original expression. However, consider the last case of $\mathcal{B}^o(e)$, in which $\mathcal{B}^o(e)$ is defined as $\mathcal{B}^\iota(e) = \mathsf{cast}_o(\textsc{true})$. Then e is necessarily a variable or an application (otherwise, some non-conversion rule would apply to construct $\mathcal{B}^o(e)$). In that case, there is a non-conversion rule that applies to construct $\mathcal{B}^\iota(e)$. A similar argument can be made for the case where $\mathcal{B}^\iota(e)$ is constructed by converting $\mathcal{B}^o(e)$ into a term. This proves that a conversion can never immediately follow another in the recursive construction of $\mathcal{B}^\iota(e)$ and $\mathcal{B}^o(e)$, and so the recursive definition above is well-founded.

This also justifies proofs by induction on the *construction* of $\mathcal{B}^\iota(e)$, $\mathcal{B}^o(e)$ and $\mathcal{B}^{op}(f)$. To illustrate, we detail the proof of the next theorem.

Theorem 4.3.4. *If e is an expression of $\mathcal{L}$, we note Γ_e the typing context whose domain is $\mathrm{FV}(e)$ and such that $\Gamma_e(x) = \iota$ for all x free in e. The context Γ_f is defined analogously for all arguments f.*

For all Σ-expression e and Σ-argument f with arity n, the following properties hold:

(i) $\Gamma_e \vdash \mathcal{B}^\iota(e) : \iota$

(ii) $\Gamma_e \vdash \mathcal{B}^o(e) : o$

(iii) $\Gamma_f \vdash \mathcal{B}^{op}(f) : \iota^n \to \iota$

Proof. The proof is by induction on the construction of $\mathcal{B}^\iota(e)$, $\mathcal{B}^o(e)$ and $\mathcal{B}^{op}(f)$. There is one case for each equation in Definition 4.3.3. Depending on the term being constructed, we prove only i, ii or iii.

Variable Let $e = x$ and $\mathcal{B}^\iota(e) = x$. $\Gamma_e(x) = \iota$ by definition of Γ_e, thus $\Gamma_e \vdash x : \iota$ by application of the typing rule VAR.

Application Let $e = K(f_1, \ldots, f_p)$ and $\mathcal{B}^\iota(e) = K(\mathcal{B}^{op}(f_1), \ldots, \mathcal{B}^{op}(f_p))$. The induction hypothesis (property iii) states that $\Gamma_{f_i} \vdash \mathcal{B}^{op}(f_i) : \iota^n \to \iota$ where n is the arity of f_i.

Let $f'_i = \mathcal{B}^{op}(f_i)$. Then we must prove $\Gamma_e \vdash K(f'_1, \ldots, f'_p) : \iota$. Let $\Sigma(K) = (n_1, \ldots, n_p)$. By assumption, e is a well-formed application, so each f_i is n_i-ary. Moreover, it is clear that $\Gamma_{f_i} \subseteq \Gamma_e$. Therefore, according to the induction hypothesis, for all i we have $\Gamma_e \vdash f'_i : \iota^{n_i} \to \iota$. By definition, $\Sigma^{\mathcal{B}}(K) = \tau_1 \times \cdots \times \tau_n \to \iota$ where $\tau_i = \iota^{n_i} \to \iota$. Then the following derivation is valid:

$$\frac{\Sigma^{\mathcal{B}}(K) = \tau_1 \times \cdots \times \tau_n \to \iota \qquad \Gamma_e \vdash f'_i : \tau_i}{\Gamma_e \vdash K(f'_1, \ldots, f'_p) : \iota}\ \textsc{App}$$

Formula-to-term conversion Let $\mathcal{B}^\iota(e) = \mathsf{cast}_o(\mathcal{B}^o(e))$. The induction hypothesis (property ii) states that $\Gamma_e \vdash \mathcal{B}^o(e) : o$. The result is obtained by the derivation

$$\frac{\Sigma^{\mathcal{B}}(\mathsf{cast}_o) = o \to \iota \qquad \Gamma_e \vdash \mathcal{B}^o(e) : o}{\Gamma_e \vdash \mathsf{cast}_o(\mathcal{B}^o(e)) : \iota}\ \textsc{App}$$

Equality Let e be $e_1 = e_2$. The induction hypothesis (property i) states $\Gamma_{e_1} \vdash \mathcal{B}^\iota(e_1) : \iota$ and $\Gamma_{e_2} \vdash \mathcal{B}^\iota(e_2) : \iota$. The result $\Gamma_e \vdash \mathcal{B}^\iota(e_1) = \mathcal{B}^\iota(e_2) : o$ is immediate by $\Gamma_{e_i} \subseteq \Gamma_e$ ($i \in \{1, 2\}$) and using the typing rule EQ.

False Let $e = \text{FALSE}$. The result $\vdash \mathcal{B}^o(e) : o$ is immediate by $\mathcal{B}^o(e) = \text{FALSE}$.

Implication Let e be $e_1 \Rightarrow e_2$. The induction hypothesis (property ii) states that $\Gamma_{e_1} \vdash \mathcal{B}^o(e_1) : o$ and $\Gamma_{e_2} \vdash \mathcal{B}^o(e_2) : o$. The result $\Gamma_e \vdash \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2) : o$ is immediate by $\Gamma_{e_i} \subseteq \Gamma_e$ ($i \in \{1, 2\}$) and using the typing rule IMP.

Universal quantification Let e be $\forall x : e'$. The induction hypothesis (property ii) states that $\Gamma_{e'} \vdash \mathcal{B}^o(e') : o$. Clearly, $\Gamma_{e'} \subseteq \Gamma_e, x : \iota$ (the free variables of e' are among x and the free variables of e). Then $\Gamma_e \vdash (\forall x^\iota : \mathcal{B}^o(e')) : o$ by application of the rule FORALL.

Term-to-formula conversion Let $\mathcal{B}^o(e)$ be $\mathcal{B}^\iota(e) = \mathsf{cast}_o(\text{TRUE})$. The induction hypothesis (property i) states that $\Gamma_e \vdash \mathcal{B}^\iota(e) : \iota$. The result is obtained by the derivation

$$\frac{\Gamma_e \vdash \mathcal{B}^\iota(e) : \iota \qquad \dfrac{\Sigma^{\mathcal{B}}(\mathsf{cast}_o) = o \to \iota \qquad \Gamma_e \vdash \text{TRUE} : o}{\Gamma_e \vdash \mathsf{cast}_o(\text{TRUE}) : \iota}\ \textsc{App}}{\Gamma_e \vdash \mathcal{B}^\iota(e) = \mathsf{cast}_o(\text{TRUE}) : o}\ \textsc{Eq}$$

Argument (expression) We have $\Gamma_e \vdash \mathcal{B}^\iota(e) : \iota$ by the induction hypothesis (property i). Then $\Gamma_e \vdash \mathcal{B}^{op}(e) : \iota^0 \to \iota$ by $\mathcal{B}^{op}(e) = \mathcal{B}^\iota(e)$ and $\iota^0 \to \iota = \iota$.

Argument (operator symbol) Let K be a first-order operator with arity $n > 0$. By definition, $\Sigma^{\mathcal{B}}(K) = \iota^n \to \iota$ and then $\mathrm{ord}(\Sigma^{\mathcal{B}}(K)) = 1$. The result $\vdash K : \iota^n \to \iota$ is obtained by the rule OpArg.

Argument (lambda-expression) Let $f = \lambda x_1, \ldots, x_n : e$ and $\mathcal{B}^{op}(f) = \lambda x_1^\iota, \ldots, x_n^\iota : \mathcal{B}^\iota(e)$. The induction hypothesis (property i) states that $\Gamma_e \vdash \mathcal{B}^\iota(e) : \iota$. It is clear that $\Gamma_e \subseteq \Gamma_f, x_1 : \iota, \ldots, x_n : \iota$. Then $\Gamma_f \vdash \lambda x_1^\iota, \ldots, x_n^\iota : \mathcal{B}^\iota(e) : \iota^n \to \iota$ by application of LamArg.

□

4.3.2 Correctness

The encoding is defined by $\mathcal{B}^o$, which maps Σ-expressions of $\mathcal{L}$ to $\Sigma^{\mathcal{B}}$-expressions of $\mathcal{L}^\iota$. The soundness property states that the validity of $\mathcal{B}^o(e)$ entails the validity of e. The dual implication is completeness. Both properties are derived as corollaries of a result about the preservation of all values by $\mathcal{B}^\iota$, $\mathcal{B}^o$ and $\mathcal{B}^{op}$. For completeness, we also need to specify the operator cast_o in the target logic.

Soundness

We map Σ-interpretations to $\Sigma^{\mathcal{B}}$-interpretations in the natural way.

Definition 4.3.5. Let I be a Σ-interpretation with domain D. The $\Sigma^{\mathcal{B}}$-interpretation $I^{\mathcal{B}}$ is defined as follows:

- $D_\iota \triangleq D$;
- For all $K \in \mathrm{Dom}(\Sigma)$, $K^{I^{\mathcal{B}}} \triangleq K^I$. This is correct because $D_{\Sigma^{\mathcal{B}}(K)}$ equals $D_{\Sigma(K)}$;
- $\mathsf{cast}_o^{I^{\mathcal{B}}}$ is the function $\left\{\top \mapsto \top^D, \bot \mapsto \bot^D\right\}$.

$I^{\mathcal{B}}$ essentially provides the interpretation for the new operator cast_o, which maps the truth values of D_o to their counterparts in $D_\iota = D$.

By Theorem 4.3.4, for all expressions e, $\mathcal{B}^\iota(e)$ and $\mathcal{B}^o(e)$ are well-formed in the context Γ_e, and their respective sorts are ι and o. Consider a TLA$^+$ valuation θ for the logic $\mathcal{L}$. θ interprets variables as elements of D. Since $\Gamma_e(x) = \iota$ for all x and $D_\iota = D$, θ is an adequate valuation for the context Γ_e, and the values $[\![\mathcal{B}^\iota(e)]\!]_\theta^{I^{\mathcal{B}}}$ and $[\![\mathcal{B}^o(e)]\!]_\theta^{I^{\mathcal{B}}}$ are well-defined respectively as elements of D and D_o. Likewise, for all n-ary f then $[\![\mathcal{B}^{op}(f)]\!]_\theta^{I^{\mathcal{B}}}$ is well-defined as an element of $D^n \to D$ (or simply D if $n = 0$).

Theorem 4.3.6. *Let Σ be a TLA$^+$ signature and I a Σ-interpretation. For all valuations θ, expression e and argument f, we have the properties:*

(i) $[\![\mathcal{B}^\iota(e)]\!]_\theta^{I^{\mathcal{B}}} = [\![e]\!]_\theta^I$

(ii.a) $[\![\mathcal{B}^o(e)]\!]_\theta^{I^{\mathcal{B}}} = \top$ *iff* $[\![e]\!]_\theta^I = \top^D$

(ii.b) $[\![\mathcal{B}^o(e)]\!]_\theta^{I^\mathcal{B}} = \bot$ *implies* $[\![e]\!]_\theta^I = \bot^D$ *when* $\mathcal{B}^o(e)$ *is not obtained by conversion*

(iii) $[\![\mathcal{B}^{op}(f)]\!]_\theta^{I^\mathcal{B}} = [\![f]\!]_\theta^I$

Proof. The result is proved by induction on the construction of $\mathcal{B}^\iota(e)$, $\mathcal{B}^o(e)$ or $\mathcal{B}^{op}(f)$. For cases constructing $\mathcal{B}^\iota(e)$, we prove *i*. For cases constructing $\mathcal{B}^o(e)$, we prove *ii.a* and *ii.b*. For cases constructing $\mathcal{B}^{op}(f)$, we prove *iii*.

For all cases constructing $\mathcal{B}^o(e)$ except the case of term-to-formula conversion, *ii.a* entails *ii.b* immediately. That is because, in all those cases, e has a Boolean top connective, and the semantics of $\mathcal{L}$ is made in such a way that $[\![e]\!]_\theta^I \neq \top^D$ implies $[\![e]\!]_\theta^I = \bot^D$ for such e.

Variable The result is immediate since $\mathcal{B}^\iota(x)$ is x. Both values equal $\theta(x)$.

Application The result is also immediate since $K^{I^\mathcal{B}} = K^I$ and using the induction hypothesis to derive $[\![\mathcal{B}^{op}(f_i)]\!]_\theta^{I^\mathcal{B}} = [\![f_i]\!]_\theta^I$ for all arguments.

Formula-to-term conversion We have $\mathcal{B}^\iota(e) = \mathsf{cast}_o(\mathcal{B}^o(e))$. By construction, $\mathcal{B}^o(e)$ is not a term-to-formula conversion, because a conversion cannot be applied immediately after another. Therefore, according to the induction hypothesis, both *ii.a* and *ii.b* hold for $\mathcal{B}^o(e)$.

$$
\begin{aligned}
[\![\mathcal{B}^\iota(e)]\!]_\theta^{I^\mathcal{B}} &= [\![\mathsf{cast}_o(\mathcal{B}^o(e))]\!]_\theta^{I^\mathcal{B}} \\
&= \mathsf{cast}_o^{I^\mathcal{B}}([\![\mathcal{B}^o(e)]\!]_\theta^{I^\mathcal{B}}) \\
&= \begin{cases} \top^D & \text{if } [\![\mathcal{B}^o(e)]\!]_\theta^{I^\mathcal{B}} = \top \\ \bot^D & \text{if } [\![\mathcal{B}^o(e)]\!]_\theta^{I^\mathcal{B}} = \bot \end{cases} && \text{(by definition)} \\
&= \begin{cases} \top^D & \text{if } [\![e]\!]_\theta^I = \top^D \\ \bot^D & \text{if } [\![e]\!]_\theta^I = \bot^D \end{cases} && \text{(induction hypothesis)} \\
&= [\![e]\!]_\theta^I
\end{aligned}
$$

Equality

$$
\begin{aligned}
[\![\mathcal{B}^o(e_1 = e_2)]\!]_\theta^{I^\mathcal{B}} = \top \text{ iff } & [\![\mathcal{B}^\iota(e_1) = \mathcal{B}^\iota(e_2)]\!]_\theta^{I^\mathcal{B}} = \top \\
\text{iff } & [\![\mathcal{B}^\iota(e_1)]\!]_\theta^{I^\mathcal{B}} = [\![\mathcal{B}^\iota(e_2)]\!]^{I^\mathcal{B}} \\
\text{iff } & [\![e_1]\!]_\theta^I = [\![e_2]\!]_\theta^I && \text{(induction hypothesis)} \\
\text{iff } & [\![e_1 = e_2]\!]_\theta^I = \top^D
\end{aligned}
$$

False The result is immediate: $[\![\mathcal{B}^o(\text{FALSE})]\!]_\theta^{I^\mathcal{B}} = \bot$ and $[\![\text{FALSE}]\!]_\theta^I = \bot^D$

Implication

$$
\begin{aligned}
[\![\mathcal{B}^o(e_1 \Rightarrow e_2)]\!]_\theta^{I^\mathcal{B}} = \top \text{ iff } & [\![\mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2)]\!]_\theta^{I^\mathcal{B}} = \top \\
\text{iff } & [\![\mathcal{B}^o(e_1)]\!]_\theta^{I^\mathcal{B}} \neq \top \text{ or } [\![\mathcal{B}^o(e_2)]\!]_\theta^{I^\mathcal{B}} = \top \\
\text{iff } & [\![e_1]\!]_\theta^I \neq \top^D \text{ or } [\![e_2]\!]_\theta^I = \top^D && \text{(induction hypothesis)} \\
\text{iff } & [\![e_1 \Rightarrow e_2]\!]_\theta^I = \top^D
\end{aligned}
$$

Remark that only property *ii.a* of the induction hypothesis is needed.

Universal quantification

$$\begin{aligned}
[\![\mathcal{B}^o(\forall x : e)]\!]_\theta^{I^\mathcal{B}} = \top \text{ iff } & [\![\forall x : \mathcal{B}^o(e)]\!]_\theta^{I^\mathcal{B}} = \top \\
\text{iff } & [\![\mathcal{B}^o(e)]\!]_{\theta^x_v}^{I^\mathcal{B}} = \top \quad \text{for all } v \text{ in } D \\
\text{iff } & [\![e]\!]_{\theta^x_v}^{I} = \top^D \quad \text{for all } v \text{ in } D \quad \text{(induction hypothesis)} \\
\text{iff } & [\![\forall x : e]\!]_\theta^{I} = \top^D
\end{aligned}$$

Term-to-formula conversion

$$\begin{aligned}
[\![\mathcal{B}^\iota(e) = \mathsf{cast}_o(\text{TRUE})]\!]_\theta^{I^\mathcal{B}} = \top \text{ iff } & [\![\mathcal{B}^\iota(e)]\!]_\theta^{I^\mathcal{B}} = \top^D \\
\text{iff } & [\![e]\!]_\theta^{I} = \top^D \quad \text{(induction hypothesis)}
\end{aligned}$$

Argument (expression) If f is just an expression e then the result is immediate by i.

Argument (operator symbol) If f is some K then the result is immediate from $K^{I^\mathcal{B}} = K^I$

Argument (lambda-expression) If f is $\lambda x_1, \ldots, x_n : e$ then we have, for all $v_1, \ldots, v_n$:

$$\begin{aligned}
[\![\mathcal{B}^{op}(f)]\!]_\theta^{I^\mathcal{B}}(v_1, \ldots, v_n) &= [\![\lambda x_1, \ldots, x_n : \mathcal{B}^\iota(e)]\!]_\theta^{I^\mathcal{B}}(v_1, \ldots, v_n) \\
&= [\![\mathcal{B}^\iota(e)]\!]_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}^{I^\mathcal{B}} \\
&= [\![e]\!]_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}^{I} \qquad \text{(induction hypothesis)} \\
&= [\![\lambda x_1, \ldots, x_n : e]\!]_\theta^I(v_1, \ldots, v_n) \\
&= [\![f]\!]_\theta^I(v_1, \ldots, v_n)
\end{aligned}$$

Thus $[\![\mathcal{B}^{op}(f)]\!]_\theta^{I^\mathcal{B}} = [\![f]\!]_\theta^I$.

□

Corollary 4.3.7 (Soundness)**.** *Let e be a closed expression of $\mathcal{L}$. If $\mathcal{B}^o(e)$ is valid then e is also valid.*

Proof. Let I be a Σ-interpretation. By assumption, $I^\mathcal{B} \models \mathcal{B}^o(e)$, *i.e.* $[\![\mathcal{B}^o(e)]\!]^{I^\mathcal{B}} = \top$. By Theorem 4.3.6, $[\![e]\!]^I = \top^D$, *i.e.* $I \models e$. Therefore e is valid. □

Completeness

Completeness is the fact that $\mathcal{B}^o$ preserves validity from $\mathcal{L}$ to $\mathcal{L}^\iota$. It is dual to soundness, which states that $\mathcal{B}^o(e)$'s validity implies e's validity. We proved soundness as a corollary of Theorem 4.3.6 by using only the implication $\Rightarrow$ from case $(ii.a)$. We may derive completeness by using the other direction $\Leftarrow$.

However, it may be the case that $I^\mathcal{B} \models \mathcal{B}^o(e)$ for all I, but $J \not\models \mathcal{B}^o(e)$ for some J. The TLA$^+$ valid formula TRUE $\neq$ FALSE is encoded as $\mathsf{cast}_o(\text{TRUE}) \neq \mathsf{cast}_o(\text{FALSE})$, but the latter formula is not valid in $\mathcal{L}^\iota$, because there are interpretations J such that cast_o^J is not injective (any interpretation where D is reduced to one element suffices). To ensure completeness, we must put a constraint on the interpretation of cast_o in the target logic.

Lemma 4.3.8. *Consider the axiom*

$$\mathsf{cast}_o(\text{TRUE}) \neq \mathsf{cast}_o(\text{FALSE}) \tag{B}$$

The mapping $I \mapsto I^{\mathcal{B}}$ is surjective as a mapping of Σ-interpretations to $\Sigma^{\mathcal{B}}$-interpretations satisfying the axiom (B).

Proof. Every $I^{\mathcal{B}}$ satisfies (B) by definition of $\mathsf{cast}_o^{I^{\mathcal{B}}}$, so the mapping is well-defined.

Let J be a $\Sigma^{\mathcal{B}}$-interpretation satisfying (B). The axiom implies that $[\![\mathsf{cast}_o(\text{TRUE})]\!]^J$ and $[\![\mathsf{cast}_o(\text{FALSE})]\!]^J$ are two distinct values in D. Thus D is an adequate domain for TLA$^+$. We let $\top^D \triangleq [\![\mathsf{cast}_o(\text{TRUE})]\!]^J$ and $\bot^D \triangleq [\![\mathsf{cast}_o(\text{FALSE})]\!]^J$. We then define the Σ-interpretation I as the restriction of J to Σ. It is then easily verified that $I^{\mathcal{B}} = J$. Therefore $I \mapsto I^{\mathcal{B}}$ is surjective. □

Corollary 4.3.9 (Completeness)**.** *Let e be a closed expression of $\mathcal{L}$. If e is valid then $\mathcal{B}^o(e)$ is valid for the class of interpretations that satisfy* (B)*.*

Proof. Suppose e is valid. Let J be a $\Sigma^{\mathcal{B}}$-interpretation. By Lemma 4.3.8, there is an I such that $J = I^{\mathcal{B}}$. By assumption, $I \models e$, *i.e.* $[\![e]\!]^I = \top^D$. By Theorem 4.3.6, $[\![\mathcal{B}^o(e)]\!]^J = \top$. Therefore, $\mathcal{B}^o(e)$ is valid. □

4.3.3 Predicate Types

$\mathcal{B}^o$ can be applied to TLA$^+$'s theory T, resulting in a new theory T' in the target logic $\mathcal{L}^\iota$. However, because the mapping $\mathcal{B}^o$ described so far preserves all operator types with ι, the resulting theory would not feature any predicate or other operator involving Booleans. For instance, applying $\mathcal{B}^o$ to the schema of set comprehension results in

$\mathsf{in} : \iota \times \iota \to \iota$

$\mathsf{setst} : \iota \times (\iota \to \iota) \to \iota$

$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setst}(a, P)) = \mathsf{cast}_o(\text{TRUE}) \Leftrightarrow \mathsf{in}(x, a) = \mathsf{cast}_o(\text{TRUE}) \wedge P(x) = \mathsf{cast}_o(\text{TRUE})$
for all $P : \iota \to \iota$

Our implementation actually assigns types with o to some TLA$^+$ primitives and defines axioms in T' closer to what one would expect:

$\mathsf{in} : \iota \times \iota \to o$

$\mathsf{setst} : \iota \times (\iota \to o) \to \iota$

$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setst}(a, P)) \Leftrightarrow \mathsf{in}(x, a) \wedge P(x) \qquad$ for all $P : \iota \to o$

This optimization is justified by the semantics of in and setst. We present the arguments for assigning those types to in and setst below. The arguments for in also apply to the subset relation and the comparison operators, which are encoded as predicates. The arguments for set comprehension apply to the choose operator, which expects a predicate argument in the target logic.

The signature $\Sigma^{\mathcal{B}}$ is modified so that $\Sigma^{\mathcal{B}}(\mathsf{in}) = \iota \times \iota \to o$, and the definitions of $\mathcal{B}^o$ and $\mathcal{B}^\iota$ are adapted:

$$\mathcal{B}^\iota(\mathsf{in}(e_1, e_2)) \triangleq \mathsf{cast}_o(\mathsf{in}(\mathcal{B}^\iota(e_1), \mathcal{B}^\iota(e_2)))$$
$$\mathcal{B}^o(\mathsf{in}(e_1, e_2)) \triangleq \mathsf{in}(\mathcal{B}^\iota(e_1), \mathcal{B}^\iota(e_2))$$

Next, we define an interpretation for in by

$$\mathsf{in}^{I^{\mathcal{B}}}(v_1, v_2) \text{ iff } \mathsf{in}^{I}(v_1, v_2) = \top^D$$

The proof of Theorem 4.3.6 is easily adapted. The key argument is that set membership is specified to be a relation in TLA$^+$, meaning in always returns a Boolean value. From this fact and the definition of $I^{\mathcal{B}}$, it follows that

$$[\![\mathsf{in}(e_1, e_2)]\!]^{I^{\mathcal{B}}} = \top \text{ iff } [\![\mathsf{in}(e_1, e_2)]\!]^{I} = \top^D$$
$$[\![\mathsf{in}(e_1, e_2)]\!]^{I^{\mathcal{B}}} = \bot \text{ iff } [\![\mathsf{in}(e_1, e_2)]\!]^{I} = \bot^D$$

The proof is trivial from here. For completeness, remark that the mapping $I \mapsto I^{\mathcal{B}}$ is still surjective, because the original interpretation of $\mathsf{in} : \iota \times \iota \to \iota$ can be defined from the interpretation of $\mathsf{in} : \iota \times \iota \to o$ (simply convert $\top$ to $\top^D$ and $\bot$ to $\bot^D$).

We now turn to set comprehension. The argument is trickier, as we need to prove that these sets are preserved if their first-order parameters are projected as predicates. To provide the intuition, remark that the equation

$$\{x \in e_1 : e_2\} = \{x \in e_1 : e_2 = \text{TRUE}\}$$

is provable in TLA$^+$ using set extensionality, because the predicates that characterize both sets are equivalent.

Let $\Sigma^{\mathcal{B}}(\mathsf{setst}) \triangleq \iota \times (\iota \to o) \to \iota$. Only one rule for $\mathcal{B}^\iota$ is adapted:

$$\mathcal{B}^\iota(\mathsf{setst}(e_1, \lambda x : e_2)) \triangleq \mathsf{setst}(\mathcal{B}^\iota(e_1), \lambda x^\iota : \mathcal{B}^o(e_2))$$

Note that the second argument to setst will always be a lambda-term, because users only have access to the notation $\{x \in e_1 : e_2\}$. Let the new interpretation for setst be defined by

$$\mathsf{setst}^{I^{\mathcal{B}}}(v, p) \triangleq \mathsf{setst}^{I}(v, p') \qquad \text{where } p'(u) \triangleq \top^D \text{ if } p(u) = \top, \text{ otherwise } \bot^D \qquad (*)$$

We now indicate how the proof of Theorem 4.3.6 is adapted. It suffices to prove that

$$[\![\mathcal{B}^\iota(\mathsf{setst}(e_1, \lambda x : e_2))]\!]^{I^{\mathcal{B}}}_\theta = [\![\mathsf{setst}(e_1, \lambda x : e_2)]\!]^{I}_\theta$$

We define the following values and functions on D:

$$v \triangleq [\![\mathcal{B}^\iota(e_1)]\!]^{I^{\mathcal{B}}}_\theta \qquad p(u) \triangleq [\![\mathcal{B}^o(e_2)]\!]^{I^{\mathcal{B}}}_{\theta^x_u}$$
$$w \triangleq [\![e_1]\!]^{I}_\theta \qquad q(u) \triangleq [\![e_2]\!]^{I}_{\theta^x_u}$$

The desired equation is verified by

$$\mathsf{setst}^{I^{\mathcal{B}}}(v, p) = \mathsf{setst}^{I}(v, p') = \mathsf{setst}^{I}(w, q)$$

where the first equality is the definition of $\mathsf{setst}^{I^{\mathcal{B}}}$ and the second equality is proved by set extensionality. Indeed, the induction hypothesis gives $v = w$ and $p(u) = \top$ iff $q(u) = \top^D$ for all u in D. But $p(u) = \top$ iff $p'(u) = \top^D$, so the elements of both sets are the same.

Let us now prove that the mapping $I \mapsto I^{\mathcal{B}}$ remains surjective when setst expects a predicate argument. Given a $\Sigma^{\mathcal{B}}$-interpretation J, we define a Σ-interpretation I such that $K^{I^{\mathcal{B}}} = K^J$ for all $K \neq \mathsf{setst}$, and

$$\mathsf{setst}^{I}(v, f) \triangleq \mathsf{setst}^{J}(v, f') \qquad \text{where } f'(u) \triangleq \top \text{ if } f(u) = \top^D, \text{ otherwise } \bot$$

We prove that $\mathsf{setst}^{I^{\mathcal{B}}} = \mathsf{setst}^J$. Let v in D and p a predicate on D. Let $f = p'$ defined as in $(*)$. Clearly, $f' = p$, thus

$$\mathsf{setst}^{I^{\mathcal{B}}}(v, p) = \mathsf{setst}^I(v, f) = \mathsf{setst}^J(v, p)$$

For choose expressions, the justifications are analogous, but instead of set extensionality, the axiom of choice determinacy is used:

$$(\forall x : P(x) \Leftrightarrow Q(x)) \Rightarrow \mathsf{choose}(P) = \mathsf{choose}(Q)$$

Here is a summary of the TLA$^+$ primitives that are assigned a type with o:

Operator	Type
choose	$(\iota \to o) \to \iota$
in	$\iota \times \iota \to o$
subseteq	$\iota \times \iota \to o$
setst	$\iota \times (\iota \to o) \to \iota$
isafcn	$\iota \to o$
lteq	$\iota \times \iota \to o$
IsFiniteSet	$\iota \to o$

To finish, let us go back to our running example from this chapter.

Example 4.3.10. The result of applying our transformation on the proof obligation is:

$$\begin{array}{rl} & \text{NEW } A : \iota,\ \text{NEW } B : \iota,\ \text{NEW } f : \iota, \\ & \mathsf{in}(f, \mathsf{arrow}(\mathsf{domain}(f), B)), \\ & \mathsf{subseteq}(\mathsf{domain}(f), A)) \\ \vdash & \mathsf{in}(f, \mathsf{union}(\mathsf{setof}_1(\mathsf{subset}(A), \lambda X^\iota : \mathsf{arrow}(X, B)))) \end{array}$$

There are no differences with the original form of the obligation, except that declared constants have sort annotations. The logical context has changed; the primitive symbols in $\Sigma^{\mathcal{B}}$ are assigned types, some of them featuring the sort o. One could verify that the proof obligation is well-sorted in the target logic; in particular, the two hypotheses and the goal are formulas.

4.4 Rewriting

Rewriting is a natural way to eliminate primitive constructs with no counterpart in the target logic. Sets are essentially defined by their characteristic predicates, so we may attempt to replace expressions of the form $x \in S$ by simpler expressions. For example:

$$x \in (S \cup T) \quad \longrightarrow \quad x \in S \vee x \in T$$

This idea is naturally extended to TLA$^+$ functions, since a function is characterized by its domain and its body:

$$[z \in D \mapsto F(z)][x] \quad \longrightarrow \quad F(x) \quad \text{when } x \in D$$

This approach leads to some difficulties. As illustrated by the line above, some rewritings are valid only under certain semantics conditions. Since D can be any set, the problem of checking the validity of that rewriting is undecidable in general.

Another difficulty is to guarantee that rewriting always terminates. Assuming the natural rule $x \in \{y \in S : P(y)\} \longrightarrow x \in S \wedge P(x)$, if we define $R \triangleq \{x \in S : \neg(x \in x)\}$ then we have the non-terminating sequence

$$R \in R \quad \longrightarrow \quad R \in S \wedge \neg(R \in R) \quad \longrightarrow \quad R \in S \wedge \neg(R \in S \wedge \neg(R \in R)) \quad \longrightarrow \quad \cdots$$

Expressions like R are rarely found in the practice of TLA$^+$. A larger obstacle to a suitable implementation of rewriting is the opposite problem: rewriting often terminates too soon, leaving primitive constructs in the obligation. The instance of $\cup$ cannot be simplified by direct application of a rewriting rule in

$$z = (S \cup T) \Rightarrow x \in z$$

The original SMT encoding, currently implemented in TLAPS, features a rewriting system that is neither terminating nor confluent. We present here an alternative rewriting system that satisfies both of these properties. Rewriting alone was already recognized as insufficient for eliminating enough TLA$^+$ primitives, which is why, in the original implementation, rewriting is supported by a set of auxiliary techniques such as elimination of equalities, or abstraction of complex subexpressions. Our objective here is not to revisit that fragment of the original implementation; we will focus on the rewriting system, even if the resulting implementation is weaker.

4.4.1 Definition and Correctness

Let $\Box$ be some variable symbol called a *hole*. An expression with a hole is any expression with exactly one occurrence of $\Box$. We denote E such expressions. If e is any other expression, the expression $E[e]$ is defined as $E\,\{\Box \mapsto e\}$. We leave implicit the sort of $\Box$, which may be ι or o depending on the context. We always assume the sorts of e and $\Box$ are identical when writing $E[e]$, so the substitution is well-defined, and the sorts of E and $E[e]$ coincide.

Definition 4.4.1 (Rewriting)**.** A rewriting rule is a pair of open expressions (l, r) such that every free variable of r is in l. We introduce a rewriting rule with the notation

$$l \triangleright r$$

Let Γ_l be the typing context such that $\Gamma_l(x) = \iota$ for all x free in l. Both l and r have a sort in the context Γ_l; we impose that their sorts coincide.

Let R be a set of rewriting rules, also called rewriting system. The relation $\longrightarrow_R$ is defined on expressions by

$$E[l\sigma] \longrightarrow_R E[r\sigma]$$

where $l \triangleright r$ is a rule in R, E is a term with a hole, and σ is a substitution. We simply note $\longrightarrow$ if the set R is obvious in context.

Clearly, if $e_1 \longrightarrow e_2$ and $\Gamma \vdash e_1 : s$ then $\Gamma \vdash e_2 : s$. The fact that $\Box$ only occurs once in expressions implies that subexpressions can only be rewritten one at a time by $\longrightarrow$. This is relevant to the question of confluence, which is a property we will discuss in the next section.

Example 4.4.2. Consider the rule

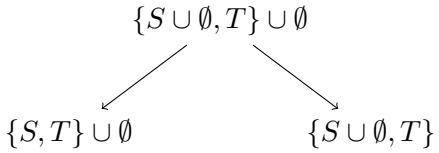
$$a \cup \emptyset \triangleright a$$

and the expression $\{S \cup \emptyset, T\} \cup \emptyset$, where S and T are two constants.

Here are two different ways to rewrite the expression, with the respective definitions for E and σ:

$$\begin{array}{lll} \{S \cup \emptyset, T\} \cup \emptyset \longrightarrow \{S, T\} \cup \emptyset & \sigma(a) \triangleq S & E \triangleq \{\Box, T\} \cup \emptyset \\ \{S \cup \emptyset, T\} \cup \emptyset \longrightarrow \{S \cup \emptyset, T\} & \sigma(a) \triangleq \{S \cup \emptyset, T\} & E \triangleq \Box \end{array}$$

We usually represent this situation with a diagram:



Proposition 4.4.3. *Let E be an expression with a hole, e an expression, I an interpretation and θ a valuation. Then $[\![E[e]]\!]^I_\theta = [\![E]\!]^I_{\theta^\Box_v}$ where $v = [\![e]\!]^I_\theta$.*

Proof. This is proved by

$$\begin{array}{lll} [\![E[e]]\!]^I_\theta = [\![E\,\{\Box \mapsto e\}]\!]^I_\theta & & \text{by definition of } E[e] \\ = [\![E]\!]^I_{\{\Box \mapsto e\}\theta} & & \text{by Proposition 4.2.9} \\ = [\![E]\!]^I_{\theta^\Box_v} & & \end{array}$$

□

Theorem 4.4.4 (Correctness)**.** *A rewriting system is defined as correct iff for all rules $l \triangleright r$, interpretations I and valuations θ, $[\![l]\!]^I_\theta = [\![r]\!]^I_\theta$.*

If R is correct, then $e_1 \longrightarrow_R e_2$ implies $[\![e_1]\!]^I_\theta = [\![e_2]\!]^I_\theta$ for all I and θ.

Proof. Let $v = [\![l\sigma]\!]^I_\theta$ and $w = [\![r\sigma]\!]^I_\theta$. Then by Proposition 4.2.9 and the correctness assumption,

$$v = [\![l]\!]^I_{\sigma\theta} = [\![r]\!]^I_{\sigma\theta} = w$$

By Proposition 4.4.3, it follows that

$$[\![E[l\sigma]]\!]^I_\theta = [\![E]\!]^I_{\theta^\Box_v} = [\![E]\!]^I_{\theta^\Box_w} = [\![E[r\sigma]]\!]^I_\theta$$

□

An immediate corollary of Theorem 4.4.4 is that $e_1 \longrightarrow e_2$ implies that $I \models e_1$ iff $I \models e_2$ for all I when e_1 and e_2 are formulas. Thus we only have to guarantee l and r always have the same value (if they are formulas, the same truth value) to guarantee both the soundness and the completeness of rewriting.

Our rewriting system for TLA$^+$ is presented in Figure 4.4. The TLA$^+$ notations are used for better readability. Note that the characters x, a, b, f, t and r all denote variables. The characters T and ϕ are not variables but placeholders for terms or formulas, respectively. The

rules that feature a T or a ϕ are schemas of rewriting rules; these rules must be presented as schemas because they involve a higher-order parameter. There are also schemas of rules for the operators that accept an arbitrary number of arguments. For instance, the actual schema for set refinement is

$$x \in \{T : y_1 \in a_1, \ldots, y_p \in a_p\} \triangleright \exists y_1, \ldots, y_p : y_1 \in a_1 \wedge \cdots \wedge y_p \in a_p \wedge x = T$$

Only the representative case for $p = 1$ is shown for the sake of readability. All the rules that involve tuples or records are understood to be generalized to any p as well.

It is easy to check that all rules preserve types from l to r, and values as well. Thus rewriting is correct by Theorem 4.4.4. An alternative way to state correctness is to assert that

$$\forall x_1, \ldots, x_n : l = r$$

is a theorem of TLA$^+$ for all $l \triangleright r$, where $x_1, \ldots, x_n$ are the free variables of l. Many of these statements are among the axioms we pass to the provers, which we discuss in the next section.

Any implementation that is based on this rewriting system is therefore correct, but so far we have described rewriting as a non-deterministic procedure represented by the relation $\longrightarrow$. Any rewriting is of the form $E[l\sigma] \longrightarrow E[r\sigma]$, but there can be several ways to put a given expression under the form on the left. The property of *confluence* ensures that diverging rewriting sequences can always be joined. Combined with *termination*, this will guarantee that any implementation of $\longrightarrow$ converges to a unique solution.

4.4.2 Termination and Confluence

The previous version of rewriting did not terminate because of this rule:

$$x \in \{y \in a : e\} \triangleright x \in a \wedge e\sigma \qquad \text{where } \sigma(y) \triangleq x$$

The problem is that the substitution $e\sigma$ may introduce a match for some rewriting rule that was not present before. We illustrated this with the formula $R \in R$ where $R \triangleq \{x \in S : \neg(R \in R)\}$. Termination is recovered by using this rule instead:

$$x \in \{y \in a : e\} \triangleright x \in a \wedge \exists y : y = x \wedge e$$

Applying this to the example:

$$R \in R \longrightarrow R \in V \wedge \exists x : x = R \wedge \neg(x \in x)$$

The subformula $x \in x$ cannot be rewritten because x is a variable.

Theorem 4.4.5 (Termination)**.** *The rewriting system of Figure 4.4 terminates, i.e. there is no infinite sequence $e_1, e_2, \ldots$ such that $e_1 \longrightarrow e_{i+1}$.*

Proof. The idea of the proof is to exhibit a measure on expressions that is strictly decreasing with $\longrightarrow$, with respect to some well-founded order.

With each expression e, we will associate a triple of natural numbers $w(e)$. We consider the lexicographic order on $\mathbb{N}^3$, noted $\prec$, which is defined by

$$\begin{aligned}(m, n, p) \prec (m', n', p') \text{ iff } & \text{either } m < m' \\ & \text{or } m = m' \text{ and } n < n' \\ & \text{or } m = m' \text{ and } n = n' \text{ and } p < p'\end{aligned}$$

$$
\begin{array}{rcl}
a \subseteq b & \triangleright & \forall x : x \in a \Rightarrow x \in b \\
x \in \{a_1, \ldots, a_p\} & \triangleright & x = a_1 \vee \cdots \vee x = a_p \\
x \in \emptyset & \triangleright & \text{FALSE} \\
x \in \text{UNION } a & \triangleright & \exists y : y \in a \wedge x \in y \\
x \in \text{SUBSET } a & \triangleright & \forall y : y \in x \Rightarrow y \in a \\
x \in \{y \in a : \phi\} & \triangleright & x \in a \wedge \exists y : y = x \wedge \phi \\
x \in \{T : y \in a\} & \triangleright & \exists y : x = T \wedge y \in a
\end{array}
\qquad
\begin{array}{rcl}
x \in a \cup b & \triangleright & x \in a \vee x \in b \\
x \in a \cap b & \triangleright & x \in a \wedge x \in b \\
x \in a \setminus b & \triangleright & x \in a \wedge \neg(x \in b) \\
x \in \text{BOOLEAN} & \triangleright & \vee\, x = \mathsf{cast}_o(\text{TRUE}) \\
 & & \vee\, x = \mathsf{cast}_o(\text{FALSE}) \\
\text{“}\texttt{foo}\text{”} \in \text{STRING} & \triangleright & \text{TRUE}
\end{array}
$$

$$
\begin{array}{rcl}
f \in [a \to b] & \triangleright & \wedge\, \mathsf{isafcn}(f) \\
 & & \wedge\, \text{DOMAIN } f = a \\
 & & \wedge\, \forall x : x \in a \Rightarrow f[x] \in b
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{isafcn}([x \in a \mapsto T]) & \triangleright & \text{TRUE} \\
\text{DOMAIN } [x \in a \mapsto T] & \triangleright & a \\
\mathsf{isafcn}([f \text{ EXCEPT } ![x] = y]) & \triangleright & \text{TRUE} \\
\text{DOMAIN } [f \text{ EXCEPT } ![x] = y] & \triangleright & \text{DOMAIN } f
\end{array}
$$

$$
\begin{array}{rcl}
t \in a \times b & \triangleright & \wedge\, t[1] \in a \\
 & & \wedge\, t[2] \in b \\
 & & \wedge\, t = \langle t[1], t[2] \rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{isafcn}(\langle x, y \rangle) & \triangleright & \text{TRUE} \\
\text{DOMAIN } \langle x, y \rangle & \triangleright & \{1, 2\} \\
\langle x, y \rangle[1] & \triangleright & x \\
\langle x, y \rangle[2] & \triangleright & y
\end{array}
$$

$$
\begin{array}{rcl}
r \in [foo : a, bar : b] & \triangleright & \wedge\, r[\text{“}\texttt{foo}\text{”}] \in a \\
 & & \wedge\, r[\text{“}\texttt{bar}\text{”}] \in b \\
 & & \wedge\, r = [foo \mapsto r[\text{“}\texttt{foo}\text{”}], \\
 & & \qquad bar \mapsto r[\text{“}\texttt{bar}\text{”}]]
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{isafcn}([foo \mapsto x, bar \mapsto y]) & \triangleright & \text{TRUE} \\
\text{DOMAIN } [foo \mapsto x, bar \mapsto y] & \triangleright & \{\text{“}\texttt{foo}\text{”}, \text{“}\texttt{bar}\text{”}\} \\
[foo \mapsto x, bar \mapsto y][\text{“}\texttt{foo}\text{”}] & \triangleright & x \\
[foo \mapsto x, bar \mapsto y][\text{“}\texttt{bar}\text{”}] & \triangleright & y
\end{array}
$$

Figure 4.4: Rewriting Rules

It remains to define $w(e) = (n_1, n_2, n_3)$. The idea is to count occurrences of certain primitive operators in e. The measure will decrease because all rewriting rules simplify e by removing at least one primitive operator. But some rules also introduce new operators which count for $w(e)$, for example:

$$\text{DOMAIN } \langle x, y \rangle \triangleright \{1, 2\}$$

The key is to define $w(e)$ in such a way that the operator $\langle \_, \_ \rangle$ weighs more more than $\{\_, \_\}$. This is why we have chosen a lexicographic order rather than the simpler order $<$ on $\mathbb{N}$.

n_1 is defined as the number of occurrences of the operators $\times$ and $[foo : \_, bar : \_]$. n_2 is the number of occurrences of $\langle \_, \_ \rangle$ and $[foo \mapsto \_, bar \mapsto \_]$. n_3 is the number of occurrences of $\subseteq$, $\{\_, \ldots, \_\}$, $\emptyset$, UNION, SUBSET, $\{\_ \in \_ : \_\}$, $\{\_ : \_ \in \_\}$, $\cup$, $\cap$, $\setminus$, BOOLEAN, STRING, $[\_ \to \_]$, $[\_ \in \_ \mapsto \_]$, [_ EXCEPT ![_] = _].

We can verify that $w(l) \succ w(r)$ for all rules $l \triangleright r$. For instance:

$$w(\text{DOMAIN } \langle x, y \rangle) = (0, 1, 0) \succ (0, 0, 1) = w(\{1, 2\})$$

Then, clearly, $w(E[l\sigma]) \succ w(E[r\sigma])$ for all $l \triangleright r$. Therefore $e \longrightarrow e'$ implies $w(e) \succ w(e')$. But $\mathbb{N}^3$ is known to be well-ordered by $\prec$, so $\longrightarrow$ terminates. □

We now turn to the confluence property, starting with some basic definitions.

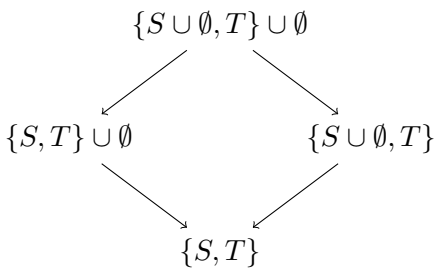
Definition 4.4.6. We denote by $\longrightarrow^\star$ the reflexive transitive closure of $\longrightarrow$. Two expressions e_1 and e_2 are joinable iff there exists e' such that $e_1 \longrightarrow^\star e'$ and $e_2 \longrightarrow^\star e'$. A rewriting system is confluent if for every e, e_1 and e_2 such that $e \longrightarrow^\star e_1$ and $e \longrightarrow^\star e_2$, e_1 and e_2 are joinable. A rewriting system is locally confluent if for every e, e_1 and e_2 such that $e \longrightarrow e_1$ and $e \longrightarrow e_2$, e_1 and e_2 are joinable.

Any confluent rewriting system is locally confluent. The converse is not true in general, but a classic result by Newman states that local confluence suffices for terminating systems.

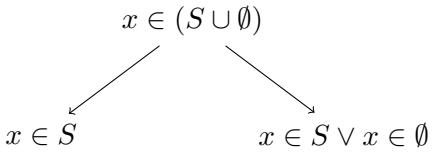
Lemma 4.4.7 (Newman)**.** *Any terminating, locally confluent rewriting system is confluent.*

Proof. A proof of this result is given by Huet [42]. The proof consists of a double induction on the lengths of the two sequences $e \longrightarrow^\star e_1$ and $e \longrightarrow^\star e_2$. □

It suffices to prove that our rewriting system is locally confluent. Let us look again at the rule $a \cup \emptyset \triangleright a$ (which is not part of our system). We saw that the expression $\{S \cup \emptyset, T\} \cup \emptyset$ could be reduced in two different ways. The two resulting formulas are joinable:



However, the introduction of this rule to our system would break (local) confluence, as this case illustrates:



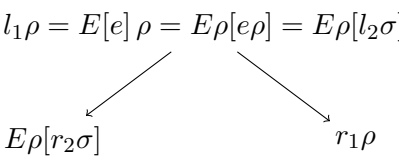
Since $x \in \emptyset$ can be rewritten as FALSE, we could join the pair by adding the schema of rules $\phi \vee \text{FALSE} \triangleright \phi$. We purposefully avoided rules that act on propositional connectives to make confluence easy to prove. In the original version of the encoding, confluence was only proved for rewriting systems that exclude applications of set extensionality. $a \cup \emptyset \triangleright a$ is an instance of set extensionality.

It can be difficult to prove local confluence directly. We may reduce the problem to a small number of easy verifications with the notion of critical pair.

Definition 4.4.8. A common instance of two expressions e_1 and e_2 is an expression e' such that $e' = e_1\rho = e_2\sigma$ for some substitutions ρ and σ. The expression e' is a most general common instance if for all common instance e'' of the same expressions, there is a substitution σ' such that $e'' = e'\sigma'$.

Definition 4.4.9. Let $l_1 \triangleright r_1$ and $l_2 \triangleright r_2$ be two rewriting rules. Suppose there exists an open expression e such that $l_1 = E[e]$ and a most general common instance $e\rho = l_2\sigma$. Then the pair of expressions $\langle E\rho[r_2\sigma]\,, r_1\rho\rangle$ is called a critical pair. A critical pair is joinable if the two expressions are joinable.

The definition of critical pairs is based on the following template for diverging rewritings:



Example 4.4.10. A critical pair can be derived from the rules

$$\begin{aligned} l_1 \triangleright r_1 &: \quad x \in (a \cup b) \triangleright x \in a \vee x \in b \\ l_2 \triangleright r_2 &: \quad a \cup \emptyset \triangleright a \end{aligned}$$

The relevant subterm of l_1 is $(a \cup b)$. It admits a most general common instance with l_2 through the substitution $\rho(b) \triangleq \emptyset$ (all other variables are ignored). This leads to the critical pair $\langle x \in a, x \in a \vee x \in \emptyset\rangle$.

Critical pairs are useful because we can enumerate them. It suffices to go through all rules $l \triangleright r$ and check if some subterm of l can be matched with the left pattern l' of another (or the same) rule. When that is the case it suffices to consider the most general match possible.

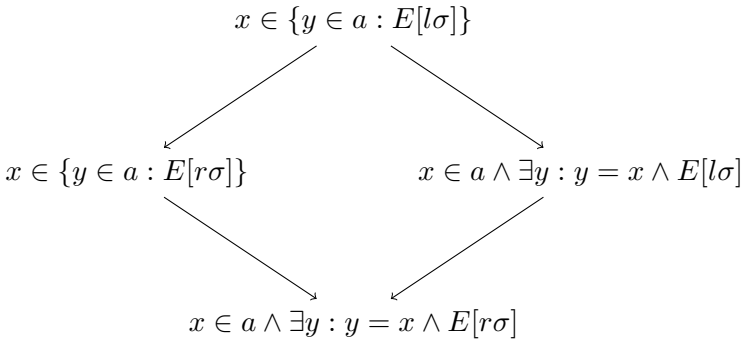
Lemma 4.4.11. *A rewriting system is locally confluent if all of its critical pairs are joinable.*

Proof. This is proven by Huet [42]. □

Theorem 4.4.12 (Confluence)**.** *The rewriting system of Figure 4.4 is confluent.*

Proof. Since it is terminating (Theorem 4.4.5), it suffices to check that all critical pairs are joinable (Lemmas 4.4.7 and 4.4.11).

There are actually almost no critical pair in our rewriting system. All critical pairs result from the rules that are parameterized by a term T or a formula ϕ. For instance, there is a class of critical pairs that consists of pairs of the form



For some rule $l \triangleright r$. But the pair is easily joinable, as the diagram also shows.

Note that $y \in a$ is not a subexpression of $\{y \in a : \phi\}$. Although we are using the readable TLA$^+$ notations here, such an expression must be read as $\mathsf{setst}(a, \lambda y : \phi)$. □

To summarize the results of this section, we state the following:

Theorem 4.4.13. *For all e, there is a unique e' such that $e \longrightarrow^\star e'$ and e' cannot be rewritten.*

Proof. Termination ensures the existence of e'. Suppose e'' also satisfies the conditions of the theorem. Then e' and e'' must be joinable, by confluence. But neither can be rewritten, therefore $e' = e''$. □

Example 4.4.14. Let us see how rewriting affects our running example. The last section ended with the obligation

$$\begin{array}{ll} \text{ASSUME} & \text{NEW } A : \iota,\ \text{NEW } B : \iota,\ \text{NEW } f : \iota, \\ & \mathsf{in}(f, \mathsf{arrow}(\mathsf{domain}(f), B)), \\ & \mathsf{subseteq}(\mathsf{domain}(f), A)) \\ \text{PROVE} & \mathsf{in}(f, \mathsf{union}(\mathsf{setof}_1(\mathsf{subset}(A), \lambda X^\iota : \mathsf{arrow}(X, B)))) \end{array}$$

We apply as many rewriting rules as possible (we let the reader translate the expressions of Figure 4.4 into our alternative notations). With some minor modifications to improve the presentation, we obtain the following result:

$$\begin{array}{ll} \text{ASSUME} & \text{NEW } A : \iota,\ \text{NEW } B : \iota,\ \text{NEW } f : \iota, \\ & \mathsf{isafcn}(f), \\ & \mathsf{domain}(f) = \mathsf{domain}(f), \\ & \forall x : \mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), B), \\ & \forall x : \mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{in}(x, A) \\ \text{PROVE} & \exists y : \ \wedge \exists X : \ \wedge \forall x : \mathsf{in}(x, X) \Rightarrow \mathsf{in}(x, A) \\ & \qquad\qquad\qquad\quad \wedge y = \mathsf{arrow}(X, B) \\ & \qquad\quad \wedge \mathsf{in}(f, y) \end{array}$$

Note that it is possible to rearrange quantifiers and substitute $\mathsf{arrow}(X, B)$ for y in the goal, and then eliminate the redundant quantifier $\exists y$. That would result in the goal

$$\begin{aligned}\exists X : \wedge\ & \forall x : \mathsf{in}(x, X) \Rightarrow \mathsf{in}(x, A) \\ \wedge\ & \mathsf{in}(f, \mathsf{arrow}(X, B))\end{aligned}$$

This can be rewritten further, eliminating the last occurrence of the operator arrow. However, implementing this kind of simplifications might result in a non-terminating encoding.

4.5 Axiomatization

All TLA$^+$ primitive operators are specified by axioms. During the axiomatization phase of the encoding, explicit declarations and axioms are inserted in the proof obligation's context for all relevant operators.

4.5.1 Definition and Correctness

The input of this step is a sequent $\Gamma \vdash \phi$, where Γ contains operator declarations with types, hypotheses which are just Boolean expressions, and the goal ϕ is a Boolean expression. We consider that the primitive operators of TLA$^+$ are bound to a standard signature Σ, and the class of possible interpretations is defined by a theory T. Axiomatization can be formally understood as the selection of operators from Σ and axioms from T to be inserted at the top of the sequent's context, resulting in a new sequent $\Delta, \Gamma \vdash \phi$. The signature Σ and the theory T are omitted; when the obligation is discharged to an automated prover, that prover will only have access to the information about TLA$^+$'s theory that is encoded in Δ.

The implementation of this step is straightforward. Let $\Gamma \vdash \phi$ be the current sequent and $K \in \mathrm{Dom}(\Sigma)$ some operator that has not been explicitly declared yet. We select a finite set of axioms T' for K and produce the new sequent

$$\textsc{new}\ K : \Sigma(K), T', \Gamma \vdash \phi$$

While the original sequent $\Gamma \vdash \phi$ is preserved, all occurrences of K in it refer to the new declaration.[1] This process must be repeated as long as the sequent contains operator symbols bound in Σ. Note that the addition of T' may introduce operators that did not occur in the sequent before; we must be mindful of dependencies between primitive operators, or axiomatization may not terminate.

Theorem 4.5.1. *Axiomatization is sound and terminating.*

We can only lay out the arguments in an abstract manner. To actually prove it, one would need to verify all axioms individually, either by hand or by automated means.

Proof. The full list of axioms for the TPTP encoding can be found in appendix B. It is clear that there is no cycle between operator dependencies, so the process terminates.

[1] Some transformation of $\Gamma \vdash \phi$ is actually required in practice. TLAPS implements bound variables as De Bruijn indices—natural numbers identifying the unique binder a variable is linked to. Any modification of the proof obligation's context must be accompanied by an update of all variable indices. This technical point about the implementation is hidden by our formalism.

For soundness, consider a Σ-interpretation I such that $I \models T$. Let $\Delta \triangleq \text{NEW } k : \Sigma(k), T'$ as described above. Appending Δ to the current context is sound if $I \models \Delta$, which is obvious as long as $T \models T'$. In general, the axioms of T' are elementary consequences of T (see the discussion below for examples). □

We do not claim that axiomatization is complete. Typical cases of incompleteness arise when the operator(s) required for expressing an existential witness do not occur in the obligation. Consider Cantor's theorem expressed as the proof obligation

$$\begin{array}{ll}\text{ASSUME} & \text{NEW } S,\ \text{NEW } f \in [S \to \text{SUBSET } S] \\ \text{PROVE} & \exists P \in \text{SUBSET } S : \forall x \in S : f[x] \neq P\end{array}$$

The expected witness for P is the set $\{x \in S : x \notin f[x]\}$, but the original obligation does not include that instance of set comprehension. To prove Cantor's theorem, users must insert an intermediary definition or proof step to introduce this set into the proof's context.

4.5.2 Second-order Axiomatization of TLA$^+$

We refer the reader to appendix B for the full list of axioms. All axioms are elementary reformulations or consequences of the axioms of TLA$^+$. Since our target language is HOL, we accept higher-order quantifiers for axioms; typically, the axiom schemas of TLA$^+$ are encoded as single axioms $\forall F^\tau : \phi$ where $\text{ord}(\tau) = 1$.

In the original theory, primitive constructs are defined from "more primitive" ones when possible, for example $a \cup b \triangleq \text{UNION } \{a, b\}$. This makes TLA$^+$'s relative consistency with ZFC more evident. But using those axioms would result in very long chains of dependencies between operators. Moreover, they reflect poorly how a user would reason about those operators. In general, we replaced a complex definition from primitive operators by a more direct characterization, with less dependencies:

$$\forall a^\iota, b^\iota, x^\iota : \mathsf{in}(x, \mathsf{cup}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \vee \mathsf{in}(x, b) \qquad \text{(CupDef)}$$

Which axioms are assigned to a given operator is always evident. In each case, the operator depends on all the other operators that appear in its associated axioms. For instance cup is specified by (CupDef) and depends on in.

The axioms of set theory are standard; we omit the axiom of infinity and the axiom of foundation, since they rarely play any role in TLA$^+$ proofs (users can explicitly invoke them if needed). Here are the basic axioms for TLA$^+$ functions:

$$\begin{array}{ll}
\forall f^\iota, g^\iota : \wedge\ \mathsf{isafcn}(f) \wedge \mathsf{isafcn}(g) & \text{(FcnExt)} \\
\qquad \wedge\ \mathsf{domain}(f) =_\iota \mathsf{domain}(g) & \\
\qquad \wedge\ (\forall x^\iota : \mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{fcnapp}(f, x) =_\iota \mathsf{fcnapp}(g, x)) & \\
\qquad \Rightarrow f =_\iota g & \\
\forall F^{\iota\to\iota}, a^\iota : \mathsf{isafcn}(\mathsf{fcn}(a, F)) & \text{(FcnIsafcn)} \\
\forall F^{\iota\to\iota}, a^\iota : \mathsf{domain}(\mathsf{fcn}(a, F)) =_\iota a & \text{(FcnDom)} \\
\forall F^{\iota\to\iota}, a^\iota, x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{fcnapp}(\mathsf{fcn}(a, F), x) =_\iota F(x) & \text{(FcnApp)} \\
\forall a^\iota, b^\iota, f^\iota : \mathsf{in}(f, \mathsf{arrow}(a, b)) \Leftrightarrow & \text{(ArrowDef)} \\
\qquad \wedge\ \mathsf{isafcn}(f) & \\
\qquad \wedge\ \mathsf{domain}(f) =_\iota a & \\
\qquad \wedge\ (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), b)) &
\end{array}$$

The operator isafcn is specified by (FcnExt). The operator fcn is specified by (FcnIsafcn), (FcnDom), (FcnApp). The operator arrow is specified by (ArrowDef). The operators domain and fcnapp are not assigned any axioms, but all other functional operators depend on them.

Tuples and records inherit from the theory of functions, but specify additional operators that essentially replace fcn and arrow. For instance, tuples are usually constructed from tup, and product sets from prod.

$$\begin{array}{ll}
\text{For all } n \geq 0, & \\
\forall x_1^\iota, \ldots, x_n^\iota : \mathsf{isafcn}(\mathsf{tup}_n(x_1, \ldots, x_n)) & \text{(TupIsafcn)} \\
\forall x_1^\iota, \ldots, x_n^\iota : \mathsf{domain}(\mathsf{tup}_n(x_1, \ldots, x_n)) = \mathsf{enum}_n(1, \ldots, n) & \text{(TupDomain)} \\
\forall x_1^\iota, \ldots, x_n^\iota : \wedge\ \mathsf{fcnapp}(\mathsf{tup}_n(x_1, \ldots, x_n), 1) = x_1 & \text{(TupApp)} \\
\qquad \wedge \ldots & \\
\qquad \wedge\ \mathsf{fcnapp}(\mathsf{tup}_n(x_1, \ldots, x_n), n) = x_n & \\
\text{For all } n \geq 2, & \\
\forall a_1^\iota, \ldots, a_n^\iota, t^\iota : \mathsf{in}(t, \mathsf{prod}_n(a_1, \ldots, a_n)) \Leftrightarrow & \text{(ProdDef)} \\
\qquad \wedge\ t = \mathsf{tup}_n(\mathsf{fcnapp}(t, 1), \ldots, \mathsf{fcnapp}(t, n)) & \\
\qquad \wedge\ \mathsf{in}(\mathsf{fcnapp}(t, 1), a_1) & \\
\qquad \wedge \ldots & \\
\qquad \wedge\ \mathsf{in}(\mathsf{fcnapp}(t, n), a_n) &
\end{array}$$

The axiom (ProdDef) is justified by the TLA$^+$ theorem

$$\forall a_1, \ldots, a_n : a_1 \times \cdots \times a_n = \{\langle x_1, \ldots, x_n \rangle : x_1 \in a_1, \ldots, x_n \in a_n\}$$

We chose this formulation because it permits a more efficient treatment of functional extensionality for tuples. If t and u are two tuples of the same length, and all of their components are equal, then $t = u$ can be derived from (ProdDef) alone.

Our theory does not include proper axiomatizations of natural, integer or real arithmetic. We would prefer arithmetic to be handled by a prover's internal engine, but this has only been implemented for our SMT encoding (Chapter 5). Still, there are a few facts about TLA$^+$'s arithmetical operators that are required so often that we decided to include them for this simple encoding. For instance, we specify *Int* to be closed under the operations $+$, $-$ (both the unary and binary versions), $*$. Likewise, $\div$ and $\%$ are defined as binary functions on *Int*, but the second argument must be positive. For every numerical constant $n : \iota$ that we declare, we specify $n \in Int$ and $n \geq 0$. Note that we treat -1 as $-(1)$, so we only declare constants for positive numbers. The set *Nat* is specified as the set of integers z such that $z \geq 0$.

Example 4.5.2. We go back to our running example, assuming rewriting has not been applied. The result with a top context Δ is

ASSUME NEW $\mathsf{in} : \iota \times \iota \to o$
NEW $\mathsf{subseteq} : \iota \times \iota \to o$
NEW $\mathsf{union} : \iota \to \iota$
NEW $\mathsf{subset} : \iota \to \iota$
NEW $\mathsf{setof}_1 : \iota \times (\iota \to \iota) \to \iota$
NEW $\mathsf{isafcn} : \iota \to o$
NEW $\mathsf{fcn} : \iota \times (\iota \to \iota) \to \iota$
NEW $\mathsf{domain} : \iota \to \iota$
NEW $\mathsf{fcnapp} : \iota \times \iota \to \iota$
NEW $\mathsf{arrow} : \iota \times \iota \to \iota$

$\forall a^\iota, b^\iota : \mathsf{subseteq}(a, b) \Leftrightarrow (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b))$, (SubseteqDef)
$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{union}(a)) \Leftrightarrow (\exists y^\iota : \mathsf{in}(x, y) \wedge \mathsf{in}(y, a))$, (UnionDef)
$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{subset}(a)) \Leftrightarrow (\forall y^\iota : \mathsf{in}(y, x) \Rightarrow \mathsf{in}(y, a))$, (SubsetDef)
$\forall F^{\iota\to\iota}, a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setof}_1(a, F)) \Leftrightarrow \exists y^\iota : \mathsf{in}(y, a) \wedge x =_\iota F(y)$, ($\text{SetofDef}_1$)

$\forall F^{\iota\to\iota}, a^\iota : \mathsf{isafcn}(\mathsf{fcn}(a, F))$, (FcnIsafcn)
$\forall F^{\iota\to\iota}, a^\iota : \mathsf{domain}(\mathsf{fcn}(a, F)) =_\iota a$, (FcnDom)
$\forall F^{\iota\to\iota}, a^\iota, x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{fcnapp}(\mathsf{fcn}(a, F), x) =_\iota F(x)$, (FcnApp)
$\forall a^\iota, b^\iota, f^\iota : \mathsf{in}(f, \mathsf{arrow}(a, b)) \Leftrightarrow$ (ArrowDef)
$\quad \wedge\, \mathsf{isafcn}(f)$
$\quad \wedge\, \mathsf{domain}(f) =_\iota a$
$\quad \wedge\, (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), b))$,

NEW $A : \iota$, NEW $B : \iota$, NEW $f : \iota$,
$\mathsf{in}(f, \mathsf{arrow}(\mathsf{domain}(f), B))$, ($\text{Hyp}_1$)
$\mathsf{subseteq}(\mathsf{domain}(f), A))$ (Hyp_2)

PROVE $\mathsf{in}(f, \mathsf{union}(\mathsf{setof}_1(\mathsf{subset}(A), \lambda X^\iota : \mathsf{arrow}(X, B))))$ (Goal)

```
thf(fresh_mem, type, ( mem: $i > $i > $o )).
thf(fresh_subsetEq, type, ( subsetEq: $i > $i > $o )).
thf(fresh_union, type, ( union: $i > $i )).
thf(fresh_subset, type, ( subset: $i > $i )).
thf(fresh_setof_1, type, ( setof_1: $i > ( $i > $i ) > $i )).
thf(fresh_fcnIsafcn, type, ( fcnIsafcn: $i > $o )).
thf(fresh_fcn, type, ( fcn: $i > ( $i > $i ) > $i )).
thf(fresh_fcnDom, type, ( fcnDom: $i > $i )).
thf(fresh_fcpApp, type, ( fcpApp: $i > $i > $i )).
thf(fresh_arrowSet, type, ( arrowSet: $i > $i > $i )).

% axioms omitted...

thf(fresh_a, type, ( a: $i )).
thf(fresh_b, type, ( b: $i )).
thf(fresh_f, type, ( f: $i )).

thf(fact35, axiom,
  ( mem @ f @
    ( arrowSet @ ( fcnDom @ f ) @ b ) )).

thf(fact36, axiom,
  ( subsetEq @ ( fcnDom @ f ) @ a )).

thf(goal, conjecture,
  ( mem @ f @
    ( union @
      ( setof_1 @ ( subset @ a ) @
        ( ^ [ X: $i ] :
          ( arrowSet @ X @ b ) ) ) ) )).
```

Figure 4.5: Running Example Encoded in TPTP

Axioms that play no part in the proof have been omitted (set and functional extensionality). All the functional axioms are added, plus some axioms of set theory.

4.6 Translation to TPTP

The proof obligation is now a sequent $\Delta, \Gamma \vdash \phi$ in the logic $\mathcal{L}^\iota$, which we view as a subset of HOL. All the relevant information about TLA$^+$'s semantics has been encoded into the Δ part of the context. The last step is the translation to TPTP/THF itself. The translation is direct and without any difficult point to address, so we will simply demonstrate it by showing the result of encoding our running example. The output is displayed below; we have omitted the axioms so that it can fit on the page.

The sorts ι and o are represented by the builtin sorts `$i` and `$o`, respectively. A functional

type $s_1 \to s_2$ is represented $s_1 > s_2$. There is no builtin type for functions of several arguments, but a type $s_1 \times s_2 \to s_3$ may be encoded as $s_1 \to (s_2 \to s_3)$ in HOL, which is written $s_1 > (s_2 > s_3)$ in TPTP. An application $f(x, y)$ is then translated as $(f\ @\ x)\ @\ y$. The connective $>$ has right associativity and @ has left associativity, therefore parentheses can be removed for both expressions: $s_1 > s_2 > s_3$ and $f\ @\ x\ @\ y$.

4.7 Evaluation

We implemented the TPTP encoding in order to integrate Zipperposition as a backend solver for TLAPS. Zipperposition is a superposition theorem prover for first-order logic with equality and theories [20]. It features an extension to support higher-order logic [7, 82, 83].

In this section, we evaluate the new Zipperposition backend on a large set of TLA$^+$ specifications. We focus our evaluation on the effects of enabling rewriting in the TPTP encoding, and the performances of Zipperposition compared to the Isabelle backend.

4.7.1 Methodology

Our data is a set of TLA$^+$ specifications coming from three different sources:

- The library of TLA$^+$ Examples;[2]
- The TLAPS Examples as found in the distribution of the tool;
- A recent specification of Lamport's *Deconstructed Bakery* algorithm [49].

The full dataset contains 35 specifications with proofs, from which TLAPS generates a total number of 4612 proof obligations. Specifications vary wildly in size: the largest file, which contains the proof of refinement for the Deconstructed Bakery, contains 892 obligations; in contrast, many files contains only elementary results of fewer than 10 obligations.

We proceeded in two steps for our experiment: first, we generated benchmarks for each solver by calling TLAPS on the TLA$^+$ files; then, we ran the solvers on their respective benchmarks. Each benchmarks corresponds to a different encoding of TLA$^+$ (Isabelle, TPTP without rewriting, TPTP with rewriting). For the Isabelle benchmark, we tested Isabelle for the three tactics `auto`, `blast` and `force`, which are the tactics available from TLAPS. We considered a given obligation to be solved by Isabelle if at least one tactic could solve it.

For all solvers, we set a timeout of 5 seconds, which is the default timeout in TLAPS. The experiment was carried out on a Dell Latitude laptop with a 1.90 GHz Intel Core i7 processor.

4.7.2 Results and Discussion

The results are presented in Table 4.1 and Table 4.2. The first table compares the results of Zipperposition with and without rewriting. The second table compares Isabelle with Zipperposition—for the latter, we combined the results obtained by disabling or enabling rewriting. For all tables, we indicate the number of obligations solved by each backend (top numbers) and the number of obligations solved *uniquely* by each backend (bottom numbers).

As Table 4.1 demonstrates, rewriting's impact on performances is only marginal. The overlap between the two versions of the TPTP encoding (2400 obligations) is substantial.

[2] `https://github.com/tlaplus/Examples`

Specification	Size (# POs)	Backend	
		Zipper.	Zipper.+RW
TLA$^+$ Distribution	1996	1124 10	1124 10
TLAPS Distribution	1402	810 13	800 3
Deconstructed Bakery	1214	510 21	498 9
Total	4612	2444 44	2422 22

Table 4.1: Number of Obligations Solved by Zipperposition

Specification	Size (# POs)	Backend	
		Isabelle	Zipperposition
TLA$^+$ Distribution	1996	1106 181	1134 209
TLAPS Distribution	1402	898 204	813 119
Deconstructed Bakery	1214	609 196	519 106
Total	4612	2613 581	2466 434

Table 4.2: Comparison of the Isabelle and Zipperposition Backends

This is not especially surprising as the rewriting system we implement is very elementary, compared to the rewriting system of the SMT encoding it is based on. Enabling rewriting actually results in *less* obligations being solved. One possible explanation is that rewriting may remove important terms by eliminating TLA$^+$ primitives, making certain existential statements harder to prove; however, we could not find a particular obligation that clearly fails for that specific reason.

Table 4.2 shows that Isabelle and Zipperposition perform rather similarly, with Isabelle solving 147 additional obligations in total. Isabelle uniquely solves 581 obligations, Zipperposition uniquely solves 434, which indicates the backends have complementary strengths. Combined in a portfolio, they solve 3047 obligations.

Chapter 5

An Optimized Encoding into First-order SMT

5.1 Overview

In Chapter 4, we described a direct encoding of TLA$^+$ into HOL, more precisely the THF dialect of TPTP. We proved that encoding sound with respect to the semantics defined in Chapter 3. In this chapter, we describe our encoding to SMT-LIB.

The encoding is presented in Figure 5.1. The general structure is identical to the encoding presented in the previous chapter (Figure 4.1), with two steps completely preserved (recovering formulas, optional rewriting). The axiomatization is adapted for SMT and completed with *user patterns* (triggers) to guide the instantiation module of SMT. Expressions are also transformed during this step to encode second-order expressions, arithmetical expressions, and equalities relevant to the axiom of set extensionality. The final translation step is of course modified to target SMT-LIB. The final problem sets the SMT logic to UFNIA by default, and UFLIA when the target solver is veriT.

The target logic $\mathcal{L}_1^{\iota,\mathsf{int}}$ is a restriction of the logic $\mathcal{L}^s$ with an uninterpreted sort ι and the *interpreted* sort int of SMT. We assume the arithmetical operators of SMT are present with their intended semantics.[1] Moreover, second-order applications are not permitted in $\mathcal{L}_1^{\iota,\mathsf{int}}$. Operator symbols cannot occur as arguments to other operators, and lambda-expressions cannot appear at all. Nested sequents cannot be parameterized by operators, as this is a form of second-order quantification. When an obligation $\Gamma \vdash e$ is such that Γ contains a sequent S, and S contains an operator declaration, like NEW $F(\_)$, then S is simply omitted to finish the translation. If S is essential to the proof then SMT will likely fail, so users should opt for a higher-order backend like Isabelle or Zipperposition in that case.

The rest of this chapter is organized as follows. We first provide some background on SMT solvers and E-matching (Section 5.2). This will also serve as a general introduction to our method supported by some examples. We then present our strategy for writing an axiomatization of TLA$^+$ with triggers (Section 5.3). We then present some fragments of the axiomatizations that are handled in a special ways: axiom schemas, integer arithmetic, and set extensionality (Section 5.4). Finally, we evaluate our SMT encoding and compare its performances with the original SMT encoding (Section 5.5).

[1] For a presentation, see the SMT documentation [6] or the page `https://smtlib.cs.uiowa.edu/theories-Ints.shtml`

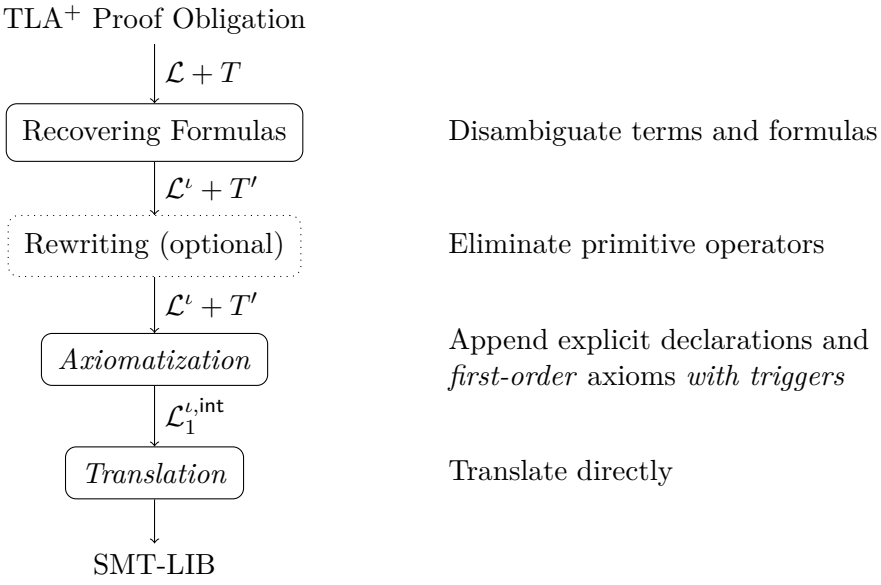


Figure 5.1: SMT Encoding Overview

5.2 E-matching Patterns

SMT solvers extend SAT solvers with procedures for first-order theories. Theories of interest for SMT users include integer and real arithmetic (linear, non-linear), bit vectors, and extensional arrays. First-order logic is undecidable in general, but quantifiers over uninterpreted domains are still supported by an *instantiation module*. The purpose of this module is to produce instances of the quantified formulas of the problem to insert into the quantifier-free problem.

Propositional logic is decidable, so we will focus on the much harder problem of generating the right instances of our first-order axioms. In this context, it is helpful to restrict our study to universally quantified formulas in prenex form:

$$\forall x_1^{s_1}, \ldots, x_n^{s_n} : \phi \quad \text{where } \phi \text{ is quantifier-free}$$

All problems of first-order logic can be reduced to problems with formulas of the form above through the application of elementary logical equivalences and Skolemization [3].

Example 5.2.1. Take the axiom specifying the subset relation in TLA$^+$:

$$\forall a^{\iota}, b^{\iota} : \mathsf{subseteq}(a, b) \Leftrightarrow \left(\forall x^{\iota} : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)\right) \qquad \text{(Subseteq)}$$

To put (Subseteq) in prenex form, we apply logical rules to move the nested quantifier $\forall x^{\iota}$ up the structure of the formula. Before this, we split the equivalence in two implications:

$$\begin{aligned} \forall a^{\iota}, b^{\iota} : \ & \wedge \mathsf{subseteq}(a, b) \Rightarrow \left(\forall x^{\iota} : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)\right) \\ & \wedge \left(\forall x^{\iota} : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)\right) \Rightarrow \mathsf{subseteq}(a, b) \end{aligned}$$

We can now apply the following rules for moving quantifiers above implications:

$$\begin{aligned} \phi \Rightarrow (\forall x : \psi) \quad &\Leftrightarrow \quad \forall x : \phi \Rightarrow \psi \\ (\forall x : \phi) \Rightarrow \psi \quad &\Leftrightarrow \quad \exists x : \phi \Rightarrow \psi \end{aligned}$$

As the second rule shows, the $\forall x^\iota$ that originally occurs on the left of a $\Rightarrow$ must become existential. Quantifiers that become existential when moved to the top are called *strong*, and the others *weak*. To finish putting the axiom in prenex form, we put the conjuncts in two separate axioms:

$$\forall a^\iota, b^\iota, x^\iota : \mathsf{subseteq}(a, b) \Rightarrow \big(\mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)\big) \qquad \text{(SubseteqElim)}$$
$$\forall a^\iota, b^\iota : \exists x^\iota : \big(\mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)\big) \Rightarrow \mathsf{subseteq}(a, b) \qquad \text{(SubseteqIntro)}$$

Skolemization is a technique for eliminating strong quantifiers. The intuition is that existentially quantified variables can be safely replaced by terms playing the role of witnesses in inner formulas. Since $\exists x^\iota$ occurs below two universal quantifiers, we introduce a new binary symbol w and rewrite (SubseteqIntro) as

$$\forall a^\iota, b^\iota : (\mathsf{in}(w(a, b), a) \Rightarrow \mathsf{in}(w(a, b), b)) \Rightarrow \mathsf{subseteq}(a, b)$$

SMT solvers vary in how they preprocess formulas and apply Skolemization, which itself admits several definitions. We will not systematically transform expressions to put them in universal prenex form as we did in Example 5.2.1, because we assume SMT solvers can handle this part more efficiently. However we will consider moving weak quantifiers at the top of our axioms, as this makes it easier to focus on quantifier instantiation. Finding relevant instances is a difficult problem, and we need to assist the SMT solvers for this task.

E-matching is one of several approaches to the problem of quantifier instantiation [24, 36, 63, 64]. The basic principle of E-matching is to look in the current ground problem for terms matching syntactical patterns. Patterns can be user-defined or heuristically generated. Formally, we represent the ground problem with a set E of propositional formulas with uninterpreted symbols and equalities, and a pattern by a term p with free variables. A match is a ground term t and a substitution σ such that $E \models t = p\sigma$. Typically, solvers will look for terms t that occur in the current problem, possibly resulting from Skolemization; those terms are said to be *known*.

Example 5.2.2. Consider the following elementary theorem of TLA$^+$:

MODULE *Subseteq*

EXTENDS *TLAPS*

THEOREM ASSUME NEW S, NEW T, NEW U,
$U = S \cap T$
PROVE $U \subseteq S$
OBVIOUS

This is proved intuitively by taking an $x \in U$ and proving $x \in S$. Because $U = S \cap T$, we have $x \in S \cap T$ and therefore $x \in S$. We will now look at a possible formalization in SMT to see how the automated proof might proceed.

The obligation is first-order and does not feature arithmetic, so the direct encoding of Chapter 4 results in a problem that SMT can handle. We insert two axioms into the problem for specifying the subset relation and the intersection operator (one unnecessary axiom is omitted). Axioms are annotated with patterns indicated between curly braces. The obligation is proved if the set of formulas below is found unsatisfiable. Note that we use different

notations to highlight the structures of first-order expressions: $\mathsf{subseteq}$ is the relation $\subseteq$, cap is the binary operator $\cap$, in is the relation $\in$.

$$\forall a^\iota, b^\iota : \{\mathsf{subseteq}(a,b)\} \qquad (\mathsf{in}(w(a,b),a) \Rightarrow \mathsf{in}(w(a,b),b)) \Rightarrow \mathsf{subseteq}(a,b) \qquad \text{(Subseteq)}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x,\mathsf{cap}(a,b))\} \quad \mathsf{in}(x,\mathsf{cap}(a,b)) \Leftrightarrow \mathsf{in}(x,a) \wedge \mathsf{in}(x,b) \qquad \text{(Cap)}$$

$$U = \mathsf{cap}(S,T) \qquad \text{(H)}$$

$$\neg\mathsf{subseteq}(U,S) \qquad \text{(G)}$$

The term $\mathsf{subseteq}(U,S)$ in the goal matches the pattern of (Subseteq). We say that the axiom is triggered by this match. The match is represented by the substitution $\{a \mapsto U, b \mapsto S\}$. That substitution is used to generate an instance of the axiom, which is introduced into the ground problem:

$$(\mathsf{in}(w(U,S),U) \Rightarrow \mathsf{in}(w(U,S),S)) \Rightarrow \mathsf{subseteq}(U,S) \qquad \text{(F1)}$$

Note that $w(U,S)$ plays the role of x in our intuitive proof, because it now suffices to assume $w(U,S)$ is an element of U and prove it is an element of S to derive a contradiction.

The subterm $\mathsf{in}(w(U,S),U)$ in (F1) matches the pattern $\{\mathsf{in}(x,\mathsf{cap}(a,b))\}$ for the substitution $\{a \mapsto S, b \mapsto T, x \mapsto w(U,S)\}$. E-matching finds matches *modulo equality*, so the equality in (H) is accounted for to match U with $\mathsf{cap}(a,b)$. The new fact is

$$\mathsf{in}(w(U,S),\mathsf{cap}(S,T)) \Leftrightarrow \mathsf{in}(w(U,S),S) \wedge \mathsf{in}(w(U,S),T) \qquad \text{(F2)}$$

Now a contradiction can be derived from the ground problem, which means the problem has been solved. That last step was essentially the expansion of $x \in S \cap T$ into $x \in S \wedge x \in T$.

Example 5.2.2 considered axioms with single patterns, but typically we want to match several patterns at the same time. A *trigger* for the variables $x_1, \ldots, x_n$ is a set of patterns $\{p_1, \ldots, p_k\}$ such that the free variables of all p_j are exactly the variables x_i. We may annotate axioms with several triggers, so the general form is

$$\begin{array}{ll} \forall x_1^{s_1}, \ldots, x_n^{s_n} : & \{p_{1,1}, \ldots, p_{1,n_1}\} \\ & \cdots \\ & \{p_{m,1}, \ldots, p_{1,n_m}\} \\ \phi & \end{array}$$

where ϕ is quantifier-free. A term t is a match if there exists a substitution σ such that $E \models t = p_{i,j}\sigma$ for some i and all $j \leq n_i$. So any individual trigger can produce an instance, but all patterns of that trigger must match simultaneously.

Modern SMT solvers implement efficient procedures for detecting terms matching triggers [24]. As a result, triggers have become a popular solution among SMT users for writing custom axiomatizations [50, 55]. Triggers offer some control over the instantiation engine to the user, which is why they are sometimes presented as a way to program SMT solvers. In fact, it is possible to define notions of termination and completeness for triggers in a satisfactory way [30]. Intuitively, we will say an axiomatization terminates if only a finite number of instances can be produced from any finite ground problem. We will say it is complete if any provable problem is still provable when only instances produced by triggers are allowed.

SMT solvers can generate triggers automatically, but we found that we could achieve better results by selecting our own triggers. The next section will detail our strategy for trigger selection through motivating examples. On the subject of soundness, there is nothing to add from our previous discussion on axiomatization (Section 4.5): triggers do not change the semantics of formulas, so they are perfectly safe to use. However, termination and completeness are not guaranteed. Our axiomatization enjoys neither of these properties, but many of our choices are still guided by considerations of termination and completeness: we select as many adequate triggers as possible to anticipate different proof scenarios, but we reject triggers that obviously result in irrelevant instances or matching loops.

5.3 Selecting Axioms and Triggers for TLA$^+$

5.3.1 Case Study

In this section, we will focus on a simple theorem of TLA$^+$, displayed below.

MODULE *Intersection*

EXTENDS *TLAPS*, *Integers*

THEOREM ASSUME NEW S,
$S \cap Int \subseteq \{\}$
PROVE $1 \notin S$
OBVIOUS

Here is an intuitive proof: we assume $1 \in S$ and derive a contradiction; since $1 \in S$ and $1 \in Int$, we have $1 \in S \cap Int$; since $S \cap Int \subseteq \{\}$, we have $1 \in \{\}$, which is contradictory.

The proof involves three set-theoretic primitive operators of TLA$^+$, namely the subset relation, the intersection and the empty set. No arithmetic is needed beyond the fact that $1 \in Int$. We will look at a few different ways to formulate the axioms with triggers, starting from a rather naive theory and finishing with the actual theory we implement. For every attempt, it is important that we consider *exclusively* instances resulting from triggers, so that the solution scales to larger problems. The problem above is simple enough so that SMT solvers can find all instances by enumerative approaches; in very large problems, the search space of ground terms may be too large, and there may be too many quantifiers for this approach to work.

Example 5.3.1 (First Attempt)**.** The encoded problem is displayed below. We treat 1 as a constant of the sort ι and assume the fact $1 \in Int$ has been derived. Details about how arithmetic is handled can be found in Section 5.4.2.

$\forall a^\iota, b^\iota : \{\mathsf{subseteq}(a, b)\}$	$\mathsf{subseteq}(a, b) \Leftrightarrow (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b))$	(Subseteq)
$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{cap}(a, b))\}$	$\mathsf{in}(x, \mathsf{cap}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \mathsf{in}(x, b)$	(Cap)
$\forall x^\iota : \{\mathsf{in}(x, \mathsf{enum}_0)\}$	$\neg\mathsf{in}(x, \mathsf{enum}_0)$	(Empty)
$\mathsf{subseteq}(\mathsf{cap}(S, Int), \mathsf{enum}_0)$		(H1)
$\mathsf{in}(1, Int)$		(H2)
$\mathsf{in}(1, S)$		(G)

Here, the only match possible is the term $\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0)$ for the axiom (Subseteq) and the substitution $\{a \mapsto \mathsf{cap}(S, \mathit{Int}), b \mapsto \mathsf{enum}_0\}$. This results in the new fact

$$\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \Leftrightarrow (\forall x^\iota : \mathsf{in}(x, \mathsf{cap}(S, \mathit{Int})) \Rightarrow \mathsf{in}(x, \mathsf{enum}_0)) \tag{F1}$$

Let us put (F1) in prenex form to continue the proof. This is the exact process we described earlier, it results in the two facts

$$\forall x^\iota : \mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \wedge \mathsf{in}(x, \mathsf{cap}(S, \mathit{Int})) \Rightarrow \mathsf{in}(x, \mathsf{enum}_0) \tag{F1a}$$

$$(\mathsf{in}(w(\dots), \mathsf{cap}(S, \mathit{Int})) \Rightarrow \mathsf{in}(w(\dots), \mathsf{enum}_0)) \Rightarrow \mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \tag{F1b}$$

where w is a new symbol and $w(\dots)$ is the term $w(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0)$.

(F1a) is a quantified formula. Since it has no trigger and we do not consider other methods of instantiation, we cannot use it. However, the path to a contradiction requires instantiating the formula with 1. Therefore, we are stuck. The fact (F1b) is ground, but it will not make the proof progress.

The problem from Example 5.3.1 is not provable using triggers only. Before we present a solution, let us stress the obvious analogy between those triggers and the rewriting rules

$$\begin{aligned} a \subseteq b &\ \triangleright\ \forall x : x \in a \Rightarrow x \in b \\ x \in a \cap b &\ \triangleright\ x \in a \wedge x \in b \\ x \in \{\} &\ \triangleright\ \textsc{false} \end{aligned}$$

Triggers were directly taken from the left patterns of the rewriting rules. We might say the axioms essentially implement those rewritings into the SMT problem. In both cases, a universal quantifier with no trigger is introduced into the problem, getting us stuck.

The solution is to reformulate (Subseteq) as we did in the previous section. The axiom is split in two; in one case, the nested quantifier $\forall x^\iota$ is moved at the top. This opens up the possibility to assign a trigger that includes the three variables a, b and x.

Example 5.3.2 (Second Attempt)**.** Compared to the previous example, only (Subseteq) has changed. We do not Skolemize (SubseteqIntro) below—the axiom plays no role in the proof. The axiom (SubseteqElim) is slightly reformulated to save some parentheses.

$$\forall a^\iota, b^\iota : \{\mathsf{subseteq}(a, b)\} \qquad (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)) \Rightarrow \mathsf{subseteq}(a, b) \tag{SubseteqIntro}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{subseteq}(a, b), \mathsf{in}(x, a)\} \qquad \mathsf{subseteq}(a, b) \wedge \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b) \tag{SubseteqElim}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{cap}(a, b))\} \qquad \mathsf{in}(x, \mathsf{cap}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \mathsf{in}(x, b) \tag{Cap}$$

$$\forall x^\iota : \{\mathsf{in}(x, \mathsf{enum}_0)\} \qquad \neg\mathsf{in}(x, \mathsf{enum}_0) \tag{Empty}$$

$$\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \tag{H1}$$

$$\mathsf{in}(1, \mathit{Int}) \tag{H2}$$

$$\mathsf{in}(1, S) \tag{G}$$

Despite the modification, the situation has not changed. The axiom (SubseteqIntro) is triggered by $\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0)$, but this does not make the proof progress.

Formulating the axiom (Subseteq) differently has not solved the issue, but it brought us closer to the solution. We know that (SubseteqElim) is the relevant axiom and we know how it must be instantiated. The expected match is $\{a \mapsto \mathsf{cap}(S, \mathit{Int}), b \mapsto \mathsf{enum}_0, x \mapsto 1\}$. The terms SMT must know about in order to find this match are $\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0)$ and $\mathsf{in}(1, \mathsf{cap}(S, \mathit{Int}))$. Only the latter is missing.

The current trigger for (Cap) can be used to generate the definition of $\mathsf{in}(x, \mathsf{cap}(a, b))$ when the term is already known. However, in situations when that term is not known but must be inferred, the axiom is useless. It is then natural to provide a new trigger matching the other side of the equivalence, so that whenever $\mathsf{in}(x, a)$ and $\mathsf{in}(x, b)$ are known, $\mathsf{in}(x, \mathsf{cap}(a, b))$ is inferred.

Example 5.3.3 (Third Attempt)**.** We add a second trigger to (Cap) and consider the problem

$$\forall a^\iota, b^\iota : \{\mathsf{subseteq}(a, b)\} \qquad (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)) \Rightarrow \mathsf{subseteq}(a, b) \qquad \text{(SubseteqIntro)}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{subseteq}(a, b), \mathsf{in}(x, a)\} \qquad \mathsf{subseteq}(a, b) \wedge \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b) \qquad \text{(SubseteqElim)}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{cap}(a, b))\} \quad \{\mathsf{in}(x, a), \mathsf{in}(x, b)\} \qquad \mathsf{in}(x, \mathsf{cap}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \mathsf{in}(x, b) \qquad \text{(Cap)}$$

$$\forall x^\iota : \{\mathsf{in}(x, \mathsf{enum}_0)\} \qquad \neg\mathsf{in}(x, \mathsf{enum}_0) \qquad \text{(Empty)}$$

$$\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \qquad \text{(H1)}$$

$$\mathsf{in}(1, \mathit{Int}) \qquad \text{(H2)}$$

$$\mathsf{in}(1, S) \qquad \text{(G)}$$

The terms $\mathsf{in}(1, \mathit{Int})$ and $\mathsf{in}(1, S)$ form an appropriate match, but so does the term $\mathsf{in}(1, S)$ by itself (both a and b are matched to S). If the instantiation engine decides to use that match, SMT will generate the fact

$$\mathsf{in}(1, \mathsf{cap}(S, S)) \Leftrightarrow \mathsf{in}(1, S) \wedge \mathsf{in}(1, S) \qquad \text{(F1)}$$

Then the new term $\mathsf{in}(1, \mathsf{cap}(S, S))$ matches (Cap) again; using this match will result in yet another matching term. Clearly, the new trigger leads to a *matching loop*.

The trigger $\{\mathsf{in}(x, a), \mathsf{in}(x, b)\}$ does make the problem solvable, but it may also result in SMT generating too many irrelevant instances. This could make large proof obligations significantly harder to prove in a short amount of time, so we need to avoid this kind of problem. Hopefully, the problem can be fixed by using triggers as *guards*. To ensure no new term of the form $\mathsf{cap}(a, b)$ is ever introduced in the ground problem, it suffices to only select triggers that include that term.

Example 5.3.4 (Final Attempt)**.** This version of the SMT problem is the one we do implement. Only (Cap) has changed compared with the previous version. The solution presented here is not the only possible one; in particular, we could have selected the trigger $\{\mathsf{in}(x, a), \mathsf{in}(x, b), \mathsf{cap}(a, b)\}$. Our solution does not require both $\mathsf{in}(x, a)$ and $\mathsf{in}(x, b)$ to

be known.

$$\forall a^\iota, b^\iota : \{\mathsf{subseteq}(a,b)\} \qquad (\forall x^\iota : \mathsf{in}(x,a) \Rightarrow \mathsf{in}(x,b)) \Rightarrow \mathsf{subseteq}(a,b) \qquad \text{(SubseteqIntro)}$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{subseteq}(a,b), \mathsf{in}(x,a)\} \qquad \mathsf{subseteq}(a,b) \wedge \mathsf{in}(x,a) \Rightarrow \mathsf{in}(x,b) \qquad \text{(SubseteqElim)}$$

$$\begin{aligned}\forall a^\iota, b^\iota, x^\iota : \ & \{\mathsf{in}(x, \mathsf{cap}(a,b))\} \\ & \{\mathsf{in}(x,a), \mathsf{cap}(a,b)\} \\ & \{\mathsf{in}(x,b), \mathsf{cap}(a,b)\}\end{aligned} \qquad \mathsf{in}(x, \mathsf{cap}(a,b)) \Leftrightarrow \mathsf{in}(x,a) \wedge \mathsf{in}(x,b) \qquad \text{(Cap)}$$

$$\forall x^\iota : \{\mathsf{in}(x, \mathsf{enum}_0)\} \qquad \neg \mathsf{in}(x, \mathsf{enum}_0) \qquad \text{(Empty)}$$

$$\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \qquad \text{(H1)}$$

$$\mathsf{in}(1, \mathit{Int}) \qquad \text{(H2)}$$

$$\mathsf{in}(1, S) \qquad \text{(G)}$$

The axiom (Cap) is triggered by the terms $\mathsf{in}(1,S)$ and $\mathsf{cap}(S, \mathit{Int})$, resulting in the fact (F1) below. Then the facts (F2) and (F3) are successively introduced as instances of (SubseteqElim) and (Empty). We do not detail the matching terms for those.

$$\mathsf{in}(1, \mathsf{cap}(S, \mathit{Int})) \Leftrightarrow \mathsf{in}(1,S) \wedge \mathsf{in}(1, \mathit{Int}) \qquad \text{(F1)}$$

$$\mathsf{subseteq}(\mathsf{cap}(S, \mathit{Int}), \mathsf{enum}_0) \wedge \mathsf{in}(1, \mathsf{cap}(S, \mathit{Int})) \Rightarrow \mathsf{in}(1, \mathsf{enum}_0) \qquad \text{(F2)}$$

$$\neg \mathsf{in}(1, \mathsf{enum}_0) \qquad \text{(F3)}$$

The ground problem is now contradictory, therefore the proof is done.

5.3.2 General Strategy

We refined our axioms and triggers progressively by inspecting some difficult TLA$^+$ obligations and looking for trigger-based solutions. Thus or method for trigger selection is essentially heuristic and does not follow a strict set of rules. Nevertheless, we were able to extract some recurring ideas. In this section, we formulate some key principles behind our strategy for trigger selection. First, we define a few general notions.

We denote by $\vec{x}$ a sequence of variables $x_1, \ldots, x_n$. The length n of the sequence is left ambiguous. A sequence of variables annotated by the sort s is written $\vec{x}^{\,s}$. Our axioms are all of the form

$$\forall \vec{x}^{\,\iota} : \{p_1, \ldots, p_n\} \ \phi$$

where ϕ is without weak quantifiers. For all terms t, let $\mathrm{Terms}(t)$ be the collection of all first-order subterms of t (including t itself). For a given axiom with a trigger as noted above, the terms *generated* by the trigger are the elements of the collection

$$\mathrm{Terms}(\phi) \setminus \bigcup_{1 \le i \le n} \mathrm{Terms}(p_i)$$

By inspecting the terms in that collection, we can predict what kinds of terms an instance may introduce into the ground problem.

Example 5.3.5. Consider again the axiom for the intersection of two sets:

$$\forall a^\iota, b^\iota, x^\iota : \mathsf{in}(x, \mathsf{cap}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \mathsf{in}(x, b)$$

Here are the four triggers we considered and the set of terms each generates:

$\{\mathsf{in}(x, \mathsf{cap}(a, b))\}$	generates	$\mathsf{in}(x, a),\ \mathsf{in}(x, b)$
$\{\mathsf{in}(x, a), \mathsf{in}(x, b)\}$	generates	$\mathsf{in}(x, \mathsf{cap}(a, b)),\ \mathsf{cap}(a, b)$
$\{\mathsf{in}(x, a), \mathsf{cap}(a, b)\}$	generates	$\mathsf{in}(x, \mathsf{cap}(a, b)),\ \mathsf{in}(x, b)$
$\{\mathsf{in}(x, b), \mathsf{cap}(a, b)\}$	generates	$\mathsf{in}(x, \mathsf{cap}(a, b)),\ \mathsf{in}(x, a)$

Borrowing the terminology of Leino [50], we call *adequate* a trigger that mentions all quantified variables, and *parsimonious* a trigger $\{p_1, \ldots, p_n\}$ that becomes inadequate when any p_i is removed. Adequacy is a strict requirement of SMT. Parsimony captures the notion of a most general trigger: a nonparsimonious trigger can be simplified into another trigger that matches more often.

Example 5.3.5 above presents the four triggers for cap that are both adequate and parsimonious. Of these four triggers, only the one generating the term $\mathsf{cap}(a, b)$ has been rejected, because it results in an obvious matching loop. The remaining triggers only generate terms that have in as a top connective. Besides the matching loop, we make the following observation: it is generally useful to generate new facts of the form $\mathsf{in}(t_1, t_2)$ where t_1 and t_2 are known, because it may lead to discover some implicit connexions between the different sets involved in a proof; on the contrary, we can hypothesize that many TLA$^+$ obligations do not require inventing sets that are not already mentioned in the obligation's statement, so generating terms of the form $\mathsf{cap}(t_1, t_2)$ is not as useful.

We found helpful to classify the operators of TLA$^+$ as either *constructors* or *accessors*. Intuitively, constructors (like cap) build complex objects from simpler ones, and accessors (like in) give information about the objects they are applied to. We assume typical valid TLA$^+$ obligations can be proved without mentioning objects (sets, functions, etc.) that are not already mentioned in the obligation; thus we avoid generating terms $C(t_1, \ldots, t_n)$ where C is a constructor. We also encourage the generation of specific terms $A(t_1, \ldots, t_n)$ where A is an accessor.

We may now formulate our general strategy for selecting triggers:

I Only axioms of the form $\forall \vec{x}^\iota : \phi$ where ϕ is without weak quantifiers are considered. Axioms are reformulated and sometimes split to obtain that form.

II To select triggers, we start from the set of adequate and parsimonious triggers that can be formulated using only subterms of ϕ.

III We reject a trigger if it generates a term $C(t_1, \ldots, t_n)$ where C is a constructor, or if it contains two terms with no variable in common.

IV For all terms $A(t_1, \ldots, t_n)$ in ϕ where A is an accessor, we look if the term is generated by at least one trigger. If not, and if generating $A(t_1, \ldots, t_n)$ is important for some proofs, we consider adding a nonparsimonious trigger.

The third principle reduces the risk of generating irrelevant terms and prevents matching loops. The fourth principle increases the chances of generating the terms necessary to a proof.

$$\forall a^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{union}(a))\} \qquad \text{(UnionElim)}$$
$$\mathsf{in}(x, \mathsf{union}(a)) \Rightarrow \exists y^\iota : \mathsf{in}(x, y) \wedge \mathsf{in}(y, a)$$

$$\forall a^\iota, x^\iota, y^\iota : \{\mathsf{in}(y, a), \mathsf{in}(x, \mathsf{union}(a))\} \qquad \text{(UnionIntro)}$$
$$\{\mathsf{in}(x, y), \mathsf{in}(x, \mathsf{union}(a))\}$$
$$\{\mathsf{in}(x, y), \mathsf{in}(y, a), \mathsf{union}(a)\}$$
$$\mathsf{in}(x, y) \wedge \mathsf{in}(y, a) \Rightarrow \mathsf{in}(x, \mathsf{union}(a))$$

Figure 5.2: Axioms for UNION

For the rest of this section, we illustrate how these principles are applied to the axioms of *set theory* and *functions* for TLA$^+$. Integer arithmetic will be dealt with elsewhere (Section 5.4.2) as it involves using SMT's native arithmetic. Other fragments of TLA$^+$'s theory (tuples, records, sequences...) are based on the axioms of functions. For the full list of axioms used in the SMT encoding, we refer the reader to Appendix C.

Set Theory

The following operators of ZF set theory are considered to be constructors: enum_n, union, subset, setst_P, setof_F, cup, cap, diff. The only accessor is the membership relation in. The following pattern is shared by many axioms of set theory:

$$\forall \vec{a}^\iota, x^\iota : \mathsf{in}(x, C(\vec{a})) \Leftrightarrow \phi(x, \vec{a}) \qquad \text{(C-Define)}$$

where C is a constructor. When the defining formula ϕ contains quantifiers, we split the axiom in two, resulting in a *introduction* and an *elimination* axioms:

$$\forall \vec{a}^\iota, x^\iota, \vec{y}^\iota : \mathsf{in}(x, C(\vec{a})) \Rightarrow \phi_1(x, \vec{a}, \vec{y}) \qquad \text{(C-Elim)}$$
$$\forall \vec{a}^\iota, x^\iota, \vec{y}^\iota : \phi_2(x, \vec{a}, \vec{y}) \Rightarrow \mathsf{in}(x, C(\vec{a})) \qquad \text{(C-Intro)}$$

The formula ϕ_1 may contain strong quantifiers. The formula ϕ_2 may contain weak quantifiers (which are strong for the axiom as a whole, since ϕ_2 occurs on the left of an implication).

To illustrate, we take the example of the TLA$^+$ operator UNION, which we denote union. Our axioms are displayed in Figure 5.2. The basic form of the axiom is

$$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{union}(a)) \Leftrightarrow \exists y^\iota : \mathsf{in}(x, y) \wedge \mathsf{in}(y, a)$$

The defining formula is an existential statement, so the axiom is split in two. The existential quantifier is still present in the elimination part; in the introduction part, it is moved at the front as a universal quantifier.

Axiom (UnionElim) receives only one trigger, $\{\mathsf{in}(x, \mathsf{union}(a))\}$. This is the only adequate trigger possible. We could apply Skolemization to the formula to have more candidate triggers. However, Skolem terms do not occur in the starting ground problem, so it is unclear if triggers

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{\mathsf{fcn}_F(a, c_1, \ldots, c_n)\} \quad \text{(FcnIsafcn)}$$
$$\mathsf{isafcn}(\mathsf{fcn}_F(a, c_1, \ldots, c_n))$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{\mathsf{fcn}_F(a, c_1, \ldots, c_n)\} \quad \text{(FcnDom)}$$
$$\mathsf{domain}(\mathsf{fcn}_F(a, c_1, \ldots, c_n)) = a$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{\mathsf{fcnapp}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), x)\} \quad \text{(FcnApp)}$$
$$\{\mathsf{in}(x, a), \mathsf{fcn}_F(a, c_1, \ldots, c_n)\}$$
$$\mathsf{in}(x, a) \Rightarrow \mathsf{fcnapp}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), x) = F(x, c_1, \ldots, c_n)$$

Figure 5.3: Axioms for Functions

that mention such terms will generate relevant instances. In general, we do not Skolemize our axioms.

Axiom (UnionIntro) is an interesting case. The first two triggers are parsimonious. Other parsimonious triggers are

$$\{\mathsf{in}(x, y), \mathsf{in}(y, a)\}$$
$$\{\mathsf{in}(x, y), \mathsf{union}(a)\}$$

We reject the first one because it generates $\mathsf{union}(a)$, and the second one because the terms $\mathsf{in}(x, y)$ and $\mathsf{union}(a)$ have no variable in common (there are too many ways to match it).

The triggers we kept both mention the term $\mathsf{in}(x, \mathsf{union}(a))$, so they can never generate this term. For the introduction axioms of set theory, we found that many obligations relied on the implicit inference of such facts. Consider for instance

$$x \in y \wedge y \in S \wedge \text{UNION } S \subseteq T \Rightarrow x \in T$$

The fact $x \in \text{UNION } S$ follows from the assumptions, but it is not mentioned explicitly. The third trigger is added to handle this kind of obligations:

$$\{\mathsf{in}(x, y), \mathsf{in}(y, a), \mathsf{union}(a)\}$$

Note that it is not parsimonious.

In general, for all axioms of the form (C-Define), (C-Elim) or (C-Intro), all selected triggers contain either $\mathsf{in}(x, C(\vec{a}))$ or $C(\vec{a})$. This ensures no term $C(\vec{a})$ is ever generated. Furthermore, for axioms of the form (C-Define) and (C-Intro), we ensure at least one trigger generates the term $\mathsf{in}(x, C(\vec{a}))$. In most cases, this is already done by a parsimonious trigger. (UnionIntro) is the only exception among the axioms of ZF set theory.

Functions

TLA$^+$'s theory of functions involves the following primitive constructs:

$$[x \in S \mapsto F(x)], \quad \text{DOMAIN } f, \quad f[x], \quad [S \to T]$$

These constructs are respectively noted using the operators fcn_F, domain, fcnapp and arrow. The operator fcn_F is rather a family of operators parameterized by the higher-order argument F. We treat it as a constructor. The operators domain and fcnapp are accessors. Internally, the class of all functions is specified by a predicate isafcn, which is not accessible in the user language. We treat is as an accessor. The operator arrow is a constructor specified by an axiom of the form (C-Define).

Let us focus first on the specification of explicit functions with their domains and values. They are specified by three axiom schemas, presented in Figure 5.3. The subject of axiom schemas will be treated in more details in the next section. Each of the three schemas specifies the return value of some accessor for an explicit function as input. To clarify using TLA$^+$'s notations:

- (FcnIsafcn) specifies $\mathsf{isafcn}([y \in a \mapsto F(y)])$ to be true in all cases;
- (FcnDom) specifies DOMAIN $[y \in a \mapsto F(y)]$ to be a in all cases;
- (FcnApp) specifies $[y \in a \mapsto F(y)][x]$ to be $F(x)$ under the condition $x \in a$.

In general, each axiom gives the definition of some term $A(C(\vec{a}), \vec{x})$ under an optional condition ϕ. Thus all the axioms of Figure 5.3 fall under the pattern

$$\forall \vec{a}^{\,\iota}, \vec{x}^{\,\iota} : \phi(\vec{a}, \vec{x}) \Rightarrow A(C(\vec{a}), \vec{x}) = t(\vec{a}, \vec{x}) \qquad \text{(C-A-Define)}$$

where C is a constructor, A an accessor, ϕ a formula and t a term. The constructor C is fcn_F for the three schemas. ϕ is just TRUE for (FcnIsafcn) and (FcnDom)

It is helpful to consider the general form (C-A-Define), because this form is also applicable to the axioms of tuples, records and sequences. These theories introduce new constructors, but also reuse the accessors isafcn, domain and fcnapp.

We describe our strategy for the general axiom. Triggers that do not generate a term $C(\vec{a})$ can be of two sorts; some mention the term $C(\vec{a})$, some mention the term $A(C(\vec{a}), \vec{x})$. When the list of variables $\vec{x}$ is empty, the simple trigger $\{C(\vec{a})\}$ is already adequate and parsimonious, so we just select this one; the axioms (FcnIsafcn) and (FcnDom) fall under this case. If $\vec{x}$ is not empty, then $\{A(C(\vec{a}), \vec{x})\}$ is adequate and parsimonious, so we select it. In the case of (FcnApp), this corresponds to the trigger

$$\{\mathsf{fcnapp}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), x)\}$$

It is also desirable to select triggers that generate the term $A(C(\vec{a}), \vec{x})$. We also look for parsimonious triggers that contain $C(\vec{a})$. This must be done on a case by case basis for every axiom. Typically, adequate triggers are obtained by selecting subterms from the condition formula ϕ. In the case of (FcnApp), we select

$$\{\mathsf{in}(x, a), \mathsf{fcn}_F(a, c_1, \ldots, c_n)\}$$

We do not look for subterms in $F(x, c_1, \ldots, c_n)$, even though there might be more parsimonious triggers to select from those subterms. This is a limitation of our implementation, as we have not gone as far as implementing a procedure for selecting triggers automatically.

Lastly, the operator arrow is treated as a set constructor, and its axiom falls under the general form (C-Define). The axiom is split into an introduction and elimination axioms, and

triggers are selected for it as we described before. Consider the introduction part:

$$\begin{aligned}\forall a^\iota, b^\iota, f^\iota : \{&\mathsf{in}(f, \mathsf{arrow}(a,b))\} \\ &\wedge \mathsf{isafcn}(f) \\ &\wedge \mathsf{domain}(f) = a \\ &\wedge (\forall x^\iota : \mathsf{in}(x,a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f,x), b)) \\ &\Rightarrow \mathsf{in}(f, \mathsf{arrow}(a,b))\end{aligned} \tag{ArrowIntro}$$

This axiom is lacking a trigger to generate $\mathsf{in}(f, \mathsf{arrow}(a,b))$. The two parsimonious triggers below do generate this term:

$$\{\mathsf{isafcn}(f), \mathsf{arrow}(a,b)\}$$
$$\{\mathsf{domain}(f), \mathsf{arrow}(a,b)\}$$

But we reject them because they both feature two terms with no variable in common; any of these triggers would lead SMT to match all functions with all sets of functions mentioned in the problem.

Our solution for generating $\mathsf{in}(f, \mathsf{arrow}(a,b))$ comes in the form of a new axiom:

$$\begin{aligned}\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, b^\iota : \{&\mathsf{fcn}_F(a, c_1, \ldots, c_n), \mathsf{arrow}(a,b)\} \\ &(\forall x^\iota : \mathsf{in}(x,a) \Rightarrow \mathsf{in}(F(x, c_1, \ldots, c_n), b)) \Rightarrow \mathsf{in}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), \mathsf{arrow}(a,b))\end{aligned} \tag{FcnTyping}$$

To clarify using TLA$^+$'s notations, this axioms allows us to prove $[x \in a \mapsto F(x)] \in [a \to b]$ when the condition $\forall x : x \in a \Rightarrow F(x) \in b$ is verified. It complements the introduction axiom for arrow by taking advantage of the fact that explicit functions with domain a are good candidates for the elements of a set $[a \to b]$.

The axiom (FcnTyping) exemplifies another recurrent pattern in our axiomatization, which is

$$\forall \vec{a}^\iota, \vec{b}^\iota, \vec{c}^\iota : \phi(\vec{a}, \vec{b}, \vec{c}) \Rightarrow \mathsf{in}(C(\vec{a}, \vec{b}), D(\vec{a}, \vec{c})) \tag{C-Typing}$$

where C is a constructor, D a set constructor, and ϕ a formula. Typing axioms are pervasive in the theories of tuples, records and sequences. When $\vec{a}$ is nonempty, the trigger

$$\Big\{C(\vec{a}, \vec{b}), D(\vec{a}, \vec{c})\Big\}$$

is a good option: the fact that one variable of the list $\vec{a}$ must be present in the two terms matching the pattern ensures less irrelevant instances are considered. When $\vec{a}$ is empty, this is not the case anymore, so we select triggers differently. Typically, the triggers selected mention $C(\vec{b})$ and some subterm of ϕ containing $D(\vec{c})$ and a variable of $\vec{b}$. Some good examples of this are found in the theory of sequences. A set of sequences is noted $Seq(a)$ and the concatenation of two sequences s and t is noted $Cat(s,t)$. The typing axiom for Cat is

$$\begin{aligned}\forall a^\iota, s^\iota, t^\iota : &\{\mathsf{in}(s, Seq(a)), Cat(s,t)\} \\ &\{\mathsf{in}(t, Seq(a)), Cat(s,t)\} \\ &\mathsf{in}(s, Seq(a)) \wedge \mathsf{in}(t, Seq(a)) \Rightarrow \mathsf{in}(Cat(s,t), Seq(a))\end{aligned} \tag{CatTyping}$$

For all $P : \iota^{n+1} \to o$,

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{setst}_P(a, c_1, \ldots, c_n))\} \quad \{\mathsf{in}(x, a), \mathsf{setst}_P(a, c_1, \ldots, c_n)\}$$
$$\mathsf{in}(x, \mathsf{setst}_P(a, c_1, \ldots, c_n)) \Leftrightarrow \mathsf{in}(x, a) \wedge P(x, c_1, \ldots, c_n) \qquad \text{(SetstDef)}$$

Figure 5.4: Axiom Schema for Set Comprehension

5.4 Other Topics in the SMT Axiomatization

Some fragments of the axiomatization are handled in a particular way. This section provides more details on the topics of axiom schemas, integer arithmetic and set extensionality. Most of these extensions involve simple transformations on proof obligations, so we will indicate how they are implemented.

5.4.1 Axiom Schemas

The following primitive operators of TLA$^+$ are specified by one or several axiom schemas: choose, setst, setof and fcn. For the encoding into HOL, we could simply declare the second-order operator and the full schema as a second-order axiom, for instance:

$$\forall P^{\iota \to o}, a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setst}(a, P)) \Leftrightarrow \mathsf{in}(x, a) \Rightarrow P(x)$$

The principle is the same for each operator, so we will focus on setst for our examples.

A reduction to FOL is achieved by replacing every occurrence $\mathsf{setst}(a, P)$ by a first-order term $\mathsf{setst}_P(a, c_1, \ldots, c_n)$, where setst_P is a new symbol and $c_1, \ldots, c_n$ are parameters that encode the context in which the original term occurs. The operator setst_P is specified by the particular instance of the set comprehension schema for P. See Figure 5.4 for the first-order axiom schema used by the SMT encoding, triggers included.

Example 5.4.1. Consider the TLA$^+$ formula

$$\exists n : \{p \in Int : p \neq n\} \neq Int$$

In HOL, the instance of set comprehension above is written $\mathsf{setst}(Int, \lambda p : p \neq n)$. The higher-order argument is $\lambda p : p \neq n$. It includes the variable n, which is bound by a quantifier in the formula. When writing the relevant instance of the comprehension schema, we cannot reference n, because it is bound locally. Therefore, we will use a parameter to pass n as an argument in the encoded term.

We declare $\mathsf{setst}_P : \iota \times \iota \to \iota$ and specify it with the axiom

$$\forall n^\iota, a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setst}_P(a, n)) \Leftrightarrow \mathsf{in}(x, a) \wedge x \neq n$$

The original formula is encoded as

$$\exists n^\iota : \mathsf{setst}_P(Int, n) \neq Int$$

In general, we must at least make parameters $c_1, \ldots, c_n$ for the free variables of P that are bound by a quantifier. However, implementing this naively, we will find that many easy proof obligations have become much harder to solve. Consider the formula

$$\exists n : \{p \in \mathit{Int} : p \neq n\} = \{p \in \mathit{Int} : p \neq 1 + 1\}$$

The set on the left is the same as in example 5.4.1. The set on the right has a different higher-order argument Q, which does not include the variable n. Applying the procedure we just described results in the formula

$$\exists n^{\iota} : \mathsf{setst}_P(\mathit{Int}, n) = \mathsf{setst}_Q(\mathit{Int})$$

which is still provable, but only by invoking set extensionality, whereas the original obligation seems trivial by instantiating $\exists n$ with $1 + 1$.

To fix this particular example, we can reuse the symbol setst_P to encode the other set as $\mathsf{setst}_P(\mathit{Int}, 1 + 1)$. In general, when treating an occurrence $\mathsf{setst}(a, \lambda x : e)$, we can anticipate reuses by selecting the most general lambda-expression P such that $\lambda x : e$ is an instance of P. For $\lambda p : p \neq 1 + 1$, we can choose $\lambda p : p \neq x$ where x is any variable other than p. The same lambda-expression works for $\lambda p : p \neq n$, which shows the two lambda-expressions have the same "shape". Let us now make this idea more formal.

Definition 5.4.2. An instance of a lambda-expression f is any lambda-expression $f\sigma$ for a substitution σ. The *matrix* of a lambda-expression f is an m such that f is an instance of m, and if f is an instance of another m' then m' is an instance of m.

Notice that it would not be very useful to generalize the definition above to expressions. The matrix of any e would just be x for the substitution $\{x \mapsto e\}$. For a lambda-expression $\lambda x_1, \ldots, x_n : e$, the matrix has to be a lambda-expression $\lambda x_1, \ldots, x_n : e'$ such that $e = e'\sigma$ and no x_i is in the domain or the range of σ. This follows from the definition and the fact that we assume σ to be compatible with the lambda-expression.

Proposition 5.4.3. *The matrix m of a lambda-expression f is unique up to renaming of free variables. Furthermore, every free variable of m has exactly one occurrence.*

Proof. Suppose m_1 and m_2 are two matrices of f. By definition, m_1 and m_2 must be instances of each other. Let σ_1 and σ_2 such that $m_1 = m_2\sigma_2$ and $m_2 = m_1\sigma_1$. We have $m_1 = m_1\sigma_1\sigma_2$. The composite substitution $\sigma_1\sigma_2$ is the identity, so clearly σ_1 and σ_2 are just renamings of free variables.

If m has a free variable x that occurs more than once, then m cannot be a matrix, because it is easy to construct a m' such that m is an instance of m' but not the converse. Simply define m' by replacing every occurrence of x in m by a distinct variable y_i. Then m is an instance of m' for $\{y_1 \mapsto x, y_2 \mapsto x, \ldots\}$. □

It remains to prove the existence of matrices for all lambda-expressions. Consider the algorithm presented in Figure 5.5. If σ and ρ are substitutions with disjoint domains then $\sigma \uplus \rho$ is their union. $\emptyset$ also denotes the unique substitution whose domain is empty. To simplify the presentation, we omit all sort annotations. The input expression e is assumed to be without second-order applications, because during preprocessing all second-order applications are rewritten from the bottom up.

$$
\begin{aligned}
\mathit{abstract}(e, X, \sigma) &:= \begin{cases} (v, \sigma \uplus \{v \mapsto e\}) & \text{if } \mathrm{FV}(e) \cap X = \emptyset \text{ with } v \notin X \cup \mathrm{Dom}(\sigma) \\ \mathit{visit}(e, X, \sigma) & \text{otherwise} \end{cases} \\
\mathit{visit}(x, X, \sigma) &:= (x, \sigma) \\
\mathit{visit}(K(e_1, \ldots, e_n), X, \sigma) &:= \text{let } \sigma_0 := \sigma \\
&\quad \text{and } (e'_i, \sigma_i) := \mathit{abstract}(e_i, X, \sigma_{i-1}) \text{ for } 1 \leq i \leq n \text{ in} \\
&\quad (K(e'_1, \ldots, e'_n), \sigma_n) \\
\mathit{visit}(e_1 = e_2, X, \sigma) &:= \text{let } (e'_1, \sigma_1) := \mathit{abstract}(e_1, X, \sigma) \text{ in} \\
&\quad \text{let } (e'_2, \sigma_2) := \mathit{abstract}(e_2, X, \sigma_1) \text{ in} \\
&\quad (e'_1 = e'_2, \sigma_2) \\
\mathit{visit}(e_1 \Rightarrow e_2, X, \sigma) &:= \text{let } (e'_1, \sigma_1) := \mathit{abstract}(e_1, X, \sigma) \text{ in} \\
&\quad \text{let } (e'_2, \sigma_2) := \mathit{abstract}(e_2, X, \sigma_1) \text{ in} \\
&\quad (e'_1 \Rightarrow e'_2, \sigma_2) \\
\mathit{visit}(\forall x : e, X, \sigma) &:= \text{let } (e', \sigma_1) := \mathit{abstract}(e, X \cup \{x\}, \sigma) \text{ in} \\
&\quad (\forall x : e', \sigma_1) \\
\mathit{matrix}(e, X) &::= \mathit{abstract}(e, X, \emptyset)
\end{aligned}
$$

Figure 5.5: Matrix Algorithm

We admit the algorithm is correct in the following sense: if $(e', \sigma) = \mathit{matrix}(e, \{x_1, \ldots, x_n\})$ then $\lambda x_1, \ldots, x_n : e'$ is the matrix of $\lambda x_1, \ldots, x_n : e$ for the substitution σ. The idea of the algorithm is to build σ incrementally while parsing the input expression from the top down. When a subterm does not contain any variable from X, it is replaced by a fresh variable and σ is updated.

We now describe how the implementation of the axiomatization phase must be modified. During this phase, the proof obligation $\Gamma \vdash e$ is parsed; for all primitive operators encountered, a declaration and axioms are inserted in the sequent. The final sequent has the form $\Delta, \Gamma \vdash e$ where Δ contains the additional declarations and axioms. We focus now on the case where a second-order application is encountered:

$$K(e_1, \ldots, \lambda x_1, \ldots, x_n : e, \ldots, e_n)$$

If the higher-order argument is a symbol F instead, we treat it as $\lambda x_1, \ldots, x_n : F(x_1, \ldots, x_n)$, where n is the arity of F. Let (e', σ) be the result of the call $\mathit{matrix}(e, \{x_1, \ldots, x_n\})$. Then the original application is replaced by

$$K_{e'}(e_1, \ldots, \sigma(v_1), \ldots, \sigma(v_m), \ldots, e_n)$$

where $\{v_1, \ldots, v_m\}$ is the domain of σ. The operator $K_{e'}$ may have been declared already, in which case we resume parsing the sequent. Otherwise, we need to make a declaration for $K_{e'}$ and insert its axioms, if there are any.

An axiom schema for K is a formula ϕ parameterized by an operator F such that ϕ contains at least one occurrence of K, and all applications of K are of the form

$$K(e_1, \ldots, F, \ldots e_n)$$

$$\forall z^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z)\} \qquad z = \mathsf{proj}_{\mathsf{int}}(\mathsf{cast}_{\mathsf{int}}(z)) \qquad \text{(CastIntInjective)}$$

$$\forall z^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z)\} \qquad \mathsf{in}(\mathsf{cast}_{\mathsf{int}}(z), \mathsf{Int}) \qquad \text{(IntIntro)}$$

$$\forall x^{\iota} : \{\mathsf{in}(x, \mathsf{Int})\} \qquad \mathsf{in}(x, \mathsf{Int}) \Rightarrow x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)) \qquad \text{(IntElim)}$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$$
$$\mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 +_{\mathsf{int}} z_2) \qquad \text{(PlusTyping)}$$

Figure 5.6: Axioms for Integer Arithmetic (Addition Only)

The first-order instance of ϕ for $K_{e'}$ is the formula $\forall v_1, \ldots, v_m : \phi'$ where ϕ' is obtained from ϕ by replacing all applications of K like the one above by

$$K_{e'}(e_1, \ldots, v_1, \ldots, v_m, \ldots, e_n)$$

The top context Δ is updated with the declaration of $K_{e'}$ (its type is easy to infer from the type of K and the sorts of the variables in $\mathrm{Dom}(\sigma)$) and the appropriate instances for all axiom schemas associated with K.

5.4.2 Integer Arithmetic

Reasoning modulo theory is a strength of SMT. We want to leverage SMT's native support for integer arithmetic to handle the TLA$^+$ obligations that involve this theory. Our encoding so far has only dealt with one sort ι besides the Boolean sort, so no term is encoded into int. We present now an extension of the axiomatic theory that effectively links SMT's and TLA$^+$'s arithmetics.

SMT's integer arithmetic is characterized by a sort int, several primitive operators like addition $+_{\mathsf{int}} : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$, and a specific interpretation of int and the primitive operators on it. TLA$^+$, in comparison, defines integers as the elements of the set *Int* and specifies operators like $+$ axiomatically. Despite the differences between the two languages, the idea is still to define a homomorphism between their respective arithmetical structures. Concretely, we declare a new operator $\mathsf{cast}_{\mathsf{int}} : \mathsf{int} \to \iota$ that is injective and ranges over the elements of *Int*. We then introduce axioms to link the operators of TLA$^+$ with their counterparts in SMT if they have one.

The axioms are displayed in Figure 5.6. Besides $\mathsf{cast}_{\mathsf{int}}$, we provide the new operator $\mathsf{proj}_{\mathsf{int}} : \iota \to \mathsf{int}$. $\mathsf{cast}_{\mathsf{int}}$ is specified as a surjective operator from ι to the subdomain of ι characterized by the predicate $x \in \mathit{Int}$ (axioms (IntIntro) and (IntElim)). It is specified as injective by (CastIntInjective). Indeed, if $\mathsf{cast}_{\mathsf{int}}(m) = \mathsf{cast}_{\mathsf{int}}(n)$ for some $m, n : \mathsf{int}$, then $m = \mathsf{proj}_{\mathsf{int}}(\mathsf{cast}_{\mathsf{int}}(m)) = \mathsf{proj}_{\mathsf{int}}(\mathsf{cast}_{\mathsf{int}}(n)) = n$. Compare (CastIntInjective) with the more natural expression of injectivity with its obvious trigger:

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)\} \quad \mathsf{cast}_{\mathsf{int}}(z_1) = \mathsf{cast}_{\mathsf{int}}(z_2) \Rightarrow z_1 = z_2$$

Suppose there are N known terms $\mathsf{cast}_{\mathsf{int}}(n)$ at a given time while SMT solves an obligation. The axiom above will generate N^2 different instances—one per couple of terms. The axiom (CastIntInjective) will only generate N.

The axiom (PlusTyping) establishes the homomorphic relation between TLA$^+$'s addition (which is noted plus) and SMT's $+_{\mathsf{int}}$. It is easily generalized to the other operators of TLA$^+$ for integer arithmetic: unary and binary substraction, multiplication, quotient and remainder, and the comparison relation. We could apply the same principle to numerical constants, which would lead to trivial axioms such as

$$1 = \mathsf{cast}_{\mathsf{int}}(1_{\mathsf{int}})$$

where 1 is the TLA$^+$ constant and 1_{int} the interpreted SMT constant. It is much simpler to just rewrite 1 as $\mathsf{cast}_{\mathsf{int}}(1_{\mathsf{int}})$ in proof obligations, so that is what we implement during the axiomatization phase. For quotient and remainder, we must restrict the axiom to positive integers for the second argument, as the operators are unspecified in TLA$^+$ otherwise:

$$\begin{aligned}&\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{div}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\} \\ &\quad z_2 >_{\mathsf{int}} 0_{\mathsf{int}} \Rightarrow \mathsf{div}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 \div_{\mathsf{int}} z_2)\end{aligned}$$

Without the condition $z_2 >_{\mathsf{int}} 0_{\mathsf{int}}$, we would be able to prove $1 \div 0 \in \mathit{Int}$ and $1 \div (-1) \in \mathit{Int}$, which are both unprovable in TLA$^+$.

Theorem 5.4.4. *The extension of the axiomatic theory for including SMT's arithmetic is sound.*

Proof. The axioms (CastIntInjective), (IntIntro) and (IntElim) essentially declare $\mathsf{cast}_{\mathsf{int}}$ as a bijection between the domain of int and the elements of TLA$^+$'s domain that are members of *Int*. The two collections have the same cardinality so there must be such a bijection. The axiom (PlusTyping) essentially states that TLA$^+$'s interpretation of plus on the elements of *Int* coincides with SMT's interpretation of $+_{\mathsf{int}}$, which we assume to be true. □

Example 5.4.5. To see how the axioms work in practice, we demonstrate with the theorem

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

We assume a constant $x : \iota$ is declared and derive a contradiction from the assumptions

$$\mathsf{in}(x, \mathit{Int}) \tag{H}$$
$$\mathsf{plus}(x, \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}})) \neq x \tag{G}$$

Here is the sequence of relevant instances that are generated from this starting set:

$$\mathsf{in}(x, \mathit{Int}) \Rightarrow x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)) \qquad \text{by (IntElim)} \qquad \text{(F1)}$$
$$\mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)), \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}})) = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x) +_{\mathsf{int}} 0_{\mathsf{int}}) \qquad \text{by (PlusTyping)} \qquad \text{(F2)}$$

The fact (F1) is the instance of (IntElim) triggered by the term $\mathsf{in}(x, \mathit{Int})$. After the insertion of (F1), the equality $x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x))$ can be derived by ground reasoning, so the subterm $\mathsf{plus}(x, \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}}))$ from (G) becomes a match for the axiom (PlusTyping), resulting in the new fact (F2). The final problem is contradictory, as shown by the following equalities:

$$\begin{aligned}\mathsf{plus}(x, \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}})) &= \mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)), \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}})) \\ &= \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x) +_{\mathsf{int}} 0_{\mathsf{int}}) && \text{by (F2)} \\ &= \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)) && \text{by SMT's arithmetic} \\ &= x\end{aligned}$$

But $\mathsf{plus}(x, \mathsf{cast}_{\mathsf{int}}(0_{\mathsf{int}})) \neq x$ by (G).

$\forall x^\iota, y^\iota : \{\mathsf{invokeSetExt}(x, y)\}$	$(\forall z^\iota : \mathsf{in}(z, x) \Leftrightarrow \mathsf{in}(z, y)) \Rightarrow x = y$	(SetExt)
$\forall x^\iota, y^\iota : \{\mathsf{setEquals}(x, y)\}$	$\mathsf{setEquals}(x, y) \Leftrightarrow x = y$	(SetEqualsDef)
$\forall x^\iota, y^\iota : \{\mathsf{setEquals}(x, y)\}$	$\mathsf{invokeSetExt}(x, y)$	(InvokeSetExt)

Figure 5.7: Set Extensionality Axioms for SMT

5.4.3 Set Extensionality

Some obligations cannot be solved without the axiom of set extensionality:

$$\forall x, y : (\forall z : z \in x \Leftrightarrow z \in y) \Rightarrow x = y$$

The axiom applies to every object in the universe of set theory. We know from experience that SMT solvers are unable to infer the relevant instances of set extensionality without triggers; but there is no obvious trigger to choose for the axiom. However, in many cases, the relevant equality to prove by extensionality occurs explicitly in the obligation. For example:

$$\begin{aligned} &\textsc{assume}\ \ \textsc{new}\ S,\ \textsc{new}\ T \\ &\textsc{prove}\ \ \ S \cup T = T \cup S \end{aligned}$$

A less trivial example may be found at the end of the Paxos proof in the distribution of TLAPS (we modified the names and removed irrelevant facts):

$$\begin{aligned} &\textsc{assume}\ \ \textsc{new}\ S,\ \textsc{new}\ T, \\ &\qquad\qquad S \subseteq T \wedge S \neq T, \\ &\qquad\qquad \forall x, y \in T : x = y \\ &\textsc{prove}\ \ \ S = \emptyset \end{aligned}$$

Here is the intuitive proof: T is either empty or a singleton set; S is a proper subset of T, so it can only be empty. Set extensionality must be invoked twice to prove the goal: one time to use $S \neq T$, the other to prove $S = \emptyset$.

The idea then is to let equalities found in the obligation trigger the axiom of set extensionality. This cannot be implemented in the form of a trigger $\{x = y\}$, because the symbol $=$ is interpreted by SMT. The solution is the set of axioms of Figure 5.7. Two new binary predicates $\mathsf{invokeSetExt}$ and $\mathsf{setEquals}$ are introduced. The axiom (SetEqualsDef) specifies $\mathsf{setEquals}$ as an alias for $=$. The operator $\mathsf{invokeSetExt}$ is not specified by anything; it is just used to control the instantiation of (SetExt). The axiom (InvokeSetExt) is a rule: it is triggered when an equality $\mathsf{setEquals}(x, y)$ is found; its only effect is to introduce $\mathsf{invokeSetExt}(x, y)$ into the ground problem, triggering (SetExt) right after.

To make this solution work, relevant equalities in the PO must be rewritten:

$$x = y \ \triangleright\ \mathsf{setEquals}(x, y) \qquad \text{in } \textit{positive contexts} \text{ when } x \text{ or } y \text{ is a } \textit{set}$$

The two conditions are heuristics to aim for more relevant instances of set extensionality. A *polarity* is assigned to every subexpression in the standard way: the top expression is positive;

the polarity is reversed when going down a negation, or left of an implication, and for the hypotheses of a sequent. A subexpression is in a positive context if its polarity is positive. Here is an example of an expression with an equality in a negative context:

$$\forall x, y : x = \emptyset \Rightarrow y \notin x$$

The subexpression $x = \emptyset$ will not be rewritten; there is no need to in order to prove that the expression is valid. The second condition is related to typing: it would be ill-advised to try solving, for example, the goal $1 + 1 = 2$ by set extensionality. In the absence of a type inference mechanism, we simply consider a *set* to be a term whose top connective is set-theoretic, like $\cup$ or UNION for instance.

5.5 Evaluation

We implemented the new SMT encoding in TLAPS. It can be used in place of the original encoding. The original encoding can still be used by passing the argument `--debug oldsmt` to TLAPS. We present now an evaluation designed to compare the performances of the two versions of the SMT encoding.

5.5.1 Methodology

We adapt the methodology of Section 4.7 to evaluate the different encodings for the SMT backend. Our set of TLA$^+$ specifications is the same as before. TLAPS is invoked to encode each obligation three times: one version is for our encoding, the other two correspond to the T_0 and T_1 type systems of the original encoding. There is a third option T_2 for the original encoding, but during our evaluation we found that the implementation of this type system is unsound in a way that is too difficult to fix. The error would cause type-checking conditions such as $x \in$ DOMAIN f to be ignored, making it very easy to prove invalid statements.

The three encodings lead to three different SMT benchmarks. Several SMT solvers are tested on the benchmarks: CVC4, cvc5, veriT and Z3. The timeout for solving is set to 5 seconds for all solvers (this is the default timeout in TLAPS). For veriT, the mention of SMT's logic UFNIA is replaced with UFLIA, because veriT does not support non-linear integer arithmetic. To obtain the results of the original encoding, we merged the results obtained from T_0 and T_1. An obligation is considered solved using the original encoding if it is solved using either T_0 and T_1.

By default, SMT solvers use our custom patterns, but this default behavior can be overriden by an option for each solver. To confirm the impact of our triggers, we tested every solver with the default behavior and with an option to ignore user patterns, for every obligation.

5.5.2 Results and Discussion

The results are presented in Tables 5.1, 5.2 and 5.3. Table 5.1 compares performances when using the original SMT encoding or our new version. We report the number of obligations solved using each version (top numbers) and the number of obligations *uniquely* solved using each version (bottom numbers). Table 5.2 is read similarly. It focuses on our new encoding and compares the results of three solvers—the results of CVC4 and cvc5 have been merged because of the substantial overlap between them. Table 5.3 looks at solvers individually to measure the impact of our custom patterns on performances. The top left numbers indicate

Specification	Size (# POs)	Encoding	
		Original	New
TLA$^+$ Distribution	1996	1688 56	1821 189
TLAPS Distribution	1402	1281 44	1333 96
Deconstructed Bakery	1214	1030 26	1188 184
Total	4612	3999 126	4342 469

Table 5.1: Comparison with the Original SMT Encoding

Specification	Size (# POs)	Solver		
		Z3	CVC4-5	veriT
TLA$^+$ Distribution	1996	1754 53	1762 67	995 0
TLAPS Distribution	1402	1278 52	1273 53	854 0
Deconstructed Bakery	1214	1104 67	1121 84	236 0
Total	4612	4136 172	4156 204	2085 0

Table 5.2: Solvers' Performances with the New Encoding

Specification	Size (# POs)	Solver									
		Z3		CVC4		cvc5		veriT		**portfolio**	
TLA$^+$ Dist.	1996	1516 21	1754 259	1695 14	1750 69	1686 22	1722 58	748 18	995 265	1745 4	1821 80
TLAPS Dist.	1402	1120 12	1278 170	1155 4	1256 105	1156 5	1240 89	510 3	854 347	1232 0	1333 101
D. Bakery	1214	927 17	1104 194	1027 48	1109 130	1036 42	1116 122	199 5	236 42	1093 2	1188 97
Total	4612	3563 50	4136 623	3877 66	4115 304	3878 69	4078 269	1457 26	2085 654	4070 6	4342 278

Table 5.3: Impact of User Patterns (Triggers)

how many obligations where solved without our triggers, the top right numbers indicate how many obligations where solved with our triggers. The bottom numbers are the numbers of obligations *uniquely* solved, for that solver. We also report the performances of a *virtual best solver* in the last column "portfolio", whose results are obtained by merging the results of all solvers: an obligation is solved in portfolio mode if it solved by one solver at least.

As Table 5.1 shows the original encoding solves 3999 obligations in total while ours solves 4342—an increase of 8,5%. There are 126 obligations that we do not solve anymore, and 469 that only our version solves. Among these 469 obligations, many are not solved with the original encoding because of a bug in the preprocessing phase: 195 obligations were not successfully encoded with T_0, 432 with T_1.

Table 5.2 details how solvers performed with the new encoding. Among the 4342 obligations solved, 4136 (95%) were solved by Z3, 4156 (95%) were solved by either CVC4 or cvc5, and 2085 (48%) were solved by veriT. The poorer performances of veriT may be due to the limitation of the solver to linear integer arithmetic. There are 172 obligations solved uniquely by Z3, and 204 solved by CVC4 or cvc5 but no other solver. Thus, the majority of TLA$^+$ obligations that can be solved by the SMT backend may be solved by combining Z3 with either CVC4 or cvc5.

If we ask solvers to ignore our triggers (user patterns), the total number of obligations solved is 4070, as reported by Table 5.3. Thus, our triggers represent an increase of 6,6% in terms of obligations solved, compared to the triggers automatically generated by the solvers. Moreover, disabling user patterns results in only 6 additional obligations to be solved. All solvers are positively impacted by enabling our triggers, but some more than others. Z3 and veriT profit the most from our triggers, with 623 and 654 obligations uniquely solved respectively. The impact is not as significant for CVC4 and cvc5, with 304 and 269 obligations solved uniquely respectively.

In summary, there is good evidence that our SMT encoding, which is an almost direct translation of TLA$^+$ completed by custom triggers, improves the SMT backend of TLAPS. This conclusion must be nuanced by the fact that 126 obligations are not solved anymore by our version. We believe our axiomatization could be improved, for example by selecting better triggers for the instances of axiom schemas.

Chapter 6

Conclusion

TLAPS is used to ensure the soundness of computer systems through theorem proving. While efficient automation is important, it is critical that TLAPS can be ensured sound as well. To this end, we defined and implemented a *direct encoding* of TLA$^+$ for automated theorem provers. The direct encoding preserves most of the source obligations' structures and is easy to prove sound. We extended the direct encoding in a minimal way to implement a new *SMT encoding*. We optimized this encoding with custom E-matching patterns, resulting in a new SMT backend that is not only safer, but also as efficient as the one previously implemented.

New Encodings of TLA$^+$

Our translations are based on a formalization of TLA$^+$ as a particular theory (collection of symbols and axioms) on top of a first-order logic. We adopted this formalism to capture the unique semantics of TLA$^+$, which is underspecified. TLA$^+$ features all the usual *logical connectives* from propositional and first-order logic, but it extends their traditional semantics to account for the new combinations the language permits. For instance, the expression $2 \land 5$ is legitimate, and this particular example is a refutable formula (if 2 and 5 are both true then they must be equal). The *primitive operators* of TLA$^+$, which include symbols from set theory, arithmetic, and symbols for functions, may also be combined in unconventional ways. For instance, $2 \cup 5$ is a legitimate expression, and while it seems nonsensical, we may derive some facts about it, like the fact that it equals $5 \cup 2$. Our formalization does not assign interpretations to the primitive operators of TLA$^+$; it only provides axioms.

This formalization of TLA$^+$ naturally leads to an encoding in two steps: first, "type errors" resulting from unconventional uses of logical connectives must be corrected, so that the usual semantics of first-order logic is recovered; second, the relevant operators must be declared as uninterpreted symbols and their axioms must be included in the problem. Regarding the first step, a similar transformation was defined for the original version of the SMT encoding, only it was not total. For instance, with the original backend, the formula $42 \Rightarrow 42$ is not supported, even though it is both legitimate and valid according to TLA$^+$'s semantics. We defined this step and proved that it results in equisatisfiable expressions of regular first-order logic. As for the second step, it is straightforward to implement. Only the operators occurring in the obligation are declared, along with the axioms we have attached to them.

Encoded problems can be directly translated to higher-order logic. We implemented a translation to the THF dialect of the TPTP standard, and added the superposition prover

Zipperposition as a backend for TLAPS. This new backend is especially useful for TLA$^+$ obligations that require higher-order reasoning. These obligations are not very common, but they are inevitable in certain situations: proofs by induction on *Nat* or other structures, or proofs involving user-defined recursive operators. Support for these obligations was lacking before the addition of Zipperposition, as only the Isabelle backend could attempt to solve them using basic automation tactics. Our evaluation shows Zipperposition and Isabelle have complementary strengths. Zipperposition is limited by the fact that it does not support interpreted arithmetic (for the version supporting HOL), but for problems involving higher-order unification, it can cope with harder problems than Isabelle.

The SMT encoding is based on the architecture of the direct encoding. Second-order constructs such as set comprehension or explicit functions are reduced to first-order applications, and the appropriate instances of their axiom schemas are generated in the process. Integer arithmetic is supported by a few special axioms that specify a correspondence between TLA$^+$'s arithmetical operators and SMT's interpreted symbols. Set extensionality is difficult to support, but the instances of set extensionality needed for most TLA$^+$ obligations fall into a few predictable patterns; we use some special axioms to generate these instances. The most important feature of our SMT encoding is the axiomatization, which is fully optimized with E-matching patterns (triggers). We carefully selected those patterns by inspecting difficult proof obligations and refining our axioms through an empirical process. Our method relies on heuristics, but the strategy can be summarized in a few principles. One important principle is to watch for the kind of terms that specific patterns may produce when they are triggered: for instance, we avoid producing terms that introduce new sets, but we encourage the production of new facts $x \in S$ for known x and S.

The resulting encoding is still very efficient despite the simple translation it is based on. Our evaluation shows that most obligations that were previously solved with the previous SMT backend are still solved with our version using a combination of CVC4 and Z3. There are also obligations that our version solves uniquely. Typically, the new version is more likely to solve *typing theorems* with less assistance from the user than before. These theorems usually consist of elementary verifications about TLA$^+$'s set theory, but for large specifications there are many verifications to perform, and several cases for the different possible transitions of the system. With the previous backend, it is often the case that at least one case of the lemma is too difficult to prove, so this particular case must be detailed or proved with Zenon. With our backend, it is more frequent that the whole lemma can be solved in a single step.

Perspectives

Our work on translating TLA$^+$ for ATPs can be continued in many different ways. We outline some ideas below. The first two are simple ways to improve the SMT encoding that we described in this work. The remaining ideas are more ambitious, and involve revisiting the type synthesis mechanism of Vanzetto's original work.

Verification of the Axioms. The direct encoding is based on a modular architecture with only two nontrivial components. We presented the arguments for why our translation is sound, so the next step would be to verify it formally. In particular, one should verify the axioms used by the SMT encoding and documented in Appendix C. All axioms are either simple consequences of TLA$^+$'s theory, or special axioms that extend the theory conservatively, like our axioms for leveraging SMT's arithmetic. This verification supposes a formalization

of the logic $\mathcal{L}^s$ (which is essentially MS-FOL with second-order applications). Ideally, the implementation of TLAPS should be linked to the verified axiomatization, to eliminate the risk of introducing errors when copying axioms into the code.

Extension of the Axiomatization. There are some fragments of TLA$^+$'s theory that our implementation does not cover. We could extend the axiomatization to support (for example) the theory of finite sets, or real arithmetic, or bags. The reals can be supported in the same way as the integers in our framework, by specifying a few special axioms to link TLA$^+$'s operators to SMT's interpreted symbols. The theory of finite sets in TLA$^+$ revolves around two operators *IsFiniteSet* and *Card*. It does not seem difficult to apply our strategy for assigning triggers to the axioms that specify finite sets. A more ambitious goal would be to implement our strategy as a general procedure for generating patterns automatically. This could lead to a better treatment of the axiom schemas of TLA$^+$ for which there is no obvious pattern (in particular the axiom schema specifying CHOOSE) and the user lemmas involved in proof obligations.

Typed Encodings. One important shortcoming of the original SMT encoding is that most type information is actually discarded during the translation. Type synthesis assigns various types to expressions, including the sorts ι, int, but also types for sets, functions, tuples, records, etc. However, these types are merely used to optimize the preprocessing phase. All expressions are encoded as terms of the sort ι in the final problem, with the exception of the sort int which is preserved. Thus the use of various typed encodings preserving constructed types in their outputs have not been explored yet. Two applications in particular seem more promising. First, SMT solvers may benefit from refined sort annotations to quantifiers and symbols, if enumerative instantiation techniques must be used. Second, the type annotations of the system T_1 may be exploited to implement the so-called *sets-as-predicates encoding*, which encodes set membership through the functional application of HOL. Types are necessary for this encoding, because the legitimate TLA$^+$ expression $x \in x$ cannot be encoded as the application of x to itself in HOL.

Soft Typing for TLA$^+$. The lack of a traditional type system allows for more creativity in formalizations and great ease of use. However, it bears some negative consequences on the experience of interactive proof with TLAPS. Users of TLA$^+$ are accustomed to the notion of proving typing lemmas of the form $Spec \Rightarrow \Box TypingOK$. In our experience with the prover, one should also prove typing lemmas for each user-defined operator, otherwise it becomes difficult to manage definitions as the proofs get larger. For instance, suppose a specification defines the composition of two functions:

$$g \circ f \triangleq [x \in \text{DOMAIN } f : g[f[x]]]$$

For each step of a proof, the user can choose to either expand every occurrence of $\circ$ into its definition, or leave the operator completely opaque. However, there are many kinds of situations in which the full definition of $\circ$ is not relevant, but some basic information about the symbol is necessary. That basic information is precisely the kind of information that is normally captured by typing. Thus it is often in the interest of the user that they prove the following lemma and make sure it is always visible in proofs:

$$\begin{array}{lll} \text{THEOREM} & \text{ASSUME} & \text{NEW } A,\ \text{NEW } B,\ \text{NEW } C, \\ & & \text{NEW } f \in [A \to B]\,,\ \text{NEW } g \in [B \to C] \\ & \text{PROVE} & (g \circ f) \in [A \to C] \end{array}$$

We want to suggest that the derivation and management of such lemmas could be further supported by automation. This could take the form of a subsystem of TLAPS that automatically infers *typing lemmas* from definitions, searches for *typing hypotheses* to infer the types of quantified variables in the manner of the T_1 type system, and performs *type inference* over whole TLA$^+$ specifications. Besides the interest for definition management, the system could be used to provide immediate feedback in the form of warnings to report type errors.

Bibliography

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[3] Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 273–333. Elsevier and MIT Press, 2001.

[4] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 19–99. Elsevier and MIT Press, 2001.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[7] Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. Superposition with lambdas. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2019.

[8] Nikolaj S. BjØrner and Mark C. Pichora. Deciding fixed and non-fixed size bit-vectors. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 376–392, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[9] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reason.*, 51(1):109–128, 2013.

[10] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[11] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[12] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[13] Samuel R. Buss. On Herbrand's theorem. In Daniel Leivant, editor, *Logical and Computational Complexity. Selected Papers. Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, Indiana, USA, 13-16 October 1994*, volume 960 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 1994.

[14] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[15] George E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[16] Alessio Coltellacci. Reconstruction of TLAPS proofs solved by VeriT in Lambdapi. In *Rigorous State-Based Methods: 9th International Conference, ABZ 2023, Nancy, France, May 30–June 2, 2023, Proceedings*, page 375–377, Berlin, Heidelberg, 2023. Springer-Verlag.

[17] The Coq Development Team. The Coq Proof Assistant, version 8.7.1, December 2017. Official Release.

[18] T. Coquand. Metamathematical investigations of a calculus of constructions. Technical Report RR-1088, INRIA, September 1989.

[19] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ Proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *18th International Symposium On Formal Methods - FM 2012*, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154, Paris, France, August 2012. Springer. The original publication is available at www.springerlink.com.

[20] Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings*, volume 10483 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2017.

[21] Nigel Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[22] Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018.

[23] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[26] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 410–425. Springer, 2008.

[27] [Rosalie] Defourné. Improving automation for higher-order proof steps. In Boris Konev and Giles Reger, editors, *Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2021.

[28] Rosalie Defourné. Encoding TLA$^+$ proof obligations safely for SMT. In Uwe Glässer, José Creissac Campos, Dominique Méry, and Philippe A. Palanque, editors, *Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings*, volume 14010 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2023.

[29] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Sci. Comput. Program.*, 94:130–143, 2014.

[30] 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*, volume 20 of *EPiC Series in Computing*, pages 22–31. EasyChair, 2012.

[31] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[32] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[33] Limor Fix. Fifteen years of formal property verification in Intel. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144. Springer, 2008.

[34] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto 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, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[35] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[36] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[37] Paul C. Gilmore. A proof method for quantification theory: Its justification and realization. *IBM J. Res. Dev.*, 4(1):28–35, 1960.

[38] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

[39] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.

[40] Joseph Y. Halpern. Presburger arithmetic with unary predicates is $\prod_1^1$ complete. *J. Symb. Log.*, 56(2):637–642, 1991.

[41] Jacques Herbrand. *Recherches sur la théorie de la démonstration.* PhD thesis, Faculté des Sciences de Paris, 1930.

[42] Gérard P. 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, 1980.

[43] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 203–208. AAAI Press / The MIT Press, 1997.

[44] Cezary Kaliszyk and Karol Pąk. Semantics of mizar as an isabelle object logic. *Journal of Automated Reasoning*, 63(3):557–595, Oct 2019.

[45] S.C. Kleene. *Mathematical Logic*. Dover books on mathematics. Dover Publications, 2002.

[46] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[47] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[48] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[49] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[50] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016.

[51] Stephan Merz. On the logic of TLA+. *Comput. Artif. Intell.*, 22(3-4):351–379, 2003.

[52] Stephan Merz. Formal specification and verification. In Dahlia Malkhi, editor, *Concurrency: the Works of Leslie Lamport*, pages 103–129. ACM, 2019.

[53] Stephan Merz and Hernán Vanzetto. Encoding TLA^+ into unsorted and many-sorted first-order logic. *Sci. Comput. Program.*, 158:3–20, 2018.

[54] Jean-François Monin. *Understanding formal methods*. Springer, 2003.

[55] Michal Moskal. Programming with triggers. *ACM International Conference Proceeding Series*, 01 2009.

[56] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[57] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[58] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

[59] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[60] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[61] Lawrence C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, 1989.

[62] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[63] Andrew Reynolds. Conflicts, models and heuristics for quantifier instantiation in SMT. In Laura Kovács and Andrei Voronkov, editors, *Vampire@IJCAR 2016. Proceedings of the 3rd Vampire Workshop, Coimbra, Portugal, July 2, 2016*, volume 44 of *EPiC Series in Computing*, pages 1–15. EasyChair, 2016.

[64] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2018.

[65] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202. IEEE, 2014.

[66] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[67] John M. Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24(9):709–720, 1998.

[68] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[69] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors,

Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.

[70] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, jul 1978.

[71] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[72] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[73] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.

[74] John Michael Spivey. *The Z notation - a reference manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[75] Ryan Stansifer. Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984.

[76] Martin Suda and Christoph Weidenbach. A PLTL-Prover based on labelled superposition with partial model guidance. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 537–543. Springer, 2012.

[77] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

[78] Geoff Sutcliffe. The 10th IJCAR automated theorem proving system competition - CASC-J10. *AI Commun.*, 34(2):163–177, 2021.

[79] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[80] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[81] Hernán Vanzetto. *Proof automation and type synthesis for set theory in the context of TLA$^+$. (Automatisation de preuves et synthèse de types pour la théorie des ensembles dans le contexte de TLA$^+$).* PhD thesis, University of Lorraine, Nancy, France, 2014.

[82] Petar Vukmirovic, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[83] Petar Vukmirovic and Visa Nummelin. Boolean reasoning in a higher-order superposition prover. In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, editors, *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual)*, volume 2752 of *CEUR Workshop Proceedings*, pages 148–166. CEUR-WS.org, 2020.

[84] Freek Wiedijk. Mizar's soft type system. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 383–399, 2007.

[85] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

Appendix A

The Standard Theory of TLA$^+$

This appendix contains an axiomatization for TLA$^+$. This theory is based on the logic $\mathcal{L}$ that is defined in chapter 3. To sum up briefly: $\mathcal{L}$ is a variant of unsorted FOL in which formulas do not form a distinct syntactic class, and second-order operators are permitted in signatures and can occur in applications.

We have organized the theory in thematic layers: epsilon-calculus, set theory, function theory, arithmetic, special functions (tuples, records and sequences) and finite sets. For each layer, we provide the list of primitive operators and their assigned shapes, followed by a list of axioms. There are several ways to formulate the theory, we have chosen one that highlights the proximity of TLA$^+$ with traditional ZFC. Many axioms are just definitions of primitive constructs from already defined constructs; this choice of presentation makes the question of TLA$^+$'s consistency relative to ZFC easier to apprehend.

The original reference for the standard theory of TLA$^+$ is [48]. While we have adapted the presentation, we believe that our axiomatization is compatible with the reference book as it only differs in superficial ways. The most notable difference is our treatment of functions, as we leave the primitive operator isafcn unspecified. We refer the reader to section 3.3 for a discussion on this topic.

Some axioms are schemas, in which case the parameters will be put in parentheses next to the axiom's name:

EnumDef $(p > 0)$ $\qquad \forall a_1, \ldots, a_p, x : x \in \{a_1, \ldots, a_p\} \Leftrightarrow x = a_1 \vee \cdots \vee x = a_p$

We use the following convention for denoting parameters: the character n will always denote a natural number, p a positive natural number, s a string of characters, and F, P, Q first-order operator symbols whose arity will be obvious in context.

Note that we will use readable notations for primitive constructs when possible. These notations will be indicated when new primitive operators are introduced. Another kind of notation we use is the multiline conjunction (or disjunction):

$$\begin{array}{l} \wedge\, \phi_1 \\ \wedge\, \phi_2 \\ \wedge\, \ldots \\ \wedge\, \phi_n \end{array} \quad \text{means} \quad \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$$

Epsilon-calculus

The epsilon-calculus consists of only one second-order operator and two axioms.

Operator	Shape	Notation
choose	(1)	CHOOSE $x : e \triangleq \mathsf{choose}(\lambda x : e)$

CHOOSEDEF (P)	$\forall x : P(x) \Rightarrow P(\mathsf{choose}(P))$
CHOOSEEXT (P, Q)	$(\forall x : P(x) \Leftrightarrow Q(x)) \Rightarrow \mathsf{choose}(P) = \mathsf{choose}(Q)$

The notation CHOOSE is not used in the axioms above because it is not general enough: CHOOSE $x : P(x)$ represents $\mathsf{choose}(\lambda x : P(x))$ but not $\mathsf{choose}(P)$. Using the extentionality axiom, we have the equality $\mathsf{choose}(\lambda x : P(x)) = \mathsf{choose}(P)$, so may use CHOOSE in the remaining axioms.

The TLA$^+$ constructs for if-then-else- and case-expressions are treated as notations:

$$\begin{aligned}
&\text{CASE } p_1 \to e_1, \ldots, p_n \to e_n \\
&\quad \triangleq \text{CHOOSE } x : \ \lor\ p_1 \land x = e_1 \\
&\qquad\qquad\qquad\ \lor\ p_2 \land x = e_2 \\
&\qquad\qquad\qquad\ \lor\ \cdots \\
&\qquad\qquad\qquad\ \lor\ p_n \land x = e_n
\end{aligned}$$

$$\begin{aligned}
&\text{CASE } p_1 \to e_1, \ldots, p_n \to e_n, \text{OTHER} \to e \\
&\quad \triangleq \text{CASE } p_1 \to e_1, \ldots, p_n \to e_n, (\neg p_1 \land \cdots \land \neg p_n) \to e
\end{aligned}$$

$$\begin{aligned}
&\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2 \\
&\quad \triangleq \text{CASE } p \to e_1, \text{OTHER} \to e_2
\end{aligned}$$

Set Theory

Operator	Shape	Notation
in	$(0,0)$	$e_1 \in e_2 \triangleq \mathsf{in}(e_1, e_2)$
		$e_1 \notin e_2 \triangleq \neg\mathsf{in}(e_1, e_2)$
$\mathsf{subseteq}$	$(0,0)$	$e_1 \subseteq e_2 \triangleq \mathsf{subseteq}(e_1, e_2)$
enum_n	$(0,\ldots,0)^\star$	$\{e_1,\ldots,e_n\} \triangleq \mathsf{enum}_n(e_1,\ldots,e_n)$
union	(0)	$\text{UNION } e \triangleq \mathsf{union}(e)$
subset	(0)	$\text{SUBSET } e \triangleq \mathsf{subset}(e)$
setst	$(0,1)$	$\{x \in e_1 : e_2\} \triangleq \mathsf{setst}(e_1, \lambda x : e_2)$
setof_p	$(0,\ldots,0,p)^{\star\star}$	$\{e_{p+1} : x_1 \in e_1,\ldots,x_p \in e_p\} \triangleq \mathsf{setof}_p(e_1,\ldots,e_p,\lambda x_1,\ldots,x_p : e_{p+1})$
cup	$(0,0)$	$e_1 \cup e_2 \triangleq \mathsf{cup}(e_1, e_2)$
cap	$(0,0)$	$e_1 \cap e_2 \triangleq \mathsf{cap}(e_1, e_2)$
diff	$(0,0)$	$e_1 \setminus e_2 \triangleq \mathsf{diff}(e_1, e_2)$
$\mathsf{boolean}$	$()$	$\text{BOOLEAN} \triangleq \mathsf{boolean}$

($\star$) The operator enum_n expects n arguments; ($\star\star$) the operator setof_p expects p constant arguments followed by one p-ary operator argument.

TLA$^+$'s set theory follows closely the axiomatic theory of ZFC. Set replacement is treated differently through the operator setof_p, but the traditional schema of replacement can be derived from the axioms. The axiom of choice is not included, but it is trivially derived from the epsilon-calculus. The axiom of foundation and the axiom of infinity are included.

SetExt	$\forall x, y : (\forall z : z \in x \Leftrightarrow z \in y) \Rightarrow x = y$
SubseteqDef	$\forall x, y : x \subseteq y \Leftrightarrow \forall z : z \in x \Rightarrow z \in y$
EnumDef (p)	$\forall a_1,\ldots,a_p, x : x \in \{a_1,\ldots,a_p\} \Leftrightarrow x = a_1 \vee \cdots \vee x = a_p$
EmptyDef	$\forall x : x \notin \{\}$
UnionDef	$\forall a, x : x \in \text{UNION } a \Leftrightarrow \exists y : y \in a \wedge x \in y$
SubsetDef	$\forall a, x : x \in \text{SUBSET } a \Leftrightarrow x \subseteq a$
SetstDef (P)	$\forall a, x : x \in \{y \in a : P(y)\} \Leftrightarrow x \in a \wedge P(x)$
SetofDef (p, F)	$\forall a_1,\ldots,a_p, x : x \in \{F(y_1,\ldots,y_p) : y_1 \in a_1,\ldots,y_p \in a_p\} \Leftrightarrow$ $\exists z_1,\ldots,z_p : \wedge z_1 \in a_1$ $\wedge \ldots$ $\wedge z_p \in a_p$ $\wedge x = F(z_1,\ldots,z_p)$
CupDef	$\forall a, b : a \cup b = \text{UNION } \{a, b\}$
CapDef	$\forall a, b : a \cap b = \{x \in a : x \in b\}$
SetminusDef	$\forall a, b : a \setminus b = \{x \in a : x \notin b\}$
BooleanDef	$\text{BOOLEAN} = \{\text{TRUE}, \text{FALSE}\}$
Foundation	$\forall x : x \neq \{\} \Rightarrow \exists y : y \in x \wedge x \cap y = \{\}$
Infinity	$\exists x : \{\} \in x \wedge \forall y : y \in x \Rightarrow y \cup \{y\} \in x$

In the axiom CapDef, the notation $\{x \in a : x \in b\}$ is ambiguous, because it can be resolved as $\mathsf{setst}(a, \lambda x : x \in b)$ or as $\mathsf{setof}(\lambda x : x \in a, b)$. The former resolution holds. In SetstDef, the notation $\{x \in a : P(x)\}$ is resolved as $\mathsf{setst}(a, \lambda x : P(x))$, but this is equal to $\mathsf{setst}(a, P)$ by extentionality, so there is no loss of generality. A similar argument can be made about SetofDef.

Function Theory

Operator	Shape	Notation
isafcn	(0)	
fcn	$(0, 1)$	$[x \in e_1 \mapsto e_2] \triangleq \mathsf{fcn}(e_1, \lambda x : e_2)$
except	$(0, 0, 0)$	$[f \text{ EXCEPT } ![x] = y] \triangleq \mathsf{except}(f, x, y)$
domain	(0)	$\text{DOMAIN } e \triangleq \mathsf{domain}(e)$
fcnapp	$(0, 0)$	$e_1[e_2] \triangleq \mathsf{fcnapp}(e_1, e_2)$
arrow	$(0, 0)$	$[e_1 \to e_2] \triangleq \mathsf{arrow}(e_1, e_2)$

In ZFC, functions are typically defined as binary relations with specific properties. TLA$^+$ defines functions axiomatically. Functions are sets like any other element of the theory, but their underlying representation as sets is not specified. The fact that functions *could* be defined from binary relations proves that the following extension is conservative.

FcnExtensionality	$\forall f, g : \land \mathsf{isafcn}(f) \land \mathsf{isafcn}(g)$ $\land \text{DOMAIN } f = \text{DOMAIN } g$ $\land \forall x : x \in \text{DOMAIN } f \Rightarrow f[x] = g[x]$ $\Rightarrow f = g$
FcnIsafcn (F)	$\forall a : \mathsf{isafcn}([x \in a \mapsto F(x)])$
FcnDomain (F)	$\forall a : \text{DOMAIN } [x \in a \mapsto F(x)] = a$
FcnApp (F)	$\forall a, y : y \in a \Rightarrow [x \in a \mapsto F(x)][y] = F(y)$
FcnSet	$\forall a, b, f : f \in [a \to b] \Leftrightarrow$ $\land \mathsf{isafcn}(f)$ $\land \text{DOMAIN } f = a$ $\land \forall x : x \in a \Rightarrow f[x] \in b$
FcnExcept	$\forall f, x, y : [f \text{ EXCEPT } ![x] = y] =$ $[z \in \text{DOMAIN } f \mapsto \text{IF } z = x \text{ THEN } y \text{ ELSE } f[z]]$

Arithmetic

Operator	Shape	Notation
Int	()	
Nat	()	
Real	()	
0, 1, 2, ...	()	
0.1, 0.2, 1.5, 5.647, ...	()	
Infinity	()	
plus	$(0, 0)$	$e_1 + e_2 \triangleq \mathsf{plus}(e_1, e_2)$
uminus	(0)	$-e \triangleq \mathsf{uminus}(e)$
minus	$(0, 0)$	$e_1 - e_2 \triangleq \mathsf{minus}(e_1, e_2)$
mult	$(0, 0)$	$e_1 * e_2 \triangleq \mathsf{mult}(e_1, e_2)$
div	$(0, 0)$	$e_1/e_2 \triangleq \mathsf{div}(e_1, e_2)$
quotient	$(0, 0)$	$e_1 \div e_2 \triangleq \mathsf{quotient}(e_1, e_2)$
remainder	$(0, 0)$	$e_1 \,\%\, e_2 \triangleq \mathsf{remainder}(e_1, e_2)$
exp	$(0, 0)$	$e_1^{e_2} \triangleq \mathsf{exp}(e_1, e_2)$
lteq	$(0, 0)$	$e_1 \leq e_2 \triangleq \mathsf{lteq}(e_1, e_2)$
		$e_1 \geq e_2 \triangleq \mathsf{lteq}(e_2, e_1)$
		$e_1 < e_2 \triangleq \mathsf{lteq}(e_1, e_2) \wedge e_1 \neq e_2$
		$e_1 > e_2 \triangleq \mathsf{lteq}(e_2, e_1) \wedge e_1 \neq e_2$
range	$(0, 0)$	$e_1 \,..\, e_2 \triangleq \mathsf{range}(e_1, e_2)$

We use the different font for the constants *Int*, *Nat* and *Real* because it is how they are written in TLA$^+$, and there is no point in introducing them as notations. The natural and decimal numbers are also introduced directly with their literal representation. Each number is a constant in TLA$^+$. The operator $*$ should not be confused with $\times$, which is used for product sets.

As for functions, TLA$^+$ does not specify an underlying representation for *Int*, *Nat* and *Real*. It merely assumes those sets exist and satisfy the theories of integer, natural and real arithmetic, respectively. This is indeed the case in the context of set theory with the axiom of infinity. TLA$^+$ furthermore specifies that $Nat \subset Int \subset Real$. Operators are defined for real numbers, but they are overloaded for integers and naturals. For instance, $1 + 1 = 2$ and $0.8 + 6.3 = 7.1$ are both valid TLA$^+$ statements; both feature the same operator $+$, but the former can be derived from the theory of naturals whereas the latter must be derived from the theory of reals.

We do not provide an explicit axiomatization for the theories of arithmetic of TLA$^+$. Details about TLA$^+$'s arithmetic can be found in [48, sec. 18.4]. We will generally want to have TLA$^+$ arithmetic handled by a solver's builtin arithmetic engine. We may assume TLA$^+$'s interpretation of arithmetic is the same as that solver's interpretation—only the domain of interpretation will generally be a type in the solver's logic.

It might be useful to clarify how TLA$^+$ defines division. a/b is division on reals, defined for all $a \in Real$ and $b \in Real \setminus \{0\}$. $a \div b$ and $a \,\%\, b$ are the euclidian quotient and its corresponding remainder, defined for all $a \in Int$ and $b \in Nat \setminus \{0\}$. Unless those expressions are unspecified, the following relation is always satisfied:

$$a = b * (a \div b) + (a \,\%\, b)$$

Moreover, $a \div b \in \mathit{Int}$ and $a \,\%\, b \in 0\,..\,(b-1)$. Therefore, when $a < 0$, TLA$^+$ truncates the euclidian quotient towards $-\infty$ and keeps the remainder positive. For instance, $-10 \div 3 = -4$ and $-10 \,\%\, 3 = 2$. When $b < 0$ then both results are unspecified.

The only operator that is not purely arithmetic is range, for which we provide an axiom.

RangeDef	$\forall m, n : m\,..\,n = \{i \in \mathit{Int} : m \leq i \wedge i \leq n\}$

Tuples

Operator	Shape	Notation
tup_n	$(0, \ldots, 0)^\star$	$\langle e_1, \ldots, e_n\rangle \triangleq \mathsf{tup}_n(e_1, \ldots, e_n)$
prod_{p+1}	$(0, 0, \ldots, 0)^\star$	$e_1 \times e_2 \times \cdots \times e_p \triangleq \mathsf{prod}_p(e_1, e_2, \ldots, e_p)$

The tuples of TLA$^+$ are just functions from an interval set $1\,..\,n$. They are defined in terms of functions, which has the benefit of allowing a form of overloading over the operators for functions. ($\star$) The operators tup_n and prod_n both expect n arguments, but prod_n is only defined for $n \geq 2$. It is indeed impossible to write $a_1 \times \cdots \times a_n$ for $n = 0, 1$ in TLA$^+$'s user language.

TupleDef (n)	$\forall x_1, \ldots, x_n : \langle x_1, \ldots, x_n\rangle = [i \in 1\,..\,n \mapsto$ case $i = 1 \to x_1$ … case $i = n \to x_n]$
ProductDef (p)	$\forall a_1, \ldots, a_{p+1} : a_1 \times \cdots \times a_{p+1} = \{\langle x_1, \ldots, x_{p+1}\rangle : x_1 \in a_1, \ldots, x_{p+1} \in a_{p+1}\}$

Strings

Operator	Shape	Notation
string	()	STRING $\triangleq$ string
"foo", "bar", …	()	

StringIntro ("foo")	"foo" $\in$ STRING
StringDistinct ("foo", "bar" distinct)	"foo" $\neq$ "bar"

Records

Operator	Shape	Notation
$\mathsf{record}_{s_1,\ldots,s_p}$	$(0,\ldots,0)^\star$	$[s_1 \mapsto e_1, \ldots, s_p \mapsto e_p] \triangleq \mathsf{record}_{s_1,\ldots,s_p}(e_1,\ldots,e_p)$
$\mathsf{recset}_{s_1,\ldots,s_p}$	$(0,\ldots,0)^\star$	$[s_1 : e_1, \ldots, s_p : e_p] \triangleq \mathsf{recset}_{s_1,\ldots,s_p}(e_1,\ldots,e_p)$
		$r.s \triangleq r[\text{“}\mathtt{s}\text{”}]$

($\star$) The operators $\mathsf{record}_{s_1,\ldots,s_p}$ and $\mathsf{recset}_{s_1,\ldots,s_p}$ both expect p arguments; string parameters $s_1,\ldots,s_p$ are assumed distinct from each other. Records are similar to tuples in that they are defined as special functions. The domain of a record is a non-empty, finite set of strings.

RecordDef $(s_1,\ldots,s_p)$	$\forall x_1,\ldots,x_p : [s_1 \mapsto x_1,\ldots,s_p \mapsto x_p] =$
	$[x \in \{\text{“}\mathtt{s_1}\text{”},\ldots,\text{“}\mathtt{s_p}\text{”}\} \mapsto \text{CASE } x = \text{“}\mathtt{s_1}\text{”} \to x_1$
	$\ldots$
	$\text{CASE } x = \text{“}\mathtt{s_p}\text{”} \to x_p]$
RecordSetDef $(s_1,\ldots,s_p)$	$\forall a_1,\ldots,a_p : [s_1 : a_1,\ldots,s_p : a_p] =$
	$\{[s_1 \mapsto x_1,\ldots,x_p \mapsto x_p] : x_1 \in a_1,\ldots,x_p \in a_p\}$

Sequences

Operator	Shape	Notation
Seq	(0)	
Len	(0)	
Cat	(0, 0)	$f \circ g \triangleq Cat(f,g)$
Append	(0, 0)	
Head	(0)	
Tail	(0)	
Subseq	(0, 0, 0)	
Selectseq	(0, 1)	

Sequences are another special case of TLA$^+$ functions. A sequence over the set S is a tuple of arbitrary length whose elements are in S. There are many new operators introduced with sequences, all are defined from previous operators. The definition of *Selectseq* stands out as the only recursive definition of this axiomatization; this particular definition is not particularly useful in practise, but it is the true definition of the operator.

SeqDef $\forall a : Seq(a) = \text{UNION } \{[1 \,..\, n \mapsto a] : n \in Nat\}$

SeqLenDef $\forall f : Len(f) = \text{CHOOSE } n \in Nat : \text{DOMAIN } f = 1 \,..\, n$

SeqCatDef
$$\forall f, g : f \circ g = [i \in 1 \,..\, Len(f) + Len(g) \mapsto \text{IF } i \leq Len(f) \text{ THEN } f[i] \text{ ELSE } g[i - Len(f)]]$$

SeqAppendDef $\forall f, x : Append(f, x) = f \circ \langle x \rangle$

SeqHeadDef $\forall f : Head(f) = f[1]$

SeqTailDef $\forall f : Tail(f) = [i \in 1 \,..\, Len(f) - 1 \mapsto f[i + 1]]$

SeqSubseqDef $\forall f, m, n : Subseq(f, m, n) = [i \in m \,..\, n \mapsto f[i - m + 1]]$

SeqSelectDef (P)
$$\begin{aligned}\forall f : Selectseq(f, P) = (&\text{CHOOSE } g : \\ &g = [i \in 0 \,..\, Len(f) \mapsto \\ &\quad \text{IF } i = 0 \text{ THEN } \langle\rangle \\ &\quad \text{ELSE IF } P(f[i]) \text{ THEN } Append(g[i-1], f[i]) \\ &\quad \text{ELSE } g[i-1]] \\ &)[Len(f)]\end{aligned}$$

Finite Sets

Operator	Shape
Bijection	$(0, 0)$
IsFiniteSet	(0)
Card	(0)

A set is finite iff it is in bijection with a set $1 \,..\, n$, in which case its cardinality is n. Cardinalities are not specified for infinite sets. Also note that $Card(a) \in Nat$ does *not* imply $IsFiniteSet(a)$ with this definition.

BijectionDef
$$\forall a, b : Bijection(a, b) = \{f \in [a \to b] : \land \forall x, y : x \in a \land y \in a \land f[x] = f[y] \Rightarrow x = y \land \forall y : y \in b \Rightarrow \exists x : x \in a \land y = f[x]\}$$

IsFiniteDef $\forall a : IsFiniteSet(a) \Leftrightarrow \exists n, f : n \in Nat \land f \in Bijection(a, 1 \,..\, n)$

CardDef $\forall a : Card(a) = \text{CHOOSE } n : n \in Nat \land \exists f : f \in Bijection(a, 1 \,..\, n)$

Appendix B

Axioms for TPTP

This section documents the axioms used in the encoding to TPTP, which is detailed in chapter 4. The organization of the theory follows the one for TLA$^+$ that we presented in appendix A (epsilon-calculus, set theory, functions, arithmetic, tuples and records). The theories of sequences and finite sets are not included here; those theories require non trivial integer arithmetic to be handled properly, so they are only supported by our SMT backend.

The underlying logic of the axioms is higher-order logic, although only the second-order fragment is used. Compared to TLA$^+$'s logic, the Boolean sort o is available, and we note the individuals sort ι. We use standard notations for types, for instance the type of a binary relation on ι is written $\iota \times \iota \to o$.

Epsilon-calculus

Operator	Type
choose	$(\iota \to o) \to \iota$

ChooseDef	$\forall P^{\iota \to o}, x^{\iota} : P(x) \Rightarrow P(\mathsf{choose}(P))$
ChooseExt	$\forall P^{\iota \to o}, Q^{\iota \to o} : (\forall x^{\iota} : P(x) \Leftrightarrow Q(x)) \Rightarrow \mathsf{choose}(P) =_{\iota} \mathsf{choose}(Q)$

Set Theory

Operator	Type
in	$\iota \times \iota \to o$
$\mathsf{subseteq}$	$\iota \times \iota \to o$
enum_n	$\iota^n \to \iota$
union	$\iota \to \iota$
subset	$\iota \to \iota$
setst	$\iota \times (\iota \to o) \to \iota$
setof_p	$\iota^p \times (\iota^p \to \iota) \to \iota$

Operator	Type
cup	$\iota \times \iota \to \iota$
cap	$\iota \times \iota \to \iota$
diff	$\iota \times \iota \to \iota$
$\mathsf{boolean}$	ι
cast_o	$o \to \iota$

SetExt	$\forall x^\iota, y^\iota : (\forall z^\iota : \mathsf{in}(z,x) \Leftrightarrow \mathsf{in}(z,y)) \Rightarrow x =_\iota y$
SubseteqDef	$\forall x^\iota, y^\iota : \mathsf{subseteq}(x,y) \Leftrightarrow (\forall z^\iota : \mathsf{in}(z,x) \Leftrightarrow \mathsf{in}(z,y))$
EnumDef (p)	$\forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \mathsf{in}(x, \mathsf{enum}_p(a_1, \ldots, a_p)) \Leftrightarrow x =_\iota a_1 \vee \cdots \vee x =_\iota a_p$
EmptyDef	$\forall x^\iota : \neg\mathsf{in}(x, \mathsf{enum}_0)$
SubsetDef	$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{subset}(a)) \Leftrightarrow (\forall y^\iota : \mathsf{in}(y,x) \Rightarrow \mathsf{in}(y,a))$
UnionDef	$\forall a^\iota, x^\iota : \mathsf{in}(x, \mathsf{union}(a)) \Leftrightarrow (\exists y^\iota : \mathsf{in}(x,y) \wedge \mathsf{in}(y,a))$
SetstDef	$\forall P^{\iota\to o}, a^\iota, x^\iota : \mathsf{in}(x, \mathsf{setst}(a,P)) \Leftrightarrow \mathsf{in}(x,a) \wedge P(x)$
SetofDef (p)	$\forall F^{\iota^p\to\iota}, a_1^\iota, \ldots, a_p^\iota, x^\iota : \mathsf{in}(x, \mathsf{setof}(a_1, \ldots, a_p, F)) \Leftrightarrow$ $\exists y_1^\iota, \ldots, y_p^\iota : \mathsf{in}(y_1,a_1) \wedge \cdots \wedge \mathsf{in}(y_p,a_p) \wedge x =_\iota F(y_1, \ldots, y_p)$
CupDef	$\forall a^\iota, b^\iota, x^\iota : \mathsf{in}(x, \mathsf{cup}(a,b)) \Leftrightarrow \mathsf{in}(x,a) \vee \mathsf{in}(x,b)$
CapDef	$\forall a^\iota, b^\iota, x^\iota : \mathsf{in}(x, \mathsf{cap}(a,b)) \Leftrightarrow \mathsf{in}(x,a) \wedge \mathsf{in}(x,b)$
SetminusDef	$\forall a^\iota, b^\iota, x^\iota : \mathsf{in}(x, \mathsf{diff}(a,b)) \Leftrightarrow \mathsf{in}(x,a) \wedge \neg\mathsf{in}(x,b)$
BoolCastInj	$\mathsf{cast}_o(\top) \neq \mathsf{cast}_o(\bot)$
BooleanDef	$\forall x^\iota : \mathsf{in}(x, \mathsf{boolean}) \Leftrightarrow x =_\iota \mathsf{cast}_o(\top) \vee x =_\iota \mathsf{cast}_o(\bot)$

Function Theory

Operator	Type
isafcn	$\iota \to o$
fcn	$\iota \times (\iota \to \iota) \to \iota$
except	$\iota \times \iota \times \iota \to \iota$
domain	$\iota \to \iota$
fcnapp	$\iota \times \iota \to \iota$
arrow	$\iota \times \iota \to \iota$

FcnExtensionality	$\forall f^\iota, g^\iota : \wedge\ \mathsf{isafcn}(f) \wedge \mathsf{isafcn}(g)$ $\wedge\ \mathsf{domain}(f) =_\iota \mathsf{domain}(g)$ $\wedge\ (\forall x^\iota : \mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{fcnapp}(f, x) =_\iota \mathsf{fcnapp}(g, x))$ $\Rightarrow f =_\iota g$
FcnIsafcn	$\forall F^{\iota\to\iota}, a^\iota : \mathsf{isafcn}(\mathsf{fcn}(a, F))$
FcnDom	$\forall F^{\iota\to\iota}, a^\iota : \mathsf{domain}(\mathsf{fcn}(a, F)) =_\iota a$
FcnApp	$\forall F^{\iota\to\iota}, a^\iota, x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{fcnapp}(\mathsf{fcn}(a, F), x) = F(x)$
ArrowDef	$\forall a^\iota, b^\iota, f^\iota : \mathsf{in}(f, \mathsf{arrow}(a, b)) \Leftrightarrow$ $\wedge\ \mathsf{isafcn}(f)$ $\wedge\ \mathsf{domain}(f) =_\iota a$ $\wedge\ (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), b))$
ExceptIsafcn	$\forall f^\iota, x^\iota, y^\iota : \mathsf{isafcn}(\mathsf{except}(f, x, y))$
ExceptDom	$\forall f^\iota, x^\iota, y^\iota : \mathsf{domain}(\mathsf{except}(f, x, y)) =_\iota \mathsf{domain}(f)$
ExceptApp	$\forall f^\iota, x^\iota, y^\iota, z^\iota : \mathsf{in}(z, \mathsf{domain}(f)) \Rightarrow$ $\wedge\ z =_\iota x \Rightarrow \mathsf{fcnapp}(\mathsf{except}(f, x, y), z) =_\iota y$ $\wedge\ z \neq_\iota x \Rightarrow \mathsf{fcnapp}(\mathsf{except}(f, x, y), z) =_\iota \mathsf{fcnapp}(f, z)$

Arithmetic and Strings

Contrary to the previous appendix, we note numerical constants with symbols num_n. This will make clearer in expressions when a natural number is an actual constant or some parameter (for instance the n in enum_n).

Our axiomatization does not cover real arithmetic, nor the characterizations of strings as sequences of characters.

Operator	Type
Int	ι
Nat	ι
num_n	ι
plus	$\iota \times \iota \to \iota$
uminus	$\iota \to \iota$
minus	$\iota \times \iota \to \iota$
mult	$\iota \times \iota \to \iota$

Operator	Type
quotient	$\iota \times \iota \to \iota$
remainder	$\iota \times \iota \to \iota$
exp	$\iota \times \iota \to \iota$
lteq	$\iota \times \iota \to o$
range	$\iota \times \iota \to \iota$
str_s	ι
string	ι

NatDef	$\forall n^{\iota} : \mathsf{in}(n, \mathsf{Nat}) \Leftrightarrow \mathsf{in}(n, \mathsf{Int}) \wedge \mathsf{lteq}(0, n)$
NumTyping (n)	$\mathsf{in}(\mathsf{num}_n, Nat)$
NumDistinct $(n_1, n_2$ distincts)	$\mathsf{num}_{n_1} \neq_{\iota} \mathsf{num}_{n_2}$
PlusTyping	$\forall m^{\iota}, n^{\iota} : \mathsf{in}(m, \mathsf{Int}) \wedge \mathsf{in}(n, \mathsf{Int}) \Rightarrow \mathsf{in}(\mathsf{plus}(m, n), \mathsf{Int})$
UminusTyping	$\forall n^{\iota} : \mathsf{in}(n, \mathsf{Int}) \Rightarrow \mathsf{in}(\mathsf{uminus}(n), \mathsf{Int})$
MinusTyping	$\forall m^{\iota}, n^{\iota} : \mathsf{in}(m, \mathsf{Int}) \wedge \mathsf{in}(n, \mathsf{Int}) \Rightarrow \mathsf{in}(\mathsf{minus}(m, n), \mathsf{Int})$
MultTyping	$\forall m^{\iota}, n^{\iota} : \mathsf{in}(m, \mathsf{Int}) \wedge \mathsf{in}(n, \mathsf{Int}) \Rightarrow \mathsf{in}(\mathsf{mult}(m, n), \mathsf{Int})$
QuotientTyping	$\forall m^{\iota}, n^{\iota} : \mathsf{in}(m, \mathsf{Int}) \wedge \mathsf{in}(n, \mathsf{Nat}) \wedge n \neq \mathsf{num}_0 \Rightarrow \mathsf{in}(\mathsf{quotient}(m, n), \mathsf{Int})$
RemainderTyping	$\forall m^{\iota}, n^{\iota} : \mathsf{in}(m, \mathsf{Int}) \wedge \mathsf{in}(n, \mathsf{Nat}) \wedge n \neq \mathsf{num}_0 \Rightarrow \mathsf{in}(\mathsf{remainder}(m, n), \mathsf{range}(\mathsf{num}_0, n))$
RangeDef	$\forall m^{\iota}, n^{\iota}, p^{\iota} : \mathsf{in}(p, \mathsf{range}(m, n)) \Leftrightarrow \mathsf{in}(p, \mathsf{Int}) \wedge \mathsf{lteq}(m, p) \wedge \mathsf{lteq}(p, n)$
StrTyping (s)	$\mathsf{in}(\mathsf{str}_s, \mathsf{string})$
StrDistinct $(s_1, s_2$ distincts)	$\mathsf{str}_{s_1} \neq_{\iota} \mathsf{str}_{s_2}$

Tuples and Records

Operator	Type
tup_n	$\iota^n \to \iota$
prod_p	$\iota^{p+1} \to \iota$
$\mathsf{record}_{s_1,\ldots,s_p}$	$\iota^p \to \iota$
$\mathsf{recset}_{s_1,\ldots,s_p}$	$\iota^p \to \iota$

TupIsafcn (n)	$\forall x_1^\iota, \ldots, x_n^\iota : \mathsf{isafcn}(\mathsf{tup}_n(x_1, \ldots, x_n))$
TupDom (n)	$\forall x_1^\iota, \ldots, x_n^\iota : \mathsf{domain}(\mathsf{tup}_n(x_1, \ldots, x_n)) = \mathsf{enum}_n 1, \ldots, n$
TupApp (n)	$\forall x_1^\iota, \ldots, x_n^\iota : \wedge\, \mathsf{fcnapp}(\mathsf{tup}_n(x_1, \ldots, x_n), \mathsf{num}_1) = x_1$
	$\wedge \ldots$
	$\wedge\, \mathsf{fcnapp}(\mathsf{tup}_n(x_1, \ldots, x_n), \mathsf{num}_n) = x_n$
ProdDef (p)	$\forall a_1^\iota, \ldots, a_p^\iota, a_{p+1}^\iota, t^\iota : \mathsf{in}(t, \mathsf{prod}_p(a_1, \ldots, a_p, a_{p+1})) \Leftrightarrow$
	$\wedge\, t = \mathsf{tup}_{p+1}(\mathsf{fcnapp}(t, \mathsf{num}_1), \ldots, \mathsf{fcnapp}(t, \mathsf{num}_{p+1}))$
	$\wedge\, \mathsf{in}(\mathsf{fcn}(t, \mathsf{num}_1), a_1)$
	$\wedge \ldots$
	$\wedge\, \mathsf{in}(\mathsf{fcn}(t, \mathsf{num}_{p+1}), a_{p+1})$
RecIsafcn $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \mathsf{isafcn}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p))$
RecDom $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \mathsf{domain}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p)) = \mathsf{enum}_p(\mathsf{str}_{s_1}, \ldots, \mathsf{str}_{s_p})$
RecApp $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \wedge\, \mathsf{fcnapp}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{str}s_1) = x_1$
	$\wedge \ldots$
	$\wedge\, \mathsf{fcnapp}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{str}s_p) = x_p$
RecsetDef $(s_1, \ldots, s_p)$	$\forall a_1^\iota, \ldots, a_p^\iota, r^\iota : \mathsf{in}(r, \mathsf{recset}_{s_1,\ldots,s_p}(a_1, \ldots, a_p)) \Leftrightarrow$
	$\wedge\, r = \mathsf{record}_{s_1,\ldots,s_p}(\mathsf{fcnapp}(r, \mathsf{str}_{s_1}), \ldots, \mathsf{fcnapp}(r, \mathsf{str}_{s_p}))$
	$\wedge\, \mathsf{in}(\mathsf{fcnapp}(r, \mathsf{str}_{s_1}), a_1)$
	$\wedge \ldots$
	$\wedge\, \mathsf{in}(\mathsf{fcnapp}(r, \mathsf{str}_{s_p}), a_p)$

Appendix C

Axioms for SMT

This section documents the axioms used in the encoding to SMT-LIB, which is detailed in chapter 5.

Axioms are expressed in multi-sorted first-order logic. Operators are assigned first-order types at declaration instead of shapes. There are axiom schemas, for which parameters are indicated after the schema's name. As before, we use the letter n for natural numbers, p for positive natural numbers, s for strings of characters, and the letters F, P, Q for operators symbols, which will be assigned types this time.

A major addition compared to previous axiomatizations is the presence of E-matching patterns. All patterns are indicated between curly-braces just below the relevant chain of quantifiers. The general syntax is:

$$\begin{aligned} \forall x_1^{s_1}, \ldots, x_n^{s_n} : & \left\{t_1^1, \ldots, t_{n_1}^1\right\} \\ & \ldots \\ & \{t_1^m, \ldots, t_{n_m}^m\} \\ \phi & \end{aligned}$$

Where, for all $1 \leq i \leq m$, the pattern $\left\{t_1^i, \ldots, t_{n_i}^i\right\}$ is a set of well-sorted terms, the free variables of which are exactly $x_1, \ldots, x_n$.

For abbreviating types, we will sometimes use the syntax $s^n \to s$. For instance the expression $\mathsf{int} \times \iota^2 \to o$ is short for the type $\mathsf{int} \times \iota \times \iota \to o$. If $n = 0$ then $s^n \to s$ is just s.

Epsilon-calculus

Operator	Type
$\mathsf{choose}_P^\star$	$\iota^n \to \iota$

$(\star)$ $P : \iota^{n+1} \to o$.

ChooseDef $(P : \iota^{n+1} \to o)$ $\quad \forall c_1^\iota, \ldots, c_n^\iota, x^\iota :$
$P(x, c_1, \ldots, c_n) \Rightarrow P(\mathsf{choose}_P(c_1, \ldots, c_n), c_1, \ldots, c_n)$

ChooseExt $(P, Q : \iota^{n+1} \to o)$ $\quad \forall c_1^\iota, \ldots, c_n^\iota :$
$(\forall x^\iota : P(x, c_1, \ldots, c_n) \Leftrightarrow Q(x, c_1, \ldots, c_n))$
$\Rightarrow \mathsf{choose}_P(c_1, \ldots, c_n) = \mathsf{choose}_Q(c_1, \ldots, c_n)$

Set Theory

Operator	Type
in	$\iota \times \iota \to o$
subseteq	$\iota \times \iota \to o$
enum_n	$\iota^n \to \iota$
union	$\iota \to \iota$
subset	$\iota \to \iota$
$\mathsf{setst}_P{}^\star$	$\iota \times \iota^n \to \iota$

Operator	Type
$\mathsf{setof}_F{}^{\star\star}$	$\iota^p \times \iota^n \to \iota$
cup	$\iota \times \iota \to \iota$
cap	$\iota \times \iota \to \iota$
diff	$\iota \times \iota \to \iota$
boolean	ι
cast_o	$o \to \iota$

$(\star)$ $P : \iota^{n+1} \to o$
$(\star\star)$ $F : \iota^{p+n} \to \iota$

SubseteqIntro $\quad \forall a^\iota, b^\iota : \{\mathsf{subseteq}(a, b)\}$
$(\forall x^\iota : \mathsf{in}(x, a) \Leftrightarrow \mathsf{in}(x, b)) \Rightarrow \mathsf{subseteq}(a, b)$

SubseteqElim $\quad \forall a^\iota, b^\iota, x^\iota : \{\mathsf{subseteq}(a, b), \mathsf{in}(x, a)\}$
$\mathsf{subseteq}(a, b) \wedge \mathsf{in}(x, a) \Rightarrow \mathsf{in}(x, b)$

EnumIntro (p) $\quad \forall a_1^\iota, \ldots, a_p^\iota : \{\mathsf{enum}_p(a_1, \ldots, a_p)\}$
$\mathsf{in}(a_1, \mathsf{enum}_p(a_1, \ldots, a_p)) \wedge \cdots \wedge \mathsf{in}(a_p, \mathsf{enum}_p(a_1, \ldots, a_p))$

EnumElim (p) $\quad \forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{enum}_p(a_1, \ldots, a_p))\}$
$\mathsf{in}(x, \mathsf{enum}_p(a_1, \ldots, a_p)) \Rightarrow x = a_1 \vee \cdots \vee x = a_p$

EmptyElim $\quad \forall x^\iota : \{\mathsf{in}(x, \mathsf{enum}_0)\} \quad \neg\mathsf{in}(x, \mathsf{enum}_0)$

SubsetDef $\quad \forall a^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{subset}(a))\}$
$\{\mathsf{subseteq}(x, a), \mathsf{subset}(a)\}$
$\mathsf{in}(x, \mathsf{subset}(a)) \Leftrightarrow \mathsf{subseteq}(x, a)$

UnionIntro $\quad \forall a^\iota, x^\iota, y^\iota : \{\mathsf{in}(y, a), \mathsf{in}(x, \mathsf{union}(a))\}$
$\{\mathsf{in}(x, y), \mathsf{in}(x, \mathsf{union}(a))\}$
$\{\mathsf{in}(x, y), \mathsf{in}(y, a), \mathsf{union}(a)\}$
$\mathsf{in}(x, y) \wedge \mathsf{in}(y, a) \Rightarrow \mathsf{in}(x, \mathsf{union}(a))$

UnionElim $\quad \forall a^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{union}(a))\}$
$\mathsf{in}(x, \mathsf{union}(a)) \Rightarrow \exists y^\iota : \mathsf{in}(x, y) \wedge \mathsf{in}(y, a)$

SetstDef ($P : \iota^{n+1} \to o$)	$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{setst}_P(a, c_1, \ldots, c_n))\}$
	$\{\mathsf{in}(x, a), \mathsf{setst}_P(a, c_1, \ldots, c_n)\}$
	$\mathsf{in}(x, \mathsf{setst}_P(a, c_1, \ldots, c_n)) \Leftrightarrow \mathsf{in}(x, a) \wedge P(x, c_1, \ldots, c_n)$
SetofIntro ($F : \iota^{p+n} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a_1^\iota, \ldots, a_p^\iota, y_1^\iota, \ldots, y_p^\iota :$
	$\{F(y_1, \ldots, y_p, c_1, \ldots, c_n), \mathsf{setof}_F(a_1, \ldots, a_p, c_1, \ldots, c_n)\}$
	$\{\mathsf{in}(y_1, a_1), \ldots, \mathsf{in}(y_p, a_p), \mathsf{setof}_F(a_1, \ldots, a_p, c_1, \ldots, c_n)\}$
	$\mathsf{in}(y_1, a_1) \wedge \cdots \wedge \mathsf{in}(y_p, a_p)$
	$\Rightarrow \mathsf{in}(F(y_1, \ldots, y_p, c_1, \ldots, c_n), \mathsf{setof}_F(a_1, \ldots, a_p, c_1, \ldots, c_n))$
SetofElim ($F : \iota^{p+n} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a_1^\iota, \ldots, a_p^\iota, x^\iota :$
	$\{\mathsf{in}(x, \mathsf{setof}_F(a_1, \ldots, a_p, c_1, \ldots, c_n))\}$
	$\mathsf{in}(x, \mathsf{setof}_F(a_1, \ldots, a_p, c_1, \ldots, c_n)) \Rightarrow$
	$\exists y_1^\iota, \ldots, y_p^\iota : \mathsf{in}(y_1, a_1) \wedge \cdots \wedge \mathsf{in}(y_p, a_p) \wedge x = F(y_1, \ldots, y_p)$
CupDef	$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{cup}(a, b))\}$
	$\{\mathsf{in}(x, a), \mathsf{cup}(a, b)\}$
	$\{\mathsf{in}(x, b), \mathsf{cup}(a, b)\}$
	$\mathsf{in}(x, \mathsf{cup}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \vee \mathsf{in}(x, b)$
CapDef	$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{cap}(a, b))\}$
	$\{\mathsf{in}(x, a), \mathsf{cap}(a, b)\}$
	$\{\mathsf{in}(x, b), \mathsf{cap}(a, b)\}$
	$\mathsf{in}(x, \mathsf{cap}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \mathsf{in}(x, b)$
SetminusDef	$\forall a^\iota, b^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{diff}(a, b))\}$
	$\{\mathsf{in}(x, a), \mathsf{diff}(a, b)\}$
	$\{\mathsf{in}(x, b), \mathsf{diff}(a, b)\}$
	$\mathsf{in}(x, \mathsf{diff}(a, b)) \Leftrightarrow \mathsf{in}(x, a) \wedge \neg\mathsf{in}(x, b)$
BoolCastInj	$\mathsf{cast}_o(\text{TRUE}) \neq \mathsf{cast}_o(\text{FALSE})$
BooleanIntro	$\mathsf{in}(\mathsf{cast}_o(\text{TRUE}), \mathsf{boolean}) \wedge \mathsf{in}(\mathsf{cast}_o(\text{FALSE}), \mathsf{boolean})$
BooleanElim	$\forall x^\iota : \{\mathsf{in}(x, \mathsf{boolean})\}$
	$\mathsf{in}(x, \mathsf{boolean}) \Rightarrow x = \mathsf{cast}_o(\text{TRUE}) \vee x = \mathsf{cast}_o(\text{FALSE})$

Function Theory

Operator	Type
isafcn	$\iota \to o$
$\mathsf{fcn}_F{}^\star$	$\iota \times \iota^n \to \iota$
except	$\iota \times \iota \times \iota \to \iota$
domain	$\iota \to \iota$
fcnapp	$\iota \times \iota \to \iota$
arrow	$\iota \times \iota \to \iota$

$(\star)\ F : \iota^{n+1} \to \iota$

FcnExtensionality	$\forall f^\iota, g^\iota : \{\mathsf{isafcn}(f), \mathsf{isafcn}(g)\}$ $\wedge\ \mathsf{isafcn}(f) \wedge \mathsf{isafcn}(g)$ $\wedge\ \mathsf{domain}(f) = \mathsf{domain}(g)$ $\wedge\ (\forall x^\iota : \mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{fcnapp}(f, x) = \mathsf{fcnapp}(g, x))$ $\Rightarrow f = g$
FcnIsafcn ($F : \iota^{n+1} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{\mathsf{fcn}_F(a, c_1, \ldots, c_n)\}$ $\mathsf{isafcn}(\mathsf{fcn}_F(a, c_1, \ldots, c_n))$
FcnDom ($F : \iota^{n+1} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{\mathsf{fcn}_F(a, c_1, \ldots, c_n)\}$ $\mathsf{domain}(\mathsf{fcn}_F(a, c_1, \ldots, c_n)) = a$
FcnApp ($F : \iota^{n+1} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{\mathsf{fcnapp}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), x)\}$ $\{\mathsf{in}(x, a), \mathsf{fcn}_F(a, c_1, \ldots, c_n)\}$ $\mathsf{in}(x, a) \Rightarrow \mathsf{fcnapp}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), x) = F(x, c_1, \ldots, c_n)$
FcnTyping ($F : \iota^{n+1} \to \iota$)	$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, b^\iota : \{\mathsf{fcn}_F(a, c_1, \ldots, c_n), \mathsf{arrow}(a, b)\}$ $(\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(F(x, c_1, \ldots, c_n), b))$ $\Rightarrow \mathsf{in}(\mathsf{fcn}_F(a, c_1, \ldots, c_n), \mathsf{arrow}(a, b))$
ArrowIntro	$\forall a^\iota, b^\iota, f^\iota : \{\mathsf{in}(f, \mathsf{arrow}(a, b))\}$ $\wedge\ \mathsf{isafcn}(f)$ $\wedge\ \mathsf{domain}(f) = a$ $\wedge\ (\forall x^\iota : \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), b))$ $\Rightarrow \mathsf{in}(f, \mathsf{arrow}(a, b))$
ArrowElim$_1$	$\forall a^\iota, b^\iota, f^\iota : \{\mathsf{in}(f, \mathsf{arrow}(a, b))\}$ $\mathsf{in}(f, \mathsf{arrow}(a, b)) \Rightarrow\ \wedge\ \mathsf{isafcn}(f)$ $\wedge\ \mathsf{domain}(f) = a$
ArrowElim$_2$	$\forall a^\iota, b^\iota, f^\iota, x^\iota : \{\mathsf{in}(f, \mathsf{arrow}(a, b)), \mathsf{in}(x, a)\}$ $\{\mathsf{in}(f, \mathsf{arrow}(a, b)), \mathsf{fcnapp}(f, x)\}$ $\mathsf{in}(f, \mathsf{arrow}(a, b)) \wedge \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(f, x), b)$

ExceptIsafcn	$\forall f^\iota, x^\iota, y^\iota : \{\mathsf{except}(f, x, y)\}$
	$\mathsf{isafcn}(\mathsf{except}(f, x, y))$
ExceptDom	$\forall f^\iota, x^\iota, y^\iota : \{\mathsf{except}(f, x, y)\}$
	$\mathsf{domain}(\mathsf{except}(f, x, y)) = \mathsf{domain}(f)$
ExceptApp$_1$	$\forall f^\iota, x^\iota, y^\iota : \{\mathsf{except}(f, x, y)\}$
	$\mathsf{in}(x, \mathsf{domain}(f)) \Rightarrow \mathsf{fcnapp}(\mathsf{except}(f, x, y), x) = y$
ExceptApp$_2$	$\forall f^\iota, x^\iota, y^\iota, z^\iota : \{\mathsf{fcnapp}(\mathsf{except}(f, x, y), z)\}$
	$\{\mathsf{except}(f, x, y), \mathsf{fcnapp}(f, z)\}$
	$\mathsf{in}(z, \mathsf{domain}(f)) \wedge z \neq x \Rightarrow \mathsf{fcnapp}(\mathsf{except}(f, x, y), z) = \mathsf{fcnapp}(f, z)$
ExceptTyping	$\forall f^\iota, x^\iota, y^\iota, a^\iota, b^\iota : \{\mathsf{except}(f, x, y), \mathsf{in}(f, \mathsf{arrow}(a, b))\}$
	$\mathsf{in}(f, \mathsf{arrow}(a, b)) \wedge (\mathsf{in}(x, a) \Rightarrow \mathsf{in}(y, b)) \Rightarrow \mathsf{in}(\mathsf{except}(f, x, y), \mathsf{arrow}(a, b))$

Arithmetic

Operator	Type
$\mathsf{cast}_{\mathsf{int}}$	$\mathsf{int} \to \iota$
$\mathsf{proj}_{\mathsf{int}}$	$\iota \to \mathsf{int}$
Int	ι
Nat	ι
plus	$\iota \times \iota \to \iota$
uminus	$\iota \to \iota$

Operator	Type
minus	$\iota \times \iota \to \iota$
mult	$\iota \times \iota \to \iota$
quotient	$\iota \times \iota \to \iota$
remainder	$\iota \times \iota \to \iota$
lteq	$\iota \times \iota \to o$
range	$\iota \times \iota \to \iota$

The symbols $+$, $-$, $\times$, $\div$, $\%$ and $\leq$ as used below are not TLA$^+$'s operators, but the builtin integer operators of SMT-LIB.

IntCastInjective	$\forall z^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z)\}\ \ z = \mathsf{proj}_{\mathsf{int}}(\mathsf{cast}_{\mathsf{int}}(z))$
IntIntIntro	$\forall z^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z)\}\ \ \mathsf{in}(\mathsf{cast}_{\mathsf{int}}(z), \mathsf{Int})$
IntIntElim	$\forall x^{\iota} : \{\mathsf{in}(x, \mathsf{Int})\}\ \ \mathsf{in}(x, \mathsf{Int}) \Rightarrow x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x))$
IntNatIntro	$\forall z^{\mathsf{int}} : \{\mathsf{cast}_{\mathsf{int}}(z)\}$ $z \geq 0 \Rightarrow \mathsf{in}(\mathsf{cast}_{\mathsf{int}}(z), \mathsf{Nat})$
IntNatElim	$\forall x^{\iota} : \{\mathsf{in}(x, \mathsf{Nat})\}$ $\mathsf{in}(x, \mathsf{Nat}) \Rightarrow x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)) \wedge \mathsf{proj}_{\mathsf{int}}(x) \geq 0$
IntPlusTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $\mathsf{plus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 + z_2)$
IntUminusTyping	$\forall z^{\mathsf{int}} : \{\mathsf{uminus}(\mathsf{cast}_{\mathsf{int}}(z))\}$ $\mathsf{uminus}(\mathsf{cast}_{\mathsf{int}}(z)) = \mathsf{cast}_{\mathsf{int}}(-z)$
IntMinusTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{minus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $\mathsf{minus}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 - z_2)$
IntMultTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{mult}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $\mathsf{mult}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 \times z_2)$
IntQuotientTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{quotient}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $z_2 > 0 \Rightarrow \mathsf{quotient}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 \div z_2)$
IntRemainderTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{remainder}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $z_2 > 0 \Rightarrow \mathsf{remainder}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) = \mathsf{cast}_{\mathsf{int}}(z_1 \,\%\, z_2)$
IntLteqTyping	$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{\mathsf{lteq}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2))\}$ $\mathsf{lteq}(\mathsf{cast}_{\mathsf{int}}(z_1), \mathsf{cast}_{\mathsf{int}}(z_2)) \Leftrightarrow z_1 \leq z_2$
IntRangeIntro	$\forall a^{\iota}, b^{\iota}, z^{\mathsf{int}} : \{\mathsf{in}(\mathsf{cast}_{\mathsf{int}}(z), \mathsf{range}(a, b))\}$ $\mathsf{lteq}(a, \mathsf{cast}_{\mathsf{int}}(z)) \wedge \mathsf{lteq}(\mathsf{cast}_{\mathsf{int}}(z), b) \Rightarrow \mathsf{in}(\mathsf{cast}_{\mathsf{int}}(z), \mathsf{range}(a, b))$
IntRangeElim	$\forall a^{\iota}, b^{\iota}, x^{\iota} : \{\mathsf{in}(x, \mathsf{range}(a, b))\}$ $\mathsf{in}(x, \mathsf{range}(a, b)) \Rightarrow x = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(x)) \wedge \mathsf{lteq}(a, x) \wedge \mathsf{lteq}(x, b)$

Tuples

Operator	Type
tup_n	$\iota^n \to \iota$
prod_{p+1}	$\iota^{p+1} \to \iota$

TupIsafcn (n)	$\forall x_1^\iota, \ldots, x_n^\iota : \{\mathsf{tup}_n(x_1, \ldots, x_n)\}$
	$\mathsf{isafcn}(\mathsf{tup}_n(x_1, \ldots, x_n))$
TupDom (n)	$\forall x_1^\iota, \ldots, x_n^\iota : \{\mathsf{tup}_n(x_1, \ldots, x_n)\}$
	$\mathsf{domain}(\mathsf{tup}_n(x_1, \ldots, x_n)) = \mathsf{enum}_n(\mathsf{cast}_{\mathsf{int}}(1), \ldots, \mathsf{cast}_{\mathsf{int}}(n))$
TupApp (p)	$\forall x_1^\iota, \ldots, x_p^\iota : \left\{\mathsf{tup}_p(x_1, \ldots, x_p)\right\}$
	$\wedge\ \mathsf{fcnapp}(\mathsf{tup}_p(x_1, \ldots, x_p), \mathsf{cast}_{\mathsf{int}}(1)) = x_1$
	$\wedge \ldots$
	$\wedge\ \mathsf{fcnapp}(\mathsf{tup}_p(x_1, \ldots, x_p), \mathsf{cast}_{\mathsf{int}}(p)) = x_p$
TupExcept $(p, p' \leq p)$	$\forall x_1^\iota, \ldots, x_p^\iota, x^\iota : \left\{\mathsf{except}(\mathsf{tup}_p(x_1, \ldots, x_p), \mathsf{cast}_{\mathsf{int}}(p'), x)\right\}$
	$\mathsf{except}(\mathsf{tup}_p(x_1, \ldots, x_{p'}, \ldots, x_p), \mathsf{cast}_{\mathsf{int}}(p'), x) = \mathsf{tup}_p(x_1, \ldots, x, \ldots, x_p)$
ProductIntro (p)	$\forall a_1^\iota, \ldots, a_p^\iota, x_1^\iota, \ldots, x_p^\iota : \left\{\mathsf{tup}_p(x_1, \ldots, x_p), \mathsf{prod}_p(a_1, \ldots, a_p)\right\}$
	$\mathsf{in}(x_1, a_1) \wedge \cdots \wedge \mathsf{in}(x_p, a_p) \Rightarrow \mathsf{in}(\mathsf{tup}_p(x_1, \ldots, x_p), \mathsf{prod}_p(a_1, \ldots, a_p))$
ProductElim (p)	$\forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \left\{\mathsf{in}(x, \mathsf{prod}_p(a_1, \ldots, a_p))\right\}$
	$\mathsf{in}(x, \mathsf{prod}_p(a_1, \ldots, a_p)) \Rightarrow x = \mathsf{tup}_p(\mathsf{fcnapp}(x, \mathsf{cast}_{\mathsf{int}}(1)), \ldots, \mathsf{fcnapp}(x, \mathsf{cast}_{\mathsf{int}}(p)))$

Strings

Operator	Type
str_s	ι
string	ι

StringIntro (s)	$\mathsf{in}(\mathsf{str}_s, \mathsf{string})$
StringsDistinct $(s_1, s_2$ distincts$)$	$\mathsf{str}_{s_1} \neq \mathsf{str}_{s_2}$

Records

Operator	Type
$\mathsf{record}_{s_1,\ldots,s_p}{}^\star$	$\iota^p \to \iota$
$\mathsf{recset}_{s_1,\ldots,s_p}{}^\star$	$\iota^p \to \iota$

$(\star)$ Fields $s_1, \ldots, s_p$ are assumed distinct

RECORDISAFCN $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \{\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p)\}$
	$\mathsf{isafcn}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p))$
RECORDDOM $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \{\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p)\}$
	$\mathsf{domain}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p)) = \mathsf{enum}_p(\mathsf{str}_{s_1}, \ldots, \mathsf{str}_{s_p})$
RECORDAPP $(s_1, \ldots, s_p)$	$\forall x_1^\iota, \ldots, x_p^\iota : \{\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p)\}$
	$\wedge\ \mathsf{fcnapp}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{str}_{s_1}) = x_1$
	$\wedge \ldots$
	$\wedge\ \mathsf{fcnapp}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{str}_{s_p}) = x_p$
RECORDEXCEPT $(s_1, \ldots, s_p, p' \leq p)$	$\forall x_1^\iota, \ldots, x_p^\iota, x^\iota : \left\{\mathsf{except}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{str}_{s_{p'}}, x)\right\}$
	$\mathsf{except}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_{p'}, \ldots, x_p), \mathsf{str}_{s_{p'}}, x)$
	$= \mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x, \ldots, x_p)$
RECSETINTRO $(s_1, \ldots, s_p)$	$\forall a_1^\iota, \ldots, a_p^\iota, x_1^\iota, \ldots, x_p^\iota : \{\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{recset}_{s_1,\ldots,s_p}(a_1, \ldots, a_p)\}$
	$\mathsf{in}(x_1, a_1) \wedge \cdots \wedge \mathsf{in}(x_p, a_p)$
	$\Rightarrow \mathsf{in}(\mathsf{record}_{s_1,\ldots,s_p}(x_1, \ldots, x_p), \mathsf{recset}_{s_1,\ldots,s_p}(a_1, \ldots, a_p))$
RECSETELIM $(s_1, \ldots, s_p)$	$\forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \{\mathsf{in}(x, \mathsf{recset}_{s_1,\ldots,s_p}(a_1, \ldots, a_p))\}$
	$\mathsf{in}(x, \mathsf{recset}_{s_1,\ldots,s_p}(a_1, \ldots, a_p))$
	$\Rightarrow x = \mathsf{record}_{s_1,\ldots,s_p}(\mathsf{fcnapp}(x, \mathsf{str}_{s_1}), \ldots, \mathsf{fcnapp}(x, \mathsf{str}_{s_p}))$

Sequences

Operator	Type
Seq	$\iota \to \iota$
Len	$\iota \to \iota$
Cat	$\iota \times \iota \to \iota$
Append	$\iota \times \iota \to \iota$

Operator	Type
Head	$\iota \to \iota$
Tail	$\iota \to \iota$
Subseq	$\iota \times \iota \times \iota \to \iota$
$\mathsf{Selectseq}_T^\star$	$\iota \times \iota^n \to \iota$

$(\star)\ T : \iota^{n+1} \to o$

SeqIntro
$$\begin{aligned}&\forall a^{\iota}, s^{\iota} : \{\mathsf{in}(s, \mathsf{Seq}(a))\}\\ &\quad \wedge \mathsf{isafcn}(s)\\ &\quad \wedge \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})\\ &\quad \wedge \forall i^{\iota} : \mathsf{in}(i, \mathsf{domain}(s)) \Leftrightarrow \wedge\, \mathsf{in}(i, \mathsf{Int})\\ &\qquad\qquad\qquad\qquad\qquad\quad \wedge\, 1 \leq \mathsf{proj}_{\mathsf{int}}(i)\\ &\qquad\qquad\qquad\qquad\qquad\quad \wedge\, \mathsf{proj}_{\mathsf{int}}(i) \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))\\ &\quad \wedge \forall i^{\mathsf{int}} : 1 \leq i \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i)), a)\\ &\quad \Rightarrow \mathsf{in}(s, \mathsf{Seq}(a))\end{aligned}$$

SeqElim$_1$
$$\begin{aligned}&\forall a^{\iota}, s^{\iota} : \{\mathsf{in}(s, \mathsf{Seq}(a))\}\\ &\quad \mathsf{in}(s, \mathsf{Seq}(a)) \Rightarrow \wedge\, \mathsf{isafcn}(s)\\ &\qquad\qquad\qquad\qquad \wedge\, \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})\\ &\qquad\qquad\qquad\qquad \wedge\, \mathsf{domain}(s) = \mathsf{range}(\mathsf{cast}_{\mathsf{int}}(1), \mathsf{Len}(s))\end{aligned}$$

SeqElim$_2$
$$\forall a^{\iota}, s^{\iota}, i^{\mathsf{int}} : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i))\}$$
$$\mathsf{in}(s, \mathsf{Seq}(a)) \wedge 1 \leq i \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) \Rightarrow \mathsf{in}(\mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i)), a)$$

CatTyping
$$\begin{aligned}&\forall a^{\iota}, s^{\iota}, t^{\iota} : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{Cat}(s, t)\}\\ &\qquad\qquad \{\mathsf{in}(t, \mathsf{Seq}(a)), \mathsf{Cat}(s, t)\}\\ &\quad \mathsf{in}(s, \mathsf{Seq}(a)) \wedge \mathsf{in}(t, \mathsf{Seq}(a)) \Rightarrow \mathsf{in}(\mathsf{Cat}(s, t), \mathsf{Seq}(a))\end{aligned}$$

CatLen
$$\begin{aligned}&\forall s^{\iota}, t^{\iota} : \{\mathsf{Cat}(s, t)\}\\ &\quad \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat}) \wedge \mathsf{in}(\mathsf{Len}(t), \mathsf{Nat})\\ &\quad \Rightarrow \mathsf{Len}(\mathsf{Cat}(s, t)) = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) + \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(t)))\end{aligned}$$

CatApp$_1$
$$\begin{aligned}&\forall s^{\iota}, t^{\iota}, i^{\mathsf{int}} : \{\mathsf{fcnapp}(\mathsf{Cat}(s, t), \mathsf{cast}_{\mathsf{int}}(i))\}\\ &\qquad\qquad \{\mathsf{Cat}(s, t), \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i))\}\\ &\quad \wedge \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat}) \wedge \mathsf{in}(\mathsf{Len}(t), \mathsf{Nat})\\ &\quad \wedge 1 \leq i \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))\\ &\quad \Rightarrow \mathsf{fcnapp}(\mathsf{Cat}(s, t), \mathsf{cast}_{\mathsf{int}}(i)) = \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i))\end{aligned}$$

CatApp$_2$
$$\begin{aligned}&\forall s^{\iota}, t^{\iota}, i^{\mathsf{int}} : \{\mathsf{fcnapp}(\mathsf{Cat}(s, t), \mathsf{cast}_{\mathsf{int}}(i))\}\\ &\quad \wedge \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat}) \wedge \mathsf{in}(\mathsf{Len}(t), \mathsf{Nat})\\ &\quad \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) + \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(t)) \wedge \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) < i\\ &\quad \Rightarrow \mathsf{fcnapp}(\mathsf{Cat}(s, t), \mathsf{cast}_{\mathsf{int}}(i)) = \mathsf{fcnapp}(t, \mathsf{cast}_{\mathsf{int}}(i - \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))))\end{aligned}$$

AppendTyping $\forall a^\iota, s^\iota, x^\iota : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{Append}(s, x)\}$
$\mathsf{in}(s, \mathsf{Seq}(a)) \wedge \mathsf{in}(x, a) \Rightarrow \mathsf{in}(\mathsf{Append}(s, x), \mathsf{Seq}(a))$

AppendLen $\forall s^\iota, x^\iota : \{\mathsf{Append}(s, x)\}$
$\mathsf{in}(\mathsf{Len}(s), \mathsf{Nat}) \Rightarrow \mathsf{Len}(\mathsf{Append}(s, x)) = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) + 1)$

AppendApp$_1$ $\forall s^\iota, x^\iota, i^{\mathsf{int}} : \{\mathsf{fcnapp}(\mathsf{Append}(s, x), \mathsf{cast}_{\mathsf{int}}(i))\}$
$\{\mathsf{Append}(s, x), \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i))\}$
$\wedge\ \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})$
$\wedge\ 1 \leq i \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))$
$\Rightarrow \mathsf{fcnapp}(\mathsf{Append}(s, x), \mathsf{cast}_{\mathsf{int}}(i)) = \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i))$

AppendApp$_2$ $\forall s^\iota, x^\iota : \{\mathsf{Append}(s, x)\}$
$\mathsf{in}(\mathsf{Len}(s), \mathsf{Nat}) \Rightarrow \mathsf{fcnapp}(\mathsf{Append}(s, x), \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) + 1)) = x$

HeadDef $\forall s^\iota : \{\mathsf{Head}(s)\}$
$\mathsf{Head}(s) = \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(1))$

TailTyping $\forall a^\iota, s^\iota : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{Tail}(s)\}$
$\wedge\ \mathsf{in}(s, \mathsf{Seq}(a))$
$\wedge\ \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) \neq 0$
$\Rightarrow \mathsf{in}(\mathsf{Tail}(s), \mathsf{Seq}(a))$

TailLen $\forall s^\iota : \{\mathsf{Tail}(s)\}$
$\wedge\ \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})$
$\wedge\ \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) \neq 0$
$\Rightarrow \mathsf{Len}(\mathsf{Tail}(s)) = \mathsf{cast}_{\mathsf{int}}(\mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) - 1)$

TailApp $\forall s^\iota, i^{\mathsf{int}} : \{\mathsf{fcnapp}(\mathsf{Tail}(s), \mathsf{cast}_{\mathsf{int}}(i))\}$
$\wedge\ \mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})$
$\wedge\ \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) \neq 0$
$\wedge\ 1 \leq i \wedge i \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s)) - 1$
$\Rightarrow \mathsf{fcnapp}(\mathsf{Tail}(s), \mathsf{cast}_{\mathsf{int}}(i)) = \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}(i + 1))$

SubseqTyping
$$\forall a^{\iota}, s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}} : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y))\}$$
$$\wedge\, \mathsf{in}(s, \mathsf{Seq}(a))$$
$$\wedge\, 1 \leq x$$
$$\wedge\, y \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))$$
$$\Rightarrow \mathsf{in}(\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y)), \mathsf{Seq}(a))$$

SubseqLen
$$\forall s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}} : \{\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y))\}$$
$$\wedge\, x \leq y + 1$$
$$\Rightarrow \mathsf{Len}(\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y))) = \mathsf{cast}_{\mathsf{int}}((y+1) - x)$$
$$\wedge\, x > y + 1 \Rightarrow \mathsf{Len}(\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y))) = \mathsf{cast}_{\mathsf{int}}(0)$$

SubseqApp
$$\forall s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}}, z^{\mathsf{int}} : \{\mathsf{fcnapp}(\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y)), \mathsf{cast}_{\mathsf{int}}(z))\}$$
$$\wedge\, 1 \leq x$$
$$\wedge\, 1 \leq z \wedge z \leq (y+1) - x$$
$$\Rightarrow \mathsf{fcnapp}(\mathsf{Subseq}(s, \mathsf{cast}_{\mathsf{int}}(x), \mathsf{cast}_{\mathsf{int}}(y)), \mathsf{cast}_{\mathsf{int}}(z))$$
$$= \mathsf{fcnapp}(s, \mathsf{cast}_{\mathsf{int}}((z + x) - 1))$$

SelectseqTyping $(T : \iota^{n+1} \to o)$
$$\forall c_1^{\iota}, \ldots, c_n^{\iota}, a^{\iota}, s^{\iota} : \{\mathsf{in}(s, \mathsf{Seq}(a)), \mathsf{Selectseq}_T(s, c_1, \ldots, c_n)\}$$
$$\mathsf{in}(s, \mathsf{Seq}(a)) \Rightarrow \mathsf{in}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n), \mathsf{Seq}(a))$$

SelectseqLen $(T : \iota^{n+1} \to o)$
$$\forall c_1^{\iota}, \ldots, c_n^{\iota}, s^{\iota} : \{\mathsf{Selectseq}_T(s, c_1, \ldots, c_n)\}$$
$$\mathsf{in}(\mathsf{Len}(s), \mathsf{Nat})$$
$$\Rightarrow \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n))) \leq \mathsf{proj}_{\mathsf{int}}(\mathsf{Len}(s))$$

SelectseqApp $(T : \iota^{n+1} \to o)$
$$\forall c_1^{\iota}, \ldots, c_n^{\iota}, s^{\iota}, x^{\iota} : \{\mathsf{fcnapp}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n), x)\}$$
$$\mathsf{in}(x, \mathsf{domain}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n)))$$
$$\Rightarrow T(\mathsf{fcnapp}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n), x))$$

SelectseqNil $(T : \iota^{n+1} \to o)$
$$\mathsf{Selectseq}_T(\mathsf{tup}_0, c_1, \ldots, c_n) = \mathsf{tup}_0$$

SelectseqAppend $(T : \iota^{n+1} \to o)$
$$\forall c_1^{\iota}, \ldots, c_n^{\iota}, s^{\iota}, x^{\iota} : \{\mathsf{Selectseq}_T(\mathsf{Append}(s, x), c_1, \ldots, c_n)\}$$
$$\wedge\, T(x, c_1, \ldots, c_n) \Rightarrow$$
$$\mathsf{Selectseq}_T(\mathsf{Append}(s, x), c_1, \ldots, c_n)$$
$$= \mathsf{Append}(\mathsf{Selectseq}_T(s, c_1, \ldots, c_n), x)$$
$$\wedge\, \neg T(x, c_1, \ldots, c_n) \Rightarrow$$
$$\mathsf{Selectseq}_T(\mathsf{Append}(s, x), c_1, \ldots, c_n)$$
$$= \mathsf{Selectseq}_T(s, c_1, \ldots, c_n)$$

TupSeqTyping (n)
$$\forall a^{\iota}, x_1^{\iota}, \ldots, x_n^{\iota} : \{\mathsf{in}(x_1, a), \ldots, \mathsf{in}(x_n, a), \mathsf{tup}_n(x_1, \ldots, x_n)\}$$
$$\mathsf{in}(x_1, a) \wedge \cdots \wedge \mathsf{in}(x_n, a) \Rightarrow \mathsf{in}(\mathsf{tup}_n(x_1, \ldots, x_n), \mathsf{Seq}(a))$$

TupSeqLen (n)
$$\forall x_1^{\iota}, \ldots, x_n^{\iota} : \{\mathsf{tup}_n(x_1, \ldots, x_n)\}$$
$$\mathsf{Len}(\mathsf{tup}_n(x_1, \ldots, x_n)) = \mathsf{cast}_{\mathsf{int}}(n)$$