Translation Validation of Tensor Compilers

Basile Clément

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

Validation de Traduction pour Compilateurs de Tenseurs

Soutenue par
Basile Clément
Le 9 Septembre 2022

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Xavier Rival Directeur de Recherche, Inria	*Président du Jury*
Christophe Alias Chargé de Recherche HDR, Inria	*Rapporteur*
George Necula (absent) Ingénieur, Google	*Rapporteur*
Corinne Ancourt Chercheuse HDR, Mines ParisTech	*Examinatrice*
Sandrine Blazy Professeure, Université de Rennes 1	*Examinatrice*
Jonathan Ragan-Kelley Assistant Professor, MIT	*Examinateur*
Xavier Leroy Professeur, Collège de France	*Directeur de thèse*
Albert Cohen Chercheur, Google	*Co-encadrant*

ENS | PSL

École Normale Supérieure

Thèse de Doctorat de l'Université PSL
Spécialité : Informatique

Validation de Traduction pour Compilateurs de Tenseurs

Translation Validation of Tensor Compilers

Basile Clément

Soutenue le 9 Septembre 2022
Préparée sous la direction de
Xavier Leroy (Collège de France)
et
Albert Cohen (Google)

Rapporteurs
Christophe Alias (Inria)
George Necula* (Google)

Examinateurs
Corinne Ancourt (Mines ParisTech)
Sandrine Blazy (Université de Rennes 1)
Jonathan Ragan-Kelley (MIT)
Xavier Rival (Inria), Président du Jury

*Absent lors de la soutenance

Acknowledgements

A PhD thesis is, ultimately, a very personal work: in many ways, I am leaving a part of my very self in this document and the work that lead up to it. It is not, however, an *individual* work: many others have had an impact on both this thesis and me, through their work and our interactions. I cannot possibly do justice to all that deserve to be cited here, but I will try my best.

Among those that I couldn't possibly forget are my advisors, without whom this journey would never have been possible. Albert Cohen has been with me from the first discussions on the original topic of the thesis to the last lines of this manuscript. I couldn't have done half of this work without your guidance, your enthusiasm, and your knowledge of the polyhedral model and the surrounding areas. Thank you, also, for your persistent support during the tough times of this adventure. I started working with Xavier Leroy later, when, at the start of my second year, I re-oriented my topic towards the verification of tensor compilers. I am honored that you accepted to take me as a student. Your humility in spite of your vast knowledge, your curiosity, and your sharp but kind questions have made these PhD years a rewarding experience. You have helped me achieve a greater understanding of my work, and have always been a stable and reassuring support when it seemed things were falling apart. There is no doubt this manuscript would not exist without you. I'd also like to acknowledge Francesco Zappa Nardelli, who was my director during the first year of this PhD, before he left for bluer pastures in the industry. Thank you for encouraging me to get out of my comfort zone and explore different directions beyond my first intuitions.

Thank you to all the members of my jury for coming to the defense, and in particular to Christophe Alias and George Necula who have honored me by accepting to report on my manuscript. Your proofreading, remarks, and questions have helped improve the quality and precision of the manuscript.

Research has been on my mind since I was a little kid and Gérard Berry, who was my first Computer Science teacher when my age fit on a 3 or 4 bit counter, deserves to be mentioned here. I did not realize my luck at the time, but you were influential nurturing my intereset for research and in instilling in me a love for computers and what is not their science — I may have not followed that path if not for you.

The first year of my PhD was spent in the PARKAS team at the École Normale Supérieure, and they deserve my gratitude for making that year memorable. Thank you, Marc, for hosting me in your team and for always being supportive, and thank you, Timothy, for your kind and encouraging presence. The life at PARKAS was also animated by its PhD students, postdocs and other engineers; thank you, Andi, Ulysse, Nicolas, Guillaume, Lélio, and Ismaïl, for the scientific discussions and the good times during lunch and other breaks.

From the second year onwards, I was supposed to move to the Cambium team at Inria — but really, I moved to my room, because Covid-19 hit. The team's weekly video calls were a welcome moment of conviviality in bleak times. I am glad we later got to spend more time together. Thank you, François, Didier, Luc, Jean-Marie, Damien, Florian, and Sébastien. The PhD students deserve a separate mention, as we shared the same struggles: thank you Nathanaëlle, Clément, Alexandre, Glen, Thomas, Léo, Paulo. I wish you the same success, for those of you that are still in this boat.

Thanks go to Christine, when I was at PARKAS, and then Hélène, at Cambium, their respective team's assistants, for their help in navigating various administrative conundrums.

As a PhD student, I found some of the fuel needed to complete the dissertation in my friends. There is the old Parisian guard that is still young in my heart: Anaël, Ulysse, Julie, Abel, Kenji, Thomas, Nofar, Najib, Annalí, Ambre. Thank you for being there. Some of my most important friendships are with people that are best named by their pseudonyms. Thank you a3nm, Corbeau, DrNo, Evarin, foncteur, loutre, Mi, Milton, Nathanaëlle, Tito, tobast. Thank you Robin for putting up with me during the years we spent as roommates, it has been a pleasure. Orphée, your strength, your talent, and your kindness are inspiring; thank you. Lwenn, you are a precious friend. Thank you for believing in me as much as I believe in you.

Finally, thanks go to my parents and my sister for your support during these years. I am sorry I moved so far from home. Thank you, Marc and Pascale, for hosting me during the lockdowns and allowing me to keep my sanity, and thank you, Luc, Sonia, Noé, Damien, and Émilie, for making them more lively.

Abstract

Tensor compilers are used in domains such as image processing and deep learning to generate efficient low-level code from high-level specifications on multidimensional tensors. After the application of both loop transformations and algebraic simplifications to the specification, the resulting low-level code can have a drastically different structure. This makes the formal verification of tensor compilers an arduous task, unsuitable for standard bisimulation techniques. I propose a new method for the verification of tensor compilers in the presence of loop and algebraic transformations. This method draws inspiration from polyhedral techniques for program representation, and relies on a refinement mapping from assignments in the low-level code to tensor definition in the specifications provided by the tensor compiler. Each run of the compiler is verified by an independent verification tool implemented in OCaml, making the method an instance of translation validation. This verification tool is tested on Halide, an industrial-grade tensor compiler.

Contents

List of Figures

List of Tables

Présentation

Note to non-French-speaking readers: this section is a substantial summary of the findings of this thesis in French. All the content within is included in the full version of the thesis in English, that is found on page 1, after this summary.

Cette partie présente un résumé substantiel en français de la thèse, rédigée en anglais. Chaque chapitre de la thèse est résumé en une section de quelues pages en suivant l'organisation du document originel. Le lecteur souhaitant des preuves, détails ou références plus précises est invité à se reporter au(x) chapitre(s) correspondant(s) de la version complète en anglais.

1 Introduction

Imaginons un langage de spécification pour des tableaux multidimensionnels potentiellement infinis, que nous appellerons *tenseurs*. Ce langage définit les tenseurs par des équations mathématiques, lues comme des écritures uniques, et implicitement quantifiées sur le domaine d'indexation du tenseur. On peut ainsi représenter, par exemple, un produit extérieur de vecteurs par la spécification :

$$C(i, j) = A(i) \times B(j)$$

Ces spécifications sont ensuites compilées en programmes impératifs pouvant être exécutés, où les tenseurs ont été remplacés par des tableaux impératifs représentant un sous-ensemble de leur domaine. À la source des travaux de cette thèse se trouve l'intuition qu'il est possible pour le compilateur de préserver une relation explicite entre tableaux et tenseurs.

La spécification ci-dessus pourrait, une fois compilée, se transformer en le code suivant :

```
for i0 = 0 to (N + 3) / 4 - 1 do
  for j = 0 to M - 1 do
    for i1 = 0 to 3 do
      let i = min(i0 * 4, N - 4) + i1 in
      c[i, j] := b[j] * a[i]
```

On remarquera à la fois des transformations de structures (ici la boucle extérieure sur `i` a été *tuilée* d'un facteur 4) et des transformations sémantiques (ici la commutativité de la multiplication a été appliquée). Les paramètres N et M représentent la taille des tableaux `a` et `b`, et ne sont pas connus statiquement : ils seront fournis par l'utilisateur au moment de l'exécution.

Dans ce cas simple, il est possible de se convaincre que le programme engendré est équivalent à la spécification, au sens où si le programme est exécuté dans une mémoire où les `a[i]` contiennent $A(i)$ et où les `b[j]` contiennent $B(j)$, alors après l'exécution, les `c[i, j]` contiendront $C(i, j)$. La preuve repose sur le fait que, lorsque l'on écrit dans `c[i, j]`, la valeur écrite est toujours $B(j) \times A(i)$, qui est égal à $A(i) \times B(j) = C(i, j)$ par commutativité de la multiplication ; de plus, l'ensemble des `c[i, j]` écrits par le programme est l'ensemble des `c[i, j]` pour $0 \leqslant i < N$ et $0 \leqslant j < M$, ce dont on se convainc à l'aide d'un peu d'arithmétique.

Cette approche fonctionne dans un cas simple comme celui ci, mais échoue dès lors que le programme contient une *récurrence*, c'est-à-dire dès lors que la valeur écrite par une itération d'une boucle dépend de la valeur écrite par l'itération précédente. En présence de récurrences, on ne connait plus statiquement la valeur stockée dans une case de tableau, un problème bien connu dans le cadre de l'équivalence de programme et pour lequel des techniques opportunistes basées sur les clotures transitives ou les enveloppes convexes ont été développées.

À l'inverse, dans cette thèse, je propose de demander un peu plus de travail au compilateur pour générer une annotation légère indiquant, en termes de spécification, la valeur que le compilateur pense être écrite par un assignement. Je nomme ces annotations des *expressions prophétiques* (car elles prédisent, en quelque sorte, la valeur qui sera calculée avant qu'elle soit effectivement

calculée), et je les note entre accolades avant une affectation. En présence de récurrences, comme dans l'implémentation d'une multiplication de matrice avec un accumulateur R, on obtient un programme annoté comme suit :

```
for i0 = 0 to (N + 3) / 4) - 1 do
  for j = 0 to M - 1 do
    for i1 = 0 to 3 do
      let i = min(i0 * 4, N - 4) + i1 in
      r {0} := 0
      for k = 0 to P - 1 do
  r {R(i,j,k)} := r + b[k, j] * a[i, k]
      c[i, j] {C(i,j)} := r
```

Les expressions prophétiques sont indiquées en violet afin de les distinguer du reste du code. Ces expressions prophétiques permettent de casser le cycle de dépendances : lorsque l'on exécute l'affectation `r := r + b[k, j] * a[i, k]`, on peut maintenant utiliser l'annotation prophétique de l'itération précédente pour en déduire que la lecture de `r`, dans le membre droit, vaut $R(i, j, k-1)$.

Contributions

Cette thèse découlent de la remarque mentionnée précédemment : certaines difficultés théoriques à la validation de compilateurs de tenseurs disparaissent lorsque le compilateur est capable de fournir des annotations prophétiques reliant les écritures dans le code engendré avec des expressions de tenseurs dans la spécification en entrée. Les contributions de la thèse explorent donc les conséquences de ces annotations, plus précisément, cette thèse fournit les contributions suivantes :

- Le développement et la formalisation de SCHED, un langage intermédiaire dédié à la validation de compilateurs de tenseurs et proche du langage intermédiaire STMT utilisé par Halide et d'autres compilateurs de tenseurs, étendu avec des annotations prophétiques.

- Le développement, la formalisation et l'implémentation d'un validateur pour programmes SCHED en rapport à une spécification exprimée

sous la forme de systèmes récurrents d'équations affines (SREA), une représentation générique de programmes.

- Une formalisation nouvelle de la sémantique de Halide sous forme d'un système d'équations, ainsi qu'une réduction de cette sémantique vers la sémantique standard des SREA.

- L'instrumentation du compilateur Halide afin d'annoter son langage intermédiaire STMT avec des annotations prophétiques, et l'implémentation des traductions de STMT annoté vers SCHED et de Halide vers un SREA.

- Une évaluation expérimentale des outils mentionnés ci-dessus sur un ensemble de programmes extraits des benchmarks officiels de Halide, et leur comparaison avec ISA, état de l'art de l'équivalence de programmes affines.

- La formalisation (mais pas l'implémentation) d'une extension au système décrit ci-dessus pour permettre le réordonnancement des operations au sein d'une réduction, une primitive importante en calcul numérique.

Organisation du chapitre

Ce résumé est organisé suivant la même structure que la version complète de la thèse en anglais. Il est composé de plusieurs sections correspondant aux chapitres de la version anglaise.

La première section introductive s'achève ici, après avoir présenté le contexte et les motivations des travaux réalisés durant cette thèse.

La seconde section présente les représentations utilisées par les compilateurs de programmes numériques que nous chercherons ensuite à vérifier, avec un intérêt tout particulier pour le modèle polyédrique qui a inspiré mes travaux.

La troisième section est fortement raccourcie dans ce résumé en français : le chapitre correspondant de la version complète présente les ensembles et relations de Presburger du modèle polyédrique tels qu'implémentés par

`isl` [112], et le lecteur intéressé est invité à s'y référer.

La quatrième section présente les idées derrière le language impératif Sched qui forment le cœur de l'aproche proposée dans cette thèse. Sched est spécifié par une sémantique et un système de types et d'effets, prouvé cohérent dans la version complète en anglais.

La cinquième section complète la présentation et étends le système de types et d'effets de Sched avec un évaluateur symbolique basé sur les *expressions prophétiques* mentionnées dans l'introduction. Cet évaluateur symbolique produit des conditions de vérifications qui impliquent la correction du code engendré relativement à la spécification de l'utilisateur.

La sixième section mentionne les résultats expérimentaux obtenus lors de l'implémentation des méthodes développées dans les précédentes sections, et en particulier son application au compilateur industriel Halide.

La septième section explique comment étendre les idées de cette thèse à la validation de réductions, une primitive permettant de réordonner les calculs lors de l'application répétée d'un opérateur associatif et commutatif.

La huitième section est omise de ce résumé en français : le chapitre correspondant dans la version anglaise compare l'approche de ce manuscrit avec d'autres travaux en validation de traduction publiés en langue anglaise.

2 Représentation des programmes comportant boucles et tableaux

Cette section propose un tour d'horizon et un historique rapide des techniques de compilation des programmes numériques intensifs opérants sur des tableaux multi-dimensionnels, un domaine riche de décennies de recherche. Puisque nous nous intéressons dans ce manuscrit principalement à la *vérification* des transformations de programmes effectuées par les compilateurs de tenseurs, ce chapitre se concentre sur les techniques permettant de représenter et d'appliquer des transformations : en particulier, les techniques d'optimisations

ne seront que brièvement mentionnées.

Pour plus de détails sur les techniques de représentation mentionnées ici, et notamment un historique du modèle polyédrique, le lecteur intéressé est invité à se référer au chapitre 2 de la version complète en anglais de cette thèse.

2.1 Le modèle polyédrique

Le modèle polyédrique est une représentation des programmes en tant que *graphe de calcul* étendu, dont les nœuds sont des *instances d'instructions*, c'est-à-dire les instructions du programmes indexées par les valeurs des itérateurs de boucles qui les contiennent. Bien que n'utilisant pas la représentation de programmes du modèle polyédrique, les techniques décrites dans ce manuscrit réutilisent certaines des intuitions du modèle.

Ensembles d'instances La première idée du modèle polyédrique est de remarquer que dans un programme de boucles impératives comme par exemple la multiplication de matrice :

```
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    for k = 0 to P - 1 do
      c[i, j] += a[i, k] * b[k, j]
```

le concept classique d'instruction (ou, plus grossièrement, de ligne de code) n'est pas suffisant pour parler du comportement dynamique du programme. Il faut plutôt parler d'instructions *paramétrées* par leur contexte afin de distinguer les "instances" de l'assignement pour des valeurs de i, j et k différentes. Par exemple, si on dénote par $I(i,j,k)$ l'assignement $c[i,j] += a[i,k] * b[k,j]$, on pourra dire qu'il est possible d'exécuter $I(i,j,k)$ et $I(i',j',k')$ en parallèle dès lors que $(i,j) \neq (i',j')$ sans impacter la sémantique du programme.

Les instructions, ou lignes de code, sont donc représentées dans ce modèle par leurs instances, des objets indexés par un espace multi-dimensionnel à coordonnées entières représentant les boucles autour de l'instruction. Les instances peuvent ainsi être vus comme des points dans cet espace multi-dimensionnel,

et le modèle est ainsi parfois connu sous le nom de *modèle géométrique*. Le nom de modèle polyédrique vient d'une restriction supplémentaire sur les ensembles d'instances – et donc les programmes – représentables : afin de rendre décidables les problèmes de transformations de programmes et d'optimisation, les premiers travaux du domaine autorisent uniquement expressions affines (c'est-à-dire de la forme $a_1x_1 + \cdots + a_nx_n + c$ avec $a_1, \ldots, a_n$ et c des constantes entières et $x_1, \ldots, x_n$ des variables de boucles ou des paramètres) dans les bornes de boucles et les indices de tableaux. D'un point de vue géométrique, cela correspond à autoriser uniquement des ensembles d'instances qui sont exactement les points à coordonnées entières contenus dans un polyèdre.

Dans ses manifestations modernes comme la bibliothèque isl utilisée dans cette thèse, le modèle polyédrique se généralise à des ensembles d'instances représentés par des unions finies de polyèdres, ce qui correspond à autoriser des expressions affines par morceaux (où les conditions sont elle-mêmes affines), avec la possibilité d'effectuer des divisions par un facteur entier constant. On appelle ces expressions des expressions *quasi-affines par morceaux*. La grammaire des programmes gérés par une telle approche est :

$$s ::= \text{for } i < e; \text{do } s \mid s\ ;\ s \mid \text{if } e \text{ then } s \text{ else } s \mid I(e_1, \ldots, e_n) \mid \text{skip}$$

Le symbole I représente un morceau de programme arbitraire sans variables libres et paramétré uniquement par ses arguments $e_1, \ldots, e_n$. Les expressions apparaissant dans les bornes de boucles, les conditionnelles, et les arguments des instructions sont des expressions quasi-affines par morceaux.

Ordre d'exécution La représentation d'un programme par un ensemble d'instances dans le modèle polyédrique n'est pas suffisante pour capturer totalement la sémantique d'un programme impératif, car un ensemble est par définition non ordonné. Pour pallier ce problème, le modèle polyédrique utilise un ordre partiel $<$ entre les instances d'une même instruction mais aussi d'instructions différentes. Si u et v sont deux instances, alors $u < v$ indique que u doit être exécutée avant v pour obtenir la sémantique correcte. Cet ordre d'exécution partiel représente une exécution parallèle de l'ensemble d'instances : deux instances incomparables peuvent être exécutées en parallèle.

Afin d'éviter l'explosion combinatoire induite par une représentation explicite de l'ordre d'exécution, le modèle polyédrique utilise la notion d'*ordonnanceur*.

Un schedule θ est une fonction associant à chaque instance un point dans un unique espace multi-dimensionnel appelé le *domaine d'ordonnancement*, muni de l'ordre lexicographique sur les entiers. L'ordre d'exécution est obtenu en composant la fonction d'ordonnancement avec l'ordre lexicographique sur les entiers. Afin de permettre plus de flexibilité dans la représentation du parallélisme, les approches modernes autorisent à marquer certaines dimensions dans le domaine d'ordonnancement comme *parallèles* : on utilise pour ces dimensions l'ordre partiel induit par l'égalité (i.e. deux valeurs différentes sont toujours incomparables) plutôt que l'ordre usuel sur les entiers.

Génération de code Le modèle polyédrique se prête à l'optimisation de programmes, qui doivent ensuite être convertis à nouveau vers une représentation textuelle compatible avec des outils de compilation classiques. Cette partie du modèle polyédrique s'appelle la *génération de code*, et reconstruit un arbre de syntaxe en suivant la structure des dimensions du domaine d'ordonnancement, chaque dimension du domaine d'ordonnancement devenant une boucle dans le code engendré. La génération de code est une composante cruciale du modèle polyédrique car une mauvaise génération de code peut introduire des inefficacités qui contrebalancent les gains obtenus grâce à la réorganisation du code permise par le modèle.

Relations d'accès et analyse de dépendences Il faut également mentionner que le modèle polyédrique permet également de représenter des relations (affines) entre paires d'instances, mais aussi entre instances et d'autres objets comme des cases mémoires. Le modèle polyédrique permet de capturer, de façon exacte en présence d'expressions quasi-affines ou approchée dans le cas général, l'ensemble des cases mémoires lues et écrites par une instance donnée. Ces ensembles de cases mémoires sont représentées sous forme de *relations d'accès* liant par des expressions quasi-affines une instance d'instruction indicée par ses itérateurs à un ensemble de cases mémoires indicées de façon multidimensionnelle.

Ces relations d'accès entre instructions et cases mémoires peuvent être utilisées pour effectuer une analyse de dépendance précise, indiquant de façon exacte quel instance est la source en écriture d'une lecture donnée. Cette analyse de dépendance permet de déterminer des contraintes sur l'ordonnancement

à respecter pour que la sémantique du programme soit préservée à travers une transformation. On peut également l'utiliser pour effectuer des transformations mémoire, comme remplacer une écriture répétée dans un scalaire `b[] = a[i] * a[i]` par l'utilisation d'un tableau `b[i] = a[i] * a[i]` en effectuant les transformations correspondantes lors des lectures.

2.2 Systèmes Récurrents d'Équations Affines

Les systèmes récurrents d'équations affines (ou SREA) prédatent et forment les fondations théoriques du modèle polyédrique. Là où le modèle polyédrique s'est concentré sur des représentations de programmes impératifs, les SREA sont des systèmes d'équations mathématiques où les équations sont de la forme :

$$\forall x_1, \ldots, x_{n_A},\ D(x_1, \ldots, x_{n_A}) \Rightarrow A(x_1, \ldots, x_{n_A}) = e$$

où D est une condition affine sur les $x_1, \ldots, x_{n_A}$, A est un nom de fonction abstrait d'arité n_A, et e est une expression arbitraires où tous les arguments des fonctions abstraites définies par le système sont des expressions affines de $x_1, \ldots, x_{n_A}$.

Les SREA ont une expressivité équivalente à celle du modèle polyédrique, et sont utilisés comme spécification dans le compilateur ALPHA de Verge, Mauras et Quinton [119]. Leur nature équationnelle en fait de bons candidats pour la spécification de programmes affines, et nous utiliserons des SREAS comme language de spécification dans cette thèse pour s'affranchir des détails d'implémentation du compilateur à vérifier.

2.3 Combinateurs fonctionnels et règles de réécriture

Une ligne de recherche assez différente dans le monde de l'optimisation est celle des règles de réécriture, généralement basées sur des combinateurs fonctionnels. Dans ces approches, le programme est successivement transformé à partir de règles de réécritures plus ou moins atomiques qui peuvent être prouvées correctes de façon indépendante jusqu'à obtenir un programme final. L'absence

de vision globale provenant de la nécéssité d'effectuer les réécritures pas à pas rend plus difficile la construction d'optimiseurs automatiques performants pour ces approches.

2.4 Le modèle de Halide

Les compilateurs optimisant basés sur le modèle polyédrique, en partie à cause de sa vaste expressivité, peinent à trouver automatiquement des optimisations aussi efficaces que celles d'un expert humain. Cette observation a mené au développement de *langages d'ordonnancement* permettant à un expert de décrire un ordonnancement à l'aide de primitives de base de façon plus efficace et plus ergonomique qu'en réécrivant manuellement le code. Le langage Halide, initialement développé par Ragan-Kelly et al. dans le cadre du traitement d'image, est fondé sur cette idée. Halide est aujourd'hui un langage industriel qui est utilisé dans la partie expérimentale de cette thèse.

Halide s'inspire à certains égards du modèle polyédrique, mais – contrairement à des outils pré-existants comme CHiLL ou URUK – n'en fait pas directement usage, préférant un compromis d'expressivité différent. La communauté polyédrique s'est ensuite inspirée de Halide pour développer Tiramisu, un compilateur polyédrique avec un langage d'ordonnancement proche de celui de Halide.

Algorithmes Dans le modèle de Halide, le code fourni par l'utilisateur est séparé en deux composantes : un *algorithme*, qui représente les calculs à effectuer sur des tenseurs, i.e. des tableaux multidimensionnels non bornés, nommés `Func`s (notamment dans les bibliothèques C++ et Python avec lesquelles l'utilisateur interagit) ou fonctions dans la terminologie de Halide.

Ainsi l'exemple traditionnel de Halide est celui d'un filtre 3×3 non-normalisé, comme on le trouve par exemple dans "Halide: decoupling algorithms from schedules for high-performance image processing" [83].

```
Func bh, bv; Var x, y;
ImageParam in(UInt(8), 2);

bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y + 1))/3;
```

L'algorithme prend en argument une image en niveaux de gris `in` et calcule une moyenne locale horizontale dans `bh`, puis verticale dans `bv`, qui est la sortie. Les dimensions de l'entrée `in` et de la sortie `bv` ne sont pas spécifiées ici : elles ne seront connus qu'à l'exécution, lorsqu'une image d'entrée concrète sera fournie et qu'une taille pour la sortie `bv` sera demandée. Halide vérifie automatiquement que les tailles d'entrée et de sortie sont compatibles, et calcule les tailles de tableaux intermédiaires pour éviter les débordements. Halide impose des restrictions qui ne sont pas expliquées ici afin de s'assurer que les définitions ne sont pas cycliques et sont uniquement définies en tout point.

Les variables `x` et `y` utilisées ici sont implicitement quantifiées sur l'ensemble des entiers relatifs, sans ordre particulier. Afin d'exprimer des récurrences de longueurs arbitraires, Halide ré-introduit dans ce monde pure une notion impératif de mise à jour en permettant des re-définitions de tenseurs. Ces re-définitions s'appliquent – sémantiquement – après la définition initiale du tenseur, et peuvent utiiser des variables de récurrence (de type `RDom`) qui représentent une itération séquentielle. À nouveau, Halide impose des restrictions qui ne sont pas expliquées ici afin de s'assurer que les tenseurs mis à jour restent bien définis et calculables.

Ordonnanceur Les algorithmes de Halide déterminent les valeurs sémantiques à calculer mais ne déterminent pas *comment* (dans quel ordre) sont calculées ses valeurs. C'est le rôle de l'ordonnanceur (schedule en anglais) qui est écrit dans un language de domaine spécifique à Halide. Ce langage spécifique permet à l'utilisateur de jongler avec différents compromis de localité et d'efficacité de travail. Le langage d'ordonnancement de Halide n'est pas décrit dans ce résumé en français, car il est assez orthogonal aux considérations de cette thèse ; le lecteur intéressé se référera à la version complète en anglais ou aux publications de référence par les auteurs de Halide.

Sémantique et SRAE Halide est un système appliqué, dont la sémantique est principalement décrite en prose et n'est spécifiée que par son unique implémentation. REINKING, BERNSTEIN et RAGAN-KELLEY [89] ont proposé une sémantique de Halide qui interprète les agorithmes comme des programmes séquentiels qui sont transformés par l'ordonnancement. Dans le cadre de cette thèse, et avec l'objectif d'obtenir une réduction vers les SRAE, je propose plutôt de donner une sémantique équationnelle aux algorithmes de Halide, qui permette de nommer les états intermédiaires de l'évaluation. Je propose ici une formalisation des algorithmes de Halide arbitraires, i.e. qui peuvent contenir des expressions non-affines ; la restriction de cette formalisation a des accès de tenseurs et des filtres affines (ou quasi-affines par morceau) peut être interprétée comme un sous-ensemble des SRAE, ce qui est démontré dans la version complète en anglais qui contient également des exemples.

Un algorithme pour un ensemble de tenseurs $\mathcal{S}$ est un tuple $\langle I, P, U, < \rangle$ tel que :

- I est un ensemble de noms, les *tenseurs d'entrées*, qui n'ont pas de définition ;
- P est une fonction qui associe chaque tenseur $A \in \mathcal{S} \backslash I$ à sa *définition pure* P_A, définie plus loin ;
- U est une fonction qui associe chaque tenseur $A \in \mathcal{S} \backslash I$ à une séquence finie (et possiblement vide) $U_A^1, \ldots, U_A^{n_A}$ de *définitions de mise à jour* pour A ;
- $<$ définit un ordre total sur les tenseurs de sortie $\mathcal{S} \backslash I$ qui représente l'ordre textuel de définition des tenseurs.

Une définition pure P_A pour un tenseur A est une équation :

$$\forall x_1, \ldots, x_{n_A},\ A(x_1, \ldots, x_{n_A}) = e$$

où l'expression e à droite de l'égalité ne peut contenir que des accès à des tenseurs d'entrée ou à des tenseurs de sortie A' avec $A' < A$. En particulier, e ne peut pas contenir d'accès à A.

Une définition de mise à jour U_A^i pour un tenseur A est aussi une équation qui contient des variables pures $x_1, \ldots, x_n$ et des variables de récurrence

$y_1, \ldots, y_r$:

$$\forall x_1, \ldots, x_n.\ \text{for}\ y_1 : R_1, \ldots, y_r : R_r.\ \phi \implies A(e_1, \ldots, e_{n_A}) = e$$

Chaque y_i est borné par R_i, un intervalle fini de $\mathbb{Z}$ dont les bornes dépendent uniquement des paramètres du problème, pas des variables $x_1, \ldots, x_n$ ni d'autres variables de récurrence. ϕ est une expression booléenne, appellée *filtre* : la mise à jour n'est effectuée que pour les points où ϕ est vrai.

La condition utilisée par Halide pour s'assurer de la bonne formation de ces définitions se formalise par la *condition de bonne formation* suivante : pour chaque tenseur A d'arité n_A et chaque définition de mise à jour U_A^i avec n variables pures, il doit exister une fonction $\pi : \{1, \ldots, n\} \to \{1, \ldots, n_A\}$ telle que, pour $1 \leqslant j \leqslant n$, et pour chaque accès $A(e'_1, \ldots, e'_{n_A})$ qui apparait dans U_A^i (y compris dans le filtre ou le membre gauche de l'égalité lui-même), on a $e'_{\pi(j)} = x_j$.

Pour définir la sémantique d'un tel algorithme, j'introduis la notion d'*indice de niveau* ψ, défini par la grammaire :

$$\begin{aligned}\psi ::= {} & \mathcal{P} \mid \mathcal{U}_n \mid \mathcal{F} \\ & \mid \mathcal{U}_n(n_1, \ldots, n_m)\end{aligned}$$

où $n > 0$ est en entier strictement positif et $n_1, \ldots, n_m$ sont des entiers arbitraires. Les indices de niveau représentent les différentes "versions" du tenseur à travers sa définition : $\mathcal{P}$ est le niveau *pur* qui se réfère à la définition initiale du tenseur ; $\mathcal{F}$ est le niveau *final* après que toutes les mises à jour aient été appliquées ; $\mathcal{U}_n$ est un niveau de mise à jour après avoir appliqué la n-ème mise à jour ; et enfin $\mathcal{U}_n(n_1, \ldots, n_m)$ est un niveau de mise à jour partiel où les $n_1, \ldots, n_m$ représentent les valeurs des variables de récurrence.

L'ordre naturel sur les indices de niveaux est :

$$\mathcal{P} \prec \cdots \prec \mathcal{U}_{n-1} \prec \mathcal{U}_n(n_1, \ldots, n_m) \prec \mathcal{U}_n \prec \cdots \prec \mathcal{F}$$

où, de plus, les $\mathcal{U}_n(n_1, \ldots, n_m)$ au sein du même niveau n sont ordonnés lexicographiquement sur les $n_1, \ldots, n_m$.

La sémantique d'un algorithme est ensuite obtenue à l'aide d'une fonction d'évaluation sur des paires $\langle A(n_1, \ldots, n_{n_A}), \psi \rangle$ d'un tenseur indexé et d'une indice de niveau. Cette fonction d'évaluation est définie de façon récursive

en lisant, pour chaque niveau, la valeur obtenue au niveau précédent du tenseur en cours de définition, et la valeur finale (i.e. au niveau $\mathcal{F}$) des autres tenseurs. Cette définition récursive est bien fondée grâce à la condition de bonne formation, comme prouvé dans la version complète en anglais de cette thèse.

3 Ensembles de Presburger

Ce chapitre, dans la version complète en anglais, rappelle les définitions de l'arithmétique de Presburger et les notations des ensembles et relations de Presburger tels qu'utilisés par `isl`. Il est largement inspiré de l'excellent tutoriel de Verdoolaege [114], avec quelques adaptations pour cette thèse. Dans ce résumé en français, je rappelle brièvement la notion d'arithmétique de Presburger, ainsi que les notations essentielles pour la compréhension des sections suivantes du résumé ; le lecteur intéressé par des détails plus complets se référera à la version complète en anglais.

Arithmétique de Presburger L'arithmétique de Presburger, aussi connue sous le nom d'arithmétique linéaire ou d'arithmétique affine, est la théorie du premier ordre des entiers naturels munis de l'addition et de l'inégalité usuelle. Les *expressions affines* sont construites à partir de variables, constantes (entières), et d'addition et sont à valeur dans $\mathbb{Z}$; les *contraintes affines* sont des égalités ou des inégalités entre expressions affines et sont à valeur booléenne. Les *formules de Presburger* sont des formules logiques construites à partir de contraintes affines, de la négation $\neg$, de la conjonction $\wedge$, de la disjonction $\vee$, et des quantificateurs de premier ordre existentiel $\exists$ et universels $\forall$. La multiplication par une constante est définie en théorie par répétition de l'addition, mais gérée nativement par les solveurs en pratique.

On peut étendre l'arithmétique de Presburger avec une infinité d'opérations de *division euclidienne* par une constante (strictement) positive. On notera $\lfloor e/n \rfloor$ le quotient de la division euclidienne de e par la constante n, et $e \bmod n$ son reste. Les expressions résultantes sont nommées *expressions quasi-affines*. L'arithmétique de Presburger et l'arithmétique quasi-affine sont particulièrement intéressantes car elles admettent une procédure d'*élimination des quantificateurs* :

c'est-à-dire que toute formule en arithmétique de Presburger ou en arithmétique quasi-affine admet une formule équivalente en arithmétique quasi-affine sans quantificateurs. Les outils modernes comme `isl` gèrent l'arithmétique quasi-affine de façon primitive, et puisqu'il n'y a autrement peu de différences pour notre travail entre les expressions affines et quasi-affines, les qualificateurs "affines" et "de Presburger" doivent dans la suite être compris comme "quasi-affine".

Tuples nommés `isl` représente des ensembles et des relations de *tuples nommés*. Un tuple nommé pour un ensemble d'arguments $\mathcal{A}$ est un objet syntaxique qui représente un arbre binaire dont les nœuds sont étiquetés par des noms d'arguments n et dont les feuilles contiennent des tuples d'arguments (i.e. d'éléments de $\mathcal{A}$).

On notera $n\langle \mathbf{t_1}, \mathbf{t_2} \rangle$ un nœud interne étiqueté par n et dont les fils sont $\mathbf{t_1}$ et $\mathbf{t_2}$, et on notera $n\langle a_1, \ldots, a_d \rangle$ une feuille étiquetée par n et contient le d-tuple $(a_1, \ldots, a_d)$.

Le nom distingué ϵ représente un tuple anonyme, que l'on notera avec des crochets au lieu d'angles. Ainsi, on notera $[3]$ pour le tuple $\epsilon\langle 3 \rangle$ et $[[x], [y]]$ pour le tuple $\epsilon\langle [x], [y] \rangle$.

Définition 3.1. La *structure* d'un tuple nommé est simplement la structure d'arbre sous-jacente, c'est-à-dire l'arbre que l'on obtient en effaçant les arguments du tuple nommé (mais en préservant les étiquettes) et en annotant chaque feuille avec son arité.

Ensembles symboliques Afin d'être à la fois générique et flexible dans notre représentation, nous utiliserons l'arithmétique de Presburger pour représenter des ensembles de tuples nommés, et des relations entre tuples nommés. Ces tuples nommés seront étiquetés, suivant les cas, par des noms de tableaux (pour représenter, par exemple, un ensemble de cases mémoires), par des contextes d'expressions à trous multiples (pour représenter des expressions arbitraires contenant des composantes affines), et, dans certains cas, par des arbres plus complexes construits sur ces primitives.

On distinguera les ensembles de tuples nommés où tous les tuples ont la même structure, que l'on appellera *ensembles homogènes*, et les ensembles de tuples nommés qui peuvent contenir des tuples de structures différentes, que l'on appellera *ensembles hétérogènes*.

Définition 3.2. Ensemble homogène Un *ensemble homogène* est une paire $\langle \mathbf{t}, \phi \rangle$, que l'on notera $\{\mathbf{t} : \phi\}$, où ϕ est une formule en arithmétique de Presburger et $\mathbf{t}$ est un tuple nommé dont les arguments sont des variables, qui sont liées dans ϕ.

Les *paramètres* de $\{\mathbf{t} : \phi\}$ sont les variables libres de ϕ qui ne sont pas liées par $\mathbf{t}$.

Un ensemble homogène peut être évalués naturellement comme un ensemble de tuples nommés dont les arguments sont des *entiers* dans un environnement qui associe des entiers à chaque paramètre de l'ensemble : il s'agit des tuples nommés de même structure que $\mathbf{t}$ pour lesquels ϕ est vrai lorsqu'on associe à chaque variable de $\mathbf{t}$ la valeur de l'argument entier correspondant.

Remarque 3.1. Une formule arithmétique ϕ peut être interprété comme un ensemble qui est soit vide (quand ϕ est fausse), soit l'ensemble de tous les tuples nommés (quand ϕ est vrai). Un tel ensemble est appelé *ensemble unitaire* ou *ensemble paramétrique*, et on le notera $\{ : \phi\}$.

Définition 3.3 (Ensemble hétérogène)**.** Un *ensemble hétérogène* est une union finie d'ensembles homogènes. On notera cette union, suivant les cas, soit en utilisant le symbole $\cup$, soit en séparant les différents ensembles homogènes constituant l'ensemble hétérogène par des points-virgules : ainsi, on notera indifféremment $\{\mathbf{t}_1 : \phi_1\} \cup \{\mathbf{t}_2 : \phi_2\}$ et $\{\mathbf{t}_1 : \phi_2 ; \mathbf{t}_2 : \phi_2\}$ le même ensemble hétérogène.

La représentation canonique d'un ensemble hétérogène est sa *décomposition structurelle* qui est définie comme suit.

Définition 3.4 (Décomposition structurelle)**.** La *décomposition structurelle* d'un ensemble hétérogène S est l'unique (modulo des formules équivalentes pour les ϕ) collection d'ensembles *homogènes* non vides et de structures deux à deux différentes $(S_i)_{i \in I}$ telle que $\bigcup_{i \in I} S_i = S$.

Les opérations usuelles sur les ensembles (union $\cup$, intersection $\cap$ et différence $-$) peuvent être définies sur les ensembles homogènes et hétérogènes à l'aide des connecteurs logiques $\wedge$, $\vee$ et $\neg$ de l'arithmétique de Presburger, en remarquant que deux tuples nommés de différentes structures sont nécéssairement différents. On notera que le complément non borné d'un ensemble hétérogène ne peut pas être facilement représenté, d'où l'utilisation de la différence comme primitive.

Les ensembles (hétérogènes ou homogènes) étant définis à partir de formules en arithmétique de Presburger, on peut *décider et calculer* un certain nombre de propriétés sur ces ensembles à l'aide de solveurs spécialisés comme `isl`. Par exemple, `isl` est capable de donner la décomposition structurelle d'un ensemble hétérogène, mais aussi de déterminer si un ensemble S est vide – noté $\nexists(S)$ – ou contient au plus un élément, auquel cas on dira que S est un *singleton*.

Il est à noter que S étant un ensemble *paramétrique*, la notion de singleton est différente de celle à laquelle le lecteur peut être habitué : ainsi, on considérera un ensemble paramétrique comme $\{A\langle i\rangle : i = 0 \wedge N > 0\}$ comme étant un singleton, bien qu'il soit vide lorsque N est négatif ou nul. En particulier, cela veut dire que l'ensemble vide est un singleton.

On peut exprimer une condition sur les paramètres pour qu'un ensemble S soit non-vide, que l'on notera $\exists S$. $\exists S$ est une formule en arithmétique de Presburger (ou, de façon équivalente, un ensemble paramétrique) qui est vraie exactement dans les environnements où S est non-vide : ainsi, un singleton S contient exactement un élément lorsque ses paramètres sont tels que $\exists S$ est vrai.

Plus généralement, cette condition de non-vacuité permet de définir la *mise à jour* d'un ensemble S_1 par un ensemble S_2, notée $S_1 \triangleright S_2$, et définie par l'équation :

$$S_1 \triangleright S_2 = S_2 \cup \{S_1 - \exists S_2\}$$

La mise à jour $S_1 \triangleright S_2$ est égale à S_2 quand S_2 est non-vide, et à S_1 autrement : on remplace S_1 par S_2, sauf si S_2 est vide, auquel cas on garde l'ancienne valeur. Si S_1 et S_2 sont des singletons, l'opération de mise à jour permet de représenter l'écriture conditionnelle (suivant la vacuité de S_2) dans un case mémoire dont la valeur actuelle est représentée par S_1.

Relations symboliques De la même façon que l'on peut représenter de façon symbolique des ensembles de tuples nommés à l'aide de formules affines, on peut représenter des relations entre tuples nommés. Bien que la structure d'arbre des tuples nommés permette déjà de représenter ces relations en tant qu'ensembles de paires, on leur donnera une structure propre munie d'opérations appropriées pour des relations.

Définition 3.5 (Relation symbolique)**.** Une relation symbolique R est une relation entre paires de tuples nommés. Comme pour les ensembles symboliques, on distinguera les relations homogènes, dénotées $\{\mathbf{s}_1 \to \mathbf{s}_2 : \phi\}$ où $\mathbf{s}_1$ et $\mathbf{s}_2$ sont des tuples nommés de variables et ϕ est une formule de Presburger dans laquelle les variables de $\mathbf{s}_1$ et $\mathbf{s}_2$ sont liées ; et les relations hétérogènes, unions de relations homogènes.

On peut définir $\mathrm{dom}(R)$ et le codomaine $\mathrm{ran}(R)$ d'une relation symbolique comme des ensembles symboliques obtenus en projetant la seconde (resp. première) composante de la relation, ainsi que les opérations $\mathrm{wrap}(R)$ qui convertit une relation symbolique en un ensemble symbolique de paires anonymes, et $\mathrm{unwrap}(S)$ qui effectue l'opération inverse.

D'une façon générale, les définitions sur les ensembles symboliques peuvent être étendues aux relations symboliques en considérant le domaine de la relation comme des paramètres. Par exemple, l'opération de mise à jour sur des relations $R_1 \triangleright R_2$ met à jour la valeur associée à chaque élément du domaine, et a donc pour domaine $\mathrm{dom}(R_1 \triangleright R_2) = \mathrm{dom}(R_1) \cup \mathrm{dom}(R_2)$, avec pour élément du codomaine associé l'élément correspondant dans R_2 s'il existe et l'élément correspondant dans R_1 autrement.

On définira également $\mathrm{sv}(R)$ qui, de façon équivalente à $\mathrm{singleton}(S)$, est une formule de Presburger qui est vraie si et seulement si R est une relation fonctionnelle, c'est-à-dire qu'il y a au plus un élément du codomaine de R associé à chaque élément du domaine de R pour toute valeur des paramètres.

D'autres opérations peuvent être définies sur les relations de Presburger et sont décrites dans la version complète de la thèse en anglais, notamment le maximum lexicographique qui permet de définir une version symbolique de la mise à jour itérée un nombre paramétrique de fois, que l'on utilisera plus loin.

4 Un langage intermédiaire pour les compilateurs de tenseurs

Je présente maintenant la spécification formelle d'un langage pensé pour être un intermédiaire entre un compilateur de tenseur et un compilateur traditionnel comme LLVM. Ce langage, nommé SCHED, est un langage impératif avec des boucles et des tableaux multi-dimensionnels, fortement inspiré du langage STMT utilisé par Halide, TVM, et d'autres compilateurs de tenseurs. La principale nouveauté de SCHED — outre sa sémantique formelle, décrite ci-après — tient dans l'incorporation d'*annotations prophétiques* qui relient écritures impératives dans un tableau et valeurs dans une sémantique mathématique purement fonctionnelle. Cette section se concentre sur la définition du langage SCHED lui-même, des concepts sous-jacents, et de sa sémantique formelle ; son utilisation dans le cadre de la vérification de traduction est décrite dans la Section 5.

Les programmes SCHED font la distinction entre les expressions représentant des calculs demandés par l'utilisateur, appelées *expressions sémantiques*, telles que `a[i, k] * b[k, j]`, et des *expressions d'indexation* apparaissant dans les bornes de boucles et les accès de tableaux.

Les expressions d'indexation peuvent être transformée par le compilateur : par exemple, l'utilisateur peut avoir écrit $\mathtt{i}$ dans sa spécification, et ce $\mathtt{i}$ devient $4\mathtt{i}_0 + \mathtt{i}_1$ dans le code engendré suite au tuilage de la boucle $\mathtt{i}$. Afin de pouvoir retrouver les expressions d'indexation originelles, nous utiliserons isl, un outil polyédrique implémentant les concepts de la Section 3, afin d'inverser les expressions engendrées par le compilateur. Il nous faut donc, comme dans les compilateurs basés sur le modèle polyédrique, restreindre les expressions d'indexation à des expressions affines (ou, plus précisément, quasi-affines par morceaux).

À l'inverse, le compilateur doit être libre d'effectuer des simplifications arbitraires sur les expressions sémantiques, tant qu'elles sont valides pour les valeurs du type sous-jacent. Le but est ici de vérifier ces simplifications de façon locales : une fois les expressions d'indexation inversées par isl pour ramener l'expression sémantique associée à une écriture particulière, nous pouvons vérifier avec un solveur SMT comme Z3 qu'elle est bien identique sémantiquement à l'expression originellement fournie par l'utilisateur. Cette

Expressions

$e, \iota, t ::= x \mid l$	variables et littéraux
$\mid a[\iota_1, \ldots, \iota_n]$	indexation de tableau
$\mid A(\iota_1, \ldots, \iota_n)$	indexation de tenseur
$\mid \text{let } x = \iota_1 \text{ in } e_2$	variable locale
$\mid \iota_1 + \iota_2 \mid n \cdot \iota$	
$\mid \lfloor \iota / n \rfloor \mid \iota \bmod n$	arithmetique linéaire
$\mid \iota_1 = \iota_2 \mid \iota_1 \leqslant \iota_2$	
$\mid \iota_1 \neq \iota_2 \mid \iota_1 < \iota_2$	comparaisons
$\mid \iota_1 \ \&\& \ \iota_2 \mid \iota_1 \ \mid\mid \ \iota_2 \mid ! \iota$	connecteurs de Boole
$\mid \text{select}(\iota, e_1, e_2)$	condition gloutonne
$\mid f(e_1, \ldots, e_n)$	appel de fonction pure

FIGURE 1 : Syntax des expressions de SCHED

comparaison est rendue possible par des annotations sur les écritures qui n'impactent pas la sémantique du programme mais sont requises par les techniques de vérification de la Section 5. Ces annotations sont engendrées automatiquement par le compilateur, comme décrit dans la Section 5.4.

4.1 Syntaxe

La syntaxe des expressions de SCHED est donnée en figure 1.

Les non-terminaux e, ι et t sont utilisés pour décrire la même grammaire d'expressions. Informellement, e est utilisé lorsqu'une expression sémantique est attendue, tandis que ι est utilisé lorsqu'une expression d'indexation est attendue. t est utilisé pour les annotations prophétiques, c'est-à-dire une expression qui fait référence aux tenseurs de la spécification $A(\iota_1, \ldots, \iota_n)$ plutôt qu'aux tableaux du programme. Afin de faciliter une extension de la

$$
\begin{aligned}
c ::= {} & \mathsf{skip} \\
& \mid c_1 \ ; \ c_2 \\
& \mid a[\iota_1, \ldots, \iota_n] \ \{t\} := e \\
& \mid \mathsf{if} \ \iota \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \\
& \mid \mathsf{let} \ x = \iota \ \mathsf{in} \ c \\
& \mid \mathsf{allocate} \ a : \tau[\iota_1 \times \cdots \times \iota_n] \ \mathsf{in} \ c \\
& \mid \mathsf{for} \ x < \iota ; \mathsf{do} \ c \\
& \mid \mathsf{par} \ x < \iota ; \mathsf{do} \ c
\end{aligned}
$$

FIGURE 2 : Syntaxe des commandes de SCHED

formalisation à des accès ou des bornes de boucles non affines, la distinction entre ces différents types d'expressions est fait par le système de type décrit dans la prochaine sous-section plutôt que par la syntaxe. Les appels de fonctions pures $f(e_1, \ldots, e_n)$ permettent d'encoder des primitives arbitraires comme l'addition ou l'exponentiation. Afin de préserver l'aspect affine des expressions d'indexation, ils y seront interdits par le système de type.

Les commandes de SCHED sont décrites en figure 2. Les écritures de tableaux sont annotées par une expression prophétique t, qui est ignorée à l'exécution : les expressions prophétiques peuvent être considérées comme du *code fantôme* qui représente la valeur écrite dans le tableau en terme de la spécification, et servent uniquement à la vérification dans la Section 5. On notera la présence d'une construction allocate permettant d'allouer localement un tableau *non initialisé*, et la présence de boucles parallèles par dont le déterminisme est forcé par la sémantique décrite dans la Section 4.2.

4.2 Sémantique dynamique

La sémantique dynamique de SCHED est donnée sous forme de sémantique à grand pas, et ne donne par construction pas de sémantique aux boucles parallèles comportant des conflits d'écriture ou de lecture, ce qui permet au

validateur de la Section 5 d'être étendu naturellement aux boucles parallèles. Ce choix non standard est rendu possible par la restriction à des expressions d'indexation affines, permettant ainsi de garder trace de façon précise des écritures et lectures de chaque itération parallèle.

La sémantique dynamique des commandes est exprimée sous la forme d'un jugement $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$. Les composantes de ce jugement sont :

- $\mathcal{E}$ est un environnement de variables locales, représenté par une liste de lieurs $x \mapsto v$ où x est une variable locale et v est une valeur entière ou booléenne.

- μ représente la mémoire sous forme d'une fonction des cases mémoire vers des valeurs dans $\mathcal{U} \uplus \{\bot\}$, où $\mathcal{U}$ est un ensemble de valeurs défini par l'utilisateur qui inclut typiquement les flottants et les entiers machines, et $\bot$ est une valeur spéciale qui représente une erreur (e.g. une valeur non initialisée). Les cases mémoires, dénotées ℓ, sont des cellules de tableaux multidimensionnels, c'est-à-dire des paires d'un nom de tableau et d'une liste d'entiers :

$$l ::= a[n_1, \ldots, n_n]$$

- c est la commande dont la sémantique est calculée.

- $\delta\mu$ est le résultat de l'évaluation, représenté sous la forme d'une *mémoire différentielle* : une fonction partielle des cases mémoires vers des valeurs dans $\mathcal{U}$, qui représente l'ensemble des écritures effectuées durant l'évaluation de c. Pour obtenir l'état de la mémoire après l'évaluation de c, $\delta\mu$ doit être combiné avec la mémoire initiale μ, comme illustré par la règle U-SEQ. L'utilisation d'une mémoire différentielle en sortie rapproche la sémantique dynamique du vérifieur décrit en Section 5, mais surtout permet de formaliser simplement la sémantique des boucles parallèles dans la règle U-PARLOOP. La condition $\forall 0 \leqslant i \neq j < n,\ \delta\mu_i \frown \delta\mu_j$ s'assure que deux itérations concurrentes de la boucle ne peuvent écrire dans la même case mémoire que si les valeurs écrites sont identiques. $\frown$ est l'opérateur de compatibilité, qui requiert que $\delta\mu_i$ et $\delta\mu_j$ associent la même valeur aux cases mémoires de leur domaine commun. Cette restriction est légèrement plus relâchée que la restriction habituelle, qui consiste à

interdire à deux itérations concurrentes d'écrire dans la même case mémoire : deux écritures concurrentes de la même valeur dans la même case mémoire est usuellement considéré comme un conflit. Les compilateurs de tenseurs comme Halide se permettent toutefois l'exploitation de ce type de conflit, dit *bénin*, qui est donc autorisé dans notre formalisation. La formalisation peut facilement être modifiée pour interdire ces conflits : il suffit pour cela de remplacer la condition par $\mathrm{dom}(\delta\mu_i) \# \mathrm{dom}(\delta\mu_j)$, où # est l'opérateur de disjonction défini ci-dessous.

- ρ est un ensemble de cases mémoires représentant l'ensemble des cases mémoires lues durant l'évaluation de la commande c. ρ est utilisé conjointement avec $\mathrm{dom}(\delta\mu)$, qui représente l'ensemble des cases mémoires écrites durant l'évaluation de c, afin d'assurer l'absence de conflits en écriture au sein d'une boucle parallèle dans la règle U-ParLoop. La condition $\forall 0 \leqslant i \neq j < n,\ \mathrm{dom}(\delta\mu_i) \# \rho_j$ s'assure qu'aucune case mémoire ne peut être en même temps lue et écrite par deux itérations concurrentes de la boucle. # est un opérateur de disjonction : $S_1 \# S_2$ est vrai si, et seulement si, $S_1 \cap S_2$ est vide.

4.3 Sémantique à petit pas

La sémantique dynamique à grand pas décrite dans la section précédente est non standard et il est approprié de se demander quelle confiance lui apporter. Pour ce faire, nous pouvons définir une sémantique de Sched à petit pas, plus standard pour un langage concurrent. Pour ce faire la syntaxe des commandes est étendue par la construction $c_1 \mid\mid c_2$ permettant de représenter la composition parallèle de deux commandes c_1 et c_2, ainsi que la construction $\mathrm{inalloc}\ \mu_a\ \mathrm{do}\ c$ permettant de représenter l'exécution partielle de la commande c faisant appel à un tableau alloué localement par la commande allocate.

La sémantique à petit pas pour Sched est définie en Fig. 3 sous la forme d'une réduction $\langle c \mid \mu\rangle \rightsquigarrow \langle c' \mid \mu'\rangle$ représentant l'évaluation d'une commande c dans l'état mémoire initiale μ, résultant en une commande c' et un état mémoire suivant μ'. Par simplicité, la sémantique à petit pas utilise des substitutions $c[x \leftarrow v]$ pour affecter des valeurs aux variables plutôt que des environnements explicites. Puisque les environnements explicites sont purs et non mutables, la

différence est principalement cosmétique.

La sémantique à petit pas permet de donner plusieurs sémantique de façon non-déterministe à des programmes comportant des conflits, tandis que la sémantique à grand pas est toujours déterministe. Il est donc impossible de prouver une quelconque équivalence entre les deux sémantiques. En revanche, il est possible de prouver la sûreté de la sémantique à grand pas vis-à-vis de la sémantique à petit pas. Ceci s'exprime à l'aide de deux théorèmes montrant l'existence et le déterminisme de la sémantique à petit pas sous hypothèse de l'existence d'une sémantique à grand pas.

Le premier théorème assure que tout programme avec une sémantique à grand pas admet une évaluation à petit pas avec le même résultat. La notation $c[\mathcal{E}]$ dénote la substitution des lieurs de $\mathcal{E}$ par leur valeur dans c.

Théorême 4.1 (Existence)**.** *Si la commande* c *exécutée à grand pas dans l'environnement* $\mathcal{E}$ *et la mémoire* μ *résulte en une mémoire différentielle* $\delta\mu$*, il existe une séquence de réductions à petit pas* $\langle c[\mathcal{E}] \mid \mu\rangle \rightsquigarrow^* \langle \text{skip} \mid \mu \triangleright \delta\mu\rangle$

Le second théorème assure le déterminisme de l'évaluation à petit pas pour les programmes qui admettent une évaluation à grand pas.

Théorême 4.2 (Déterminisme)**.** *Si la commande* c *exécutée à grand pas dans l'environnement* $\mathcal{E}$ *et la mémoire* μ *résulte en une mémoire différentielle* $\delta\mu$*, et que par ailleurs* $\langle c[\mathcal{E}] \mid \mu\rangle$ *se réduit en zéro, une, ou plusieurs étapes en* $\langle \text{skip} \mid \mu'\rangle$*, alors* μ' *est égal à* $\mu \triangleright \delta\mu$*.*

La preuve de ces deux théorèmes est omise de ce résumé : le lecteur intéressé se référera à la version complète en anglais.

4.4 Typage

Afin de distinguer les expressions sémantiques des expressions d'indexations et des expressions prophétique, nous utilisons un système de type. Dans

SEQ-CTX
$$\frac{\langle c_1 \mid \mu\rangle \rightsquigarrow \langle c_1' \mid \mu'\rangle}{\langle c_1 \,;\, c_2 \mid \mu\rangle \rightsquigarrow \langle c_1' \,;\, c_2 \mid \mu'\rangle}$$

SEQ-SKIP
$$\frac{}{\langle \text{skip} \,;\, c \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

PAR-L
$$\frac{\langle c_1 \mid \mu\rangle \rightsquigarrow \langle c_1' \mid \mu'\rangle}{\langle c_1 \parallel c_2 \mid \mu\rangle \rightsquigarrow \langle c_1' \parallel c_2 \mid \mu'\rangle}$$

PAR-R
$$\frac{\langle c_2 \mid \mu\rangle \rightsquigarrow \langle c_2' \mid \mu'\rangle}{\langle c_1 \parallel c_2 \mid \mu\rangle \rightsquigarrow \langle c_1 \parallel c_2' \mid \mu'\rangle}$$

PAR-SKIP-R
$$\frac{}{\langle c \parallel \text{skip} \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

PAR-SKIP-L
$$\frac{}{\langle \text{skip} \parallel c \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

IF-TRUE
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = \text{true}}{\langle \text{if } \iota \text{ then } c_1 \text{ else } c_2 \mid \mu\rangle \rightsquigarrow \langle c_1 \mid \mu\rangle}$$

IF-FALSE
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = \text{false}}{\langle \text{if } \iota \text{ then } c_1 \text{ else } c_2 \mid \mu\rangle \rightsquigarrow \langle c_2 \mid \mu\rangle}$$

LET
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = v}{\langle \text{let } x = \iota \text{ in } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow v] \mid \mu\rangle}$$

SEQLOOP
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = n \in \mathbb{Z}}{\langle \text{for } x < \iota; \text{do } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow 0] \,;\, \ldots \,;\, x[c \leftarrow n-1] \mid \mu\rangle}$$

PARLOOP
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = n \in \mathbb{Z}}{\langle \text{par } x < \iota; \text{do } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow 0] \parallel \ldots \parallel x[c \leftarrow n-1] \mid \mu\rangle}$$

ASSIGN
$$\frac{[\![\iota_i]\!]_{\emptyset;\mu} = n_i \text{ for all } 1 \leqslant i \leqslant n \qquad [\![e]\!]_{\emptyset;\mu} = v \qquad a[n_1, \ldots, n_n] \in \text{dom}(\mu)}{\langle a[\iota_1, \ldots, \iota_n]\ \{t\} := e \mid \mu\rangle \rightsquigarrow \langle \text{skip} \mid \mu[a[n_1, \ldots, n_n] \leftarrow v]\rangle}$$

ALLOCATE
$$\frac{\begin{array}{c}[\![\iota_i]\!]_{\emptyset;\mu} = n_i \text{ for all } 1 \leqslant i \leqslant n \\ \mu_a = \{a[i_1, \ldots, i_n] \mapsto \bot \mid 0 \leqslant i_1 < n_1 \wedge \cdots \wedge 0 \leqslant i_n < n_n\} \\ a \notin \text{arrays}(\mu)\end{array}}{\langle \text{allocate } a : \tau[\iota_1 \times \cdots \times \iota_n] \text{ in } c \mid \mu\rangle \rightsquigarrow \langle \text{inalloc } \mu_a \text{ do } c \mid \mu\rangle}$$

INALLOC-CTX
$$\frac{\begin{array}{c}\text{dom}(\mu) \# \text{dom}(\mu_a) \\ \langle c \mid \mu \uplus \mu_a\rangle \rightsquigarrow \langle c' \mid \mu' \uplus \mu_a'\rangle \qquad \text{dom}(\mu_a') = \text{dom}(\mu_a)\end{array}}{\langle \text{inalloc } \mu_a \text{ do } c \mid \mu\rangle \rightsquigarrow \langle \text{inalloc } \mu_a' \text{ do } \tau c' \mid \mu'\rangle}$$

INALLOC-SKIP
$$\frac{}{\langle \text{inalloc } \mu_a \text{ do skip} \mid \mu\rangle \rightsquigarrow \langle \text{skip} \mid \mu\rangle}$$

FIGURE 3 : Sémantique à entrelacements pour SCHED

ce système de type, le type $\mathbb{A}$ représentant les expressions quasi-affines par morceaux est utilisé pour les accès de tableaux et les bornes de boucles, tandis que le type $\mathbb{B}$ des contraintes quasi-affines par morceaux est utilisé pour les conditionnelles. Le système de type s'assure que les expressions d'indexation sont de type $\mathbb{A}$ ou $\mathbb{B}$, tandis que les expressions sémantiques et prophétiques ont des types "utilisateurs" (tel que float32 ou int32) dénotés par τ. On supposera que les calculs sur les types $\mathbb{A}$ et $\mathbb{B}$ sont effectués à l'aide d'arithmétique exacte, et on ne permettra pas leur stockage directement dans des tableaux. En revanche, les valeurs de ces types peuvent être convertis en types utilisateurs à l'aide de fonctions de conversions.

L'environnement de typage Γ représente à la fois l'environnement dynamique $\mathcal{E}$ et la mémoire μ. On trouve trois types de lieurs dans un environnement de typage Γ :

- Des lieurs affines $x : \mathbb{A}$ et $x : \mathbb{B}$ indiquant l'existence d'une variable (globale ou locale) du type d'indexation correspondant. Les variables de types utilisateurs sont représentés par des tableaux de dimension 0.

- Des lieurs de tableaux $a : \tau[\iota_1 \times \cdots \times \iota_n]$ indiquant un tableau n-dimensionnel a, où la i-ème dimension a une longueur ι_i, et contenant des valeurs de type τ (ou la valeur d'erreur $\bot$). Une variable mutable est représentée par un tableau de zéro-dimensionnel $a : \tau[]$.

- Des expressions booléennes ι (de type $\mathbb{B}$) qui contraignent les variables d'indexations présentes dans le contexte. Ces expressions sont utilisées pour représenter les contraintes sur les bornes des boucles ainsi que les conditionnelles. Ces expressions peuvent être rapprochées des *conditions de chemin* dans un évaluateur symbolique, et sont présentes dans le jugement de typage afin d'être exploitées par l'évaluateur symbolique décrit dans la Section 5.

Les règles de typage pour les expressions de SCHED sont données en figure 4.

$$\text{T-Var}\ \frac{}{\vdash \Gamma, x : \tau}\quad \frac{}{x,\ \tau :\ \vdash x : \tau}$$

$$\text{T-Array}\ \frac{a : \tau[\iota'_1, \ldots, \iota'_n] \in \Gamma \qquad \forall 1 \leqslant i \leqslant n,\ \Gamma \vdash \iota_i : \mathbb{A}}{\Gamma \vdash_a a[\iota_1, \ldots, \iota_n] : \tau}$$

$$\text{T-Tensor}\ \frac{A \in \mathcal{S} \qquad \forall 1 \leqslant i \leqslant n_A,\ \Gamma \vdash \iota_i : \mathbb{A}}{\Gamma \vdash_A A(\iota_1, \ldots, \iota_{n_A}) : \tau_A}$$

$$\text{T-Bool}\ \frac{b \in \{\text{true}, \text{false}\}}{\Gamma \vdash b : \mathbb{B}}$$

$$\text{T-Int}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathbb{A}}$$

$$\text{T-Call}\ \frac{\begin{array}{c}\forall 1 \leqslant i \leqslant n, k_i \in \{k, \emptyset\} \Rightarrow \Gamma \vdash_{k_i} e_i : \tau_i \\ k \in \{a, A\} \qquad f \in \mathcal{F} \qquad \tau_f = \tau_1 \times \cdots \times \tau_n \to \tau\end{array}}{\Gamma \vdash_k f(e_1, \ldots, e_n) : \tau}$$

$$\text{T-Select}\ \frac{\begin{array}{c}k \in \{a, A, \emptyset\} \\ \Gamma \vdash \iota_1 : \mathbb{B} \qquad \Gamma, e_1 \vdash_k e_2 : \tau \qquad \Gamma, \neg e_1 \vdash_k e_3 : \tau\end{array}}{\Gamma \vdash_k \text{select}(\iota_1, e_2, e_3)}$$

$$\text{T-Let}\ \frac{k \in \{a, A, \emptyset\} \qquad \Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A} \vdash_k e : \tau}{\Gamma \vdash_k \text{let } x = \iota \text{ in } e : \tau}$$

$$\text{T-Add}\ \frac{\Gamma \vdash \iota_1 : \mathbb{A} \qquad \Gamma \vdash \iota_2 : \mathbb{A}}{\Gamma \vdash \iota_1 + \iota_2 : \mathbb{A}}$$

$$\text{T-Mul}\ \frac{\Gamma \vdash \iota : \mathbb{A}}{\Gamma \vdash n \cdot e : \mathbb{A}}$$

$$\text{T-Div}\ \frac{\Gamma \vdash \iota : \mathbb{A} \qquad n > 0}{\Gamma \vdash \lfloor \iota / n \rfloor : \mathbb{A}}$$

$$\text{T-Mod}\ \frac{\Gamma \vdash \iota : \mathbb{A} \qquad n > 0}{\Gamma \vdash \iota \bmod n : \mathbb{A}}$$

$$\text{T-Cmp}\ \frac{\Gamma \vdash \iota_1 : \mathbb{A} \qquad \Gamma \vdash \iota_2 : \mathbb{A} \qquad \odot \in \{=, \leqslant\}}{\Gamma \vdash \iota_1 \odot \iota_2 : \mathbb{B}}$$

$$\text{T-And}\ \frac{\Gamma \vdash \iota_1 : \mathbb{B} \qquad \Gamma \vdash \iota_2 : \mathbb{B}}{\Gamma \vdash \iota_1\ \&\&\ \iota_2 : \mathbb{B}}$$

$$\text{T-Not}\ \frac{\Gamma \vdash \iota : \mathbb{B}}{\Gamma \vdash !\,\iota : \mathbb{B}}$$

FIGURE 4 : Règles de typages pour les expressions de SCHED

5 Validation d'un compilateur de tenseurs

Cette section explore une approche pratique pour la validation d'un compilateur de tenseurs à l'aide du langage SCHED présenté dans la section précédente. Cette approche consiste à construire un évaluateur symboliques pour les programmes SCHED qui construit, à l'aide des annotations prophétiques, des conditions de vérifications dont la validité implique la correction de l'évaluation symbolique elle-même.

5.1 Évaluation prophétique

Les commandes de SCHED sont impératives et fonctionnent par effets de bord (lectures et écritures depuis la mémoire). Nous étendons en conséquence le système de type de la Section 4 avec un système de type et d'effets pour les commandes de SCHED qui capture précisément les effets d'une commande sur la mémoire du programme. Ce système de type et d'effets est nommé *évaluation prophétique* car il capture les écritures que le compilateur a prédit à l'aide des annotations prophétiques — et non pas les écritures effectives du programme.

L'évaluation prophétique est implémentée par le jugement $\Gamma \vdash c : \Delta h$ décrit dans la Fig. 5, où Δh est un *tas symbolique* représentant l'ensemble des écritures prophétiques effectuées par c. Les détails de la représentation des tas symboliques sous forme de relations de Presburger est omise de ce résumé, il est donc recommandé au lecteur intéressé de se référer à la version complète en anglais pour sa description. Il suffit ici de savoir qu'un tas symbolique Δh peut s'évaluer en une mémoire partielle $[\![\Delta h]\!]_{\mathcal{E};M}$ dans un environnement $\mathcal{E}$ et un modèle M, et que les opérations suivantes, définies sur les tas symboliques, sont fidèles par rapport à cette évaluation :

- $\Delta h \backslash a$ représente le tas symbolique Δh où les cases mémoires associées au tableau a ont été supprimées
- $\Delta h_1 \triangleright \Delta h_2$ représente le tas symbolique Δh_1 mis à jour avec le tas symbolique Δh_2, i.e. on a $[\![\Delta h_1 \triangleright \Delta h_2]\!]_{\mathcal{E};M} = [\![\Delta h_1]\!]_{\mathcal{E};M} = [\![\Delta h_2]\!]_{\mathcal{E};M}$.

T-ALLOCATE
$$\frac{\Gamma, a : \tau[e_1 \times \cdots \times e_n] \vdash c : \Delta h \qquad \forall 1 \leqslant i \leqslant n,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash \text{allocate } a : \tau[e_1 \times \cdots \times e_n \text{ in } c : \Delta h \backslash a}$$

T-SKIP
$$\frac{}{\Gamma \vdash \text{skip} : \emptyset}$$

T-SEQ
$$\frac{\Gamma \vdash c_1 : \Delta h_1 \qquad \Gamma \vdash c_2 : \Delta h_2}{\Gamma \vdash c_1 \ ; c_2 : \Delta h_1 \triangleright \Delta h_2}$$

T-IF
$$\frac{\Gamma \vdash e : \mathbb{B} \qquad \Gamma, e \vdash c_1 : \Delta h_1 \qquad \Gamma, \neg e \vdash c_2 : \Delta h_2}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : (\Delta h_1 \cap e) \uplus (\Delta h_2 \cap \neg e)}$$

T-SEQLOOP
$$\frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < e \vdash c : \Delta h}{\Gamma \vdash \text{for } x < e; \text{do } c : \blacktriangleright_{0 \leqslant x < e} \Delta h}$$

T-PARLOOP
$$\frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < e \vdash c : \Delta h \qquad \upsilon\left(\bigcup_{0 \leqslant x < e} \Delta h\right) = \Delta h'}{\Gamma \vdash \text{par } x < e; \text{do } c : \Delta h'}$$

T-ASSIGN
$$\frac{\begin{array}{c}\Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \qquad E\langle \iota''_1, \ldots, \iota''_m \rangle = \text{decompose}(t) \\ a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \qquad \Gamma \vdash \iota_i : \mathbb{A} \text{ pour tout } 1 \leqslant i \leqslant n \\ \Gamma \vdash 0 \leqslant \iota_i < \iota'_i \text{ pour tout } 1 \leqslant i \leqslant n\end{array}}{\Gamma \vdash a[\iota_1, \ldots, \iota_n]\ \{t\} := e : \{a\langle \iota_1, \ldots, \iota_n \rangle \rightarrow E\langle \iota''_1, \ldots, \iota''_m \rangle\}}$$

FIGURE 5 : Évaluation prophétique des commandes SCHED

- $\Delta h_1 \uplus \Delta h_2$ représente l'union disjointe de deux tas symboliques, défini si leur domaines sont disjoints.

- $\Delta h \cap e$ est identique à Δh lorsque e est vrai, et est vide autrement.

- $\rhd_{0 \leqslant x < \iota} \Delta h$ est une représentation symbolique de la mise à jour itérée de Δh, où x est libre dans Δh. On a $[\![\rhd_{0 \leqslant x < \iota} \Delta h]\!]_{\mathcal{E};M} = \rhd_{0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}} [\![\Delta h]\!]_{\mathcal{E}+x \mapsto i;M}$. Cette opération n'est calculable exactement que lorsque ι est une expression affine des variables d'indexation du contexte, ce qui est assuré par le typage.

- $\upsilon(\Delta h)$ renvoie un nouveau tas symbolique $\Delta h'$ qui représente une version simplifiée de Δh. υ permet de simplifier les tas symboliques qui contiennent plusieurs expressions sémantiquement équivalentes (par exemple $A(i) + B(j)$ et $B(j) + A(i)$) associées à une même case mémoire en un tas symbolique qui choisit arbitrairement parmi ces représentations.

- decompose extrait d'un terme t un contexte E à plusieurs trous et sans variables libres et des expressions d'indexations $\iota_1, \ldots, \iota_n$ telles que la substitution des trous par $\iota_1, \ldots, \iota_n$ dans E, dénotée $E[\iota_1, \ldots, \iota_n]$, est égale à t. decompose est utilisé pour construire la représentation d'un tas symbolique singleton dans la règle T-ASSIGN.

5.2 Évaluation symbolique

Dans la section précédente, nous avons introduit un système de type et d'effets nommé *évaluation prophétique* pour SCHED qui calcule l'évaluation d'un programme telle qu'annoncée par ses annotations prophétiques et qui ignore le membre droit des assignements. Il nous faut maintenant définir un évaluateur symbolique qui calcule l'évaluation du programme en utilisant sa définition réelle, c'est-à-dire en évaluant le membre droit des assignements. Cet évaluateur symbolique utilise l'évaluation prophétique pour briser les cycles créés par les boucles séquentielles, et générera des conditions de vérifications qui impliquent la correction de l'évaluation prophétique.

Le jugement $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ est présenté en Fig. 6 et suit les

règles d'évaluation dynamique de SCHED. L'environnement de typage Γ et le tas d'entrée h sont des représentations symboliques de l'environnement dynamique $\langle \mathcal{E}; \mu \rangle$. La paire $\langle \Delta h; R \rangle$ est une représentation symbolique de l'état dynamique $\langle \delta\mu; \rho \rangle$. h et Δh sont représentés en utilisant des relations de Presburger : leur domaine est constitué de tuples nommés étiquetés par des noms de tableaux, et leur codomaine est constitué de tuples nommés étiquetés par des contextes multi-trous. R est représenté comme un ensemble de Presburger contenant des cases mémoires. C correspond aux conditions de vérifications et est une relation de Presburger entre paires de contextes multi-trous qui doivent être égaux pour que l'évaluation soit correcte.

L'évaluateur symbolique fait usage de quelques définitions auxiliaires, rapportées ici.

rw-safe(x, ι, W, R) est une formule de Presburger qui assure l'absence de condition critique entre les lectures dans R et les écritures dans W pour deux valeurs distinctes de x entre 0 et ι. Elle est définie comme suit (y est une variable fraîche) :

$$\text{rw-safe}(x, \iota, W, R) = \nexists\left(\bigcup_{0 \leqslant x < \iota} \bigcup_{0 \leqslant y < \iota} (W \cap R[x := y] \cap \{x \neq y\})\right)$$

Pour éviter les conditions critiques entre deux écritures, on autorisera plus de comportement qu'habituellement en théorie de la concurrence : en effet, Halide autorise (et exploite) une certaine catégorie de conflits bénins où plusieurs itérations parallèles peuvent écrire la même valeur à la même adresse. Pour représenter cela nous définisions d'abord l'ensemble des cases mémoires qui peuvent être écrites par des itérations différentes :

$$\text{conflicts}(x, \iota, W) = \bigcup_{0 \leqslant x < \iota} \bigcup_{0 \leqslant y < \iota} (\text{fst}(W) \cap \text{snd}(W)[x := y] \cap \{x \neq y\})$$

Ensuite, nous définissons la condition suivante, qui requiert que la valeur écrite dans une case mémoire conflictuelle doit être unique (rappelons que sv(R) est

S-ALLOCATE

$$\frac{\Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]; h \vdash C \implies c : \langle \Delta h; R \rangle \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{pour tout} 1 \leqslant i \leqslant n}{\Gamma; h \vdash C \implies \text{allocate } a : \tau[\iota_1 \times \cdots \times \iota_n] \text{ in } c : \langle \Delta h \backslash a; R \backslash a \rangle}$$

S-SKIP

$$\frac{}{\Gamma; h \vdash \emptyset \implies \text{skip} : \langle \emptyset; \emptyset \rangle}$$

S-SEQ

$$\frac{\Gamma; h \vdash C_1 \implies c_1 : \langle \Delta h_1; R_1 \rangle \qquad \Gamma; h \triangleright \Delta h_1 \vdash C_2 \implies c_2 : \langle \Delta h_2; R_2 \rangle}{\Gamma; h \vdash C_1 \cup C_2 \implies c_1 \,;\, c_2 : \langle \Delta h_1 \triangleright \Delta h_2; R_1 \cup R_2 \rangle}$$

S-LET

$$\frac{\Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, x = \iota; h \vdash C \implies c : S}{\Gamma; h \vdash C[x := \iota] \implies \text{let } x = \iota \text{ in } c : S[x := \iota]}$$

S-SEQLOOP

$$\frac{\begin{array}{c} z \text{ fresh} \\ \Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h \triangleright \underset{0 \leqslant z < x}{\blacktriangleright} \Delta h[x := z] \vdash C \implies c : \langle \Delta h; R \rangle \end{array}}{\Gamma; h \vdash \bigcup_{0 \leqslant x < \iota} C \implies \text{for } x < \iota \text{; do } c : \langle \underset{0 \leqslant x < \iota}{\blacktriangleright} \Delta h; \bigcup_{0 \leqslant x < \iota} R \rangle}$$

S-PARLOOP

$$\frac{\begin{array}{c} \Gamma \vdash \iota : \mathbb{A} \\ \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h \vdash C \implies c : \langle \Delta h; R \rangle \qquad \Gamma \vdash \text{rw-safe}(x, \iota, \text{dom}(\Delta h), R) \\ \text{ww-covered}(\Gamma, x, \iota, \Delta h) = C' \qquad R' = \bigcup_{0 \leqslant x < \iota} R \qquad \upsilon\left(\bigcup_{0 \leqslant x < \iota} \Delta h\right) = \Delta h' \end{array}}{\Gamma; h \vdash C' \cup \bigcup_{0 \leqslant x < \iota} C \implies \text{par } x < \iota \text{; do } c : \langle \Delta h'; R' \rangle}$$

S-IF

$$\frac{\begin{array}{c} \Gamma \vdash \iota : \mathbb{B} \qquad \Gamma, \iota; h \vdash C_1 \implies c_1 : \Delta h_1 R_1 \\ \Gamma, \neg\iota; h \vdash C_2 \implies c_2 : \Delta h_2 R_2 \qquad C = (C_1 \cap \iota) \uplus (C_2 \cap \neg\iota) \\ \Delta h = (\Delta h_1 \cap \iota) \uplus (\Delta h_2 \cap \neg\iota) \qquad R = (R_1 \cap \iota) \uplus (R_2 \cap \neg\iota) \end{array}}{\Gamma; h \vdash C \implies \text{if } \iota \text{ then } c_1 \text{ else } c_2 : \langle \Delta h; R \rangle}$$

S-ASSIGN

$$\frac{\begin{array}{c} a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \\ \Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{pour tout } 1 \leqslant i \leqslant n \qquad \hat{\ell} = a\langle \iota_1, \ldots, \iota_n \rangle \\ \Gamma \vdash \{\hat{\ell}\} \subseteq \{a\langle x_1, \ldots, x_n \rangle \mid 0 \leqslant x_1 < \iota'_1, \ldots, 0 \leqslant x_n < \iota'_n\} \qquad \Gamma \vdash \text{reads}(e) \subseteq \text{dom}(h) \\ \hat{C} = [\![e]\!]_h = \{\text{decompose}(t)\} \qquad \Delta h = \{\hat{\ell} \to \text{decompose}(t)\} \end{array}}{\Gamma; h \vdash \hat{C} \implies a[\iota_1, \ldots, \iota_n]\,\{t\} := e : \langle \Delta h; \text{reads}(e) \rangle}$$

FIGURE 6 : Évaluateur symbolique

une formule en arithmétique de Presburger qui exprime que la relation R est fonctionnelle) :

$$\text{sv-conflicts}(x, \iota, W) = \text{sv}((\left(\bigcup_{0 \leqslant x < \iota} W \right) \cap \text{conflicts}(x, \iota, W))$$

On pourra ici considérer que ww-covered(Γ, x, ι, W) est l'ensemble vide lorsque $\Gamma \vdash$ sv-conflicts(x, ι, W) et le singleton $\{0\langle \mapsto \rangle 1\langle\}\rangle$ (qui représente la condition de vérification *fausse* $0 = 1$) autrement. En pratique, afin d'autoriser des écritures concurrentes avec des expressions syntaxiquement différentes mais de même valeur (par exemple, $A(i) + B(i)$ et $B(i) + A(i)$, la définition de ww-covered est plus complexe : le lecteur intéressé se référera à la version complète en anglais pour cette définition complète.

Les règles de l'évaluateur symbolique sont des adaptations symboliques assez directes des règles d'exécution dynamiques et, à part pour la règle S-SEQLOOP, elles sont algorithmiques : on peut calculer les sorties Δh, R et C récursivement à partir des entrées Γ, h et c. Dans le cas de la règle S-SEQLOOP, la sortie Δh apparaît comme entrée de l'appel récursif dans la mise à jour itérée, ce qui nécessite d'inventer Δh avant l'appel récursif. L'évaluation prophétique permet de résoudre ce problème : en effet, on peut prouver par un simple raisonnement inductif que l'évaluation prophétique produit toujours le même Δh que l'évaluation symbolique, qui est donc indépendante du tas d'entrée h. On utilisera ainsi l'évaluation prophétique pour calculer le Δh qui sera utilisé lors de l'appel récursif à a règle d'évaluation symbolique, ce qui vérifiera a posteriori sa correction.

5.3 Preuve de correction

La preuve que l'évaluateur symbolique capture correctement le comportement de la sémantique dynamique présentée dans la section précédente est omise de ce résumé en français, et peut être trouvée dans la version complète de la thèse en anglais.

5.4 Génération des expressions prophétiques

Cette section propose une méthodologie de vérification qui présuppose la présence d'annotations prophétiques dans le code liant écritures dans un tableau avec une expression représentant la valeur écrite en terme des tenseurs de la spécification. Cela peut sembler au premier abord comme imposant une contrainte supplémentaire à l'utilisateur, qui devrait alors fournir ces annotations, mais il n'en est rien. En effet, j'affirme que ces annotations peuvent en pratique être engendrées automatiquement par le compilateur à partir de la spécification de façon peu coûteuse. Pour justifier cette affirmation, considérons une équation $A(\iota_1, \ldots, \iota_n) = e$ de la spécification. On peut la transformer en une équation alternative $A(\iota_1, \ldots, \iota_n) = f_A(e, \iota_1, \ldots, \iota_n)$ où f_A est une fonction opaque pour le compilateur de tenseurs, qui est en fait implémenté par la première projection. Après compilation de cette spécification transformée, on obtiendra des écritures dont le membre droit est de la forme $f_A(e', \iota'_1, \ldots, \iota'_n)$ dont on pourra extraire l'annotation prophétique $A(\iota'_1, \ldots, \iota'_n)$. On voit donc ainsi que le compilateur de tenseur doit, par nécessité, avoir la capacité de tracer les annotations prophétiques durant la génération de code, et qu'il devrait être possible de le modifier sans trop de mal pour produire automatiquement du code annoté. Cette approche a été utilisée sur le compilateur Halide dans l'évaluation expérimentale décrite dans la prochaine section.

6 Évaluation expérimentale

Durant cette thèse, j'ai implémenté les algorithmes de validation de traduction décrit dans les sections précédents en OCaml, et appliqué l'approche au compilateur Halide. J'ai également comparé l'approche à ISA, un outil d'équivalence de programmes affine issue du modèle polyédrique, qui utilise une approche complétement automatisée et ne dépend pas d'annotations prophétiques.

Dans l'implémentation, les calculs de l'évaluation prophétique et symboliques sont effectués à l'aide de la bibliothèque `isl`. La spécification ainsi que les conditions de vérifications obtenues sont ensuites fournies au solveur Z3 pour déterminer leur véracité.

Les détails de l'implémentation et de l'instrumentation du compilateur Halide pour ces expériences sont décrits dans la version complète de cette thèse en anglais. Dans ce résumé en français, je me contente de rappeler et de commenter les résultats de l'évaluation expérimentale effectuée sur un ensemble de programmes issus des *benchmarks* officiels de Halide.

Les résultats de la vérification expérimentale, effectuée sur une machine avec un processeur Intel® Core™ i9 – 9900 et un timeout de 15 minutes, sont disponibles en Table 1. On notera que dans l'outil ISA effectue son analyse en 3 étapes implémentés par les outils `c2pdg`, `da`, et `eqv`, dont les temps d'exécution sont indiqués séparément. Le temps total pris par ISA doit être obtenu en sommant ces trois temps d'exécution. De plus, lorsqu'un opérateur associatif ou commutatif est en jeu, cela doit être explicitement indiqué à ISA, ce qui peut augmenter significativement son temps d'exécution. Cette information est disponible sous la forme d'annotations A (associatif) et C (commutatif) pour l'opérateur correspondant dans la colonne `eqv`.

Il est à noter que certains exemples sont des variations du même exemple : par exemple, `sgemm1024` et `sqsgemm` sont des variantes de `sgemm` (une multiplication de matrice $N \times M$ par une matrice $M \times P$) avec les contraintes $N = M = P = 1024$ et $N = M = P$, respectivement. De même, les variantes préfixées par `big` correspondent à des cas où les tailles de matrices sont plus grandes que 512, ce qui limite le nombre de spécialisations à considérer et permet à ISA de réussir la vérification. Pour plus de détails sur les benchmarks utilisés, le lecteur intéressé se référera à la version complète de la thèse en anglais.

On remarquera que lorsque mon outil et ISA parviennent tous deux à valider la correction d'une compilation, leur performance est comparable ou plus avantageuse pour mon outil — sauf pour le benchmark `conv` qui implique des conditions affines complexes, et la conversion depuis la représentation affine vers la représentation interne de Z3 est un facteur limitant important dans ce benchmark. Par ailleurs, mon outil est capable de vérifier plus d'exemples que ISA.

TABLE 1 : Resultats de l'évaluation expérimentale (temps en secondes)

	isa				Ours	
	c2pdg	da	eqv			
blur	<1	<1	1.1^{AC+}	✓	<1	✓
cmm1024	<1	1.2	10	✓	2.6	✓
sgemm1024	<1	2.5	$27.3^{C\times}$	✓	1	✓
sqsgemm	2.5	4min34	$> 15min^{C\times}$	?	15.1	✓
bigsqsgemm	1.6	17.5	$8min12^{C\times}$	✓	2.7	✓
sgemm	7.	>15min	N/A	?	3min24	✓
bigsgemm	2.8	1min19	>15min	?	12.1	✓
sc1	3.44	4m20	3.3	✗	12.5	✓
sc32	1min41	>15min	N/A	?	4min53	✓
dsc	10.2	13min49	>15min	?	1min43	✓
conv	2.	12.2	27.9^{Cmax}	✓	2min12	✓
sdot	<1	<1	<1	✓	<1	✓
harris*	7.9	31.5	1min8	✓	44.8	✓
unsharp*	1.3	6.1	13min44	✗	6.1	✓
nl_means*	>15min	N/A	N/A	?	>15min	?
sgemmTA†	21.5	>15min	N/A	?	10.4	✗
sgemmTB†	21.1	N/A	N/A	✗	34.7	✗

*Pas d'optimisations flottantes

†Échec attendu

7 Vérification des réductions

Une *réduction* est l'application répétée d'un opérateur binaire sur les éléments d'une séquence de valeurs, les *réduisant* ultimement en une unique valeur. L'exemple le plus commun est l'opérateur de somme Σ qui itère l'opérateur d'addition +. Lorsque l'opérateur binaire sous-jacent est associatif et commutatif, comme c'est généralement le cas pour l'addition, le résultat de la réduction ne dépend pas de l'ordre dans lequel le calcul est effectué. Cette propriété est utilisée par les compilateurs de tenseurs afin de mieux optimiser la localité des calculs mais aussi pour révéler du parallélisme caché.

L'approche décrite jusqu'ici dans ce manuscrit ne s'applique pas à de tels

réordonnancement, car elle requiert un encodage rigide de l'ordre d'exécution d'une telle réduction. Ce chapitre décrit une extension à l'approche développée jusqu'ici, basée sur la remarque que dans l'application d'une réduction à une case de tableau :

```
for j in 0 to N - 1 do
  a[] += b[j]
```

on peut considérer l'affectation `a[] += b[j]` comme ajoutant la valeur `b[j]` à l'ensemble des valeurs réduites dans la case mémoire `a[]`.

Comme on ne peut pas tracer directement les valeurs `b[j]` de façon symbolique, il faut utiliser une représentation plus complexe, qui représente à la fois l'index `j` réduit et la valeur `b[j]` associée. Les détails de cette représentation et sa formalisation sont disponibles dans la version complète de la thèse en anglais, mais n'ont pas été implémentés dans l'outil utilisé pour l'évaluation expérimentale.

8 Travaux liés

Le chapitre 8 présente de nombreux travaux liés à cette thèse, notamment en validation de traduction et en équivalence de programmes affines. Ces travaux étant rédigés en anglais, ils sont discutés en détail uniquement dans la version complète de cette thèse en anglais.

9 Conclusion

La spécificité des compilateurs de tenseurs est d'effectuer principalement des transformations qui modifient en profondeur la structure du programme optimisé, ce qui rend leur vérification difficile pour les techniques de vérification traditionnelles basées sur des bisimulations. Cette thèse s'intéresse au développement de techniques de *validation de traduction* pour ces compilateurs, dans lesquelles un validateur est développé indépendemment et peut être utilisé pour plusieurs compilateurs différents.

En toute généralité, le validateur doit résoudre un problème indécidable, celui de l'équivalence de programmes entre l'entrée et la sortie du compilateur. La validation de traduction est donc nécessairement incomplète, et doit soit accepter de ne pouvoir valider certaines transformations, soit demander au compilateur des aides pour la vérification, qui prennent habituellement la forme d'une preuve complète de correction des transformations instanciées sur le programme source. Cette thèse propose, dans le cadre des compilateurs de tenseurs, une approche intermédiaire, à partir de la remarque suivante : de nombreux compilateurs de tenseurs sont construit comme des générateurs de code à partir d'une spécification de haut niveau. En ajoutant à la spécification des annotations triviales, celles-ci sont transformées comme le code par le compilateur. Le code engendré peut ensuite, grâce à ces annotations, être relié à la spécification initiale, et cette thèse montre que – sous l'hypothèse d'un flot de contrôle affine – ces annotations sont suffisantes pour construire un générateur de conditions de vérifications qui garantisse la correction du code engendré.

Dans cette thèse je fournis les fondations théoriques et une implémentation en OCaml d'un tel algorithme. Il se base sur des spécifications de haut niveau inspirée des SRAE, une représentation intermédiaire provenant du monde polyédrique, utilisée comme langage commun pouvant facilier, à terme, l'utilisation du système par différent compilateurs de tenseurs. En appliquant avec succès cette approche au compilateur Halide, je montre qu'il est possible d'instrumenter des compilateurs de tenseurs pour générer les annotations nécessaires à cette approche et sa faisabilité en pratique, montrant ainsi la viabilité de l'approche de validation de traduction pour les compilateurs de tenseurs.

Certaines questions restent toutefois non résolues, notamment concernant l'applicabilité de l'approche au-delà du cas purement affine (ou quasi-affine) : j'explore dans la version complète de la thèse en anglais une dizaine de telles questions qui ouvrent les perspectives futures de mon approche.

1 Introduction

Parallel Computing

In the second half of the twentieth century, the silicon industry was growing at a fast rate. In 1975, Gordon Moore predicted that the number of transistors in a circuit would double every two years — an exponential rate that has empirically held since, and has been known under the name of "Moore's Law". At first, the exponential increase in transistor density has translated directly into an exponential increase in computing power: switching to a new generation of processors would often directly translate into faster execution times, with little to no change to the program. Thermal dissipation issues put an end to that golden age of free performance improvements, and the additional transistors now translate to an increase in the number of functional units, whose clock speed remains roughly the same as that of the previous generation. Today, even that is slowly coming to an end as the chip fabrication process reaches the limits of what physics allow — plotting out a future where chip space is ever so valuable, and encouraging the design of purpose-built hardware tailored to the specific needs of performance-intensive applications.

The democratization of numerical computing The improvements to chip design allowed by Moore's law has fueled an explosion of computing devices both in data centers; through the rise of cloud computing, and in the hands of everyone; through the development of the now ubiquitous smartphones and otherwise increasingly "smart" appliances and devices. These heterogeneous devices embed general-purpose processors but also more specialized processors that can be orders of magnitude more efficient both in terms of performance and power usage for certain tasks such as graphics processing or cryptography.

The most common of these specialized processors are *graphics processing units*, or GPUs: originally developed as independent processors responsible for outputting pixels on a display, their highly parallel structure has proven invaluable for the implementation of many computationally heavy algorithms for image processing and linear algebra. The use of GPUs to implement non-graphics algorithms is known as GPGPU: general-purpose computing on graphics processing units. GPGPU, through frameworks such as CUDA for programming Nvidia GPUs, was a critical piece in making deep learning techniques viable at a previously impractical scale, leading to their widespread use since the early 2010s. This use is now pervasive, both in the data center and — fueled by privacy concerns — directly on users' devices, with applications ranging from computer vision to natural language processing. In turn, this is increasing the demand for performant numerical code outside the confines of traditional high-performance computing application domains such as the numerical simulations of scientists and engineers.

Writing efficient code for parallel architecture such as GPUs is notoriously difficult: in addition to the usual considerations about program semantics, the programmer needs to think about synchronization, shared caches, and memory coalescing optimizations in a programming model that has much different performance characteristics than the CPUs most are used to. Unoptimized programs written by novices that fail to account for the hardware's specificities can easily be orders of magnitude slower than properly optimized programs written by an expert. Thus, traditionally, efficient code has been written by experts and packaged into libraries (such as BLAS libraries for linear algebra) that provide highly optimized primitives. Users then have to express their problem using those optimized primitives, leaving massive performance improvements on the table when doing so is impossible or complex. With the inclusion of GPUs as components of increasingly diverse systems and the development, driven by the needs for both performance and energy efficiency, of specialized hardware such as Google's Tensor Processing Unit and Microsoft's Brainwave project, the reliance on a small set of handwritten primitives for each platform cannot scale. This drives a renewed interest in languages and frameworks aiming to improve the readability, portability, and efficiency of user-defined array programs.

A second era of parallel computing While improvements to chip design have led to steadily increasing clock frequencies for decades, memory has not followed, and the speed at which data can be accessed in a Von Neumann architecture compared to the speed at which it can be processed has drastically slowed, relatively speaking. Caches are part of the solution to this problem. Another solution is to increase instruction-level parallelism to ensure that the processor has work to do while memory transfers is in progress. Hardware architects have developed dynamic techniques such as out-of-order execution, branch prediction, and speculative execution to increase the instruction-level parallelism. These techniques work well but are local by nature and not able to exploit long-range parallelism opportunities. The necessary hardware is also competing with functional units for what is now extremely valuable chip space. The alternative is to rely on the programmer or the compiler to ensure that instructions and data are appropriately laid out in memory, taking into consideration cache sizes and the available parallelism in order to achieve optimal performance on a given hardware.

The Role of Languages

GPUs and other specialized processors require such precise compiler optimizations to achieve peak performance. This leads to an interest in the design of new tools and frameworks aiming at democratizing the availability of high performance numerical programming. This includes tools using true-and-tried techniques such as polyhedral compilation in Polymage [75], Tiramisu [7], and Tensor Comprehensions [111]; but also the exploration of new approaches. One line of research concerns the design of languages suited to the optimization of programs through *rewrite rules* and includes tools such as Accelerate [72], Futhark [50] or Rise [48]. Another line of research, introduced by Ragan-Kelley et al. [82] in the Halide domain-specific language and compiler, takes some ideas from polyhedral compilation through the separation of *algorithms* (with semantic properties) and *schedules* that guide the compiler through the process of generating efficient code. The schedule can be either written by an expert, or found through the use of automated techniques. Halide was originally developed in the context of image processing pipelines; with TVM [25] using the same approach with a focus on deep learning kernels.

These different approaches are *tensor compilers*, because they compile tensor specifications. To differentiate these tools that work with "pointful" specification (i.e. a specification that gives a value to each point in the tensor's domain) from higher-level frameworks such as TensorFlow [1] that operate on multidimensional operators, I will sometimes explicitly talk about *low-level* tensor compilers.

Due to the different factors mentioned above, new techniques are being developed to generate highly efficient numerical programs for an increasingly wide landscape of hardware. At the same time, more and more algorithms must be implemented efficiently. This poses the question: how do we ensure that the low-level programs generated for these combinations of algorithms, compilation techniques, and hardware are correct, and effectively implement the program that the user had in mind? A recent study by Shen et al. [99] on the source of bugs in deep learning compilers shows semantics issues in the compilation can cause bugs that are hard to catch because they silently return incorrect results. Testing is the traditional solution to this problem, but it is necessarily partial and full coverage cannot easily be achieved. Fuzzing is an extension of testing where test cases are generated automatically to try and uncover bugs in the compiler. The design of formally verified implementations of the compiler algorithms used is an interesting direction, but it has the main issue that as new compilation algorithms are being designed, a formally verified implementation of the new algorithms must be provided.

Translation validation provides an attractive middle ground between testing and formal verification. In translation validation approaches, a separate tool (that can be itself formally verified) is designed. The tool takes as input the source program and the output of the compiler, possibly extended with annotations, and tries to produce a proof that the transformations performed by the compiler are correct *on this instance*, i.e. that *this* optimized code is properly implementing *this* input program. Program equivalence being undecidable in general, translation validation is most often used to verify a single transformation pass in an optimizing compiler, ensuring the correction of the full compilation process through many small hops between consecutive states of the intermediate representation. Within the verified transformation pass, however, translation validation is mostly agnostic to the actual implementation, and only cares about the type of transformations that can be performed. Translation-validation is usually applied to specific pass of an optimizing compiler and can be specific to the transformation performed by the optimization pass.

In the domain of numerical computing, translation validation algorithms for array and loop programs have been proposed [59, 98, 116]. These algorithms have originally been developed under the assumption of human programmers performing code transformations manually e.g. in the context of embedded systems. These approaches have been successful at verifying transformations of realistic multimedia systems [117]; however, they do not compose well with the possibility to perform program transformations based on complex algebraic or (non-linear) arithmetic reasoning.

Existing approaches try to prove the equivalence between the original and the compiled program by asserting that the outputs of the computation must be identical and propagating that information backwards to determine which equalities must be satisfied by intermediate and input arrays. This limits the transformations on the array values that can be handled; even transformations that can be handled (e.g. associative and commutative operator re-orderings) can result in drastically increased execution times due to considering all possible permutations. These issues cannot easily be lifted: the equivalence problem that these approaches try to solve is undecidable by nature, as relations between the intermediate arrays in both programs must be inferred.

Instead, I propose an alternative approach relying on the presence of a compiler that can give hopefully reasonable hints through annotations, as I now outline.

Motivating Examples

Picture yourself a specification language for n-dimensional arrays. The exact details of the specification language may vary, but can be roughly thought of as follows. This language deals with tensors: functional values indexed by integer points in a multidimensional space. The tensors are defined by equations, read as assignments, and implicitly quantified over the multidimensional domain. For instance, the outer product of two vectors A and B is a two-dimensional tensor defined by the equation:

$$C(i, j) = A(i) \times B(j) \tag{1.1}$$

Here i and j are implicitly bound variables ranging over arbitrary integer values.

These high-level specifications are compiled into imperative programs through a code generation process guided by a (user-provided or compiler-generated) schedule, whose concrete details vary. In the generated program, the tensors are replaced by arrays representing finite subdomains of the corresponding tensor. Our first intuition is that the nature of the code generator makes it possible to keep track of a mapping between array writes and tensor definitions: this property will be at the basis of our verification algorithm.

Going back to Eq. (1.1), in an implementation of that specification, A, B and C become arrays, and the range of `i` and `j` are computed through auxiliary mechanisms such as inference from the shape of the input arrays (as in Tensor Comprehensions), or an explicit realization domain for the output array (as in Halide). In Halide, expert users can manually write schedules using a domain-specific language; in our example, the schedule `C.split(i, i0, i1, 4).reorder(i1, j, i0)` would generate the following implementation, expressed in pseudocode (recall that lower-case names and brackets are used for array accesses while upper-case names and parentheses are used for tensor accesses).

```
for i0 = 0 to (N + 3) / 4 - 1 do
  for j = 0 to M - 1 do
    for i1 = 0 to 3 do
      let i = min(i0 * 4, N - 4) + i1 in
      c[i, j] := b[j] * a[i]
```

The `split` directive indicates that the loop over `i` should be separated into an outer loop `i0` and an inner loop `i1`. The loop nesting order, from innermost to outermost, is specified by the `reorder` directive. More examples of the Halide scheduling language are shown by Ragan-Kelley et al. [83]. The commutativity of $\times$ (i.e. `b[j] * a[i]` instead of `a[i] * b[j]`) has been applied manually for illustration purposes: it does not appear in the schedule. More generally, Halide features a simplification algorithm which can propagate constants and make use of algebraic properties such as commutativity, associativity and distributivity: the values stored in the array `c` can be computed using a different, but equivalent, expression than that defining tensor C.

Halide makes fast-math assumptions on floating-point numbers, which we discuss in Section 9.4.

The parameters N and M used in the code correspond to the size of the arrays `a` and `b`, and are not statically known; therefore, the program ought to be valid for any such sizes.

In this case, it is relatively easy to convince oneself that the implementation follows the semantics prescribed by the specification. More precisely, for any tensors A, B and C such that $C(i,j) = A(i) \times B(j)$, and any initial memory mapping the `a[i]` to $A(i)$ and the `b[j]` to $B(j)$, running the above program will produce a final memory which, in addition, maps the `c[i, j]` to $C(i,j)$, where i ranges over $0, \ldots, N-1$ and j over $0, \ldots, M-1$.

There are multiple proofs of this fact, but we will focus on the following one, which exploits the remark that, when executing an assignment `c[i, j] := b[j] * a[i]`, the value written is always the corresponding $C(i,j)$ from the specification. More precisely, we can make the following observations:

- The set of locations written by the assignment `c[i, j] := b[j] * a[i]` is precisely

$$\left\{ c[\min(4i_0, N-4) + i_1, j] \;\middle|\; \begin{array}{l} 0 \leqslant i_0 < \lceil N/4 \rceil \\ 0 \leqslant i_1 < 4 \wedge 0 \leqslant j < M \end{array} \right\}$$

  This set contains the "required" set of locations $\{c[i,j] \mid 0 \leqslant i < N \wedge 0 \leqslant j < M\}$, which is expressed as an inclusion of sets defined by quasi-affine formulas. Quasi-affine formulas are affine formulas extended with division and modulo operations where the denominator is a constant. The decidability properties of Presburger arithmetic [78] extend to quasi-affine formulas, and efficient specialized solvers such as `isl` [112] can be used to check its validity.

- When the assignment is executed for some values of i and j, we have $0 \leqslant i < N$ and $0 \leqslant j < M$, which can also be checked using `isl`, and ensures that reads from arrays `a` and `b` are in bounds. Note that this is not the same check as above: here we bound the set of locations *upwards* to stay within the bounds for `a` and `b`, while earlier we bounded that set *downwards* to at least contain the required writes to `c`. Combining these two checks, the set of computed locations must be exactly $\{c[i,j] \mid 0 \leqslant i < N \wedge 0 \leqslant j < M\}$.

- When the assignment is executed for some values of i and j, the value in `b[j]` is $B(j)$ and the value in `a[i]` is $A(i)$, as per the previous point, and because arrays `a` and `b` are never written to. Hence, the value written to `c[i, j]` is equal to $B(j) \times A(i)$. This value is equal to $C(i,j)$, since $C(i,j) =$

$A(i) \times B(j)$ by definition and assuming $\times$ is commutative. In general, proving this type of equality requires unfolding tensor definitions and checking the algebraic reasoning performed by the compiler's simplifier. Off-the-shelf general purpose SMT solvers such as Z3 [74] used in this work, are quite good at proving such formulas.

These three checks — coverage of writes, definedness of reads, equality of values — form the backbone of a verifier for array programs. Here, we have seen a simple example where a single value is ever written to each location; in general, the resulting program can contain recurrences: an iteration of a loop which depend on values read by a previous iteration of the same loop. This would make the proof fail: we have assumed that, when reading from arrays a and b, we know exactly the value they hold. In the presence of recurrences, this is in general impossible, as it requires unfolding many iterations of the loop at once. This property is also the main difficulty for program equivalence checking approaches, where best-effort techniques based on transitive closure [97] or affine hulls [116] have been developed.

Instead, we can side-step the issue entirely by relying on the fact we are dealing with implementations generated by a scheduling compiler, which has a rich set of information about the assignments available. Let us assume that the assignment is annotated with a *prophetic expression*. The prophetic expression lives in the specification world, and predicts the value that will be written by the assignment in terms of tensors. Assuming that the compiler can produce those annotations along with the code, this breaks the cycle, because the prophetic expression uses tensors, and hence is independent of the program memory. In particular, this means that we can always know the value of the prophetic expression for any iteration of a loop without having to execute the previous iterations of the loop.

Let us examine an example of this by considering matrix multiplication, implemented with an explicit accumulator R:

$$\begin{aligned} R(i,j,k) &= \text{if } k \leqslant 0 \text{ then } 0 \text{ else } R(i,j,k-1) + A(i,k) \times B(k,j) \\ C(i,j) &= R(i,j,P-1) \end{aligned}$$

Reusing the same schedule as earlier, we would get the following implementation, where we have annotated each assignment with a prophetic expression:

```
for i0 = 0 to (N + 3) / 4) - 1 do
  for j = 0 to M - 1 do
    for i1 = 0 to 3 do
      let i = min(i0 * 4, N - 4) + i1 in
      r {0} := 0
      for k = 0 to P - 1 do
  r {R(i,j,k)} := r + b[k, j] * a[i, k]
      c[i, j] {C(i,j)} := r
```

Prophetic expressions, here and throughout the manuscript, are written using math font and (on the electronic version of this manuscript) a different color in order to distinguish them from the surrounding code.

In an assignment such as `c[i, j] {C(i,j)} := r`, the prophetic expression $C(i,j)$ between brackets is ignored during the execution of the program, and only used for the validation. It is an assertion than the value written when executing the statement will be $C(i,j)$.

Without the annotations, it would not be immediately clear what the value of `r` should be at iteration k of the loop. However, by using the annotations, we can build a *prophetic* version of the program, which reads from the specification and executes assignments by using their prophetic expression instead of the right-hand side.

This prophetic version of the program never reads from mutable memory, making its analysis much easier: knowing the last write to an array cell is enough to know its value. In particular, it is clear from the prophetic version of the program that `r` is equal to either 0 (when $k = 0$) or $R(i,j,k-1)$ (otherwise) when updating `r` within the loop on k, and equal to $R(i,j,P-1)$ (assuming $P > 0$) when writing to `c[i, j]`. In addition, the fact that $C(i,j)$ is written by the prophetic program to the cell `c[i, j]` for all $0 \leqslant i < N$ and $0 \leqslant j < M$ is also clear, by the same set inclusion as in the outer product case.

Finally, we have to prove that the regular version of the program has the same behavior as the prophetic version of the program. To do so, we have to prove the following side conditions for any values of i, j and k reachable during

execution of the program:

$$R(i,j,k) = \text{if } k \leqslant 0 \text{ then } 0 \text{ else } R(i,j,k-1) + B(k,j) \times A(i,k)$$
$$C(i,j) = \text{if } P \leqslant 0 \text{ then } 0 \text{ else } R(i,j,P-1)$$

The right-hand side of the equalities are computed from the right-hand side of the assignments of `r` and `c[i, j]`, where the reads to `r` are computed using the prophetic expressions. The side conditions are an inductive invariant: they ensure that our reasoning propagates from one iteration of the loop to the next.

We now have reduced the correctness of our program to these two quantified equalities in the specification. In and of itself, that is valuable, thanks to the simplifications performed above: in the formulas, the structure of the implementation has been erased, yielding a simpler domain. The equalities in this case can be proven easily by an SMT solver.

To recapitulate, the approach outlined here requires two key ingredients to be automated:

- The assignments must be annotated with prophetic expressions in the specification, using tensors and independent of the program memory. This enables the use of symbolic evaluation for loops of parametric size.
- The expressions used in array indices (both for reads and writes), loop bounds, and conditionals must be quasi-affine. This ensures that we can keep track of the values written to and read from arrays.

Contributions

The contributions of this thesis follow from the remark that most of the difficulties in the verification of tensor compilers vanish if the compiler is able to provide sufficiently precise annotations relating the intermediate values in the compiled program and in the original specification. This leads to the development of a validator for the *black-box* combination of affine transformations including structure-modifying loop nest transformations or array

layout transformations, and algebraic transformations of the right-hand side of assignments, under the only assumption that the compiler is instrumented to provide the appropriate annotations. More precisely, this thesis makes the following contributions:

- The design of an intermediate language for the validation of tensor compilers, Sched, that closely follows the Stmt intermediate language of Halide augmented with *prophetic annotations*.

- Two semantics for Sched, a traditional interleaving small-step semantics and a novel deterministic big-step semantics that captures the behavior of race-free parallel programs, and a reduction from the big-step semantics to the interleaving semantics, ensuring that programs with a big-step semantics have a *deterministic* interleaving semantics.

- The design, implementation, and formalization of a verifier for Sched programs with respect to a system of affine recurrence equations (SARE) that relies on the refinement mapping provided by the prophetic annotations.

- A novel formalization of Halide as a system of equations, and the derivation of a reduction from affine Halide specifications to SAREs.

- The instrumentation of the Halide compiler to augment its intermediate language Stmt with annotations, and the implementation of both the reduction from Halide to SAREs and of annotated Stmt to Sched.

- The experimental evaluation of the verifier for Sched programs, using the instrumented Halide compiler on affine specifications extracted from the official Halide benchmarks.

- The formalization (but not the implementation) of an extension to the verifier to handle the reordering of *reductions*, an important primitive in numerical computing representing the possible re-orderings of associative and commutative operators and exploited by tensor compilers to expose additional parallelism.

Organization of this Manuscript The first chapter is this introduction and presents the context and motivations for the need to validate low-level tensor compilers.

The second chapter goes over an historical overview of the representations used by compilers of computation-intensive numerical programs, with an assumed focus on the polyhedral representation that has driven much research interest and on the lightweight scheduling approach originating in the Halide DSL and compiler that was used in the experimental portion of this thesis.

The third chapter is a reference on the abstraction of Presburger sets and Presburger relations, originating from polyhedral compilers and implemented by libraries such as `isl` [112]. This representation is used to build a symbolic representation of programs in the later chapters of the manuscript.

The fourth chapter presents SCHED, an imperative parallel language of loops and arrays with annotations relating the assignments to so-called "prophetic" expressions in a tensor specification. The chapter gives a small-step interleaving semantics to programs, and a big-step deterministic semantics to race-free programs that is proven sound with respect to the small-step semantics. Finally, a type and effect system is given for SCHED program capturing all the writes it performs, expressed as prophetic expressions. The type and effect system is proved sound for programs that have a big-step semantics where all annotations hold at runtime.

In the fifth chapter, a symbolic evaluator is added to the SCHED language by augmenting the type and effect system with a verification condition generator. It is proved that when the verification conditions hold, the symbolic evaluator exactly captures the writes and the reads performed by a program, which allows verifying a generated program with respect to its specification.

The sixth chapter discusses an implementation of the symbolic evaluator from the previous chapter in OCaml, and its application on affine specifications extracted from the officials benchmarks provided with the Halide compiler. The implementation compares favorably in terms of runtime and coverage with the isa tool from Verdoolaege, Janssens, and Bruynooghe [116], a state-of-the-art program equivalence checking tool for affine programs.

The seventh chapter discusses the context of *reductions*, the repeated application

of an associative-commutative operator such as a summation that appear in many tensor specifications. The computations in a reduction can be arbitrarily reordered, a property that tensor compilers exploit to better optimize for parallelism and cache locality. Such reorderings cannot be verified by the previously introduced approach, nor by most of the literature on affine program equivalence checking. In the chapter, an extension of the prophetic annotations and of the symbolic evaluator is proposed to be able to handle these reorderings. This extension has not been implemented in the OCaml tool, and no experimental evaluation is available.

The eighth chapter compares the solution proposed by this thesis with related works in translation validation, affine program equivalence checking, and formalizations of tensor specifications or compilers.

The ninth and final chapter is a conclusion, recapitulating the work performed in the thesis and placing it in a larger context. The conclusion also contains an ample discussion of possible future work to apply the approach of this thesis to larger classes of programs such as histograms and compiler transformations such as parametric tiling.

Notations and Conventions Throughout this manuscript, we will use the term "tensor" and "array" abundantly with a rather precise semantics. A *tensor* is a function from a tuple of integers to a value. We denote tensor accesses using uppercase letters and parentheses, e.g. $A(3,7)$ denotes the value of tensor A at position $(3,7)$. Tensors are used in functional specifications and a tensor access refers to a unique, immutable value. On the other hand, an *array* is used in an imperative programs, and is a mutable object that can be updated throughout the execution of the program. We denote array accesses using lowercase letters and brackets, e.g. $a[3,7]$ denotes the access to array a at position $(3,7)$. The values associated with an array access can change during the execution of a program and depend on the computer memory. In examples, it is generally assumed that values stored in an array correspond to the values associated with the tensor of the same (uppercase) name and corresponding index in the specification.

Previous Publications Part of this thesis (notably, parts of chapter 4, chapter 5 and chapter 6) are published in [27].

2 Representations of Programs with Loops and Arrays

The need to optimize numerical programs operating on arrays is not a recent development: the use of computers to help in the resolution of numerical problems occurring in physics and mathematics has been a driving force behind the development of computer science since its infancy. The techniques that are used today to represent and optimize numerical array programs are the fruits of a rich history of array optimization research and have evolved through the years along with the hardware they target. This chapter is intended to give some background and history on the compilation techniques for computation-intensive array programs that are used in modern tools and research.

The reader should keep in mind, while reading this chapter, that this manuscript is concerned with the *verification* of the program transformations performed by tensor compilers. Hence, this chapter is intended to help put the verification techniques developed in later chapters in the context of the transformations they are designed for. This means that *optimization* is neither a concern nor a focus here, and work that focuses on these aspects will only be mentioned when it contributes to the discussion on the representation of programs or transformations. Generally speaking, we are interested in understanding *what* transformations the compiler *can* apply, not *when* or *where* they will be applied.

This chapter is split into three sections. First, I present the history of the *polyhedral model*: a framework for the representation and optimization of array programs that has long been the focus of parallel programming research. The polyhedral model enables the compact representation of combinations of many loop and data-layout transformations; it also draws interest for its ability to express performance-impacting characteristics such as parallelism and cache

locality as objectives of an optimization problem. Then, I briefly present work in a different line of research on using a functional representation of array programs with combinators optimized using rewrite rules. Although this line of research is promising, especially regarding the possibility of building formally verified compilers, it is not the focus of this thesis, which is concerned with techniques to validate and test existing compilers. Finally, I present the approach to the organization of computation championed by Halide: by taking some ideas from the polyhedral model (namely, a separation between *what* is computed — the "algorithm" — and *how* it is computed) while rejecting some of its core ideas (e.g. Halide eschews the expressive power of Presburger arithmetic fundamental to the polyhedral model and replaces it with a much simpler interval analysis), the approach seems to hit a sweet spot between expressiveness, performance, and ease-of-use by non-expert practitioners. This section also includes a new formalization of Halide algorithms as a set of equations, and a reduction from affine Halide algorithms to SAREs, both being novel contributions.

2.1 The polyhedral model

The polyhedral model is an abstract representation of a program as a computation graph [40]. The nodes of the computation graph represent *statement instances*, i.e. iterations of a statement parameterized by their position in a multidimensional space. Also known as the polyhedron model, the polytope model, or the geometric model, the polyhedral model has had many incarnations and alternate presentations; it is used both for program transformation in compilers and for program analysis and verification.

In this section, I present an overview of the polyhedral model as a tool for the representation and transformation of programs with arrays and loops. The rest of the ideas presented in this thesis have roots in the polyhedral model, notably in the exact representation of dependencies used by polyhedral tools; as such, having a good grasp of the semantics representation of programs used by the polyhedral model should help the reader put the rest of the thesis in context. The presentation is roughly chronological, with different axes for the concepts of the model: instance sets and schedules for the representation of programs in the model; code generation for the extraction of traditional

program representations out of the model; dependence analysis for the validity of transformations in the model. The history of the polyhedral model is tightly related to the mathematical concept of Systems of Affine Recurrence Equations (SAREs) that have been developed jointly. Any loop nest whose semantic is exactly captured by the polyhedral model can be represented as a SAREs, and they are used as intermediate or input representations in some polyhedral compilers [119]. SARE can be used as a pure, equational specification for polyhedral programs and are presented in section 2.2.

2.1.1 Instance sets

While the terminology was different and some concepts implicit, key ingredients of the polyhedral model can already be found in the work of Lamport [65]. In this seminal work, the computations performed by a perfect loop nest are modelled as the repetition of the loop body for each value of the loop iterators. Under this view, it is possible to construct a different loop nest with the same body but with a different iteration order while preserving the program's behavior. A well-chosen transformation can reveal latent parallelism opportunities, resulting in a program that can be executed more efficiently on a multiprocessor machine such as the Illiac IV computer that motivated his work.

Using modern terminology, the work of Lamport could be presented as follows. At the core of the approach is a simple idea: a counted for loop can be understood as the set of its iterations. Assuming a single base instruction I representing its body, a perfect loop nest can be described by the following grammar:

$$s ::= \text{for } i < e; \text{do } s \mid \mathrm{I}(e_1, \ldots, e_n)$$

where the loop body I is parameterized by expressions $e_1, \ldots, e_n$. The expressions appearing in the loop bounds and as argument of the loop body can only use the outer loop variables, as well as *program parameters*: variables that are in scope for the whole execution of the program and whose value is unknown but constant during said execution. Typically, the program parameters correspond to the dimensions of the input and output arrays. The representation of the loop body I is arbitrary; it will be assumed to be context-independent, i.e. I

does not contain any free variables so that all uses of the loop variables go through the expressions $e_1, \ldots, e_n$.

It should be noted that while we talk about a "program", the appropriate terminology would be that of a program fragment, such as the body of a function, or even part of the body of a function. An example of such a perfect loop nest is the computationally-intensive main loop of a matrix multiplication algorithm:

```
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    for k = 0 to P - 1 do
      c[i, j] += a[i, k] * b[k, j]
```

where the base instruction is $I(i, j, k) = c[i,j] += a[i,k] * b[k,j]$. Another example adapted from Lamport is:

```
for i = 2 to M do
  for j = 1 to N do
    a[i, j] := b[i, j] + c[i] ;
    c[i] := b[i - 1, j] ;
    b[i, j] := a[i + 1, j] * a[i + 1, j ] ;
```

where the base instruction $I(i, j)$ is the full loop body.

While the instruction I is present only once in the program text or in traditional compiler representations such as abstract syntax trees and control-flow graphs, it will be executed many times when running the program, for each value of the loop iterators. The gist of the polyhedral representation of programs is to make explicit this dynamic set of executions. For instance in the case of the matrix multiplication it is the parametric set:

$$\{I(i, j, k) \mid 0 \leqslant i < N \wedge 0 \leqslant j < M \wedge 0 \leqslant k < P\}$$

Definition 2.1.1 (Instance set)**.** The *instance set* of a program P is the set of all dynamic executions of the program's instructions. Elements of the instance set such as $I(0, 1, 3)$ are called *instruction instances*, *statement instances*, or simply *instances*.

Instance sets are also called *index set* and *iteration domain* in the literature. Different representations of instance sets exist. While I have chosen a presentation where the elements of the instance sets contain both an instruction and its position in the iteration space, many authors choose to instead have one instance set per statement. In this case, the instance set only contains points in the iteration space, and I will prefer the terminology of *index set*. The elements of an index set are ordered tuples containing the value of the loop iterators surrounding the statement, and are also known as iteration vectors

I have already mentioned "points in the iteration space"; indeed, as tuples, iteration vectors are points with integer coefficients in a multidimensional space, and can be thought of and drawn as such. This gives a geometric representation of index sets, giving the model the name of *geometric model*. This is a natural remark, and Lamport's article already contains representations of a program's iterations as points in a two-dimensional space to explain the transformations. The name of "polyhedral model" comes from the necessity to impose restrictions on the sets of points that we are able to handle while keeping the representation practical: one of the original goals of the model, and certainly that of Lamport, is to enable the automatic parallelization of programs. To satisfy this requirement, the representation of index sets must be expressible using a reasonably "nice" fragment of logic so that automated tools can manipulate them. A naturally "nice" fragment of logic is obtained by adding restrictions to the type of expressions appearing in loop bounds: only affine combinations of the outer loop variables and the program parameters are allowed. Loop bounds are of the form $a_1x_1 + \cdots + a_nx_n + c$ where $x_1, \ldots, x_n$ are loop variables or program parameters and $a_1, \ldots, a_n$ and c are integer constants. With this restriction, the index set of a program can be represented as a conjunction of affine constraints: equivalently, when interpreting the index set geometrically, it represents the interior points with integral coordinates of a convex polyhedron.

2.1.2 One polyhedron, many polyhedra

This restriction to affine or mostly affine programs has been crucial to the practical applicability of approaches based on the polyhedral model, as it makes many of the problems presented below decidable using linear programming

techniques. It has also, historically, been seen as somewhat of a weakness of the model, and I should mention that techniques have been developed to improve the representative power of the model, although they are outside the scope of this presentation. The work of Benabderrahmane et al. [16] gives a good overview of some of these techniques.

It is still worth mentioning how modern approaches to the "core" polyhedral representation fare compared to Lamport's restrictions half a century ago so as not to build an opinion of the model that is too biased. The proper restriction that still allows programs to be represented and manipulated using linear programming techniques is a bit wider: modern polyhedral tools are able to handle the full expressive power of Presburger arithmetic extended with divisions by a constant positive number, i.e. quasi-affine arithmetic. This is described in more details in chapter 3; for the goal of the current overview of the polyhedral model, it suffices to say that in modern approaches, instance sets are represented using unions of polyhedra and that the answers and approaches mentioned here apply just as well to piece-wise quasi-affine expressions and unions of polyhedra as they do to affine expressions and polyhedra.

In a similar vein, for the sake of simplicity and following Lamport's research, I have only mentioned perfectly nested loops whose body contains a single instruction. Modern approaches based on the polyhedral model are concerned with *im*perfectly nested loop nests with conditionals and potentially many statements, a more realistic characterization of scientific and numerical code. Imperfect loop nests can be defined by the grammar:

$$s ::= \text{for } i < e; \text{do } s \mid s\ ;\ s \mid \text{if } e \text{ then } s \text{ else } s \mid I(e_1, \ldots, e_n) \mid \text{skip}$$

The symbol I stands for an arbitrary piece of code that does not have free variables, but can depend on the value of the contextual arguments $e_1, \ldots, e_n$. Like the expressions for loop bounds, the expressions appearing in the conditionals and as contextual arguments to the instructions are restricted to affine expressions of the loop iterators — or rather, piece-wise quasi-affine expressions thereof. This again ensures that programs can be represented and questions answered using linear integer programming. The fragment of programs of imperfect loop nests that can be represented using this approach is commonly known as a "Static Control Part" of the program, referring to the data-independent nature of the control flow within. We will use if e $\{s\}$ as a shorthand for a conditional whose else branch is empty, i.e. if e then s else skip. Note

that conditionals are included as a convenience, and are not strictly necessary in the presence of piece-wise expressions. Assuming that select(c, t, e) is a conditional that returns the value of t when c is true and e when c is false, the construct if e then s_1 else s_2 can also be expressed as follows, assuming that x is a variable that does not appear in either s_1 or s_2:

$$\text{for } x < \text{select}(e, 0, 1); \text{do } s_1 \text{ ; for } x < \text{select}(e, 1, 0); \text{do } s_2$$

In a polyhedral representation of an imperfect loop nest, it is implicitly assumed that each statement occurs only once in the program source, because the polyhedral model has no real notion of multiplicity. There is now a distinct index set for each of the statements, which can a priori have different dimensionalities if they are not nested under the same number of loops. Equivalently, the instance set now contains points in spaces of potentially heterogeneous dimensionalities for each of the statements in the program. Often, and in particular for code generation, instances in these different spaces are *aligned* up to the highest dimensionality by filling in the missing dimensions with zeroes; while crude, this is a good way to think about the polyhedral model for imperfect loop nests as a generalization of the case of perfect loop nests.

In some representations it is possible to duplicate statements, e.g. by tagging multiple copies of the statement as virtually distinct; however, this must be done ahead of time and is not well-supported by automated optimizers.

2.1.3 Program order

The instance set represents the set of *dynamic* instructions that gets executed during a run of the program, and is one of the core abstractions of the polyhedral model. However, the instance set alone is not enough to represent the behavior of a program: instructions are imperative in nature and their semantics depends on their execution order, which cannot be captured by the unordered nature of a set representation. Instead, the polyhedral model seeks to explicitly model the execution order as a strict partial order $<$ over the instances: if u and v are two instances such that $u < v$, u is executed before v during all executions of the program. The partial nature of $<$ represents parallel programs: two incomparable instances u and v can be executed in parallel.

The execution order relates pairs of instances; in the case of the matrix multiplication algorithm presented above, the original execution order is:

$$I(i, j, k) < I(i', j', k') \Leftrightarrow i < i' \vee (i = i' \wedge (j < j' \vee (j = j' \wedge k < k')))$$

On the other hand, the computations for different values of i and j are independent and can be represented in parallel, as in the following pseudocode, where par represents a loop whose iterations are executed in parallel:

```
par i = 0 to N - 1, j = 0 to M - 1 do
  for k = 0 to P - 1 do
    c[i, j] += a[i, k] * b[k, j]
```

The corresponding execution order only relates instances at the same iteration of loops i and j:

$$I(i, j, k) \prec I(i', j', k') \Leftrightarrow i = i' \wedge j = j' \wedge k < k'$$

This split representation of programs as an instance set equipped with a partial order raises the following questions about its practical use:

- How do we build this program representation from an input representation such as an abstract syntax tree or other compiler intermediate representations?
- What is a good representation for the partial order that is practical for use within a compiler or analysis framework?
- What are the conditions for two programs with the same instance set but different partial execution orders to be semantically equivalent?
- Given an input program, how do we find an efficient execution order (e.g. in terms of exposed parallelism) that is semantically equivalent?
- How can we build a concrete program that can be compiled down to machine code from such a representation?

In the context of this presentation, we are only considering well-formed loops without arbitrary control flow that renders the first question mostly uninteresting, and it will not be treated further; however, it should not be dismissed as entirely obvious, in particular when applying the polyhedral model to fragments of general-purpose programming languages such as C and Fortran. The remainder of these questions are at the root of problems such as

dependence analysis, scheduling, optimization, and code generation — all of which are cornerstones of the polyhedral model.

2.1.4 Scheduling

The partial order $<$ is rarely represented explicitly both due to the inherently quadratic nature of such a representation, and because it is unclear how to perform code generation directly using the partial order. Instead, more compact representations, called *schedules*, are sought. A schedule is a function θ that maps each statement instance to a point in a partially ordered set, whose representation is chosen adequately to satisfy some "good properties". A good representation for the partial order must be compatible with code generation, that is, it should be possible to generate a loop structure from the schedule representation that is independent of the value of the program parameters. In addition, one of the goals of the polyhedral model is program optimization and parallelization: it should be possible to express profitability heuristics as optimization problems on the schedule representation. Since I am mainly concerned about questions of semantic equivalence between programs, I will eschew questions about the inner workings of such optimizers; instead, I will focus on what is representable as a schedule without worrying about the possibility for a specific algorithm to exhibit a particular schedule.

A formal treatment of the schedules used in the polyhedral model can be found in [71]. A schedule θ is a function mapping statement instances to a multidimensional *schedule domain*, a subset of integer tuples of a fixed size equipped with the lexicographic order $\prec$. To ensure that they are compatible with automated tools, the schedules of the polyhedral model are limited to piece-wise affine functions (or piece-wise quasi-affine functions using modern tools) of the iteration vector. For instance the execution order of the original loop nest can be captured by the trivial schedule $\theta_0(I(i,j,k)) = (i,j,k)$ while the following program where the iterations of loops i and j are performed in parallel has the same instance set but schedule $\theta_1(I(i,j,k)) = (k)$:

```
for k = 0 to P - 1 do
  par i = 0 to N - 1, j = 0 to M - 1 do
    c[i, j] += a[i, k] * b[k, j]
```

The attentive reader can notice that there is one sequential loop in the program representation for each schedule dimension, a property that will be ensured by code generators.

Schedules are a compact representation of the program order $\prec$, but not all program orders can be represented by schedules. In particular, since the lexicographic order is a total order, polyhedral schedules can only express inner parallelism by mapping several instances to the same value. The "maximally parallel" program order mentioned above cannot be expressed using such schedules. This is noted by Pugh [79], who also shows that in spite of such restrictions, affine schedules can be used to represent most of the sequential loop transformations studied in the literature such as loop interchange, loop skewing, loop tiling, and loop reversal. Pugh focuses on the use of unidimensional schedules with a proposed recursive application of his method when multiple sequential loops are needed. Lu [71] provides a more general presentation of multidimensional schedules, and extends the representation to piece-wise affine schedules. Such multidimensional schedules are now a staple of the polyhedral model, although there has been some interest in exploring the use of uni-dimensional polynomial schedules instead [39].

The *permutable bands* of the Pluto algorithm propose a solution to the problem of representing outer parallelism using polyhedral schedules: certain dimensions of the schedule space can be marked as parallel, and are considered as unordered by the lexicographic order. For instance, if we consider a three-dimensional space (i, j, k) where j is such a parallel dimension, the partial execution order is given by:

$$(i, j, k) \prec (i', j', k') \Leftrightarrow i < i' \vee (i = i' \wedge j = j' \wedge k < k')$$

Using this approach, the schedule for the maximally parallel program earlier is $\theta(I(i, j, k)) = (i, j, k)$ where both i and j are marked as parallel. Multiple consecutive parallel dimensions can be implemented by a combined parallel loop in any order: such consecutive parallel dimensions are called permutable bands.

The idea behind permutable bands can be traced back to older scheduling algorithms such as that of Lim, Cheong, and Lam [68]. Those algorithms could not express the nesting of parallel bands within sequential loops. Modern schedule representations such as that of Verdoolaege et al. [118] represent schedules as a

hierarchy of nodes representing permutable bands, sequential/lexicographic schedules, and additional structuring nodes.

2.1.5 Code generation

Once a schedule has been found for a program, a new problem arises: the polyhedral model has served its goal, and a transformed program (often under the form of an abstract syntax tree) must be generated from the schedule to be used by further compilation passes. Like the schedule optimization, the code generation algorithm depends on the representation of the schedule; however, unlike schedule optimization, code generation raises non-trivial semantics questions.

In the case of a single instruction and an affine schedule, Lamport [65] shows that it is possible to extend the schedule into a one-to-one affine mapping to a larger space, so that the coordinates of the schedule coincide with the first few coordinates of the one-to-one mapping. The construction of the one-to-one mapping does not depend on the program parameters and can thus be used to reconstruct a loop nest compatible with the schedule's order: the first components corresponding to the schedule dimensions are implemented using nested sequential loops, while the remaining components are implemented as a single multidimensional parallel loop (that might be split up or recomposed by later compilation passes). Since every relevant piece of information is affine, the bounds of the loops can be recovered from the index set and the one-to-one mapping using integer linear programming. Other early works in the domain such as those of Pugh [79] use similar techniques; with piece-wise schedules such as those of Lu [71], extra conditionals are introduced for each piece, leading to coarse code duplication.

Ancourt and Irigoin [4] give the first formal treatment of the code generation process in the polyhedral model. Their polyhedra scanning method uses Fourier-Motzkin elimination to generate a perfect loop nest scanning the points in a single polyhedron obtained by applying an affine schedule to an input affine loop nest. The schedule is required to be unimodular, i.e. it must have determinant $+1$ or -1; the inverse schedule S^{-1} is applied to the vector representing the loop iterators to reconstruct the original statement position

in the input program. Fourier-Motzkin elimination is used to project the constraints on an increasing number of dimensions, yielding minimum and maximum bounds that only depends on the outer loop iterators. The original constraints are kept as guards on the inner loop and redundant constraints are removed using Fourier-Motzkin elimination, a sound but incomplete procedure due to Fourier-Motzkin elimination not being exact on integer sets. The technique can be thought of as generating code with inefficient control flow, then trying to "clean up" the code to remove control overhead. It has been improved further by Fur [41] using the simplex method instead of Fourier-Motzkin to eliminate redundant constraints. Kelly, Pugh, and Rosser [61] adapted the technique to handle multiple polyhedra by generating multiple perfectly nested loops then removing redundant conditionals when possible. Last in this line of work, Chen [24] proposed several improvements to the simplification phase in order to minimize control overhead further.

The techniques described in the previous paragraph use a two-step approach of first generating naïve code with inefficient control flow, then eliminating redundancies to improve the control overhead in a second step. Another line of research tries to directly generate efficient code with no control overhead, obviating the need for the second phase entirely. The first approach following this line of thought is presented by Quilleré, Rajopadhye, and Wilde [81]. This top-down approach relies on the idea of *separating* a polyhedron or union of polyhedra into two or more disjoint unions of polyhedra according to an appropriate criterion. Starting from a top-level loop, the domain is separated into disjoint parts that can be sorted textually in accordance with the schedule order: the projection of each part on the current loop are disjoint intervals. Applied recursively, this allows generating code that is free of conditionals, except for some conditions involving modulo expressions; on the other hand, this approach is prone to code explosion.

Bastoul [15] improved upon Quilleré's algorithm to limit the code explosion by preventing some unneeded splitting and introduced a technique to undo splitting after the fact when possible. Implemented in the CLooG code generator, another key innovation of this work is the ability to handle arbitrary schedule functions without restrictions such as unimodularity. This is achieved by keeping the original dimensions in the polyhedron that is given to the code generation algorithm. The original dimensions are considered as additional inner dimensions; once the code generation process reaches these dimensions, loops are only generated for them if they cannot be directly expressed in

terms of the schedule dimensions. The "un-splitting" technique of Bastoul is reminiscent of the control overhead removal techniques of the previous paragraph; Vasilache, Bastoul, and Cohen [110] goes further in that direction by proposing to use additional control overhead removal techniques on top of the code generation algorithm.

Schedule trees are a modern and flexible schedule representation introduced by Verdoolaege et al. [118] and refined by Grosser, Verdoolaege, and Cohen [44]. Schedule trees are built as a tree of nodes of various types capturing realistic use cases in applications such as the PPCG polyhedral compiler, and traditional polyhedral schedules are but one type of node (albeit central). The code generation algorithm for schedule trees is built on top of Quilleré's code generation algorithm, that the authors augmented with extensions such as using strided loops to replace conditionals with modulo when possible and an isolation mechanism giving the user some control code generation process by specifying portions of the space to be processed separately. More interestingly in terms of reducing code explosion, the improved algorithm uses a component analysis exploiting the fact that two instances scheduled at the same time can be arbitrarily reordered to avoid separating lone statements when possible, resulting in more compact code compared to Bastoul's post-processing approach. Razanajato, Loechner, and Bastoul [84] investigate the impact of the separation heuristics on the performance of the generated code and show that more compact code is not necessarily more performant. Less compact code can be more specialized and can often end up with simpler expressions for loop bounds when decomposing polyhedrons as a composition of simpler shapes such as rectangles and triangles. Compared to existing approach, their aggressively splitting method generates code with better performance in some cases, and worse performance in other cases, showing that the code generation problem in the polyhedral model might not yet be solved in a satisfactory way.

Quilleré's code generation algorithm, as well as some of the improvements by Bastoul, has been formally verified in Coq by Courant and Leroy [29]. This formalization has led to the discovery of an intermediate language for loops over polyhedron (not only intervals) that shares some properties with schedule trees but is overall simpler, and uncovered some corner cases in the polyhedra sorting algorithm that however do not seem to occur in practice. The formalization uses the Verified Polyhedron Library [23] to implement polyhedral operations. The proof is only focused on the code generation problem and assumes that

the schedule is correct and respect dependencies. The code generator could be plugged into a verified compiler such as the CompCert compiler of Leroy [67] to obtain a polyhedral code generator down to assembly language. The more recent improvements such as the components analysis and fine-grained control over the polyhedron splitting decisions mentioned in the previous paragraph are not part of the verified generator.

All the approaches mentioned here use structured loop nests to iterate over the points of the polyhedron. Boulet and Feautrier [22] proposed an alternative approach to code generation in the polyhedral model by generating unstructured programs using goto statements instead. The technique is based on the computation of a piece-wise affine function computing the "next" instance to execute after the current one according to the schedule, and using an appropriate goto statement after setting the loop iterators to their "next" value. A technique based on Boolean guards is also proposed to avoid recomputing derived variables when their dependencies have not changed.

2.1.6 Access Relations

The polyhedral model is not limited to the representation of programs as sets of instances. It is also possible to represent relations between statements and the memory locations they write to or read from. For instance, consider the statement S defined as $c[i,j] = c[i,j] + a[i,k] * b[k,j]$. An instance $S(i,j,k)$ of statement S reads from $a[i,k]$, $b[k,j]$ and $c[i,j]$, and writes to $c[i,j]$. This information is used to compute the dependence analysis described in the next section, which gives criteria to ensure that a schedule is valid.

The representation not only of statements across time dimensions but also of memory locations across space dimensions enables additional optimizations related to memory layout to be expressed in the polyhedral model. For instance, consider the following program:

```
allocate b[] in
for i = 0 to N - 1 do
  b[] = a[i] * a[i]
  c[i] = b[] * b[]
```

The same location b[] is used within each iteration of the loop to hold the intermediate value a[i], hence this loop cannot be parallelized, even though there are no semantic dependencies between distinct iterations. In order to be able to parallelize the loop, a data layout transformation called *array expansion* must be applied to the array b[] to introduce an extra dimension and store the intermediate values in separate memory locations:

```
allocate b[N] in
for i = 0 to N - 1 do
  b[i] = a[i] * a[i]
  c[i] = b[i] * b[i]
```

Much like schedules can be expressed as binary relations mapping a statement in the original iteration space to a position in the new schedule space, data layout transformations are expressed in the polyhedral model using binary relations mapping memory locations in an old data space to memory locations in a new data space where arrays can have different dimensionalities and layouts. Data layout transformations can also be expressed using more precise relations that depend on more that just the original memory location but also on the occurrence of the array access within the statement (e.g. in the statement `b[] = a[i] * a[i]`, it is possible to apply a transformation only to the first occurrence of `a[i]` but not the second), on the value of the iteration dimensions for the current instance, and — when applied coincidentally with a schedule — on the value of the schedule dimensions.

For instance, assuming the statement `b[] = a[i] * a[i]` is called S_1, the array expansion performed above can be expressed using the following ternary relation:

$$\{S_1(i) \to b[] \to b[i] \mid 0 \leqslant i < N\}$$

2.1.7 Dependence analysis

The general problem of equivalence between programs is undecidable; however, the question of a schedule' *validty* (i.e. preserves program semantics) must be answered. The groundwork for what would become the standard polyhedral approach to answer this question can be found in Lamport's seminal work. The general idea is to examine the dependencies between statement instances to build a sufficient condition for equivalence that can be expressed as an integer linear problem. Roughly speaking, we say that there is a dependency between two instances if they access the same memory location, at least one of the accesses being a write. Two schedules of the same instance set that assign the same (strict) order to any pair of two dependent instances represent semantically equivalent programs. This idea of looking at dependencies is not

unique to the polyhedral model, and has been extensively considered by the parallel programming community. What is more unique to the polyhedral model is the way in which these dependencies are computed.

The original approaches to dependence analysis classified dependencies coarsely by treating full arrays as variables: instead of considering memory locations, there is a dependency between two instances in that coarse model if they access the same array (at possibly different indices) with at least one of the access being a write. On the other hand, the approach proposed by Lamport and formalized in Pugh [79] and Lu [71] use an *exact* dependence analysis: by inspecting the array accesses performed by the instructions, parametric representations of the array locations accessed by each instruction can be constructed. From these, a more precise over-approximation of the dependence relation can be constructed using (once again) integer linear programming techniques.

Computing the data dependencies between statements require inspecting the representation of the statement I in order to collect the memory locations it accesses. In order to make this tractable, this is usually done under the assumption that arrays do not alias, and that array subscripts are always within bounds, two properties that have to be checked separately. A strength often touted by proponents of the polyhedral model is that, unlike other approaches to automatic parallelization, the data dependencies between statement instances can be computed exactly. The fact that the treatment of data dependencies by the polyhedral model is by necessity correct for any over-approximation of the dependencies is sometimes overlooked: it means that approximate dependencies for any statement can be computed, although more precise analyses and more transformations can be performed when only affine array subscripts are in use.

2.2 Systems of Affine Recurrence Equations

While the polyhedral model was developed as a tool for the optimization of imperative programs with arrays and loops, the theoretical foundations for the semantics of programs that can be exactly represented in the model can be found in the theory of Systems of Affine Recurrence Equations (SAREs).

The foundations for the study of SAREs are described by Karp, Miller, and Winograd [60], predating Lamport's parallelization scheme by a few years. Karp, Miller, and Winograd [60] are interested in the efficient organization of computation for the recurrence equations that arise when applying finite-difference approximations to systems of partial differential equations. These recurrence equations are defined over a multidimensional grid space that can be represented as a subset of $\mathbb{Z}^n$, and exhibit a uniformity property: the dependencies can be expressed as constant vectors that invariant to translation on the grid.

SAREs are a generalization of the Systems of Uniform Recurrence Equations (SUREs) introduced by Karp, Miller, and Winograd [60]. A SURE can be understood by considering the case of a single equation for a function a_1 defined over a domain $\mathcal{D}_1 \subseteq \mathbb{Z}^n$:

$$a_1(p) = f_1(a_1(p - w_1), \ldots, a_1(p - w_k))$$

where $p \in \mathcal{D}_1$ and $w_1, \ldots, w_k \in \mathbb{Z}^n$ are constant n-dimensional vectors with integral coordinates. The SUREs studied by Karp, Miller, and Winograd [60] are systems of such equations over a set of functions $a_1, \ldots, a_m$ each having one defining equation over domains $\mathcal{D}_1, \ldots, \mathcal{D}_m$, subsets of a shared space $\mathbb{Z}^n$. The equations can be mutually recursive, and are represented by a dependence graph, abstracting away the right-hand sides except for their dependencies. There is a direct correspondence between the functions $a_1, \ldots, a_m$ of a SURE and the instructions in a polyhedral program representation. Karp, Miller, and Winograd [60] are interested in the study of SUREs as they often occur when applying finite-difference approximations to systems of partial differential equations, and are hence motivated to find efficient solvers for SUREs. They introduce the notion of schedules on SUREs from which the polyhedral schedules are derived, although the schedules of Karp, Miller, and Winograd [60] are unidimensional. A schedule of particular theoretical interest (but of limited practical importance) is the *free schedule*: each computation is schedules to the first time at which all its dependencies are available. The authors show criteria for the existence of a schedule and for the amount of parallelism exposed by a schedule, laying the theoretical foundation for the techniques of Lamport [65] — and hence the whole polyhedral compilation field.

The equations in a SURE are *uniform* because the dependence vectors w_i are constants independent of the arguments of the function being computed. On

the other hand, arbitrary affine expressions can be used as dependence vectors in a SARE; furthermore, multiple defining equations with mutually disjoint domains and different right-hand sides can be associated with a single function. Hence, a SARE is a set of equations:

$$\forall x_1, \ldots, x_{n_A},\ \phi \implies A(x_1, \ldots, x_{n_A}) = e$$

where ϕ is a Boolean-valued affine formula of $x_1, \ldots, x_{n_A}$ (and of global constants), and e is an expression built from functions and tensor accesses. The domain of the equation above is $\mathcal{D} = \{(x_1, \ldots, x_{n_A}) \mid \phi\}$. If there are multiple defining equations for the same tensor A, their domains must be disjoint. The indices of all tensor accesses in e must also be affine expressions of the variables. Equivalently, a single equation for each tensor can be provided that performs a case analysis on the domain, and piece-wise affine expressions in the tensor indices can be allowed. The inputs of a SARE are the tensors that never appear on the left-hand side of an equation. There is no requirement that the domain of a SARE be total, or that equations be free of self-references such as $A(i) = A(i) + 1$. SAREs are typically given a semantic through a schedule: each point in the domain of each equation is assigned a timestamp later than all of its dependencies; the disjointness condition on the equations ensures that this gives a unique semantics to a given SARE. Some SAREs such as those with self-references do not have valid schedules, and hence do not have a semantics.

Polyhedral programs can be converted to SAREs by using dataflow analysis as introduced by Collard and Griebl [28]: using the Parametric Integer Programming algorithm of [36], it is possible to express the last statement instance that writes to a given memory location as a piece-wise affine expression. Tensors can then be introduced for each assignment statement, with as many dimensions as the statement has outer loop iterators, and the right-hand side of the assignment can be used as the definition of the tensor where array accesses are replaced with the corresponding tensor access obtained through the previous analysis — a process also known as *array expansion* [38]. Dataflow analysis and array expansion can be used to remove false dependencies in polyhedral programs.

SAREs are used as intermediate representations in polyhedral compilers, and form the foundation of the ALPHA equational language of Verge, Mauras, and Quinton [119]. The mathematical nature of SAREs are also a good candidate for expressing the specification of affine programs, and the verifier

presented in chapter 5 uses SAREs as a specification language to abstract away from the details of a specific implementation such as Halide or Tensor Comprehensions.

2.3 Functional Combinators and Rewrite Rules

Another line of research aiming to generate high-performance code for multi-dimensional workloads stems from the functional programming community and relies on rewrite rules for compiler optimizations. In languages such as Accelerate [72], Futhark [50] or Rise [48], a successor to Lift [102] programs are written using high-level functional combinators such as map and reduce. Optimization and implementation choices are expressed using rewrite rules introducing many specialized versions of the combinators that ultimately guide an imperative code generator. The rewrite rules approach is flexible and can be extended through the introduction of new combinators, new rewrite rules, or both, to support various application domains such as stencils [47] and hardware constructs such as specific vectorized instructions [101].

Compared to polyhedral compilation and especially to scheduling approaches such as that used by Halide, approaches based on functional combinators are in practice more limited in their ability to deliver the high-performance required by real-world applications. Rise has proposed to combine functional approaches and schedule-based approaches by introducing Elevate, a strategy language that can express complex combinations of rewrite rules programmatically and re-implement many of the scheduling primitives of TVM using this strategy language.

Approaches based on functional combinators typically use the same functional language to implement the original specification and the final implementation. The rewrite rules can be proved correct independently. The approach of Liu et al. [69] gives a formal specification to a functional language with map and reduce combinators, and the available rewrite rules of the system are theorems stating the correctness of the rule and proved in Coq. Using manually crafted rules, they can match the performance obtained by a well-optimized Halide schedule.

2.4 The Halide model

The polyhedral model makes the representation and optimization of many array programs possible; however, in part due to its wide expressiveness, optimizers that work by trying to find polyhedral schedules often fail to compete with programs hand-optimized by experts, a situation for which the user has no resort but to optimize the program by hand. This observation has led to the design and implementation of *scheduling languages* that decouple the writing of an algorithm with the application of a schedule using user-facing scheduling primitives. Multiple such attempts have been made using the tools and abstractions provided by the polyhedral model, but projects such as AlphaZ, CHiLL or URUK have largely been unsuccessful at reaching a non-expert audience. One of the reasons for this might be the very reliance of these tools on the polyhedral model: the polyhedral model "leaks" in various ways into the user interface. This may not deter, and even attract, a researcher familiar with the model, but it makes these tools harder than necessary to use and understand for the uninitiated. In comparison, the Halide compiler and language, developed by Ragan-Kelly et al. in the context of image processing pipelines, provides the combination of a familiar array-based syntax and a powerful but succinct scheduling language. Originally designed for computational photography and computer vision algorithms, Halide's scheduling language provides a mental model that is more familiar and easier to use for members of related communities. The success of Halide — used both in an industrial setting and the focus of ongoing research — has inspired the development of TVM, originally a fork of Halide, with a focus on deep learning operators, and has inspired the polyhedral community to build Tiramisu, a user-directed polyhedral compiler with a focus on distributed computing and whose scheduling language is heavily inspired by Halide's.

2.4.1 Algorithms

Halide represents tensors of arbitrary dimensionality as pure functions defined over an infinite integer domain. The expressions defining the tensors can refer to other tensors, which must have been previously defined: the graph of tensor definitions must be acyclic. Halide algorithms are defined in a DSL embedded in C++ or Python, where the user defines tensors (called `Funcs`) as symbolic

multidimensional functions over an infinite domain. Definitions are written using an overloaded = operator. The traditional Halide example is that of a 3×3 unnormalized box filter, reproduced below from "Halide: decoupling algorithms from schedules for high-performance image processing" [83].

```
Func bh, bv; Var x, y;
ImageParam in(UInt(8), 2);

bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y + 1))/3;
```

In this algorithm, we take as input a grayscale image represented by the $N \times M$ `uint8` matrix `in`, and computes a horizontal blur in `bh` followed by a vertical blur in `bv`. The first definition of any `Func` is its unique *pure definition*, and it is treated specially to ensure that the `Func` is defined over all the points in its multidimensional domain. Pure definitions make use of *pure variables*, which have type `Var` in the DSL, and are implicitly quantified over *all* the integers (including negative ones). Distinct pure variables must appear as arguments to the tensor in the left-hand side of a pure definition, and are bound on the right-hand side. Only pure variables bound on the left-hand side can appear in the right-hand side.

A Halide specification is called an *algorithm*, and an algorithm containing only pure definitions bears similarity to a system of affine recurrence equation used in the polyhedral model, with a few key differences allowing Halide to thrive on different trade-offs. First, Halide algorithms are laid out in a textual order: pure definitions can only refer to `Func`s that have already been defined, ensuring an acyclic dependence graph between the pure definitions that make them more restrictive than the arbitrary mutual recursion of SAREs. Second, equations in a SARE have as domain a sub-set of $\mathbb{Z}^n$ defined using affine inequalities, while pure definitions in Halide are always defined over the full $\mathbb{Z}^n$ space. In a SARE each access must be checked to be in bounds and falls back to a pre-determined value otherwise; on the other hand, arbitrary indices (including the result of indexing another `Func`) can be used on the right-hand side of a pure Halide definition. Bounds must be approximated to ensure infinite arrays do not have to be materialized, as described below.

Using only pure definitions, Halide algorithms are fairly limited in their expressiveness and are not able to express recurrences of parametric length.

Halide's solution is to introduce some imperative mutability to the algorithms through the concept of *update definitions*, that effectively enables the user to implement algorithms using sequential loops. An example of an algorithm using update definition is the following specification for the general matrix multiplication D = αAB + βC, where K is the size of the reduced dimension

```
Var i, j;
Func D;
RDom k(0, K);
D(i, j) = beta * C(i, j);
D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j);
```

The tensor `D` in this example has two different definitions. The pure definition `D(i, j) = beta * C(i, j)` initializes the tensor, and the update definition `D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j)` applies on top of that initialization for each value of `k`, iteratively. In addition to pure variables, update definitions can make use of *recurrence variables*. Using, recurrence variables is it possible to define tensors iteratively in multiple steps, by having the value at each step depend on the value at the previous step. The value of `D(i, j)` used in the update definition refers to the previous value of the tensor `D(i, j)` in an imperative way, according to the iteration order of the recurrence variables appearing in either the left- or right- hand side of the definition. Recurrence variables are declared using the `RDom` constructor, representing a multidimensional loop as a tuple of recurrence variables iterating over a bounded rectangular domain. In the example, the update definition uses the recurrence variable `k`, and is semantically equivalent to the imperative loop:

Halide calls recurrence variables "reduction variables", but their use is not limited to reductions, hence I prefer the term of "recurrence variables". See chapter 7 for an in-depth discussion of reductions.

```
for k = 0 to K - 1 do
  D(i, j) = D(i, j) + alpha * A(i, k) * B(k, j)
```

As in the example, update definitions can refer to the tensor currently being defined, in an imperative way. They are not equations: pure variables appearing in update definitions should be thought of informally as parallel loops over an infinite domain. As such, it is not possible to give a semantics to some update definitions such as `D(i, j) = D(j, i) + 1`. In order to rule out such impossible definitions, Halide ensures that within a single update definition, the same tensor indices can only be accessed *at the same value of the pure variables*. This is enforced by ensuring that each pure variable appearing in the definition must appear as an argument in the same position in all accesses to that tensor

in the definition. The pure variable can also appear in other positions, for instance, it is valid to write `D(i, i + 1) = D(i, i - 1) + D(i, 0)` because `i` is the only pure variable, and it is the first argument in all accesses to tensor `D`.

Definitions for distinct `Func`s cannot be interleaved: a definition `Func` only sees the values of another `Func` after all the update definitions for that other `Func` have been processed. To define mutually recursive tensors, users must use functions whose value is a tuple, whose components must be computed together. Finally, update definitions can include an arbitrary boolean expression as a "filter" restricting the points where the update is performed, and which can refer to the recurrence variables and any pure variables used in the definition. If all the filters and all the index expressions appearing in a Halide algorithm are piece-wise quasi-affine combinations of pure and recurrence variables, we say that the algorithm is an *affine algorithm*. An affine algorithm is equivalent to an affine program that can be represented exactly in the polyhedral model, and can be readily converted to a SARE, as explained in subsection 2.4.4.

2.4.2 Schedules

Halide algorithms specify what values should be computed, but does not say much about how to compute those values. More precisely, the algorithm does not specify the order in which computations should occur (except for semantic dependencies), or where intermediate results should be stored, if applicable. Like in the polyhedral model, this information is encoded separately in a *schedule* — but Halide schedules bear little resemblance to polyhedral schedules. In fact, I would argue that Halide's scheduling primitives, tailored to the needs of computational photography and computer vision algorithms, reveal a fundamentally different way of thinking about the organization of computation compared to traditional polyhedral tools, a way of thinking materialized in the `compute_at` and `store_at` scheduling directives.

Halide schedules are written in the same DSL as the algorithms by applying *scheduling directives* to the `Func`s. Schedules are often written manually by experts, but Halide also provides automated schedulers that search for efficient schedules using machine learning techniques [3, 5]. By default, scheduling

directives apply to the pure definition of the `Func`; they can also be applied to update definitions by using the `.update()` accessor which return a scheduling object for the corresponding update definition. Describing the complete interface of Halide's scheduling API is out of scope for this manuscript, and I will instead focus on a few simple examples to give a taste of its design philosophy.

When scheduling a single function, Halide's scheduling directives can express a restricted set of loop transformations using the `split` and `reorder` primitives. For instance, in the outer product $C(i,j) = A(i) \times B(j)$, the dimension `i` can be strip-mined and sunk using the schedule

```
C.split(i, i0, i1, 4).reorder(i1, j, i0)
```

leading to the following low-level code:

```
for i0 = 0 to (N + 3) / 4 - 1 do
  for j = 0 to M - 1 do
    for i1 = 0 to 3 do
      let i = min(i0 * 4, N - 4) + i1 in
      c[i, j] := a[i] * b[j]
```

These primitives can express arbitrary tiling of the perfect loop nest computing the function's definition, but not more complex transformations such as skewing that can be represented using the affine transformations of the polyhedral model. On the other hand, the `split` primitive can express non-affine *parametric* tiling (i.e. tiling by a factor that is not a statically known constant). The differences in expressiveness between Halide schedules and polyhedral schedules are discussed in more depth in Baghdadi et al. [7].

The `split` directive indicates that the loop over `i` should be separated into an outer loop `i0` and an inner loop `i1`. The loop nesting order, from innermost to outermost, is specified by the `reorder` directive. Additional scheduling directives such as `.parallel` or `.vectorize` can be used to mark loops as parallel or to use vectorized instructions. Recurrence variables can typically not be made parallel unless they satisfy specific requirements ensuring that parallelization is semantically sound. It should be noted that in this example, when `N` is not a multiple of four, the last tile of `i` is "shifted inwards" and recomputes some elements that were already computed in the previous tile.

This is only one of the modes provided by Halide's `split` directive: by ensuring that the loop over `i1` has a statically known constant size, it enables its implementation using vectorized instructions (expressed in Halide with the `.vector(i1)` directive). Later optimizations performed by the Halide compiler (not specified by the schedule) may split the last iteration of the `i0` loop and simplify away the `min` computation except for that last iteration.

The bounds of the generated loops are computed using a process called *bounds inference* based on user-provided bounds for the area of C to be computed, here $[\![0, N[\![\times [\![0, M[\![$. Bounds inference uses interval analysis to compute an over-approximation of the necessary range of the loops for the requested domain of the output tensor to be computed. By using interval analysis, Halide is not restricted to affine expressions in loop bounds and tile sizes; on the other hand, Halide's bounds inference algorithm can sometimes be too conservative and infer larger bounds than necessary, resulting in wasted computation. If the bounds inference fails (e.g. the computed over-approximation would have an infinite size in some dimension), the programmer is met with an error and has to explicitly clamp the indices to ensure they are within finite bounds, an information that can be picked up by bounds inference. It is the programmer's responsibility to ensure that this does not unexpectedly change the algorithm's semantics.

The strength of Halide's scheduling language appears when scheduling a complete pipeline composed of multiple dependent tensors. In the polyhedral model, each tensor definition is interpreted as a statement, and polyhedral schedules map each statement instance (i.e. each point in a tensor's definition domain, in the case of pure definitions) to a single point in the final iteration space, building schedules that are work-efficient by construction. In the modern world where the cost of accessing memory is orders of magnitude larger than the cost of performing a computation, work-efficiency is not necessarily a desirable property: when possible, it is often better to recompute derived values from data that is available locally rather than to communicate those through a slower memory shared amongst many processing units, a technique known as rematerialization. Halide recognizes the need for rematerialization and Halide's schedules are designed for a proper exploitation of the trade-off between work-efficiency and locality.

Going back to the box filter shown earlier, "Halide: decoupling algorithms from schedules for high-performance image processing" [83] shows the result of two

schedules illustrating this tension. Locality can be maximized by computing the required values of bh at each iteration of the loop over bv, essentially inlining bh's computation. Halide's default schedule is aggressively inlining intermediate tensors, and results in the following low-level code:

```
for y:
  for x:
    bv(x, y) = // bh(x,y-1) + bh(x,y) + bh(x,y+1) =
      (in(x-1,y-1)+in(x,y-1)+in(x+1,y-1))
     +(in(x-1,y  )+in(x,y  )+in(x+1,y  ))
     +(in(x-1,y+1)+in(x,y+1)+in(x+1,y+1))
```

The values of $bh(x, y-1)$, $bh(x, y)$ and $bh(x, y+1)$ are computed immediately before being used, ensuring that they fit in registers or, at worst, a fast cache, minimizing memory transfers. The price to pay for that locality is that each point of bh is computed three times: there is an overlap between the points of bh computed at each iteration of the loop.

At the other end of the scale, all values of bh could be pre-computed once, in a work-efficient way. Since Halide prevents mutually recursive tensor definitions, each tensor in a pipeline is scheduled independently, and this work-efficient schedule can be expressed using `bh.compute_root()`, forcing the computation of bh to be performed in a separate loop nest from that of bv: `bh.compute_root()` in Halide.

```
for y:
  for x:
    bh(x, y) = in(x-1, y)+in(x,y)+in(x+1,y)

for y:
  for x:
    bv(x, y) = bh(x,y-1)+bh(x,y)+bh(x,y+1)
```

Halide's bounds inference algorithm is used to automatically infer the bounds on the loops over bh that are needed to compute the requested rectangle of bv. No redundant work if performed in this work-efficient schedule, but locality is completely lost: by first computing all of bh before computing bv, we essentially ensure that it will not fit in cache (unless the dimensions are very small) and that additional slow memory transfers from the memory will be needed.

To explore the trade-offs between the local default schedule and the work-efficient schedule of `compute_root`, Halide provides the `compute_at` primitive. This primitive specifies where the computation of a function should happen within the loop nest of its consumer. For instance, assume we have tiled the computation of `bv` using the schedule directive

```
bv.split(x, tx, px, T)
  .split(y, ty, py, T)
  .reorder(px, py, tx, ty)
```

Ignoring the alignment issues when the dimension sizes are not divisible by the tile size, this results in a loop nest that looks like the following:

```
for ty:
  for tx:
    for py:
      for px:
        let x = tx * T + px in
  let y = ty * T + py in
  bv(x, y) = ...
```

Then, using the scheduling directive `bh.compute_at(bv, tx)` indicates that the value of `bh` should be computed independently for each tile: the bounds inference algorithm is used on the set of `bv` locations computed within a specific iteration of loop `tx`, and Halide inserts at the beginning of that loop code that computes the inferred over-approximation of `bh` needed for the computation of `bv`, resulting in the following loop nest:

```
for ty:
  for tx:
    for y = ty * T to ty * T + T - 1 do
      for x = tx * T to tx * T + T - 1 do
  bh(x, y) = ...

    for py:
      for px:
        x = tx * tile_size + px
  y = ty * tile_size + py
  bv(x, y) = ...
```

The sizes of the loops for `y` and `x` are inferred automatically by Halide's bounds

inference pass to ensure that all points required by the computation of `bv` is computed exactly once between the tile. By adjusting the tile size, we get a trade-off between work-efficiency and locality: only the points on the border of the tile are re-computed across tiles, and the size of the portion of `bh` which must be kept is (roughly) that of the tile. Although not crucial to the discussion, it should be noted that in practice Halide re-aligns the loops and array bounds so that they start at zero.

In some cases, `compute_at` forces the materialization of intermediate buffers that are too large compared to their use. In this case, the `store_at` schedule directive can be used. Like `compute_at`, `store_at` indicates a loop level. Instead of allocating the temporary buffer for the intermediate computation at the level of the `compute_at`, it is introduced at the level of the `store_at`. Halide ensures that there are only sequential loops between the `store_at` and `compute_at` level, and does not recompute values that it can prove have already been computed by previous iterations using interval analysis. In addition, Halide provides "storage folding" facilities to store intermediate values in a rolling buffer when applicable.

Halide schedules can express trade-offs between work efficiency and data locality concisely. The reliance on rectangular regions and interval analysis gives a different point in the design space compared to polyhedral schedules. The design of the `compute_at` and `store_at` scheduling primitives provide a simple vision of tiling by computing the dependencies of a tile independently of other tiles, leading naturally to overlapping tiles when appropriate. On the other hand, the global view of the polyhedral model is biased towards work-efficient schedules that are not necessarily optimal on today's hardware. Research exists to incorporate non work-efficient schedules in the polyhedral model, but it is typically separate from the general framework and done using specific approaches and ad-hoc schedulers. An example of this is the recent work of Zhao and Cohen [121] that uses the expansion node of schedule trees to represent overlapped tiling by using a single point in the original domain to represent a full tile with modified dependencies.

2.4.3 Semantics of Halide Specifications

Halide has been developed as a practical system, whose semantics is mostly described in prose and represents the behavior of the existing implementation. Reinking, Bernstein, and Ragan-Kelley [89] proposed a semantics of Halide algorithms understood as sequential programs, in order to formalize the effect of a subset of Halide's scheduling directives and prove the correctness of the code generation process. The authors of that paper also formalize Halide's bounds inference algorithm as solving a program synthesis problem.

I am more interested in giving an equational semantics to Halide algorithms, in order to be able to name and keep track of the intermediate states of the evaluation process. Hence, I propose the following formalization of Halide algorithms that are not necessarily affine and may contain arbitrary expressions. In the next subsection, I propose a reduction of *affine* Halide algorithms (i.e. algorithms where all tensor accesses and all filters are affine in the pure and recurrence variables) to SAREs that follow the same general structure as the semantics in this section.

A Halide algorithm over a set of tensors $\mathcal{S}$ can be represented as a tuple $\langle I, P, U, < \rangle$ where:

- I is the set of input tensors, which have no associated definition;
- P maps each tensor $A \in \mathcal{S} \backslash I$ to its *pure definition* P_A;
- U maps each tensor $A \in \mathcal{S} \backslash I$ to a (possibly empty) finite sequence $U_A^1, \ldots, U_A^{n_A}$ of *update definitions* for A;
- $<$ defines a total order over the non-input tensors in $\mathcal{S} \backslash I$, representing the textual order of the first pure definition to the tensor

The pure definition P_A for tensor $A \in \mathcal{S}$ is an equation written as follows:

$$\forall x_1, \ldots, x_{n_A},\ A(x_1, \ldots, x_{n_A}) = e$$

The right-hand side of a pure definition can only refer to input tensors and the defined tensors A' defined before A (i.e. with $A' < A$). In particular, the definition of A cannot refer to A itself.

An update definition U^i_A for tensor $A \in \mathcal{S}$ is also an equation, which involve some pure variables $x_1, \ldots, x_n$ and an arbitrary number of recurrence variables $y_1, \ldots, y_r$:

$$\forall x_1, \ldots, x_n.\ \text{for}\ y_1 : R_1, \ldots, y_r : R_r.\ \phi \implies A(e_1, \ldots, e_{n_A}) = e$$

Each of the y_i has a recurrence domain R_i, a parametrically bounded interval of $\mathbb{Z}$. The bounds of R_i can only depend on the parameters, not the pure variables nor other recurrence variables. The boolean expression ϕ is the filter: an additional condition on the points where the update is performed.

The well-formedness condition can be formalized as follows. We require the existence of a function π from the pure variables $\{1, \ldots, n\}$ to the argument positions $\{1, \ldots, n_A\}$ such that, for all $1 \leqslant j \leqslant n$:

1. $e_{\pi(j)} = x_j$, and
2. For each $A(e'_1, \ldots, e'_{n_A})$ appearing as a sub-term of either the right-hand side e or the filter ϕ, we have $e'_{\pi(j)} = x_j$.

π maps each pure variable to a shared position in all accesses of the tensor being defined in the update definition. In general, there might be fewer pure variables than argument positions ($n \leqslant n_A$); the other arguments can be any expression of the pure and reduction variables. In any case, this ensures that there is no circular dependencies between iterations of an update definition: the expression defining a location at some value of the pure variables can not read from locations defined for other values of the pure variables, since the pure variables are shared indices of all the accesses.

I now propose a semantics for Halide algorithms based on model theory. Let us define a *stage index* ψ as one of:

$$\begin{aligned}\psi ::= {} & \mathcal{P} \mid \mathcal{U}_n \mid \mathcal{F} \\ & \mid \mathcal{U}_n(n_1, \ldots, n_m)\end{aligned}$$

where $n > 0$ is a non-negative integer and $n_1, \ldots, n_m \in \mathbb{Z}$ are arbitrary integers.

Stage indices are used to distinguish between each of the "versions" of a tensor that occurs when evaluating the Halide algorithm. They encode the updates that have been applied to a tensor. A stage index can be:

- The *pure* stage index $\mathcal{P}$ refers to the value of the tensor's pure definition.
- The *final* stage index $\mathcal{F}$ refers to the value of the tensor after all updates have been performed. This is the value of the tensor that is used by dependent tensor definitions.
- An *update* stage index $\mathcal{U}_n$ refers to the value of the tensor after evaluating the n-th update stage.
- A *partial update* stage index $\mathcal{U}_n(n_1, \ldots, n_m)$ refers to the value of the tensor during the evaluation of the n-th update stage. The values $n_1, \ldots, n_m$ are the values of the reduction variables used in the n-th update stage; updates for lexicographically smaller values of the reduction variables are already taken into consideration.

There is a natural order, corresponding to the evaluation order, on stage indices.

In practice, Halide orders the recurrence variables in reverse lexicographic order (inside-out instead of outside-in), but it makes no practical difference here.

$$\mathcal{P} \prec \cdots \prec \mathcal{U}_{n-1} \prec \mathcal{U}_n(n_1, \ldots, n_m) \prec \mathcal{U}_n \prec \cdots \prec \mathcal{F}$$

In addition, the $\mathcal{U}_n(n_1, \ldots, n_m)$ for the same stage n are ordered lexicographically on the recurrence variables $n_1, \ldots, n_m$.

A model M of the algorithm is a function from pairs $\langle A(n_1, \ldots, n_n), \psi \rangle$ of a tensor indexing and a stage index. The evaluation of an expression e in an environment $\mathcal{E}$ and model M, denoted $[\![e]\!]_M$, is defined by mapping each function to their usual interpretation and maps tensor indices to their final stage:

$$[\![A(e_1, \ldots, e_n)]\!]_{\mathcal{E};M} = M(\langle A([\![e_1]\!]_{\mathcal{E};M}, \ldots, [\![e_n]\!]_{\mathcal{E};M}), \mathcal{F} \rangle)$$

We also define the intermediate evaluation of expression e in environment $\mathcal{E}$ and model M at stage ψ for tensor A, denoted $[\![e]\!]^{A \mapsto \psi}_{\mathcal{E};M}$, by mapping each function to its usual interpretation and maps tensor indices to the current stage as follows:

$$[\![A(e_1, \ldots, e_n)]\!]^{B \mapsto \psi}_{\mathcal{E};M} = \begin{cases} M(\langle A([\![e_1]\!]^{B \mapsto \psi}_{\mathcal{E};M}, \ldots, [\![e_n]\!]^{B \mapsto \psi}_{\mathcal{E};M}), \psi \rangle) & \text{if } A = B \\ M(\langle A([\![e_1]\!]^{B \mapsto \psi}_{\mathcal{E};M}, \ldots, [\![e_n]\!]^{B \mapsto \psi}_{\mathcal{E};M}), \mathcal{F} \rangle) & \text{otherwise} \end{cases}$$

We then say that M is a model for a Halide algorithm $\mathcal{A}$ if the following holds:

- For any pure definition in $\mathcal{A}$ of the form

  $$\forall x_1, \ldots, x_n,\ A(x_1, \ldots, x_n) = e$$

  and for all integers $n_1, \ldots, n_n$, the following equality holds:

  $$M(\langle A(n_1, \ldots, n_n), \mathcal{P}\rangle) = [\![e]\!]^{A \mapsto \mathcal{P}}_{\mathcal{E}, x_1 \mapsto n_1, \ldots, x_n \mapsto n_n; M}$$

  This states that the evaluation of tensor A at its pure index satisfies Halide's pure definition for A. Recursive tensor accesses are not allowed in pure definitions, hence the choice to map A to its pure index when evaluating e is arbitrary.

- For any update definition of tensor A at position s in $\mathcal{A}$:

  $$\forall x_1, \ldots, x_n.\ \text{for } y_1 : R_1, \ldots, y_r : R_r.\ \phi \implies A(e_1, \ldots, e_{n_A}) = e$$

  and for all integers $n_1, \ldots, n_n$ and $m_1, \ldots, m_r$ such that $m_i \in [\![R_i]\!]_{\mathcal{E}}$ for any $1 \leqslant i \leqslant r$, let ψ be the current stage index $\mathcal{U}_s(m_1, \ldots, m_r)$ and let ψ' be the previous stage index for A. ψ' is either $\mathcal{U}_s(m'_1, \ldots, m'_r)$ where $m'_i \in [\![R_i]\!]_{\mathcal{E}}$ for $1 \leqslant i \leqslant r$ and $(m'_1, \ldots, m'_r)$ is the lexicographic predecessor of $(m_1, \ldots, m_r)$; or $\mathcal{U}_{s-1}$ (with the convention that $\mathcal{U}_0 = \mathcal{P}$) if no such lexicographic predecessor exists. Moreover, let $\mathcal{E}'$ be $\mathcal{E}, x_1 \mapsto n_1, \ldots, x_n \mapsto n_n, y_1 \mapsto m_1, \ldots, y_r \mapsto m_r$. Then, if $[\![\phi]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}$ holds, the following equality holds:

  $$M(\langle A([\![e_1]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}, \ldots, [\![e_{n_A}]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}), \psi\rangle) = [\![e]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}$$

  This states that when the condition ϕ holds, the value of A at the new stage index ψ is equal to the evaluation of e at the previous stage index ψ'.

  Otherwise, the following equality holds:

  $$M(\langle A([\![e_1]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}, \ldots, [\![e_{n_A}]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}), \psi\rangle) = M(\langle A([\![e_1]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}, \ldots, [\![e_{n_A}]\!]^{A \mapsto \psi'}_{\mathcal{E}'; M}), \psi'\rangle)$$

This states that when ϕ does not hold, the value of A at the new stage index ψ is unchanged and equal to its value at the previous stage index ψ'.

All expressions are evaluated in the environment $\mathcal{E}'$ that defines the current values of the pure and recurrence variables, but at the previous stage index ψ' instead of the current stage index ψ. This ensures the absence of dependence cycles.

- The value at stage index $\mathcal{U}_s$ is equal to the value at the last stage index $\mathcal{U}_s(n_1, \ldots, n_n)$ that is within the recurrence bounds, or to the evaluation at stage index $\mathcal{U}_{s-1}$ (still with the convention $\mathcal{U}_0 = \mathcal{P}$) if there is no such stage index.

- The value at stage $\mathcal{F}$ is equal to the value at $\mathcal{U}_s$ where s is the last update associated with the tensor, or at $\mathcal{P}$ if the tensor has no update.

This construction is deterministic in the sense that for any Halide algorithm $\mathcal{A}$ and initial environment $\mathcal{E}$ assigning a well-typed value to each of the parameters, for any model M_0 for the input tensor of $\mathcal{A}$, there is a unique model M of $\mathcal{A}$ that is an extension of M_0.

The proof is obtained by defining a schedule assigning a unique computation point to each pair of a tensor access and a stage index so that all its dependencies are computed beforehand. The schedule is given as a lexicographically ordered tuple. Since Halide requires a valid dependency graph between *tensors*, the first component of the schedule is the tensor name, ordered according to the Halide dependency order. Hence, we only need to build a schedule for the equations relating to a given tensor, assuming that the tensors it depends on are fully defined.

We take the stage index as second component of the schedule. Indeed, the construction above ensures by construction that any equation for tensor A at stage index ψ only depends on the values of A at a lexicographically smaller stage index $\psi' \prec \psi$. The only exception is the pure definition of A at stage $\mathcal{P}$, however, Halide forbids recursive uses of tensor A in its pure definition, ensuring that A at $\mathcal{P}$ has no dependence on itself.

Since the bounds on recurrence variables can only depend on parameters that

are defined by the initial environment $\mathcal{E}$, for each initial environment $\mathcal{E}$ the set of valid stage indices is finite and the construction is well-founded.

Finally, we must check that we never compute two different values for the same tensor access and stage index. This could only happen in an update definition, and since the stage index exactly defines the value of the pure variables, it would require two distinct values of the pure variables to map to the same tensor access. Since Halide requires that all pure variables used in an assignment must appear at least once as an argument of the tensor being defined on the left-hand side, this is not possible.

Note that this construction corresponds to an inner parallel construction: for each stage index (that can be understood as a sequential iteration), we compute the value of all the affected tensor accesses in parallel. We did not use the second part of Halide's update restriction, namely that all accesses to the tensor in the right-hand side must share its pure variable indices with the left-hand side definition. This second part is not necessary here because our semantics only reads from the value of the tensor at the previous stage index, preventing possible conflicts. This would not be practical in an implementation, as it would require making a copy of the whole tensor at each step. By forcing the pure variables to appear at the same position in all tensor accesses, Halides ensures that the updates can be performed independently for each value of the pure variables, making a practical implementation possible.

The semantics that is given here can be understood as defining a set of equations between the values in the model. If the original Halide semantics was affine (i.e. all array accesses and filter definitions are affine in the pure and recurrence variables), then this set of equations form a SARE.

Example 1. Let us consider a simple matrix multiplication, written in Halide as follows:

```
Var i, j;
RDom k(0, K);
C(i, j) = 0;
C(i, j) = C(i, j) + A(i, k) * B(k, j)
```

A model $\mathcal{M}$ is a model for this algorithm in environment $\mathcal{E}$ if the following constraints hold, for all $i, j \in \mathbb{Z}$:

For the pure definition of C, we have

$$M(\langle C(i,j)\rangle, \mathcal{P}\rangle) = 0 \tag{2.1}$$

For the partial updates at stage 1, when $\mathcal{E}(K) > 0$, the previous stage index is $\mathcal{P}$ when $k = 0$ and $\mathcal{U}_1(k-1)$ otherwise, hence we have

$$M(\langle C(i,j)\rangle, \mathcal{U}_1(0)\rangle) = M(\langle C(i,j), \mathcal{P}\rangle) + M(\langle A(i,k), \mathcal{F}\rangle) \times M(\langle B(k,j), \mathcal{F}\rangle) \tag{2.2}$$

and for all $0 < k < K$

$$M(\langle C(i,j)\rangle, \mathcal{U}_1(k)\rangle) = M(\langle C(i,j), \mathcal{U}_1(k-1)\rangle) + M(\langle A(i,k), \mathcal{F}\rangle) \times M(\langle B(k,j), \mathcal{F}\rangle) \tag{2.3}$$

For the update at stage 1, when $\mathcal{E}(K) > 0$ the last partial update is $\mathcal{U}_1(K-1)$, hence we have

$$M(\langle C(i,j)\rangle, \mathcal{U}_1\rangle) = M(\langle C(i,j)\rangle, \mathcal{U}_1(K-1)\rangle) \tag{2.4}$$

and otherwise there are no partial updates and we have

$$M(\langle C(i,j)\rangle, \mathcal{U}_1\rangle) = M(\langle C(i,j)\rangle, \mathcal{P}\rangle) \tag{2.5}$$

Finally, $\mathcal{U}_1$ is the stae of the last update, and we have

$$M(\langle C(i,j)\rangle, \mathcal{F}\rangle) = M(\langle C(i,j)\rangle, \mathcal{U}_1) \tag{2.6}$$

The whole process is not without reminding of the array expansion procedure of Feautrier [38], although our construction is more verbose because we do not assume that tensor indices are affine.

2.4.4 Reduction from affine Halide algorithms to SAREs

The model theoretic semantics of Halide algorithms described in the previous section can be adapted to define a reduction from *affine* Halide algorithms to

SAREs. For each tensor A in the Halide algorithm, we introduce a tensor A^S to represent A in the SARE, as well as intermediate tensors A_0 to represent the pure definition and $A_1, \ldots, A_n$ to represent the update definitions of A. The tensor A_0 is directly defined using the pure definition of A, where each other tensor B is replaced with its SARE equivalent B^S. The tensor A_i representing an update definition U^i_A has r extra indices, where r is the number of recurrence variables in U^i_A: conceptually, the tensor is replicated for each value of the recurrence variables. Note that due to the update restriction, knowing the value of the array indices is enough to know the value of all pure variables appearing in the right-hand side.

The transformation is essentially the same as the model theoretic semantics in the previous section, except that we build equations for A^ψ for the appropriate stage indices for the tensor A. Instead of formal minutiae, we thus explain the construction of these tensors through examples. In the matrix multiplication algorithm above, the tensor D_1 has an extra index for dimension k. Within the reduction domain $0 \leqslant k < K$, we replace accesses to other tensors with their SARE equivalent, and replace accesses to D to accesses to D_i with the previous value of k. Outside the reduction domain, we replicate the last value of the previous stage, which is just $D_0(i, j)$ in this case:

$$\begin{aligned} 0 \leqslant k < K &\Rightarrow D_1(i, j, k) = D_1(i, j, k-1) + A^S(i, k) \times B^S(k, j) \\ k < 0 \vee k \geqslant K &\Rightarrow D_1(i, j, k) = D_0(i, j) \end{aligned}$$

The SARE tensor for D, D^S, is defined as the last value of D_1 lexicographically, i.e. $D^S(i, j) = D_1(i, j, K-1)$. Note that when $K \leqslant 0$, $D_1(i, j, K-1)$ is equal to $D_0(i, j)$.

There are a few subtleties here. First, if there are multiple reduction variables, the previous value must be computed lexicographically within the definition rectangle: for two reduction variables $0 \leqslant x < X$ and $0 \leqslant y < Y$, the extra indices to a recursive access would be select$(y \leqslant 0, x-1, x)$ and select$(y \leqslant 0, Y-1, y-1)$ respectively. Second, if there is a filter, when the filter is false, the previous value of the current stage is directly reused. Finally, if the indices depend on the reduction variables, all non-updated locations are defined using the previous value of the current stage. For instance, the update `D(i, 2k) += D(i, k)` where `i` is a pure variable and $0 \leqslant k < K$ is a reduction variable becomes

(within the recurrence domain):

$$0 \leqslant k < K \Rightarrow D_1(i, 2k, k) = D_1(i, 2k, k-1) + D_1(i, k, k-1)$$
$$0 \leqslant k < K \wedge j \neq 2k \Rightarrow D_1(i, j, k) = D_1(i, j, k-1)$$

Because it is derived from the general case where the Halide algorithm is not necessarily affine, the construction can introduce unneeded "copies", i.e. equations that are just defined to reindex another tensor. While in general it is not possible to eliminate such copies, for an affine specification, we can use a library such as `isl` to symbolically solve for the last non-copy definition of each location following the same approach as that of Feautrier [37].

Presburger sets 3

Presburger sets and relations form the cornerstone of modern polyhedral representations, and form the basis of the validator presented in chapter 5. This chapter is a compact retelling of (parts of) the excellent tutorial of Verdoolaege [114] on polyhedral concepts, with some omissions and adaptations to make it more suited to our use cases. It is not intended to be read linearly: rather, it should be seen as a definition of background concepts the reader can refer to while reading the rest of this manuscript.

In the electronic version of this document, uses of the concepts defined in this chapter are hyperlinked to their definition; in compatible PDF readers, hovering over a notation should bring up a window with the definition of a notation or concept. This is done using Thomas Colcombet's knowledge LaTeX package [31].

3.1 Presburger arithmetic

Presburger arithmetic, also known as affine arithmetic and linear arithmetic, is the first-order theory of natural numbers equipped with addition, equality, and inequality $(\mathbb{N}, +, =, <)$. *Affine expressions* are built from variables, constants, and addition; *affine constraints* are equalities or inequalities between affine expressions. *Presburger formulas* are logical formulas built from affine constraints, negation $\neg$, conjunction $\wedge$, disjunction $\vee$, and first-order existential $\exists$ and universal $\forall$ quantifiers. Constant multiplication can be defined in Presburger arithmetic using repeated addition:

$$n \cdot x = x + \cdots + x$$

Solvers for Presburger arithmetic support constant multiplication as a primitive.

Presburger arithmetic is well-known for being a decidable subset of Peano arithmetic. While deciding the satisfiability of a Presburger arithmetic formula has a triply-exponential worst case complexity, specialized solvers for Presburger arithmetic or restricted subsets thereof exist and are used in a variety of applications. Solvers for Presburger arithmetic are the foundation of the polyhedral model, a collection of techniques for the representation and optimization of array programs presented in chapter 2.

Another desirable and well-known property of Presbuger arithmetic is that it admits a *quantifier elimination* procedure: any formula in Presburger arithmetic is logically equivalent to a *quantifier-free* formula in Presburger arithmetic... almost. To admit quantifier elimination, Presburger arithmetic must be extended with either one *constant division* or *constant remainder* operation for each natural integer, i.e. terms of the form $\lfloor e/n \rfloor$ or $e \bmod n$ where $n > 1$ is an integer must be allowed (consider for instance the formula $\exists y,\ x = 2y$ that otherwise would not admit an equivalent quantifier-free formula). Affine expressions with constant division or constant remainder (both of which can be defined in terms of the other) are called *quasi-affine*; modern polyhedral tools such as `isl` [112] support quasi-affine expressions everywhere. Because there are otherwise not many differences in the theoretical properties of affine and quasi-affine expressions or constraints, the reader should generally understand the "affine" or "Presburger" qualifiers to be "quasi-affine" instead, unless explicitly specified otherwise.

The rest of this chapter describes the formalization of *symbolic sets* and *symbolic relations* over tuples of integers constrained by symbolic formulas. All the notations and operations presented in this chapter generally apply to arbitrary first-order formulas, but might not be *decidable or computable* for arbitrary formulas. Dealing with arbitrary first-order formulas would require relying on SMT solver heuristics and completeness issues could come up virtually anywhere. On the other hand, all the operations presented in this chapter *are* decidable when using Presburger arithmetic, and admit efficient implementations in libraries such as `isl`. A general goal of the approach presented in this thesis will be to rely on the good theoretical properties of Presburger arithmetic as much as possible.

3.2 Named tuples

Presburger sets are modelled as symbolic sets of *named tuples*, representing arbitrary identifiers indexed by symbolic variables. The symbolic variables (but not the identifiers) are constrained by formulas in Presburger arithmetic, and polyhedral libraries such as `isl` provide efficient symbolic operations on such sets. Named tuples are simply terms representing an arbitrary tree structure using nodes of variable arity:

Definition 3.2.1 (Named tuples)**.** A *named tuple* over a sort $\mathcal{A}$ of arguments ranged over by a is either:

- A name n along with $d \geqslant 0$ arguments a_i for $1 \leqslant i \leqslant d$, written $n\langle a_1, \ldots, a_d \rangle$, or
- A name n along with two named tuples $\mathbf{t_1}$ and $\mathbf{t_2}$, written $n\langle \mathbf{t_1}, \mathbf{t_2} \rangle$.

In the following, we will use three types of named tuples. Named tuples whose arguments are integers are called *integer tuples*; named tuples whose arguments are *distinct* variables are called *integer tuple templates* or simply *variable tuples*; and named tuples whose arguments are affine expressions are called *affine tuples*.

Example 2. $A\langle B\langle 2\rangle, C\langle 3, 4, 5\rangle\rangle$ is a named integer pair. $[x, y, z]$ is a flat anonymous variable tuple. $Q\langle x + 1, z \bmod 2\rangle$ is a flat named affine tuple.

We assume the existence of a distinguished name ϵ which is used to represent anonymous tuples. *Anonymous tuples* are written using square brackets instead of angle brackets: we write $[3]$ to represent the integer tuple $\epsilon\langle 3\rangle$, and $[[x], [y]]$ to represent the variable tuple $\epsilon\langle [x], [y]\rangle$. The `isl` notation uses square brackets instead of angle brackets everywhere; this presentation uses angle brackets in order to distinguish syntactically between the named tuple $a\langle i + 1, j\rangle$ and the array access $a[i + 1, j]$.

Like regular tuples, named tuples represent collections of elements; as such, we denote using **bold face** variables representing named tuples. We permit

"lifting" constructs on elements to tuples when it makes sense to do so: for instance, if $\mathbf{x}$ is a variable tuple and $\mathbf{i}$ is an integer tuple with the same structure we write $\mathbf{x} \mapsto \mathbf{i}$ to represent the mapping from each variable in $\mathbf{x}$ to the integer at the corresponding position in $\mathbf{i}$. Similarly, if $\mathbf{x}$ is a variable tuple, we can write $\exists \mathbf{x},\ \phi$ to quantify ϕ existentially over the variables in $\mathbf{x}$.

Example 3. $A\langle B\langle x\rangle, [y, z]\rangle \mapsto A\langle B\langle 1\rangle, [2, 3]\rangle$ represents the mapping $x \mapsto 1, y \mapsto 2, z \mapsto 3$.

$\exists A\langle B\langle x\rangle, [y, z]\rangle,\ x + y = z$ is the formula $\exists x,\ \exists y,\ \exists z,\ x + y = z$.

This notion of "structure" is called the *space* of the tuple in `isl`, and we will use the two terms interchangeably.

Definition 3.2.2 (Space of a tuple)**.** The space $\mathcal{S}\mathbf{t}$ of the named tuple $\mathbf{t}$ represents the shape or structure of the tuple. It is defined as:

- $\mathcal{S}\mathbf{t} = n/d$ if $\mathbf{t}$ is of the form $n\langle a_1, \ldots, a_d\rangle$
- $\mathcal{S}\mathbf{t} = (n, \mathcal{S}\mathbf{d}, \mathcal{S}\mathbf{r})$ is $\mathbf{t}$ is of the form $n\langle \mathbf{d}, \mathbf{r}\rangle$

The space of a pair $(\mathbf{d}, \mathbf{r})$ of named tuples is the pair of the spaces $(\mathcal{S}\mathbf{d}, \mathcal{S}\mathbf{r})$.

3.3 Symbolic sets

Following standard mathematical notation, a set of named tuples is written as the list of elements in the set, enclosed by braces and separated by commas. Following `isl` notation, we also allow separating elements using semicolon interchangeably with commas. The named tuples in a set do not need to all have the same structure for instance we can build the set $\{A\langle 1, 2\rangle, [3]\}$ out of $A\langle 1, 2\rangle$ and $[3]$ that have different structure.

Definition 3.3.1 (Symbolic set)**.** Let $\mathbf{t}$ be a variable tuple and ϕ be a (Presburger) formula, possibly involving more variables than those of $\mathbf{t}$. Then $\{\mathbf{t} : \phi\}$, where the variables of $\mathbf{t}$ are bound in ϕ, is a *homogenous symbolic set*.

A *symbolic set* is a finite union of homogenous symbolic sets with possibly (but not necessarily) different structures. A symbolic set is represented by separating the components of the union with a semicolon:

$$\begin{aligned} C &::= \mathbf{t} : \phi \mid \mathbf{t} : \phi; C \\ S &::= \emptyset \mid \{C\} \end{aligned}$$

If ϕ is the constant formula true, we omit both ϕ and the preceding colon, so that for instance $\{[a\langle i\rangle, b\langle j\rangle]\}$ is the set $\{[a\langle i\rangle, b\langle j\rangle] : \text{true}\}$.

The formula ϕ can contain free variables; the symbolic set $\{\mathbf{t} : \phi\}$ is *parametric* and has the free variables of ϕ not bound by $\mathbf{t}$. The free variables of a symbolic set are called the *parameters* of the symbolic set.

Example 4. The set $\{A\langle i\rangle : 0 \leqslant i < N\}$ is a parametric set with N as unique parameter.

A symbolic set S can be *evaluated* to a concrete set in an environment $\mathcal{E}$ containing bindings for the parameters of the symbolic set. The evaluation of S in $\mathcal{E}$ is denoted $[\![S]\!]_{\mathcal{E}}$ and, assuming that $[\![e]\!]_{\mathcal{E}}$ represents the evaluation of expression e to an integer and $[\![\phi]\!]_{\mathcal{E}}$ is the truth value of formula ϕ, we have:

$$\begin{aligned} [\![\mathbf{t} : \phi]\!]_{\mathcal{E}} &= \{\mathbf{i} \mid [\![\phi]\!]_{\mathcal{E}+\mathbf{t}\mapsto\mathbf{i}}\} \\ [\![\mathbf{t} : \phi; C]\!]_{\mathcal{E}} &= \{\mathbf{i} \mid [\![\phi]\!]_{\mathcal{E}+\mathbf{t}\mapsto\mathbf{i}}\} \cup [\![C]\!]_{\mathcal{E}} \\ [\![\emptyset]\!]_{\mathcal{E}} &= \emptyset \\ [\![\{C\}]\!]_{\mathcal{E}} &= [\![C]\!]_{\mathcal{E}} \end{aligned}$$

Note that the expressions on the right-hand side are set comprehensions in the meta-language whereas the expressions within the brackets are syntactic objects. The metavariable $\mathbf{i}$ here represents an integer tuple, and the notation $\mathbf{t} \mapsto \mathbf{i}$ implies that $\mathbf{t}$ and $\mathbf{i}$ have the same structure.

The representation of symbolic sets is not unique: a given heterogeneous set can be represented using many equivalent formulations of the formula ϕ; furthermore, multiple homogenous sets in a heterogeneous set can live in the same space.

Example 5. The sets $\{A\langle i, j\rangle : i < j; A\langle i, j\rangle : i = j; [i] : i = 3\}$ and $\{A\langle i, j\rangle : i \leqslant j; [i] : i = 3\}$ are different representations of the same symbolic set, i.e. they both evaluate to the same concrete set in all environments.

Since the formulas defining a homogenous set is defined using Presburger arithmetic, and formulas in Presburger arithmetic are decidable, it is possible to decide the emptiness of a symbolic set:

Definition 3.3.2 (Empty set)**.** A homogenous symbolic set $\{\mathbf{t} : \phi\}$ is *empty*, written $\nexists(\{\mathbf{t} : \phi\})$, if the formula $\exists\mathbf{t},\ \phi$ is false.

A heterogeneous symbolic set S is *empty*, written $\nexists(S)$, if all of its component homogenous sets are empty. Equivalently, S is *empty* if $[\![S]\!]_{\mathcal{E}}$ is equal to the empty set for all environments $\mathcal{E}$.

Many operations on symbolic sets are easier to formalize on disjoint homogenous sets. The notion of space can be used to define the *space decomposition* of a symbolic set as a disjoint union of homogenous sets, an operation implemented using Presburger arithmetic by polyhedral libraries such as `isl`.

Definition 3.3.3 (Space decomposition)**.** The *space decomposition* of a symbolic set S is the unique collection of nonempty *homogenous* sets S_i such that $\bigcup_i S_i = S$ and no two distinct components S_i and S_j have the same space.

Operations on symbolic sets can be defined first on homogenous sets, then lifted to a symbolic set using its space decomposition. This is the case of the union $S_1 \cup S_2$, intersection $S_1 \cap S_2$ and difference $S_1 - S_2$ that can be defined on pairs of homogenous sets with the same space as follows (where the set definitions have been α-renamed to the same variables):

$$\{\mathbf{t} : \phi_1\} \cup \{\mathbf{t} : \phi_2\} = \{\mathbf{t} : \phi_1 \vee \phi_2\}$$
$$\{\mathbf{t} : \phi_1\} \cap \{\mathbf{t} : \phi_2\} = \{\mathbf{t} : \phi_1 \wedge \phi_2\}$$
$$\{\mathbf{t} : \phi_1\} - \{\mathbf{t} : \phi_2\} = \{\mathbf{t} : \phi_1 \wedge \neg\phi_2\}$$

These operations can then be lifted to heterogeneous symbolic sets by applying them independently to each component of the space decomposition. An empty homogenous set is used when one of the two sets has no component of the appropriate shape. The unbounded complement of a homogenous set $\{\mathbf{t} : \phi\}$ can be defined as $\{\mathbf{t} : \neg\phi\}$ but cannot easily be lifted to heterogeneous symbolic sets.

There is a special case of symbolic sets: sets that are either empty or contain a single element. Such sets can be thought of as representing (conditional) constants: for any value of its parameters, either the set is empty and the constant is undefined, or the set contains the value of the constant as single element. Abusing the term, and following Verdoolaege [114], we define a singleton set as follows:

Definition 3.3.4 (Singleton set)**.** A symbolic set S is a *singleton* if, in any environment $\mathcal{E}$ mapping the free variables of S to integer values, there is at most one element in $[\![S]\!]_{\mathcal{E}}$.

Presburger arithmetic libraries such as `isl` can decide whether a set is a singleton by checking that two elements in the set are necessarily equal. This definition of singleton sets might make more sense when thinking of it as the set version of a single-valued relation as defined below.

Use of symbolic sets In a polyhedral representation of programs, relations between pairs of named tuples are used to represent both the scheduling of statement instances and the access relations between instances. The names of the tuples are identifier representing either statements or arrays, depending on the type of relation considered. The presentation of section 2.1 can fairly easily be expressed in terms of named tuples; for more details, the interested reader can refer to the tutorial of Verdoolaege [114]. For our purpose of the translation-validation of a tensor compiler, we will use named tuples to represent and transport sets of expressions living in a multidimensional space between the specification and the implementation. As such, instead of identifiers, we will use expression contexts with multiple holes as names: for instance, we can represent the set

$$\{A(i,k) \times B(k,j) \mid 0 \leqslant i,j,k < N\}$$

by extracting its affine components into an expression context used as an opaque name:

$$\{(A(\square_1, \square_2) \times B(\square_2, \square_3))\langle i, k, j\rangle \mid 0 \leqslant i, j, k < N\}$$

This representation enables handling the affine parts of an expression semantically using Presburger arithmetic while still being able to handle arbitrary expressions; this representation is explained in more details in chapter 5.

3.4 Unit sets

It is sometimes useful to represent conditions on the parameters of a set such as the conditions under which the set is empty or nonempty. To do so, we can use unit sets, that can be thought of as symbolic boolean, and are represented using a formula within braces, prefixed with a colon:

Definition 3.4.1 (Unit set)**.** A *unit set* is written $\{ : \phi\}$ where ϕ is a (Presburger) formula. Unit sets are also called parametric sets and represent a constraint on the free variables of ϕ. Unit sets are evaluated to the truth value of the formula ϕ in an environment $\mathcal{E}$.

The union, intersection, and difference of unit sets can be defined in the same way as for symbolic sets by computing the appropriate boolean combination on the formulas. We also allow the intersection (resp. difference) of a symbolic set with a unit set by adding a conjunction with the unit set's formula (resp. its negation) to the symbolic set. The union of a symbolic set with a unit set is not meaningful and hence disallowed. We occasionally abuse notations and write the formula ϕ directly instead of the unit set $\{: \phi\}$; in particular, the intersection of a set S with a formula ϕ is to be understood as:

$$S \cap \phi = S \cap \{: \phi\}$$

A common use of unit sets is to represent the *non-emptiness condition* of a symbolic set.

Definition 3.4.2 (Non-emptiness condition)**.** If S is a symbolic set, then $\exists S$ is a unit set representing the non-emptiness of S. More specifically, if $S = \bigcup_i \{\mathbf{t_i} : \phi_i\}$ is the space decomposition of S, then:

$$\exists S = \{: \bigvee_i \exists \mathbf{t_i},\ \phi_i\}$$

Proposition 3.4.1. *In any environment $\mathcal{E}$ that binds the parameters of S, $[\![S]\!]_{\mathcal{E}}$ is empty if, and only if, $[\![\exists]\!]_{\mathcal{E}}$ is* false*.*

With this non-emptiness condition we can define the update combinator $\triangleright$ on sets:

Definition 3.4.3 (Update)**.** If S_1 and S_2 are symbolic sets, the *update* of S_1 with S_2, denoted $S_1 \triangleright S_2$, is defined as:

$$S_1 \triangleright S_2 = S_2 \cup (S_1 - \exists S_2)$$

$S_1 \triangleright S_2$ is equal to S_2 when S_2 is nonempty, and to S_1 otherwise, hence the notion of *update*. Note that when $\exists S_1$ and $\exists S_2$ are disjoint, $S_1 \triangleright S_2$ is simply $S_1 \cup S_2$.

3.5 Symbolic relations

Symbolic sets represent a set of named tuples in terms of latent parameters. On the other hand, it is often useful to represent relations between *pairs* of tuples: for instance, in polyhedral representations, we want to represent dependence relations and access relations. For the validator presented in chapter 5, relations enables representing a heap symbolically by relating (symbolic) locations in an array and an expression that location is currently equal, if known.

Definition 3.5.1 (Symbolic relation)**.** A *symbolic relation*, ranged over by R, is a relation between argument tuples. An *homogenous symbolic relation* is denoted

$\{\mathbf{s}_1 \to \mathbf{s}_2 : \phi\}$ where $\mathbf{s}_1$ and $\mathbf{s}_2$ are variable tuples and ϕ is a formula. The variables of $\mathbf{s}_1$ and $\mathbf{s}_2$ are bound in ϕ.

Like sets, symbolic relations can be heterogeneous and are represented using an explicit union separated by a semicolon.

Like for symbolic sets, union, intersection, and difference of symbolic relations can be defined by applying the appropriate boolean connectives to the underlying formula. The domain and range of a symbolic relation can be obtained by projecting out the tuples on the right and on the left of the arrow, respectively:

$$\begin{aligned} \mathrm{dom}(\{\mathbf{s}_1 \to \mathbf{s}_2 : \phi\}) &= \{\mathbf{s}_1 : \exists \mathbf{s}_2,\ \phi\} \\ \mathrm{ran}(\{\mathbf{s}_1 \to \mathbf{s}_2 : \phi\}) &= \{\mathbf{s}_2 : \exists \mathbf{s}_1,\ \phi\} \end{aligned}$$

Furthermore, we allow the intersection and difference of a symbolic relation with a symbolic set, understood as restricting the *domain* of the relation by considering both decompositions and adding the appropriate conjunction where the spaces match.

Example 6. If $R = \{a\langle i, j\rangle \mapsto b\langle i + j\rangle; c\langle k\rangle \mapsto b\langle k + 2\rangle\}$ and $S = \{a\langle i, j\rangle : i > j; b\langle k\rangle\}$, then $R \cap S$ is the relation:

$$R \cap S = \{a\langle i, j\rangle \mapsto b\langle i + j\rangle : i > j\}$$

We also allow the intersection and difference of a symbolic relation with a unit set, by adding or subtracting the corresponding formula to all the homogenous parts.

Symbolic relations with at most one range element per domain element are *single-valued*.

Definition 3.5.2 (Single-valued relation)**.** A *single-valued relation* is a symbolic relation which has at most one output associated with each input. In this case we also say that the relation is a partial function.

The single-valuedness of a relation R is denoted by sv(R) and indicates that $[\![R]\!]_{\mathcal{E}}$ is single-valued in *any* environment $\mathcal{E}$ that binds the parameters of R.

Definition 3.5.3 (Inverse of a relation)**.** A symbolic relation R can be inverted by exchanging its domain and range. We denote by R^{-1} the inverse of R, defined on its space decomposition:

$$\left(\bigcup_i \{\mathbf{d}_i \to \mathbf{r}_i : \phi_i\}\right)^{-1} = \bigcup_i \{\mathbf{r}_i \to \mathbf{d}_i : \phi_i\}$$

Symbolic relations can be *applied* to symbolic sets, yielding the set of elements that are in relation with any element of the set it is applied to.

Definition 3.5.4 (Relation application)**.** The application R(S) of a binary relation R to a set S is the set containing the elements that appears as the right-hand side in R when the left-hand side is an element of S. In other words:

$$\begin{aligned} R(S) &= \mathrm{ran}(R \cap S) \\ &= \{\mathbf{j} : \exists \mathbf{i},\ \mathbf{i} \in S \wedge \mathbf{i} \to \mathbf{j} \in R\} \end{aligned}$$

Moreover, parameters can be bound to a set in order to build a relation between the values in the set and the values of the parameters.

Definition 3.5.5 (Parameter binding)**.** If S is a symbolic set and $\mathbf{x}$ a variable tuple, $\mathbf{x}$ can be *bound* in S to a symbolic relation, mapping the elements of S to the possible values of the variables in $\mathbf{x}$. Binding the variable tuple $\mathbf{x}$ in set S is denoted $\lambda\mathbf{x}.S$, borrowing the notation from the lambda calculus since $(\lambda\mathbf{x}.\ S)(\{\mathbf{x}\}) = S$, and defined on the space decomposition of S as:

$$\lambda\mathbf{x}.\ \bigcup_i \{\mathbf{t}_i : \phi_i\} = \bigcup_i \{\mathbf{x} \to \mathbf{t}_i : \phi_i\}$$

In practice, it is often more convenient to inverse the relation, so that the bound variable is in the range rather than the domain of the resulting relation. We use the keyword bind to indicate this "inverted binding", i.e.:

$$\mathrm{bind}_{\mathbf{x}}(S) = (\lambda\mathbf{x}.\ S)^{-1}$$

If $\mathbf{x}$ is a single variable $[x]$, we allow writing an inequality involving x as a subscript, with the convention that the inequality should be intersected with the set S:

$$\mathsf{bind}_{\iota_1 \leqslant x < \iota_2}(S) = \mathsf{bind}_{[x]}(S) \cap \{ : \iota_1 \leqslant x < \iota_2\}$$

Symbolic relations can be converted to sets of pairs, and sets of pairs can be converted to symbolic relations, using the wrap and unwrap operations.

Definition 3.5.6 (Wrap and unwrap)**.** The wrap operation converts a symbolic relation $R = \bigcup_i \{\mathbf{t}_i \to \mathbf{s}_i : \phi_i\}$ to a symbolic set containing anonymous pairs of the domain and range elements:

$$\mathsf{wrap}\left(\bigcup_i \{\mathbf{t}_i \to \mathbf{s}_i : \phi_i\}\right) = \bigcup_i \{[\mathbf{t}_i, \mathbf{s}_i] : \phi_i\}$$

The unwrap operation is the inverse of wrap: it converts a symbolic set contains only anonymous pairs to a symbolic relation, with the first component of the pair as domain and the second component of the pair as range.

$$\mathsf{unwrap}\left(\bigcup_i \{[\mathbf{t}_i, \mathbf{s}_i] : \phi_i\}\right) = \bigcup_i \{\mathbf{t}_i \to \mathbf{s}_i : \phi_i\}$$

If R is a symbolic relation, abstraction and application automatically wraps. $\lambda\mathbf{x}.\ R$ is $\lambda\mathbf{x}.\ \mathsf{wrap}(R)$, and $R_1(R_2)$ is $R_1(\mathsf{wrap}(R_2))$.

Note that the unwrap operation is partial and is not defined on all symbolic sets.

dom and ran cannot be used on wrapped relations, but we can use fst and snd instead.

Definition 3.5.7 (First and Second)**.** The fst (resp. snd) operator can be used on a set of pairs to access the first (resp. second) component of the pair. More

precisely, if $S = \bigcup_i \{[\mathbf{t}_i, \mathbf{s}_i] : \phi_i\}$ is a symbolic set, fst and snd are defined as:

$$\begin{aligned} \text{fst}(S) &= \text{dom}(\text{unwrap}(S)) \\ &= \bigcup_i \{\mathbf{t}_i : \exists \mathbf{s}_i.\ \phi_i\} \\ \text{snd}(S) &= \text{ran}(\text{unwrap}(S)) \\ &= \bigcup_i \{\mathbf{s}_i : \exists \mathbf{t}_i.\ \phi_i\} \end{aligned}$$

fst and snd can be lifted to relations, where they apply to the range component of the relation (e.g. $\text{fst}(\{\mathbf{t} \to [\mathbf{s}, \mathbf{u}] : \phi\}) = \{\mathbf{t} \to \mathbf{s} : \exists \mathbf{u}.\ \phi\}$).

The equivalent of wrap and unwrap "lifted" to relations are the curry and uncurry operations.

Definition 3.5.8 (Curry and Uncurry)**.** The curry operation converts a symbolic relation of shape

$$R = \bigcup_i \{[\mathbf{t}_i, \mathbf{s}_i] \to \mathbf{u}_i : \phi_i\}$$

to a symbolic relation

$$\text{curry}(R) = \bigcup_i \{\mathbf{t}_i \to [\mathbf{s}_i, \mathbf{u}_i] : \phi_i\}$$

It is named curry due to the similarity with the operator of the same name in functional programming, that transforms a function on pairs to a function returning a function, represented in wrapped form by the $[\mathbf{s}_i, \mathbf{u}_i]$ pair.

The uncurry operation is the reverse of the curry operation: it converts a symbolic relation of shape

$$R = \bigcup_i \{\mathbf{t}_i \to [\mathbf{s}_i, \mathbf{u}_i] : \phi_i\}$$

into

$$\text{uncurry}(R) = \bigcup_i \{[\mathbf{t}_i, \mathbf{s}_i] \to \mathbf{u}_i : \phi_i\}$$

Note that curry and uncurry are partial, and only defined on symbolic sets of the appropriate shapes.

3.6 Piece-wise Expressions

It is often useful to represent a tuple expression as a function of another tuple expressions. Piece-wise expressions fulfil that role, and can be thought of as an explicit representation of singleton sets and single-valued relations.

Definition 3.6.1 (Piece-wise expression)**.** An expression tuple $\mathbf{f}$ is a named tuple whose arguments are affine or quasi-affine expressions of variables. A *piece-wise expression* P maps a variable tuple to an expression tuple when a given formula holds, and can be written as a disjoint union:

$$P ::= \{\mathbf{x}^1 \mapsto \mathbf{e}^1 : \phi_1; \ldots \mathbf{x}^n \mapsto \mathbf{e}^n : \phi_n\}$$

The domain of the piece-wise expression P, denoted $\mathrm{dom}(P)$, is the symbolic set:

$$\mathrm{dom}(P) = \{\mathbf{x}^1 : \phi_1, \ldots, \mathbf{x}^n : \phi_n\}$$

As a special case, a *constant* or *parametric* piece-wise expression consists of a union of expression tuples, in which case the domain is a unit set:

$$P ::= \cdots \mid \{\mathbf{e}^1 : \phi_1; \ldots \mathbf{e}^n : \phi_n\}$$

Example 7. $\{[i, j] \mapsto a\langle i + 1, j\rangle : i < N; [i, j] \mapsto a\langle N, j\rangle : i \geqslant N\}$ is a piece-wise expression representing the location `a[min(i, N), j]`.

We allow the intersection and difference of a piece-wise expression with a symbolic set, understood as restricting the domain of the piece-wise expression by considering both decompositions and adding the appropriate conjunction to the piece-wise formula on the piece where the domain space of the expression

matches the domain space of the heterogeneous set. Similarly, we allow the intersection and difference of a piece-wise expression and a unit set.

A piece-wise expression P can be evaluated in an environment $\mathcal{E}$ to a function from integer tuples to either integer tuples or the special undefined value $\perp$ indicating that no pieces matches the corresponding input tuple.

3.7 Lexicographic optimization

Binding parameters allows building maps from (symbolic) array locations to the iteration of a sequential loop that write to said location. Projecting out the maximal value associated with each location would yield a piece-wise expression representing the last iteration that write to that location. This is the basis of the array expansion procedure that is used to compute an exact dataflow analysis. In general, we want to be able to compute the latest element across a multidimensional tuple; hence, we use a lexicographic order:

Definition 3.7.1 (Lexicographic order)**.** Given two vectors **a** and **b** of equal length, **a** is said to be lexicographically smaller than **b** if it is equal to **b** or if it is smaller in the first position in which it differs from **b**.

The lexicographic order is written $\preceq$ and its strict version is written $\prec$.

Assuming a strict partial order $<_{\mathcal{N}}$ on the set of names, the lexicographic order can be extended to named tuples by comparing the names before their arguments.

Definition 3.7.2 (Lexicographic order of named tuples)**.** If $\mathbf{t_1}$ and $\mathbf{t_2}$ are two integer tuples and the names appearing in the tuples are ordered by a strict partial order $<_{\mathcal{N}}$, the lexicographic order on named tuples is defined as follows:

- If $\mathbf{t_1}$ and $\mathbf{t_2}$ have distinct names n_1 and n_2, then $\mathbf{t_1} \prec \mathbf{t_2}$ if, and only if, $n_1 <_{\mathcal{N}} n_2$.

- If $\mathbf{t_1}$ and $\mathbf{t_2}$ are flat tuples with the same name $n\langle \mathbf{i}_1 \rangle$ and $n\langle \mathbf{i}_2 \rangle$, then $\mathbf{t_1} \prec \mathbf{t_2}$ if $\mathbf{i}_1$ and $\mathbf{i}_2$ have the same length and $\mathbf{i}_1 \prec \mathbf{i}_2$.

- If $\mathbf{t_1}$ and $\mathbf{t_2}$ are pairs with the same name $n\langle \mathbf{t_{1,1}}, \mathbf{t_{1,2}} \rangle$ and $n\langle \mathbf{t_{2,1}}, \mathbf{t_{2,2}} \rangle$, then $\mathbf{t_1} \prec \mathbf{t_2}$ if either $\mathbf{t_{1,1}} \prec \mathbf{t_{2,1}}$ or $\mathbf{t_{1,1}} = \mathbf{t_{2,1}}$ and $\mathbf{t_{1,2}} \prec \mathbf{t_{2,2}}$.

- Otherwise, $\mathbf{t_1}$ and $\mathbf{t_2}$ are incomparable.

Generally, the lexicographic order on named tuples is not a total order. It is however total when restricted to sets of tuples where the order on names $<_{\mathcal{N}}$ is total and each name is used to denote a unique space. In particular, it is total on a heterogenous set.

The definition of the lexicographic order on named tuples given above is more general than the usual definition which is restricted to comparing named tuples in the same space by their argument vector. This is equivalent to stating that distinct names are never ordered in the above definition, which can be assumed in most of this manuscript, except when called out explicitly.

If S and S' are symbolic sets, we can build the set $S \prec S'$ defined as $S \prec S' = \{[\mathbf{s}, \mathbf{s}'] \mid \mathbf{s} \in S \wedge \mathbf{s}' \in S' \wedge \mathbf{s} \prec \mathbf{s}'\}$ that contains pairs of elements of S and greater elements of S'. $S \preceq S'$, $S \succ S'$ and $S \succeq S'$ can be defined similarly.

The lexicographic maximum and minimum allow computing the lexicographically maximal and minimal elements of a set, respectively.

Definition 3.7.3 (Lexicographic maximum and minimum)**.** The lexicographic maximum (resp. minimum) of a totally ordered symbolic set S, denoted lexmax(S) (resp. lexmin(S)), is a piece-wise expression with unit domain such that, in any environment $\mathcal{E}$, one of the following is true:

- $[\![\text{lexmax}(S)]\!]_{\mathcal{E}}$ is undefined (resp. $[\![\text{lexmin}(S)]\!]_{\mathcal{E}}$) and $[\![S]\!]_{\mathcal{E}}$ is empty, or

- $[\![\text{lexmax}(S)]\!]_{\mathcal{E}} \in [\![S]\!]_{\mathcal{E}}$ (resp. $[\![\text{lexmin}(S)]\!]_{\mathcal{E}} \in [\![S]\!]_{\mathcal{E}}$) and for any $s \in [\![S]\!]_{\mathcal{E}}$, $s \preceq [\![\text{lexmax}(S)]\!]_{\mathcal{E}}$ (resp. $[\![\text{lexmin}(S)]\!]_{\mathcal{E}} \preceq s$).

For a symbolic set $S = \{\mathbf{t}_i : \phi_i\}$, the lexicographic maximum can be defined as

a quantified first-order formula:

$$\text{lexmax}(S) = \mathbf{s} \Leftrightarrow \left(\bigvee_i \exists \mathbf{t}_i,\ \mathbf{t}_i = \mathbf{s} \wedge \phi_i \right) \wedge \bigwedge_i \forall \mathbf{t}_i,\ \phi_i i \Rightarrow \mathbf{t}_i \prec \mathbf{s}$$

If the formulas ϕ_i are expressed in a logic that admits a quantifier elimination procedure such as Presburger arithmetic, the quantifiers introduced by the lexicographic maximum or minimum operations can be eliminated to yield an equivalent quantifier-free formula. In the case of Presburger arithmetic, parametric integer programming (PIP) of Feautrier [36] can be used instead. PIP directly computes the lexicographic maximum as a symbolic piece-wise affine expression, and is implemented in the `isl` library.

The lexicographic maximum and minimum operations can be defined similarly for symbolic relations, and computed in the same way using PIP. For a symbolic relation R, the lexmax(R) and lexmin(R) are piece-wise expressions with the same domain as R, and each element in the domain is mapped to the lexicographically greatest (resp. lowest) value associated with that domain element.

3.8 Notations and Conventions

In explicit notations for symbolic constructs such as sets or relations, if the formula ϕ entails an equality between a variable and a piece-wise quasi affine expression using only free variables and variables that appear on the left of that variable, we allow writing the equality inline. For instance, we can write:

`isl` supports this shorthand notation, and uses it when displaying objects, but it only uses quasi-affine expressions that are not piece-wise.

$$\{[x_1, x_2] \mapsto [x_3 = x_1 + x_2, x_4] \mid x_3 \geqslant x_4\}$$

instead of

$$\{[x_1, x_2] \mapsto [x_3, x_4] \mid x_3 = x_1 + x_2 \wedge x_3 \geqslant x_4\}$$

If the variable no longer appears in the formula after having eliminated the equality, its name can be omitted entirely. For instance, we can also write

$$\{[x_1, x_2] \mapsto [x_1 + x_2, x_4] \mid x_1 + x_2 \geqslant x_4\}$$

to represent the same relation.

Note that this can change the precise representation of the underlying structure. In general, we assume that sets (and relations, etc.) represent classes of equivalent sets: changing the representation does not change the underlying mathematical object. Changing the representation of objects can however change the *runtime characteristics* of algorithms and operations (e.g. projections). For instance, `isl` provides a coalescing primitive that exploits properties of integer linear arithmetic to simplify the representation, and notably merge multiple homogenous sets when possible. This can drastically improve the performance of many algorithms that often are quadratic in the number of homogenous sets present in the representation. Coalescing has been described by Verdoolaege [113] who has shown its impact on improvements and a reduction of time-outs in several applications. The implementation described in chapter 6 relies on adequate use of the coalescing operation to get good performance, notably to perform on-the-fly "re-rolling" of unrolled loops.

4 An intermediate language for tensor compilers

This chapter presents the formal specification of an intermediate language designed to stand in-between a tensor compiler and a traditional compiler such as LLVM. This language, called Sched, is a simple imperative language with arrays and loops. Sched is a subset of the Stmt language used internally by Halide, TVM, and Tensor Comprehensions, extended with annotations for the purpose of translation-validation that are explained in this chapter and used in chapter 5. The presentation of this chapter focuses on the language itself and aims to explain the syntax and semantics of Sched. It is intended to provide relatively complete information to the compiler writer (with a background in formal semantics) interested in targeting Sched: the details of the verification techniques proposed for Sched are postponed until chapter 5.

In this chapter, and the rest of this thesis, we will assume the existence of an ambiant specification and a model M for that specification. In order to abstract away the details of a specific tensor compiler, we assume that the specification is expressed as a SARE. If the compiler to validate uses a different specification formalism, as is the case with Halide, we assume that the specification has previously been translated to a SARE, for instance using the algorithm presented in subsection 2.4.4.

The underlying goals for Sched lead to the following design decisions.

Sched programs should distinguish between the expressions representing computation requested by the original user of the tensor compiler, which we will call *semantic expressions* (e.g. `a[i, k] * b[k, j]` in a matrix multiplication is a semantic expression), and the index expressions appearing in loop bounds and array accesses that can be transformed as a result of compiler transformations

(e.g. the user might have written `i`, `j`, and `k` in the multiplication above, but after compilation, those expressions became `4 * i0 + i1`, `max(0, j)` and `16 * k0 + 4 * k1 + k2`).

The structure-modifying transformations performed by the compiler have an effect on the index expressions, and we want our verifier to be able to *invert* those transformations in order to recover the original indices written by the programmer. To do so, we want to use polyhedral tools such as `isl` (see chapter 3); hence, we must restrict index expressions to piece-wise affine combinations of outer loop iterators and program parameters. Although most useful loop transformations can be represented using affine transformations, some transformations applied by compilers such as Halide can generate non-affine index expressions; moreover, the user may want to write non-affine specifications such as histograms. I restrict SCHED to affine index expressions for now, and discuss in section 9.6 ways in which this restriction could be relaxed.

On the other hand, we want the compiler to be free to perform arbitrary transformations on semantic expressions provided that their value does not change; for instance, the compiler should be free to introduce bit tricks and to simplify commutative, associative or distributive operators. The intent is to verify each such transformation *locally*, at the level of an array write: once the transformations performed by the compiler through index expressions have been, in a way, "inverted" to bring the semantic expression into the specification space, we can compare it with its expected value using verification tools such as SMT solvers. To make this verification possible, SCHED programs contain annotations on assignments that express the expected value of the write using the user's original specification. These annotations do not impact the semantics of SCHED programs, but are required by the verification techniques of the next chapter. section 5.6 discusses the generation of these annotations.

Finally, with the goal of facilitating translation validation in the next chapter, and perhaps surprisingly for a language that contains parallel loops, the semantics for SCHED is given in a big-step operational style. This big-step semantics evaluates a program to a set of updates performed while evaluating the program that can represent deterministic communication-free parallel loops, the likes of which are typically targeted by tensor compilers.

In formal definitions, we assume disjoint, countably infinite sets of array names

(ranged over by a, b), variables (ranged over by x, y), tensor names (ranged over by A, B), and function names (range over by f, g). We also assume a set of values $\mathcal{U}$, and a set of types (ranged over by τ). Each type τ can be interpreted by a set of values $[\![\tau]\!] \subseteq \mathcal{U}$. In general, the $[\![\cdot]\!]$ notation is used to denote an interpretation or evaluation. We assume the existence of two distinguished types $\mathbb{A}$, which evaluates to the set of all integers, and $\mathbb{B}$, which evaluates to $\{\text{true}, \text{false}\}$. Each function name f has a function type $\tau_f = \tau_1 \times \cdots \times \tau_{a_f} \to \tau$, where a_f is the arity of f, and a semantic interpretation $[\![f]\!]$ as a well-typed function of a_f values. Constants are nullary functions.

A signature $\mathcal{S}$ is a pair $\langle \mathcal{P}, \mathcal{A} \rangle$ where $\mathcal{P}$ is a finite set of integer variables called the *program parameters*, and $\mathcal{A}$ is a finite set of *tensor names* ranged over by A. A tensor A is equipped with a type τ_A and an arity n_A. A model M over $\mathcal{S}$ is an assignment of a value $[\![x]\!]_M$ for each parameter $x \in \mathcal{P}$, and of a function $[\![A]\!]_M$ from $\mathbb{Z}^{n_A}$ to $[\![\tau_A]\!]$ for each $A \in \mathcal{A}$.

4.1 Syntax

Although we want to enforce a distinction between affine and non-affine expressions in order to make program analysis tractable, the distinction is not made at the syntax level but through the use of a type system. The hope is to make it easier to extend the system to handle non-affine expressions in loop bounds and array accesses in a controlled way. In particular, programs that do not respect the affine restrictions of the type system are still given a semantics: although the techniques developed in chapter 5 do not work with such programs, extensions such as those discussed in section 9.6 can be designed for them and rely on the same underlying semantics. The grammar for the expressions of Sched is given in Fig. 4.1 Even though there is no syntactic distinction, to make intentions clearer, the metavariable ι is used where an index expression is expected, the metavariable e is used when a semantic expression is expected, and the metavariable t is used when a tensor expression is expected.

As discussed in section 9.6, allowing non-affine indices in array reads is fairly easy, whereas allowing non-affine indices in array writes and in conditionals is harder.

The syntax for expression includes tensor accesses $A(\iota_1, \ldots, \iota_n)$. This is because Sched is designed to be used with a validator: tensor accesses referring to the specification can be used only in assertions (as enforced by the type system);

Expressions

$$
\begin{array}{rr}
e, \iota, t ::= x \mid l & \text{variable and literals} \\
\mid a[\iota_1, \ldots, \iota_n] & \text{array indexing} \\
\mid A(\iota_1, \ldots, \iota_n) & \text{tensor indexing} \\
\mid \text{let } x = \iota_1 \text{ in } e_2 & \text{let expression} \\
\mid \iota_1 + \iota_2 \mid n \cdot \iota & \\
\mid \lfloor \iota / n \rfloor \mid \iota \bmod n & \text{linear arithmetic} \\
\mid \iota_1 = \iota_2 \mid \iota_1 \leqslant \iota_2 & \\
\mid \iota_1 \neq \iota_2 \mid \iota_1 < \iota_2 & \text{comparisons} \\
\mid \iota_1 \ \&\& \ \iota_2 \mid \iota_1 \ || \ \iota_2 \mid ! \iota & \text{Boolean connectives} \\
\mid \text{select}(\iota, e_1, e_2) & \text{eager conditional} \\
\mid f(e_1, \ldots, e_n) & \text{pure function call}
\end{array}
$$

Figure 4.1: Syntax of expressions

they do not have an executable semantics. The pure function calls $f(e_1, \ldots, e_n)$ can refer to any operator or functions on values, and have a call-by-value semantics. Pure function calls are not allowed in index expressions: instead, the quasi-affine expressions $\iota_1 + \iota_2$, $n \cdot \iota$, $\lfloor \iota / n \rfloor$ and $\iota \bmod n$ are called out specifically. Pure functions are allowed to take both semantic expressions and index expressions as arguments, and hence can include casts from the arbitrary precision indices to machine integers. Any primitive operation on values such as multiplication, addition, exponentiation, etc. is implemented using such function calls, and is opaque to the rest of the system.

The grammar for the commands (or statements) of SCHED is given in Fig. 4.2, and describes an imperative language of arrays and loops. Assignments are annotated with a prophetic expression t representing an assertion that the value written by the assignment is equal to the value denoted by the prophetic expression in the original specification, as will be explained in chapter 5. Prophetic expressions are ignored at runtime: they are a form of ghost code used for verification purposes only. The allocation command

$$
\begin{aligned}
c ::= {} & \mathsf{skip} \\
& \mid c_1 \ ; \ c_2 \\
& \mid a[\iota_1, \ldots, \iota_n] \ \{t\} := e \\
& \mid \mathsf{if} \ \iota \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2 \\
& \mid \mathsf{let} \ x = \iota \ \mathsf{in} \ c \\
& \mid \mathsf{allocate} \ a : \tau[\iota_1 \times \cdots \times \iota_n] \ \mathsf{in} \ c \\
& \mid \mathsf{for} \ x < \iota ; \mathsf{do} \ c \\
& \mid \mathsf{par} \ x < \iota ; \mathsf{do} \ c
\end{aligned}
$$

Figure 4.2: Syntax of Commands

allocate $a : \iota_1, \ldots, \iota_n$ in c allocates a new n-dimensional array a without initializing its contents. Parallel loops par $x < \iota ; \mathsf{do}\ c$ are non-communicating: they should be thought of as executing each iteration in an independent thread that only synchronizes at the end of the loop. It particular, the semantics will enforce the determinism of parallel loops.

Parallel loops are the only source of concurrency in the language. This is consistent not only with polyhedral techniques and scheduling approaches such as that of Halide, but also with the hierarchical structure of GPUs. In a realistic compilation pipeline (e.g. in Halide), parallel loops are tagged with the level in the hierarchy that they belong to, but this has no impact on the high-level semantics, and hence is ignored in this presentation.

4.2 Dynamic semantics

A Sched expression evaluates to a value in $\mathcal{V} = \mathcal{U} \cup \mathbb{Z} \cup \{\mathsf{true}, \mathsf{false}\} \uplus \{\bot\}$. $\bot$ is a distinguished value representing an undefined or unknown value. Index expressions have values in $\mathbb{Z} \cup \{\mathsf{true}, \mathsf{false}\} \uplus \bot$ while semantic expressions have values in $\mathcal{U} \uplus \{\bot\}$. The evaluation function $[\![e]\!]_{\mathcal{E};\mu}$ is defined in a local environment $\mathcal{E}$ and a memory μ. The stack $\mathcal{E}$ maps variable names to affine

values in $\mathbb{Z} \cup \{\text{true}, \text{false}\}$, and the memory μ maps memory locations to $\mathcal{U} \uplus \{\bot\}$. Memory locations, ranged over by ℓ, are multidimensional array cells, that is, array names indexed by integers:

$$\ell ::= a[n_1, \ldots, n_n]$$

The evaluation function $[\![e]\!]_{\mathcal{E};\mu}$ for expressions is given in Fig. 4.3. If none of the rules apply, the result of the evaluation is the distinguished value $\bot$ representing an error. The use of metavariables v and n in rules imply that the corresponding value is not $\bot$; as such, $\bot$ is propagating: the result of any computation involving $\bot$ is itself $\bot$. The select operator is an eager conditional and evaluates all its arguments before choosing a value depending on the value of the conditional. This follows the design of the Halide compiler; it could easily be replaced with a lazy version instead. A function name f is assumed to evaluate independently of the environment to a function $[\![f]\!]$ that applies to an arbitrary number of arguments and returns $\bot$ if the arity is incorrect.

We also define the evaluation $[\![t]\!]_{\mathcal{E};M}$ of a prophetic expression t in environment $\mathcal{E}$ and model M, where M assigns a semantic to tensors. The evaluation rules for $[\![t]\!]_{\mathcal{E};M}$ are identical to the rules for $[\![e]\!]_{\mathcal{E};\mu}$, except that the rule for array accesses is replaced with a rule for tensor accesses:

$$\frac{[\![\iota_n i]\!]_{\mathcal{E};M} = n_i \in \mathbb{Z} \text{ for all } 1 \leqslant i \leqslant n}{[\![A(\iota_1, \ldots, \iota_n)]\!]_{\mathcal{E};M} = [\![A]\!]_M(n_1, \ldots, n_n)}$$

The question of the semantics to give to SCHED commands needs to be considered carefully. As an intermediate language targeted to compilers of array languages for heterogeneous hardware, one goal of which is the parallelization of programs, it needs some notion of concurrency. Because they need to reason about possible interleavings of concurrent threads, formalizations of concurrent programming languages are usually presented using so-called "small-step semantics" or structural operational semantics. A small-step semantics can be described in terms of a reduction relating a command c and program state σ to a new command c' and new program state σ' obtained after performing one "step" of evaluation, such as performing a single assignment: $\langle x := e\ ;\ c, \sigma\rangle \to \langle c, \sigma[x := [\![e]\!]_\sigma]\rangle$. Small-step semantics are the standard way of reasoning about concurrent programs because they capture interactions between concurrent executions by considering all the possible interleaved

$$\frac{x \mapsto v \in \mathcal{E}}{[\![x]\!]_{\mathcal{E};\mu} = v} \qquad \frac{}{[\![l]\!]_{\mathcal{E};\mu} = l}$$

$$\frac{[\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{N} \text{ for all } 1 \leqslant i \leqslant n \qquad a[n_1, \ldots, n_n] \mapsto v \in \mu}{[\![a[\iota_1, \ldots, \iota_n]]\!]_{\mathcal{E};\mu} = v}$$

$$\frac{x \notin \mathrm{fv}(e) \cup \mathrm{dom}(\mathcal{E}) \qquad [\![\iota]\!]_{\mathcal{E};\mu} = v_0 \qquad [\![e]\!]_{\mathcal{E}+x\mapsto v_0;\mu} = v}{[\![\text{let } x = \iota \text{ in } e]\!]_{\mathcal{E};\mu} = v}$$

$$\frac{[\![\iota_1]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad [\![\iota_2]\!]_{\mathcal{E};\mu} = m \in \mathbb{Z}}{[\![\iota_1 + \iota_2]\!]_{\mathcal{E};\mu} = n + m} \qquad \frac{[\![\iota]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad m \in \mathbb{N}}{[\![m \cdot \iota]\!]_{\mathcal{E};\mu} = m \cdot n}$$

$$\frac{[\![\iota]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad m \in \mathbb{N} \qquad m > 0}{[\![\lfloor \iota / m \rfloor]\!]_{\mathcal{E};\mu} = \lfloor n/m \rfloor} \qquad \frac{[\![\iota]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad m \in \mathbb{N} \qquad m > 0}{[\![\iota \bmod m]\!]_{\mathcal{E};\mu} = n \bmod m}$$

$$\frac{[\![\iota_1]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad [\![\iota_2]\!]_{\mathcal{E};\mu} = m \in \mathbb{Z} \qquad \odot \in \{=, <, \neq, \leqslant\}}{[\![\iota_1 \odot \iota_2]\!]_{\mathcal{E};\mu} = n \odot m}$$

$$\frac{\begin{array}{c}[\![\iota_1]\!]_{\mathcal{E};\mu} = b_1 \in \{\text{true}, \text{false}\} \\ [\![\iota_2]\!]_{\mathcal{E};\mu} = b_2 \in \{\text{true}, \text{false}\} \qquad \odot \in \{\&\&, ||\}\end{array}}{[\![\iota_1 \odot \iota_2]\!]_{\mathcal{E};\mu} = b_1 \odot b_2} \qquad \frac{[\![\iota]\!]_{\mathcal{E};\mu} = b \in \{\text{true}, \text{false}\}}{[\![\,!\,\iota]\!]_{\mathcal{E};\mu} = \neg b}$$

Select-True
$$\frac{[\![\iota]\!]_{\mathcal{E};\mu} = \text{true} \qquad [\![e_1]\!]_{\mathcal{E};\mu} = v_1 \qquad [\![e_2]\!]_{\mathcal{E};\mu} = v_2}{[\![\text{select}(\iota, e_1, e_2)]\!]_{\mathcal{E};\mu} = v_1}$$

Select-False
$$\frac{[\![\iota]\!]_{\mathcal{E};\mu} = \text{false} \qquad [\![e_1]\!]_{\mathcal{E};\mu} = v_1 \qquad [\![e_2]\!]_{\mathcal{E};\mu} = v_2}{[\![\text{select}(\iota, e_1, e_2)]\!]_{\mathcal{E};\mu} = v_2}$$

Call
$$\frac{[\![e_i]\!]_{\mathcal{E};\mu} = v_i \quad \text{for all } 1 \leqslant i \leqslant n}{[\![f(e_1, \ldots, e_n)]\!]_{\mathcal{E};\mu} = [\![f]\!](v_1, \ldots, v_n)}$$

Figure 4.3: Evaluation function for semantic expressions

executions. Furthermore, the semantics of non-terminating programs can be naturally captured by infinite reduction sequences.

Small-step semantics' ability to reason about race conditions and deadlocks is invaluable when proving handwritten code that uses synchronization primitives in subtle ways that can actually exhibit these behaviors. In the context of compiling mostly equational high-level arrays program whose source does not include any synchronization primitives, the resulting code usually does not depend directly on low-level synchronization primitives but rather on higher level constructs abstracting away the low-level primitives. The main interaction a compiler for array languages has with concurrency is through the use of parallel and vectorized loops evaluating all their instructions in a loosely constrained order. Compilers such as Halide or Tensor Comprehension abstract away parallelism through the use of non-communicating parallel loops; the loops are converted into a single outer parallel loop (or to thread and block identifiers on GPUs) with appropriate barriers as a transformation late in the compilation process. This transformation is relatively simple and could be proved correct independently. Effectively, the generated code represents a sequence of parallel stages represented by parallel loops, with a single unconditional global barrier between the stages, and without additional synchronization primitives within the stages. This allows communication between concurrent execution units as long as the communication does not occur within the execution of the same parallel loop, encompassing communication techniques based on message passing [21]. One notable exception is Halide's `async()` scheduling primitive that generate dependent threads communicating through a queue within the same parallel loop. Furthermore, the input languages to tensor compilers typically lack both recursion and unbounded while loops, making programs in those languages necessarily terminating and the languages themselves not Turing-complete by design. As such, because the code generated by tensor compiler is always terminating and uses parallelism rather than true concurrency, the use of small-step semantics in this context appears less necessary than is usually the case for concurrent programs.

On the other hand, so-called "big-step semantics" or natural semantics relate a command c and a program state σ to a final program state σ' obtained after executing the whole command. Whereas small-step semantics perform the computation piece by piece, making a bit of progress each time until no work remain, big-step semantics directly evaluate a program to its final result in one go. Big-step semantics abstract away many details of the computation and its

order. This is attractive for symbolic evaluation since a symbolic evaluator does not have to explore the many possible interleavings of small-step reductions. At the same time, the requirement of fully evaluating a program to a value in a big-step semantics make them awkward to use for nonterminating computations, often requiring the use of a separate semantics for nonterminating programs. In addition, in a big-step semantics, information about the computation order is hidden within the derivation tree, making them difficult if not impossible to use in the context of concurrent programs where reasoning about interleavings of concurrent executions interacting through shared communication channels is necessary. As was mentioned above, however, in the context of a tensor compiler, these usual weaknesses of big-step semantics do not apply, while a big-step semantics is attractive as the basis of a symbolic evaluator.

We want to build a big-step semantics for Sched. To make things concrete, let us assume we have a big-step judgement $\mu \vdash c \Downarrow \mu'$ that evaluates a command c in memory μ, resulting in a new memory μ'. The rules for the evaluation of assignment, sequential composition, and sequential loops can be written relatively easily and compose well; for instance, the rule for sequential composition can be written as:

$$\frac{\mu \vdash c_1 \Downarrow \mu_1 \qquad \mu_1 \vdash c_2 \Downarrow \mu_2}{\mu \vdash c_1 \; ; \; c_2 \Downarrow \mu_2}$$

One problem remain: it is still unclear how to capture the semantics of parallel loops using this big-step semantics. Recall that we are interested in synchronization-free loops, i.e. we assume that the only synchronization mechanism is a global barrier at the end of the execution of the loop that waits for all iterations of the loop to finish before continuing. Since this implies that distinct iterations of the loop cannot depend on their relative execution order in any way, the first idea would be to execute the body of the loop in an arbitrary order, and require that all orders evaluate to the same final memory. However, this is too coarse, as is shown by the following example:

Active waiting schemes could be used to implement synchronization mechanism on top of simple memory writes and reads, but our goal here is to prevent such "bogus" programs.

```
par i = 0 to 1 do
  if (i == 0) {
    y[] := x[] ;
    x[] := 0 ;
    x[] := y[] ;
  }
  if (i == 1) {
    z[] := x[] ;
  }
done
```

Any execution of the program for one value of `i` followed by the other would yield a final memory where all three arrays `x`, `y` and `z` have the same value, but an interleaving semantics would allow a final state with `z[] = 0` instead. The issue here is the presence of a *data race*: both threads access to the array `x`, and the first thread writes to it without consideration for the second thread's read. Because data races lead to nondeterminism, and the original program that was compiled is always deterministic, the introduction of this nondeterminism can be considered a compiler bug. Instead of trying to capture the semantics of data races, we will prevent them: programs that exhibit data races should not have semantics, because data races can only be introduced by a compiler bug.

In practice, tensor compilers such as Halide can allow a very specific kind of data races, sometimes called "benign" data races. A benign data race is a race where multiple threads write concurrently to the same location, but they all write the same value. On all the hardware targeted by Halide, and all existing hardware I know of, the location will always end up containing that value on the next synchronization. Such benign races are exploited by Halide to shift the start of the last tile of a tiled loop, ensuring that all tiles are full (an example is given in chapter 1). In turn, this enables further optimizations, such as using vectorized instructions even for the last tile.

To prevent data races, our semantics must capture the set of memory locations that are read and written during the execution of the program. To allow benign data races as used by Halide, we must further capture all the possible values written to each location. Hence, the result of the evaluation of a command c should not only contain the final memory μ' but also a set ρ of read locations and a mapping ω from locations to a set of values written to that location. We can enforce the absence of races between a read access and a write access by

requiring the read-set ρ_i and the write-set $\mathrm{dom}(\omega_j)$ to be disjoint when i and j are distinct threads. Further, we can enforce the absence of non-benign races between write accesses by requiring that all possible values written by two distinct threads i and j to the same location ℓ are identical (i.e. for all $v_i \in \omega_i(\ell)$ and $v_j \in \omega_j(\ell)$, we must have $v_i = v_j$).

Because read-write races are forbidden, it is not possible for a thread to see the writes performed by another thread, and we can define a big-step semantics by evaluating each iteration of the loop independently in the initial memory at the start of the loop. However, if we do so, it is not clear what the final memory should be after evaluating the parallel loop, i.e. it is not clear how to recombine the memories obtained by evaluating each iteration independently. Consider the following pair of programs:

```
x[] := 0 ;
par i = 0 to 1 do
  if (i == 1)
    x[] := i ;
done
```

```
x[] := 1 ;
par i = 0 to 1 do
  if (i == 0)
    x[] := i ;
done
```

In both cases, after evaluating each iteration of the loop independently, we would get one thread where $x[] \mapsto 0$ and another thread where $x[] \mapsto 1$, and we need to somehow reconcile the values. To do so, we must either keep the value that is different from the initial value before evaluating the loop, or examine the sets of written locations in both cases to determine which thread writes to `x[]` in each case. None of these solutions seem very satisfactory, and we can instead think of a third solution: instead of capturing the final memory after executing the command, we can make our big-step semantics return a *differential memory* $\delta\mu$, a partial mapping that contains the last value written to each location. If a location is not written at all, it should have no associated value in the differential memory. Using such a formulation of the semantics, it is easy to recombine the evaluation of all the iterations in a parallel loop: we can simply take the union of the differential memories, since the absence of non-benign races ensures that the union has at most one unique value per memory location. By returning a differential memory instead of a "complete" memory as the result of the evaluation, the result of an evaluation only depends on the locations that are actually touched during said evaluation, giving natural framing properties to the semantics. In particular, this will make

it possible to break the self-dependent loop between iterations of sequential loops in chapter 5, because we can write a symbolic evaluator for annotated programs that is independent of the memory it is executed in.

Now that we have motivated the need for differential memories, the attentive reader may notice some duplication of purpose with the set of reads ω. The difference is indeed small: where ω maps each location to the set of all values written to it during the execution of the program, $\delta\mu$ maps each location to the *last* of those values. Could we completely get rid of ω and only use $\delta\mu$ instead? The answer depends on the amount of slack we allow ourselves regarding the kind of races that should be deemed acceptable. Clearly, read-write races are inacceptable, and $\delta\mu$ can be used instead of ω to prevent those since they have the same domains. We could also prevent all write-write reads using $\delta\mu$ by requiring that the $\delta\mu$ for different threads must be disjoint, but we want to allow benign races, because compilers such as Halide rely on them. Unfortunately, it is not possible to allow benign races while preventing all non-benign races using only $\delta\mu$. Consider the following pair of programs:

```
par i = 0 to 1 do
  x[] := i ;
  x[] := 7 ;
done
```

```
par i = 0 to 1 do
  x[] := 7 ;
done
```

In both cases, the last value written by the loop body to location `x[]` is 7, and hence they would have the same $\delta\mu$; for the program on the left, distinct intermediate values are written by each thread, which is captured by ω. The program on the left has non-benign data races (there are races between the writes of 0, 1 and 7 to `x[]`), while the program on the right has only benign data races. However, the non-benign data races for the left program are, in some sense, "covered" by a write of the same value, 7 — hence, even though there can be data races during the execution of the program, the final value is deterministic: once the parallel loop has finished executing, the value in location `x[]` is always 7. This is true if all writes are atomic: in this case, we can order all the writes depending on the time they are committed to the main memory for `x[]`, and the last write to `x[]` is necessarily the last write to `x[]` by *some* thread, and hence necessarily a write of 7. Even if the writes to `x[]` are not atomic, we can consider them as a combination of atomic writes to components

of `x[]`, and the same reasoning applies for each of the atomic components of `x[]`.

In the end, there seems to be no substantial difference between allowing only benign data races (which requires ω) and also allowing data races that are "covered" by an identical write (which can be expressed using $\delta\mu$ only). A compiler assuming that no data race occurs can of course perform incorrect program transformations if it assumes that no data race occurs, which may be thought less likely if only benign races are possible. Unfortunately, even only in the presence of benign races, compilers can perform incorrect transformations changing program meaning, as described by Boehm [20]. Since this is not an issue I aim to solve in this thesis, and in order to simplify the presentation here, I only include the computation of $\delta\mu$ in the evaluation rules below, omitting ω completely.

To express the evaluation of a Sched command, we must define the set of locations read during the evaluation of an expression, written $\mathrm{rd}_{\mathcal{E};\mu}(e)$, and defined in Fig. 4.4. The definition of $\mathrm{rd}_{\mathcal{E};\mu}(e)$ depends on the evaluation function $[\![\cdot]\!]_{\mathcal{E};\mu}$. If any evaluation fails during the computation of $\mathrm{rd}_{\mathcal{E};\mu}(e)$ (i.e. no rule applies, for instance because we are evaluating a memory location that has no associated value), the result is undefined. When we write $\mathrm{rd}_{\mathcal{E};\mu}(e) = \rho$, we implicitly assume that the result is defined.

The set of locations read by an expression e is always defined in any environment where the evaluation of e succeeds; moreover, whenever the set of locations read is defined, it captures the locations that influence the evaluation of the expression.

Lemma 4.2.1. *If an expression e evaluates to a value v in environment $\mathcal{E}$ and memory μ, then $\mathrm{rd}_{\mathcal{E};\mu}(e)$ is defined and equal to a set of locations.*

Proof. The proof is immediate by induction on the structure of e after generalizing over both $\mathcal{E}$ and μ. □

To express that the set of read locations captures the locations influencing the evaluation, we first define what it means for memories to agree on a set of locations.

$$\frac{}{\mathrm{rd}_{\mathcal{E};\mu}(x) = \emptyset} \qquad \frac{}{\mathrm{rd}_{\mathcal{E};\mu}(l) = \emptyset}$$

$$\frac{[\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{N} \text{ for all } 1 \leqslant i \leqslant n \qquad \mathrm{rd}_{\mathcal{E};\mu}(\iota_i) = \rho_i \quad \text{for all } 1 \leqslant i \leqslant n}{\mathrm{rd}_{\mathcal{E};\mu}(a[\iota_1, \ldots, \iota_n]) = \{a[n_1, \ldots, n_n]\} \cup \bigcup_{1 \leqslant i \leqslant n} \rho_i}$$

$$\frac{\begin{array}{c} x \notin \mathrm{fv}(e) \cup \mathrm{dom}(\mathcal{E}) \\ [\![\iota]\!]_{\mathcal{E};\mu} = v_0 \qquad \mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho_1 \qquad \mathrm{rd}_{\mathcal{E}[x \mapsto v_0];\mu}(e) = \rho_2 \end{array}}{\mathrm{rd}_{\mathcal{E};\mu}(\text{let } x = \iota \text{ in } e) = \rho_1 \cup \rho_2}$$

$$\frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho \qquad n \in \mathbb{Z}}{\mathrm{rd}_{\mathcal{E};\mu}(n \cdot \iota) = \rho} \qquad \frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho \qquad n > 0}{\mathrm{rd}_{\mathcal{E};\mu}(\lfloor \iota / n \rfloor) = \rho} \qquad \frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho \qquad n > 0}{\mathrm{rd}_{\mathcal{E};\mu}(\iota \bmod n) = \rho}$$

$$\frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota_1) = \rho_1 \qquad \mathrm{rd}_{\mathcal{E};\mu}(\iota_2) = \rho_2 \qquad \odot \in \{+, =, <, \neq, \leqslant, \&\&, ||\}}{\mathrm{rd}_{\mathcal{E};\mu}(\iota_1 \odot \iota_2) = \rho_1 \cup \rho_2}$$

$$\frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho}{\mathrm{rd}_{\mathcal{E};\mu}(!\,\iota) = \rho} \qquad \frac{\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \rho \qquad \mathrm{rd}_{\mathcal{E};\mu}(e_1) = \rho_1 \qquad \mathrm{rd}_{\mathcal{E};\mu}(e_2) = \rho_2}{\mathrm{rd}_{\mathcal{E};\mu}(\mathrm{select}(\iota, e_1, e_2)) = \rho \cup \rho_1 \cup \rho_2}$$

$$\frac{\mathrm{rd}_{\mathcal{E};\mu}(e_i) = \rho_i \text{ for all } 1 \leqslant i \leqslant n}{\mathrm{rd}_{\mathcal{E};\mu}(f(e_1, \ldots, e_n)) = \rho_1 \cup \cdots \cup \rho_n}$$

Figure 4.4: Read locations for semantics expressions

Definition 4.2.1. We say that memory μ is *compatible with memory* μ' *over the set of locations* ρ, or that μ *agrees with* μ' *on* ρ, and we write $\mu(\rho) = \mu'(\rho)$, if the locations in ρ are mapped to the same value in both μ and μ':

$$\forall \ell \in \rho,\ \mu(\ell) = \mu'(\ell)$$

The compatibility over a set of location ρ is an equivalence relation; moreover, two memories are compatible on a union if, and only if, they are compatible on both components of the union. In particular, two memories compatible on a set of locations are compatible on any subset thereof.

We can now state that the evaluation of an expression only depends on the locations it reads:

Theorem 4.2.2. *If the set of read locations* $\mathrm{rd}_{\mathcal{E};\mu}(e)$ *is defined, then for any memory* μ' *compatible with* μ *over* $\mathrm{rd}_{\mathcal{E};\mu}(e)$ *we have* $\mathrm{rd}_{\mathcal{E};\mu'}(e) = \mathrm{rd}_{\mathcal{E};\mu}(e)$ *and* $[\![e]\!]_{\mathcal{E};\mu'} = [\![e]\!]_{\mathcal{E};\mu}$.

Remark 4.2.1. Theorem 4.2.2 always apply when μ' is an extension of μ, since it is necessarily compatible with μ over $\mathrm{rd}_{\mathcal{E};\mu}(e) \subseteq \mathrm{dom}(\mu)$.

Note that $[\![e]\!]_{\mathcal{E};\mu'}$ and $[\![e]\!]_{\mathcal{E};\mu}$ can be both equal to $\bot$ if there is a non-memory-related error such as evaluating $1 + \mathrm{true}$.

Proof of Theorem 4.2.2. By induction on the structure of e, after generalizing over $\mathcal{E}$ and μ:

Case $e = x$ We always have $\mathrm{rd}_{\mathcal{E};\mu'}(x) = \emptyset = \mathrm{rd}_{\mathcal{E};\mu}(x)$ and $[\![x]\!]_{\mathcal{E};\mu}$ does not depend on μ.

Case $e = a[\iota_1, \ldots, \iota_n]$ $\mathrm{rd}_{\mathcal{E};\mu}(a[\iota_1, \ldots, \iota_n])$ is defined, hence for any $1 \leqslant i \leqslant n$, $[\![\iota_i]\!]_{\mathcal{E};\mu} = n_i$ is an integer.

By definition, $\mathrm{rd}_{\mathcal{E};\mu}(\iota_i) \subseteq \mathrm{rd}_{\mathcal{E};\mu}(a[\iota_1, \ldots, \iota_n])$ hence by induction hypothesis $\mathrm{rd}_{\mathcal{E};\mu'}(\iota_i) = \mathrm{rd}_{\mathcal{E};\mu}(\iota_i)$ and $[\![\iota_i]\!]_{\mathcal{E};\mu'} = [\![\iota_i]\!]_{\mathcal{E};\mu} = n_i$ for any memory μ' agreeing with μ on $\mathrm{rd}_{\mathcal{E};\mu}(a[\iota_1, \ldots, \iota_n])$.

Moreover, $a[n_1, \ldots, n_n]$ is in $\mathrm{rd}_{\mathcal{E};\mu}(a[\iota_1, \ldots, \iota_n])$ over which μ and μ' agree, hence either $a[n_1, \ldots, n_n]$ is not present in both μ and μ' or it maps to the same value in both, from which we conclude.

Case $e = \mathrm{let}\ x = \iota\ \mathrm{in}\ e'$ By induction hypothesis, we have:

$$\mathrm{rd}_{\mathcal{E};\mu'}(\iota) = \mathrm{rd}_{\mathcal{E};\mu}(\iota)$$
$$[\![\iota]\!]_{\mathcal{E};\mu'} = [\![\iota]\!]_{\mathcal{E};\mu}$$

for any memory μ' agreeing with μ on $\mathrm{rd}_{\mathcal{E};\mu}(e)$. Hence, we can apply the induction hypothesis on e' in environment $\mathcal{E} + x \mapsto [\![\iota]\!]_{\mathcal{E};\mu}$ to conclude.

The other cases are simple applications of the induction hypothesis. □

If no locations are read during the evaluation of e in $\mathcal{E}$ and μ (i.e. $\mathrm{rd}_{\mathcal{E};\mu}(e) = \emptyset$), then the evaluation of e does not depend on μ, i.e. for any memory μ' $[\![e]\!]_{\mathcal{E};\mu'} = [\![e]\!]_{\mathcal{E};\mu}$; moreover, we also have $\mathrm{rd}_{\mathcal{E};\mu'}(e) = \emptyset$ and $[\![e]\!]_{\mathcal{E};\mathcal{M}} = [\![e]\!]_{\mathcal{E};\mu}$ for any model $\mathcal{M}$.

Definition 4.2.2. If $\mathrm{rd}_{\mathcal{E};\emptyset}(e)$ is defined and equal to the empty set (e.g. if e does not contain any array read), we define $[\![e]\!]_{\mathcal{E}} = [\![e]\!]_{\mathcal{E};\emptyset}$ and we have $[\![e]\!]_{\mathcal{E}} = [\![e]\!]_{\mathcal{E};\mu}$ for any memory μ.

Lemma 4.2.3. *If* $\mathrm{rd}_{\mathcal{E};\mu}(\iota) = \emptyset$ *for some memory* μ *then for any environment* $\mathcal{E} \vDash \Gamma$, *memory* μ' *and model* $\mathcal{M}$, *we have* $[\![\iota]\!]_{\mathcal{E};\mu'} = [\![\iota]\!]_{\mathcal{E};\mathcal{M}} = [\![e]\!]_{\mathcal{E}}$.

We are now finally able to define the evaluation of a command c in an environment $\langle \mathcal{E}; \mu \rangle$ into a pair $\langle \delta\mu; \rho \rangle$ of a differential memory and a read-set. This evaluation is written $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$ and is defined through inference rules in Fig. 4.5. $\delta\mu$ is a partial mapping from locations to values representing the updates performed by the evaluation, and ρ is the set of locations that were read during the evaluation.

The update operator $\mu_1 \triangleright \mu_2$, read "$\mu_1$ then μ_2", where μ_1 and μ_2 are mappings, represents the mappings of μ_2 and those of μ_1 not overwritten by μ_2. The

U-SKIP
$$\frac{}{\mathcal{E};\mu \vdash \mathsf{skip} \Downarrow_u \langle \emptyset;\emptyset\rangle}$$

U-SEQ
$$\frac{\mathcal{E};\mu \vdash c_1 \Downarrow_u \langle \delta\mu_1;\rho_1\rangle \qquad \mathcal{E};\mu \triangleright \delta\mu_1 \vdash c_2 \Downarrow_u \langle \delta\mu_2;\rho_2\rangle}{\mathcal{E};\mu \vdash c_1 \,;\, c_2 \Downarrow_u \langle \delta\mu_1 \triangleright \delta\mu_2;\rho_1 \cup \rho_2\rangle}$$

U-IF-TRUE
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = \mathsf{true} \qquad \mathcal{E};\mu \vdash c_1 \Downarrow_u \langle \delta\mu;\rho\rangle}{\mathcal{E};\mu \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Downarrow_u \langle \delta\mu;\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \rho\rangle}$$

U-IF-FALSE
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = \mathsf{false} \qquad \mathcal{E};\mu \vdash c_2 \Downarrow_u \langle \delta\mu;\rho\rangle}{\mathcal{E};\mu \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \Downarrow_u \langle \delta\mu;\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \rho\rangle}$$

U-LET
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = v \qquad \mathcal{E} + x \mapsto v;\mu \vdash c \Downarrow_u \langle \delta\mu;\rho\rangle}{\mathcal{E};\mu \vdash \mathsf{let}\ x = e\ \mathsf{in}\ c \Downarrow_u \langle \delta\mu;\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \rho\rangle}$$

U-FOR
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad \forall 0 \leqslant i < n,\ \mathcal{E} + x \mapsto i;\mu \triangleright \bigtriangleup_{0\leqslant j<i} \delta\mu_j \vdash c \Downarrow_u \langle \delta\mu_i;\rho_i\rangle}{\mathcal{E};\mu \vdash \mathsf{for}\ x < e;\mathsf{do}\ c \Downarrow_u \langle \bigtriangleup_{0\leqslant i<n} \delta\mu_i;\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \bigcup_{0\leqslant i<n} \rangle\rho_i}$$

U-PARLOOP
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad \forall 0 \leqslant i < n,\ \mathcal{E} + x \mapsto i;\mu \vdash c \Downarrow_u \langle \delta\mu_i;\rho_i\rangle \qquad \forall 0 \leqslant i \neq j < n,\ \delta\mu_i \supset \delta\mu_j \qquad \forall 0 \leqslant i \neq j < n,\ \mathrm{dom}(\delta\mu_i) \# \rho_j}{\mathcal{E};\mu \vdash \mathsf{par}\ x < e;\mathsf{do}\ c \Downarrow_u \langle \bigcup_{0\leqslant i<n} \delta\mu_i;\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \bigcup_{0\leqslant i<n} \rho_i\rangle}$$

U-ASSIGN
$$\frac{[\![e]\!]_{\mathcal{E};\mu} = v \qquad \forall 1 \leqslant i \leqslant n,\ [\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{Z} \qquad \ell = a[n_1,\ldots,n_n] \in \mathrm{dom}(\mu)}{\mathcal{E};\mu \vdash a[\iota_1,\ldots,\iota_n]\ \{t\} := e \Downarrow_u \langle \{\ell \mapsto v\};\mathrm{rd}_{\mathcal{E};\mu}(e) \cup \bigcup_{1\leqslant i\leqslant n} \mathrm{rd}_{\mathcal{E};\mu}(\iota_i)\rangle}$$

U-ALLOCATE
$$\frac{\mu_a = \{a[i_1,\ldots,i_n] \mapsto \bot \mid 0 \leqslant i_1 < n_1 \wedge \cdots \wedge 0 \leqslant i_n < n_n\} \qquad \forall 1 \leqslant i \leqslant n,\ [\![e_i]\!]_{\mathcal{E};\mu} = n_i \qquad \mathcal{E};(\mu \backslash a) \triangleright \mu_a \vdash c \Downarrow_u \langle \delta\mu;\rho\rangle}{\mathcal{E};\mu \vdash \mathsf{allocate}\ a : \tau[e_1,\ldots,e_n]\ \mathsf{in}\ c \Downarrow_u \langle \delta\mu \backslash \mathrm{dom}(\mu_a);\rho \backslash \mathrm{dom}(\mu_a)\rangle}$$

Figure 4.5: Update semantics for SCHED statements

intuition for $\triangleright$ can be understood through the rule U-SEQ, and it can be defined as $\mu_1 \triangleright \mu_2 = (\mu_1 \setminus \mathrm{dom}(\mu_2)) \uplus \mu_2$, where $\uplus$ denotes the union of disjoint mappings. It can also be defined extensionally:

$$\mu_1 \triangleright \mu_2(\ell) = \begin{cases} \mu_2(\ell) & \text{if } \ell \in \mathrm{dom}(\mu_2) \\ \mu_1(\ell) & \text{otherwise} \end{cases}$$

$\triangleright$ is associative, hence, we can define the iterated update $\blacktriangleright_{0 \leqslant i < n} \mu_i = \mu_0 \triangleright \ldots \triangleright \mu_{n-1}$. The iterated update is used to specify sequential loops in rule U-FOR.

Memories are expected to contain a value for each accessible memory location, some of which may be potentially undefined (i.e. contain $\perp$). This is respected by rule U-ALLOCATE that introduces a fresh local array and initializes its domain with $\perp$. Local arrays are properly scoped: the notation $\mu \setminus a$ in rule U-ALLOCATE indicates that we remove all locations associated with array a from μ before evaluating c. This ensures that array names are properly shadowed (i.e. if there was already an array named a, it can no longer be accessed within an allocate a block). Note that local arrays allocated in different threads do not alias, and local arrays allocated outside a parallel loop can be accessed by each thread. To ensure scoping, writes and reads to the local array are erased after returning from the inner scope of the allocate statement, ensuring that the values are no longer accessible. The assumption that a value is present for each accessible memory location is exploited by rule U-ASSIGN to make out-of-bounds writes a runtime error. In fact, we can show using a trivial induction that both reads and writes are only performed within the domain of μ:

Remark 4.2.2. If c evaluates to $\langle \delta\mu; \rho \rangle$ in environment $\mathcal{E}$ and memory μ, then $\mathrm{dom}(\delta\mu) \subseteq \mathrm{dom}(\mu)$ and $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\mu)$.

Conversely, the program c cannot distinguish between sufficiently big memories that agree on the set of reads:

Lemma 4.2.4. *If c evaluates to $\langle \delta\mu; \rho \rangle$ in environment $\mathcal{E}$ and memory μ, then it also evaluates to $\langle \delta\mu; \rho \rangle$ in any memory μ' that contains $\mathrm{dom}(\delta\mu)$ and agrees with μ on ρ.*

Proof. Under the conditions of the lemma, by induction on the judgement $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$:

U-Skip $\mathcal{E}; \mu' \vdash \text{skip} \Downarrow_u \langle \emptyset; \emptyset \rangle$ holds for any memory μ'.

U-Seq We have $\mathcal{E}; \mu \vdash c_1 \Downarrow_u \langle \delta\mu_1; \rho_1 \rangle$ and $\mu' \restriction_{\rho_1 \cup \rho_2} = \mu \restriction_{\rho_1 \cup \rho_2}$ hence in particular, $\mu' \restriction_{\rho_1} = \mu \restriction_{\rho_1}$ and, by induction hypothesis, we have $\mathcal{E}; \mu' \vdash c_1 \Downarrow_u \langle \delta\mu_1; \rho_1 \rangle$.

Moreover, we have $\mathcal{E}; \mu \triangleright \delta\mu_1 \vdash c_2 \Downarrow_u \langle \delta\mu_2; \rho_2 \rangle$ and we must prove that $(\mu' \triangleright \delta\mu_1) \restriction_{\rho_2} = (\mu \triangleright \delta\mu_1) \restriction_{\rho_2}$ to apply the induction hypothesis again and conclude. The equality holds because if we consider a location in ρ_2, either it is in $\delta\mu_1$ and the result from $\delta\mu_1$ is used in both sides, or it is not, and thus the results in μ' and μ are used, respectively — but by hypothesis we had $\mu' \restriction_{\rho_1 \cup \rho_2} = \mu \restriction_{\rho_1 \cup \rho_2}$, and we conclude.

U-For By Theorem 4.2.2, the evaluation of e in μ and μ' is identical. By iteration, for each $0 \leqslant i < n$, $\mu \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j$ and $\mu' \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j$ agree on $\bigcup_{0 \leqslant j < n} \rho_j$, hence in particular on ρ_i, and we get $\mathcal{E}; \mu' \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j \vdash c \Downarrow_u \langle \delta\mu_i; \rho_i \rangle$. We conclude using Theorem 4.2.2.

U-Par By Theorem 4.2.2, the evaluation of e in μ and μ' is identical. By induction hypothesis, we have $\mathcal{E} + x \mapsto i; \mu' \vdash c \Downarrow_u \langle \delta\mu_i; \rho_i \rangle$ for each $0 \leqslant i < n$ and we conclude using Theorem 4.2.2.

The other cases (U-If-True, U-If-False, U-Let, U-Assign and U-Allocate) follow immediately from the induction hypothesis and Theorem 4.2.2. □

The absence of races is enforced by rule U-ParLoop. The condition

$$\forall 0 \leqslant i \neq j < n,\ \mathrm{dom}(\delta\mu_i) \# \rho_j$$

ensures the absence of read-write races using the disjointness operator #, defined as follows.

Definition 4.2.3. Two sets S_1 and S_2 are *disjoint*, written $S_1 \# S_2$, if they have no elements in common.

$$S_1 \# S_2 := S_1 \cap S_2 = \emptyset$$

By abuse of notation, we say that a partial memory μ and a set ρ are disjoint, and write $\mu \# \rho$, if ρ and the domain of μ are disjoint, i.e. when $\mathrm{dom}(\mu) \# \rho$.

Similarly, we say that two partial memories μ_1 and μ_2 are disjoint, and write $\mu_1 \# \mu_2$, when their domains are disjoint, i.e. when $\mathrm{dom}(\mu_1) \# \mathrm{dom}(\mu_2)$.

The condition

$$\forall 0 \leqslant i \neq j < n,\ \delta\mu_i \supset \delta\mu_j \tag{4.1}$$

ensures that all write-write races are covered by a benign race, as explained earlier. This condition uses the compatibility operator, defined as follows:

Definition 4.2.4. We say that two partial memories μ and μ' are *compatible*, denoted $\mu \supset \mu'$, if they agree on their shared domain, i.e. for any location ℓ associated to a value v in μ and to a value v' in μ', v and v' are equal.

Remark 4.2.3. Two partial memories μ and μ' are compatible if, and only if, their respective updates commute:

$$\mu \supset \mu' \Leftrightarrow \mu \triangleright \mu' = \mu' \triangleright \mu$$

It is possible to replace condition Eq. (4.1) with a disjointness condition

$$\forall 0 \leqslant i \neq j < n,\ \mathrm{dom}(\delta\mu_i) \# \mathrm{dom}(\delta\mu_j)$$

in order to ensure the absence of all data races, however, as mentioned above, languages such as Halide do generate code with benign data races that we want to verify, hence the use of the more complex compatibility condition. If the semantics is extended with the write-set ω recording *all* the values written to a given location (ω follows the same rules as $\delta\mu$, except that an the union of the associated sets must be taken whenever $\triangleright$ is used), the condition can also be replaced with a compatibility condition on ω instead.

4.3 Soundness

To justify the use of the big-step semantics presented in the previous section, and increase confidence that it properly captures the expected behavior of programs

in SCHED, it is worthwhile to prove it sound with respect to a more traditional small-step concurrent semantics with interleaved executions allowing data races.

As discussed in the previous section, the definition of a proper small-step semantics must be carefully considered. Formulation of concurrent small-steps semantics usually rely on mechanisms to represent a *single* pool of concurrent execution units, akin to the threads in a processor or the processors in a distributed system. Memory is split between a local memory for each thread and a global memory that can be used by multiple threads to communicate. Most transition rules are local to the thread, but synchronization primitives and accesses to the global memory require conditions on the other threads to make progress. On the other hand, the parallel loops introduced in SCHED can introduce arbitrarily nested "levels" of parallelism and a sequence of local memories; for instance, the semantics of the following program:

```
par i = 0 to N do
  allocate b[M] in
  par j = 0 to M do
    b[j] := a[i, j] ;
  par j = 0 to M do
    allocate c[] in
    c[] := b[M - j] ;
    d[i, j] := c[] * c[] ;
```

requires the introduction of two nested parallel levels with their own local memories: there are $N \times M$ versions of the zero-dimensional array `c` and N versions of the array `b`. A proper semantics would take into account this nesting property of the language, and could be modelled along the lines of concurrent languages for GPU programming [51, 63] that also deal with hierarchical memory structures.

In order to represent a small-step semantics for the SCHED language, we introduce additional constructs to represent intermediate computation stages that are not needed for the big-step semantics: the parallel composition operator $c_1 \parallel c_2$ that is used to expand parallel loops, and the inalloc μ_a do c construct used to represent computation within an allocate block. inalloc takes as argument a partial memory μ_a representing the current state of the memory for the locally allocated array. By explicitly storing the local memory in the

command, this can properly express the semantics of locally allocated arrays in the presence of parallel composition.

The small step reduction rules $\langle c \mid \mu \rangle \rightsquigarrow \langle c' \mid \mu' \rangle$ are given in Figure 4.6. These reduction rules are similar to the reduction rules of Reinking, Bernstein, and Ragan-Kelley [89], with two differences: compared to ours, semantics of Reinking, Bernstein, and Ragan-Kelley [89] uses an explicit grammar for contexts instead of the *-CTX rules, and uses an imperative store instead of substitutions. Since the imperative store is never written to except when entering a loop, those differences are essentially stylistic. Reinking's semantics also uses a different formulation of memories where each array name is associated with a partial function on integer tuples.

Rule ASSIGN can only write to a location that is already present in the memory, hence the small step semantics never introduce new locations:

Lemma 4.3.1. *If $\langle c \mid \mu \rangle$ reduces to $\langle c' \mid \mu' \rangle$, then μ' and μ have the same domain.*

In order to focus on the interesting part (i.e. the array language and the constraints in rule U-PARLOOP that are meant to ensure the absence of races), the reduction rules are expressed using capture-avoiding substitutions (whose standard rules are not reproduced here) rather than an explicit environment. Relating the small-step to the big-step semantics (that does use an explicit environment) requires the following technical lemma, whose proof is omitted here (the general case is required for the induction):

Lemma 4.3.2. *If $\mathcal{E}, \mathcal{E}'$ denotes the concatenation of environments $\mathcal{E}$ and $\mathcal{E}'$, and $e[\mathcal{E}]$ (resp. $c[\mathcal{E}]$) denotes the application of $\mathcal{E}$ as a capture-avoiding substitution to e (resp. to c), then the following holds:*

$$[\![e]\!]_{\mathcal{E},\mathcal{E}';\mu} = [\![e[\mathcal{E}]]\!]_{\mathcal{E}';\mu}$$

and

$$\mathcal{E}, \mathcal{E}'; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle \Leftrightarrow \mathcal{E}'; \mu \vdash c[\mathcal{E}] \Downarrow_u \langle \delta\mu; \rho \rangle$$

In particular, when $\mathcal{E}' = \emptyset$, we have:

$$[\![e]\!]_{\mathcal{E};\mu} = [\![e[\mathcal{E}]]\!]_{\emptyset;\mu}$$

Seq-Ctx
$$\frac{\langle c_1 \mid \mu\rangle \rightsquigarrow \langle c_1' \mid \mu'\rangle}{\langle c_1 \,;\, c_2 \mid \mu\rangle \rightsquigarrow \langle c_1' \,;\, c_2 \mid \mu'\rangle}$$

Seq-Skip
$$\frac{}{\langle \text{skip} \,;\, c \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

Par-L
$$\frac{\langle c_1 \mid \mu\rangle \rightsquigarrow \langle c_1' \mid \mu'\rangle}{\langle c_1 \parallel c_2 \mid \mu\rangle \rightsquigarrow \langle c_1' \parallel c_2 \mid \mu'\rangle}$$

Par-R
$$\frac{\langle c_2 \mid \mu\rangle \rightsquigarrow \langle c_2' \mid \mu'\rangle}{\langle c_1 \parallel c_2 \mid \mu\rangle \rightsquigarrow \langle c_1 \parallel c_2' \mid \mu'\rangle}$$

Par-Skip-R
$$\frac{}{\langle c \parallel \text{skip} \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

Par-Skip-L
$$\frac{}{\langle \text{skip} \parallel c \mid \mu\rangle \rightsquigarrow \langle c \mid \mu\rangle}$$

If-True
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = \text{true}}{\langle \text{if } \iota \text{ then } c_1 \text{ else } c_2 \mid \mu\rangle \rightsquigarrow \langle c_1 \mid \mu\rangle}$$

If-False
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = \text{false}}{\langle \text{if } \iota \text{ then } c_1 \text{ else } c_2 \mid \mu\rangle \rightsquigarrow \langle c_2 \mid \mu\rangle}$$

Let
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = v}{\langle \text{let } x = \iota \text{ in } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow v] \mid \mu\rangle}$$

SeqLoop
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = n \in \mathbb{Z}}{\langle \text{for } x < \iota; \text{do } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow 0] \,;\, \ldots \,;\, x[c \leftarrow n-1] \mid \mu\rangle}$$

ParLoop
$$\frac{[\![\iota]\!]_{\emptyset;\mu} = n \in \mathbb{Z}}{\langle \text{par } x < \iota; \text{do } c \mid \mu\rangle \rightsquigarrow \langle c[x \leftarrow 0] \parallel \ldots \parallel x[c \leftarrow n-1] \mid \mu\rangle}$$

Assign
$$\frac{[\![\iota_i]\!]_{\emptyset;\mu} = n_i \text{ for all } 1 \leqslant i \leqslant n \qquad [\![e]\!]_{\emptyset;\mu} = v \qquad a[n_1, \ldots, n_n] \in \text{dom}(\mu)}{\langle a[\iota_1, \ldots, \iota_n]\ \{t\} := e \mid \mu\rangle \rightsquigarrow \langle \text{skip} \mid \mu[a[n_1, \ldots, n_n] \leftarrow v]\rangle}$$

Allocate
$$\frac{\begin{array}{c}[\![\iota_i]\!]_{\emptyset;\mu} = n_i \text{ for all } 1 \leqslant i \leqslant n \\ \mu_a = \{a[i_1, \ldots, i_n] \mapsto \bot \mid 0 \leqslant i_1 < n_1 \wedge \cdots \wedge 0 \leqslant i_n < n_n\} \\ a \notin \text{arrays}(\mu)\end{array}}{\langle \text{allocate } a : \tau[\iota_1 \times \cdots \times \iota_n] \text{ in } c \mid \mu\rangle \rightsquigarrow \langle \text{inalloc } \mu_a \text{ do } c \mid \mu\rangle}$$

InAlloc-Ctx
$$\frac{\begin{array}{c}\text{dom}(\mu) \# \text{dom}(\mu_a) \\ \langle c \mid \mu \uplus \mu_a\rangle \rightsquigarrow \langle c' \mid \mu' \uplus \mu_a'\rangle \qquad \text{dom}(\mu_a') = \text{dom}(\mu_a)\end{array}}{\langle \text{inalloc } \mu_a \text{ do } c \mid \mu\rangle \rightsquigarrow \langle \text{inalloc } \mu_a' \text{ do } \tau c' \mid \mu'\rangle}$$

InAlloc-Skip
$$\frac{}{\langle \text{inalloc } \mu_a \text{ do skip} \mid \mu\rangle \rightsquigarrow \langle \text{skip} \mid \mu\rangle}$$

Figure 4.6: Small-Step Interleaving Semantics

and

$$\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle \Leftrightarrow \emptyset; \mu \vdash c[\mathcal{E}] \Downarrow_u \langle \delta\mu; \rho \rangle$$

Usually, one would expect big-step and small-step semantics to be equivalent, in the sense that there should be a big-step evaluation for program c in memory μ to memory μ' if, and only if, there is a finite sequence of reductions from $\langle c \mid \mu \rangle$ to $\langle \text{skip} \mid \mu' \rangle$. However, imperative small-step semantics in a concurrent setting are not deterministic due to the possibility of races, while our big-step semantic is deterministic: hence, such a strong equivalence theorem would be inconsistent and is necessarily false.

Recall that our big-step semantics is deterministic by design: we are only interested in deterministic programs because we consider that if a tensor compiler generates a racy, nondeterministic program from a deterministic specification, it can only indicate a bug in the tensor compiler and should be disallowed by the verifier. Hence, rather than true equivalence, we are interested in proving that programs with a big-step semantics have a deterministic interleaving small-step semantics. This can be expressed as the combination of two theorems: if c evaluates in big-step to μ' in μ, then $\langle c \mid \mu \rangle$ reduces in many small steps to $\langle \text{skip} \mid \mu' \rangle$ (existence), and if c evaluates in big-step to μ' in μ, then all sequences of small-step reductions starting from $\langle c \mid \mu \rangle$ converge to $\langle \text{skip} \mid \mu' \rangle$ (determinism).

The existence property is the easiest to state and prove.

Theorem 4.3.3 (Existence)**.** *If a command* c *evaluates to* $\langle \delta\mu; \rho \rangle$ *in environment* $\mathcal{E}$ *and memory* μ*, then there exists a sequence of reductions from* $\langle c[\mathcal{E}] \mid \mu \rangle$ *to* $\langle \text{skip} \mid \mu \triangleright \delta\mu \rangle$*.*

The proof of the theorem proceeds by building a leftward sequence of reduction steps and composing them as appropriate, always evaluating the left branch first in the case of parallel composition. Because the big-step semantics evaluates all iterations of a parallel loop in the *same* initial memory while this construction results in evaluating iteration i in the memory resulting of the evaluation of all iterations $0 \leqslant j < i$, an induction on Theorem 4.3.3 would not be well-founded; we must instead generalize its statement before performing the proof:

Lemma 4.3.4. *If a command* c *evaluates to* $\langle \delta\mu; \rho \rangle$ *in environment* $\mathcal{E}$ *and memory* μ*, then there exists a sequence of reduction from* $\langle c[\mathcal{E}] \mid \mu' \rangle$ *to* $\langle \text{skip} \mid \mu' \triangleright \delta\mu \rangle$ *for any memory* μ' *that contains* $\text{dom}(\delta\mu)$ *and agrees with* μ *on* ρ*.*

The proof of Theorem 4.3.4 relies on a few technical lemmas relating the update operation with inclusion and compatibility of maps.

Lemma 4.3.5. *If* $\delta\mu_1 \subseteq \delta\mu_1'$ *then* $\delta\mu_1 \triangleright \delta\mu_2 \subseteq \delta\mu_1' \triangleright \delta\mu_2$ *for all partial maps* $\delta\mu_2$*.*

Lemma 4.3.6. *If* μ *is compatible with* μ' *on* ρ*, then for any* $\delta\mu$*,* $\mu \triangleright \delta\mu$ *is compatible with* $\mu' \triangleright \delta\mu$ *on* ρ*.*

Proof. Consider a location ℓ. If ℓ is in the domain of $\delta\mu$, then $\mu \triangleright \delta\mu(\ell) = \delta\mu(\ell) = \mu' \triangleright \delta\mu(\ell)$. If ℓ is not in the domain of $\delta\mu$, then $\mu \triangleright \delta\mu(\ell) = \mu(\ell) = \mu'(\ell) = \mu' \triangleright \delta\mu(\ell)$. □

We can now prove Theorem 4.3.4 by induction, and the proof of Theorem 4.3.3 immediately follows by applying Theorem 4.3.4 with $\mu' = \mu$.

Proof. By induction on the derivation of $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$, we build a sequence of reduction steps without using rules Par-R or Par-Skip-R by treating the parallel composition as a sequential composition.

U-Skip We have $\delta\mu = \emptyset$ and $\text{skip} \mid \mu'$ reduces in 0 steps to $\text{skip} \mid \mu'$ for any memory μ'.

U-Assign Theorem 4.2.2 ensures that the evaluation of each expression in μ and μ' are identical, and we can apply rule Assign to $c[\mathcal{E}] \mid \mu$ using Theorem 4.3.2.

U-If-True, U-If-False By induction hypothesis, we get a sequence of small-step reductions for the body in any compatible memory μ', which we can prefix with If-True (resp. If-False).

U-Let By induction hypothesis, we get a sequence of small-step reductions for the body $\langle c[\mathcal{E} + x \mapsto v] \mid \mu' \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu' \triangleright \delta\mu \rangle$. Moreover, we have let $x = \iota$ in $c[\mathcal{E}]$ = let $x = \iota[\mathcal{E}]$ in $c[\mathcal{E}]$, hence by rule LET we have $\langle \text{let } x = \iota \text{ in } c[\mathcal{E}] \mid \mu' \rangle \rightsquigarrow \langle c[\mathcal{E}][x \leftarrow v] \mid \mu' \rangle$ from which we conclude since $c[\mathcal{E} + x \mapsto v] = c[\mathcal{E}][x \leftarrow v]$.

U-Seq By induction hypothesis, we have $\langle c_1[\mathcal{E}] \mid \mu_1 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu_1 \triangleright \delta\mu_1 \rangle$ for any μ_1' compatible with μ on ρ_1. We also have $\langle c_2[\mathcal{E}] \mid \mu_2 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu_2 \triangleright \delta\mu_2 \rangle$ for any μ_2 compatible with $\mu \triangleright \delta\mu_1$ on ρ_2.

Assume that we have μ' compatible with μ on $\rho_1 \cup \rho_2$; in particular, μ' is compatible with μ on ρ_1 and we have $\langle c_1[\mathcal{E}] \mid \mu' \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu' \triangleright \delta\mu_1 \rangle$. Moreover, $\mu' \triangleright \delta\mu_1$ is compatible with $\mu \triangleright \delta\mu_1$ on ρ_2 by Theorem 4.3.6, hence we have $\langle c_2[\mathcal{E}] \mid \mu' \triangleright \delta\mu_1 \rangle \rightsquigarrow^* \langle \text{skip} \mid (\mu' \triangleright \delta\mu_1) \triangleright \delta\mu_2 \rangle$.

By applying rule SEQ-CTX to the steps in the reduction of $c_1[\mathcal{E}]$ and eliminating the resulting skip with rule SEQ-SKIP, we can connect it with the reduction of $c_2[\mathcal{E}]$ to obtain $\langle c_1 \; ; \; c_2[\mathcal{E}] \mid \mu' \rangle \rightsquigarrow^* \langle \text{skip} \mid (\mu' \triangleright \delta\mu_1) \triangleright \delta\mu_2 \rangle$ and conclude by associativity.

U-Par Similar to rule U-SEQ, we get $\langle c_1[\mathcal{E}] \mid \mu_1 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu_1 \triangleright \delta\mu_1 \rangle$ for any μ_1 compatible with μ on ρ_1 and $\langle c_2[\mathcal{E}] \mid \mu_2 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu_2 \triangleright \delta\mu_2 \rangle$ for any μ_2 compatible with μ on ρ_2.

For μ' compatible with μ on $\rho_1 \cup \rho_2$, we get $\langle c_1[\mathcal{E}] \mid \mu' \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu' \triangleright \delta\mu_1 \rangle$ which we can wrap in rule PAR-L and PAR-SKIP-L to obtain $\langle c_1 \; ; \; c_2[\mathcal{E}] \mid \mu' \rangle \rightsquigarrow^* \langle c_2[\mathcal{E}] \mid \mu' \triangleright \delta\mu_1 \rangle$.

Because of the constraint $\text{dom}(\delta\mu_1) \# \rho_2$, $\mu' \triangleright \delta\mu_1$ is compatible with μ' and by transitivity with μ on ρ_2, hence we have $\langle c_2[\mathcal{E}] \mid \mu' \triangleright \delta\mu_1 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu' \triangleright \delta\mu_1 \triangleright \delta\mu_2 \rangle$ and we conclude.

U-For By induction hypothesis, for $0 \leqslant i < n$ and a memory μ_i compatible with $\mu \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j$ on ρ_i, we have $\langle c[\mathcal{E} + x \mapsto i] \mid \mu_i \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu_i \triangleright \delta\mu_i \rangle$.

If μ' is compatible with μ on $\bigcup_{0 \leqslant i < n} \rho_i$, then $\mu' \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j$ is compatible with $\mu \triangleright \bigtriangleright_{0 \leqslant j < i} \delta\mu_j$ by repeated application of Theorem 4.3.6 on $\bigcup_{0 \leqslant i < n} \rho_i$, and in particular on ρ_i.

Hence, we get a series of reductions from $\langle c[\mathcal{E} + x \mapsto i] \mid \mu' \triangleright \blacktriangleright_{0 \leqslant j < i} \delta\mu_j \rangle$ to $\langle \text{skip} \mid \mu' \triangleright \blacktriangleright_{0 \leqslant j \leqslant i} \delta\mu_j j \rangle$ that can be combined with rules Seq-Ctx and Seq-Skip and prefixed with rule For to conclude.

U-ParLoop The proof is similar to that of rule U-For using rules Par-L, Par-Skip-L and Par instead of Seq-Ctx, Seq-Skip and For. Moreover, we must check that $\mu' \triangleright \blacktriangleright_{0 \leqslant j < i} \delta\mu_j$ is compatible with μ on ρ_i in order to apply the induction hypothesis, which is the case because of the conditions $\text{dom}(\delta\mu_j) \# \rho_i$ ensuring that $\mu' \triangleright \blacktriangleright_{0 \leqslant j < i} \delta\mu_j$ is compatible on ρ_i with μ' and hence with μ by transitivity.

□

Let us now consider the proof of the determinism property for programs with a big-step semantics. Determinism can be stated as follows:

Theorem 4.3.7 (Determinism)**.** *If* c *evaluates to* $\langle \delta\mu; \rho \rangle$ *in environment* $\mathcal{E}$ *and memory* μ*, and* $\langle c[\mathcal{E}] \mid \mu \rangle$ *reduces in zero, one, or several steps to* $\langle \text{skip} \mid \mu' \rangle$*, then* μ' *is equal to* $\mu \triangleright \delta\mu$*.*

In order to prove Theorem 4.3.7, the first intuition is to perform an induction on the judgement $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$. However, consider the case of rule U-ParLoop: to apply the induction hypothesis, we would need to build separate executions $\langle c[\mathcal{E} + x \mapsto i] \mid \mu \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu'_i \rangle$ for each $0 \leqslant i < n$ from the interleaved execution $\langle \text{for } x < \iota; \text{do } c \mid \mu \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu' \rangle$. Doing so would require many re-ordering of the underlying reduction steps, and it is not clear it would be easy to prove the correctness of such re-orderings.

If we cannot perform an induction on the big-step semantics, we can try to perform an induction on the small-step semantics instead, since this is the only other hypothesis available to us. Doing so requires generalizing the statement of Theorem 4.3.7, since within the induction, the result of the reduction is not necessarily skip. Considering the case of a single-step reduction first, the statement of the induction step would need to be similar to the following lemma:

Lemma 4.3.8 (Single-step Preservation). *If* $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$ *and* $\langle c[\mathcal{E}] \mid \mu \rangle \rightsquigarrow \langle c' \mid \mu' \rangle$ *both hold, then there exists* $\delta\mu'$ *and* ρ' *such that* $\emptyset; \mu' \vdash c' \Downarrow_u \langle \delta\mu'; \rho' \rangle$ *holds.*

Moreover, we have $\delta\mu' \subseteq \delta\mu$, $\rho' \subseteq \rho$, *and* $\mu' \triangleright \delta\mu' = \mu \triangleright \delta\mu$.

Theorem 4.3.8 can be repeatedly applied through an immediate induction to generalize its statement to a sequence of reductions:

Corollary 4.3.9 (Many-steps Preservation). *If* $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$ *and* $\langle c[\mathcal{E}] \mid \mu \rangle \rightsquigarrow^* \langle c' \mid \mu' \rangle$ *both hold, then there exists* $\delta\mu'$ *and* ρ' *such that* $\emptyset; \mu' \vdash c' \Downarrow_u \langle \delta\mu'; \rho' \rangle$ *holds.*

Moreover, we have $\delta\mu' \subseteq \delta\mu$, $\rho' \subseteq \rho$, *and* $\mu' \triangleright \delta\mu' = \mu \triangleright \delta\mu$.

By instantiating c' with skip and ignoring the inclusion conditions, the proof of Theorem 4.3.7 is immediate from Theorem 4.3.9:

Proof of Theorem 4.3.7. By Theorem 4.3.9, we get $\delta\mu'$, and ρ' such that $\emptyset; \mu' \vdash \text{skip} \Downarrow_u \langle \delta\mu'; \rho' \rangle$, $\delta\mu' \subseteq \delta\mu$, $\rho' \subseteq \rho$, and $\mu' \triangleright \delta\mu' = \mu \triangleright \delta\mu$.

However, the only applicable rule giving a semantic to skip is U-SKIP, hence we get that $\delta\mu' = \emptyset$ and we conclude that $\mu \triangleright \delta\mu = \mu' \triangleright \emptyset = \mu'$. □

It now remains to prove Theorem 4.3.8. Except that Theorem 4.3.8 is trivially false: we first need to introduce big-step evaluation rules for the intermediate constructs introduced by the small step semantics, namely parallel composition and the inalloc block. These evaluation rules are reproduced below and are derived from the rules U-PARLOOP and U-ALLOCATE.

U-PAR

$$\frac{\mathcal{E}; \mu \vdash s_1 \Downarrow_u \langle \delta\mu_1; \rho_1 \rangle \qquad \mathcal{E}; \mu \vdash s_2 \Downarrow_u \langle \delta\mu_2; \rho_2 \rangle \qquad \text{dom}(\delta\mu_1) \# \rho_2 \qquad \text{dom}(\delta\mu_2) \# \rho_1 \qquad \omega_1 \supset \omega_2}{\mathcal{E}; \mu \vdash s_1 \ ; \ s_2 \Downarrow_u \langle \delta\mu_1 \triangleright \delta\mu_2; \rho_1 \cup \rho_2 \rangle}$$

U-INALLOC

$$\frac{\mathcal{E}; \mu \triangleright \mu_a \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle}{\mathcal{E}; \mu \vdash \text{inalloc } \mu_a \text{ do } c \Downarrow_u \langle \delta\mu \setminus \text{dom}(\mu_a); \rho \setminus \text{dom}(\mu_a) \rangle}$$

The proofs of Theorem 4.2.4 and Theorem 4.3.4 remain valid in the presence of these additional rules, and we can finally prove Theorem 4.3.8.

Proof of Lemma 4.3.8. By induction on the reduction step $\langle c[\mathcal{E}] \mid \mu\rangle \rightsquigarrow \langle c' \mid \mu'\rangle$.

Seq-Ctx The big-step semantics must use rule U-Seq, hence we have $\mathcal{E}; \mu \vdash c_1 \Downarrow_u \langle \delta\mu_1; \rho_1\rangle$. By induction hypothesis, since $\langle c_1[\mathcal{E}] \mid \mu\rangle \rightsquigarrow \langle c_1' \mid \mu'\rangle$ holds, we get that $\emptyset; \mu' \vdash c_1' \Downarrow_u \langle \delta\mu_1'; \rho_1'\rangle$ holds with the inclusions and $\mu' \triangleright \delta\mu_1' = \mu \triangleright \delta\mu_1$.

From the original application of U-Seq, we also have $\mathcal{E}; \mu \triangleright \delta\mu_1 \vdash c_2 \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$ hence, by Theorem 4.3.2, we have $\emptyset; \mu \triangleright \delta\mu_1 \vdash c_2[\mathcal{E}] \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$.

Since $\mu' \triangleright \delta\mu_1' = \mu \triangleright \delta\mu_1$, we get $\emptyset; \mu' \triangleright \delta\mu_1' \vdash c_2[\mathcal{E}] \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$ and we conclude using set reasoning and Theorem 4.3.5 to prove the inclusions $\delta\mu_1' \triangleright \delta\mu_2 \subseteq \delta\mu_1 \triangleright \delta\mu_2$, and $\rho_1' \cup \rho_2 \subseteq \rho_1 \cup \rho_2$.

Seq-Skip The first two rules in the proof tree for $\mathcal{E}; \mu \vdash \text{skip} \; ; c \Downarrow_u \langle \delta\mu; \rho\rangle$ are U-Seq and U-Skip, which we can remove.

Par-L and Par-R Assume without loss of generality that we are considering rule Par-L; the proof is similar to the case of Seq-Ctx. By induction hypothesis, we get $\emptyset; \mu' \vdash c_1' \Downarrow_u \langle \delta\mu_1'; \rho_1'\rangle$ with $\mu' \triangleright \delta\mu_1' = \mu \triangleright \delta\mu_1$ and the appropriate inclusions.

We also have $\mathcal{E}; \mu \vdash c_2 \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$ from the big-step semantics, which gives $\emptyset; \mu \vdash c_2[\mathcal{E}] \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$ by Theorem 4.3.2.

We now claim that μ and μ' are compatible over ρ_2. First, since $\text{dom}(\delta\mu_1) \# \rho_2$, $\mu' \triangleright \delta\mu_1' = \mu \triangleright \delta\mu$ is compatible with μ over ρ_2. Second, since $\delta\mu_1' \subseteq \delta\mu_1$ and $\delta\mu_1 \# \rho_2$, we also have $\delta\mu_1' \# \rho_2$, μ' and $\mu' \triangleright \delta\mu_1'$ are compatible over ρ_2, and we conclude by transitivity.

Since μ and μ' are compatible over ρ_2, we get $\emptyset; \mu' \vdash c_2[\mathcal{E}] \Downarrow_u \langle \delta\mu_2; \rho_2\rangle$ and we conclude since the side conditions of U-Par still apply when one of the argument gets smaller.

When applying Par-R, note that $\delta\mu_2$ is compatible with $\delta\mu_1$ and $\delta\mu_1'$, hence

the updates $\delta\mu_2 \triangleright \delta\mu_1$ and $\delta\mu_2 \triangleright \delta\mu'_1$ commute.

Par-Skip-L and Par-Skip-R The proof tree of the big-step semantics is composed of U-PAR and one U-SKIP, which simplify to the proof tree of the right (resp. left) hand side.

If-True, If-False and Let We have $\mu' = \mu$ and by Theorem 4.3.2, the evaluation in the small- and big-step semantics are identical. We conclude using the induction hypothesis and the monotony of $\cup$ for the inclusion.

SeqLoop We have $\mu' = \mu$ and the evaluations in the small- and big-step semantics are identical by Theorem 4.3.2. The set of reads $\mathrm{rd}_{\mathcal{E};\mu}(e)$ no longer contributes to ρ, which might get smaller; this is allowed. If $n > 0$, the comb-like application of rule U-SEQ for the sequence of assignments corresponds to the unfolding of the quantified premise in U-FOR. If $n \leqslant 0$, rule U-FOR is otherwise equivalent to U-SKIP.

ParLoop We have $\mu' = \mu$ and the evaluations in the small- and big-step semantics are identical by Theorem 4.3.2. The set of reads $\mathrm{rd}_{\mathcal{E};\mu}(e)$ again no longer contributes to ρ. If $n \leqslant 0$, rule U-PAR is otherwise equivalent to U-SKIP. If $n > 0$, the comb-like application of rule U-PAR corresponds to the unfolding of the quantified premise in U-PARLOOP, except that the same memory is used as input for each application of U-PAR. Because each $\mathrm{dom}(\delta\mu_i)$ is disjoint with all the other ρ_j, we can apply Theorem 4.2.4 to change the input memory from μ to $\mu \triangleright \bigtriangleup_{0 \leqslant j < i} \delta\mu_j$. Finally, the side conditions are equivalent in both case using De Morgan's laws for # and $\cup$, and noting that $\delta\mu_1 \supset \delta\mu_2 \cup \delta\mu_3$ if and only if $\delta\mu_1 \supset \delta\mu_2$ and $\delta\mu_1 \supset \delta\mu_3$.

InAlloc-Ctx Since $\langle c[\mathcal{E}] \mid \mu \uplus \mu_a \rangle \rightsquigarrow \langle c' \mid \mu' \uplus \mu'_a \rangle$ and $\mathcal{E}; \mu \uplus \mu_a \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$, by induction hypothesis we get $\emptyset; \mu' \uplus \mu'_a \vdash c \Downarrow_u \langle \delta\mu'; \rho' \rangle$ with $\delta\mu' \subseteq \delta\mu$, $\rho' \subseteq \rho$ and $(\mu' \uplus \mu'_a) \triangleright \delta\mu' = (\mu \uplus \mu_a) \triangleright \delta\mu$.

By standard set reasoning, the inclusions remain when removing $\mathrm{dom}(\mu_a)$, and we get the big-step evaluation for inalloc μ'_a do c by applying U-INALLOC.

InAlloc-Skip The proof tree for the big-step evaluation applies U-INALLOC to U-SKIP, hence evaluates to $\langle \emptyset; \emptyset \rangle$ and can be transformed to U-SKIP.

□

Theorem 4.3.9 and Theorem 4.3.3 can also be used to prove the confluence of reductions starting in a program with a big-step semantics:

Corollary 4.3.10 (Confluence). *If* c *has a big-step evaluation to* $\langle \delta\mu; \rho \rangle$ *in* $\mathcal{E}$ *and* μ*, and* $\langle c[\mathcal{E}] \mid \mu \rangle$ *reduces in many small steps to both* $\langle c_1 \mid \mu_1 \rangle$ *and* $\langle c_2 \mid \mu_2 \rangle$*, then they both reduce in many steps to some common pair* $\langle c' \mid \mu' \rangle$*.*

Proof. By Theorem 4.3.9, c_1 has a big-step semantics $\emptyset; \mu_1 \vdash c_1 \Downarrow_u \langle \delta\mu_1; \rho_1 \rangle$ with $\mu_1 \triangleright \delta\mu_1 = \mu \triangleright \delta\mu$. Hence, by Theorem 4.3.3, there is a reduction $\langle c_1 \mid \mu_1 \rangle \rightsquigarrow^* \langle \text{skip} \mid \mu \triangleright \delta\mu \rangle$.

The same reasoning applies to c_2, and we conclude with $c' = \text{skip}$ and $\mu' = \mu \triangleright \delta\mu$. □

4.4 Typing

In order to distinguish between semantic expressions and index expressions, we will introduce two distinct type systems. In these type systems, the distinguished type $\mathbb{A}$ of affine expressions (or, more accurately, piece-wise quasi-affine expressions) is used for array indices and loop bounds, while the distinguished type $\mathbb{B}$ of affine constraints (or, rather, piece-wise affine constraints) is used in conditionals. The type system enforces that expressions of these distinguished types are expressed using piece-wise quasi-affine combinations of the program parameters and outer loop iterators. For now, we will assume that computation on the types $\mathbb{A}$ and $\mathbb{B}$ is performed using exact arithmetic and ignore the issue of possible overflows in index computations throughout. The handling of integer overflows is discussed in Section 9.5. Moreover, we do not allow values of types $\mathbb{A}$ and $\mathbb{B}$ to be directly stored into arrays; however, we do allow casting them to value types such as int32 using a conversion function.

The typing environments abstract over a portion of both the runtime environment $\mathcal{E}$ and the memory μ. Typing environments are ranged over by Γ, and can contain three type of bindings:

- Affine bindings $x : \mathbb{A}$ or $x : \mathbb{B}$ from a name to an affine type. We only allow variables of affine types, because variables of other types can be represented using local zero-dimensional arrays, as discussed below. Affine bindings are introduced using let expressions and commands.

- Array bindings from names to array shapes $a : \tau[\iota_1 \times \cdots \times \iota_n]$. An array shape $\tau[\iota_1 \times \cdots \times \iota_n]$ represents an n-dimensional, rectangular array containing values of type τ, where dimension i has length ι_i. Indices start at 0 in each dimension. Arrays are mutable, and are not initialized. A mutable variable can be represented using a zero-dimensional array.

- Affine boolean expressions e, to keep track of the bounds on loop indices and other conditionals. These are similar to path conditions in a symbolic evaluator, and are included in the typing judgement for the purpose of symbolic evaluation in the next section.

Whenever we write a context $\Gamma, x : \tau$ (resp. $\Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]$), we implicitly assume that x (resp. a) is not bound in Γ. In the case of an array binding, we also assume that the ι_i are well-typed of type $\mathbb{A}$, using the typing rules presented below. More generally, we follow the Barendregt convention of α-renaming the bound variables to avoid name conflicts. Moreover, we assume that all contexts Γ start with a common prefix Γ_P that only contain variable bindings for the parameters and are always usable in expressions.

The expressions of SCHED can be separated into two categories. *Affine expressions* are used in array indices, loop bounds, and conditionals. They are restricted to syntactically affine combinations of the program parameters and outer affine variables, and are typed using the judgement $\Gamma \vdash \iota : \tau$ (read "under assumptions Γ, the affine expression ι has type τ"), where $\tau \in \{\mathbb{A}, \mathbb{B}\}$. We will also state "$\iota$ is an affine expression in Γ" for $\Gamma \vdash \iota : \mathbb{A}$ and "ι is an affine constraint in Γ" for $\Gamma \vdash \iota : \mathbb{B}$, omitting the "in Γ" part when it can be inferred from the context. *Semantic expressions* have a value type and can contain array reads and function calls. They appear on the right-hand side of array assignments, and are typed using the judgement $\Gamma \vdash_a e : \tau$, which is read "under assumptions Γ, the expression e has semantic type τ". We will also state simply "e has type τ in Γ", omitting the "in Γ" part when it can be inferred from context. Finally, *prophetic expressions* have a value type and contains tensor accesses reading directly from the specification tensors instead of array accesses. Prophetic expressions t are typed using the judgement $\Gamma \vdash_A t : \tau$, read "t has prophetic type τ in Γ".

The typing rules for both judgements are given in Fig. 4.7. Most of the rules are fairly standard. select is an eager affine conditional that can appear in both kinds of expressions.

A typing environment Γ can be related with the runtime environments $\mathcal{E}$ that it represents.

Definition 4.4.1. A runtime environment $\mathcal{E}$ is *compatible* for a typing environment Γ, written $\mathcal{E} \vDash \Gamma$, if the variables of Γ are associated in $\mathcal{E}$ to values that satisfy the assertions in Γ. Formally, $\mathcal{E} \vDash \Gamma$ is defined using the inference rules:

$$\frac{}{\mathcal{E} \vDash \emptyset} \qquad \frac{\mathcal{E}(x) \in \mathbb{Z} \qquad \mathcal{E} \vDash \Gamma}{\mathcal{E} \vDash \Gamma, x : \mathbb{A}} \qquad \frac{\mathcal{E}(x) \in \{\text{true}, \text{false}\} \qquad \mathcal{E} \vDash \Gamma}{\mathcal{E} \vDash \Gamma, x : \mathbb{B}}$$

$$\frac{\forall 1 \leqslant i \leqslant n,\ [\![\iota_i]\!]_{\mathcal{E}} = n_i \in \mathbb{Z} \qquad \mathcal{E} \vDash \Gamma}{\mathcal{E} \vDash \Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]} \qquad \frac{\mathcal{E} \vDash \Gamma \qquad [\![e]\!]_{\mathcal{E}} = \text{true}}{\mathcal{E} \vDash \Gamma, e}$$

If b is a boolean expression in context Γ, whether written directly as a formula on affine expressions or expressed as a set-theoretic formula on Presburger sets (such as a set inclusion or an expression using empty or sv), we use the notation $\Gamma \vdash b$ to indicate that the expression b is implied by the affine constraints of Γ. That is, $\Gamma \vdash b$ holds iff $[\![b]\!]_{\mathcal{E}}$ is true for all environments $\mathcal{E}$ compatible with Γ. This can be expressed as a piecewise quasi-affine problem and decided using `isl`. For instance, if S_1 and S_2 are two symbolic sets, $\Gamma \vdash S_1 \subseteq S_2$ holds iff $[\![S_1]\!]_{\mathcal{E}} \subseteq [\![S_2]\!]_{\mathcal{E}}$ for all $\mathcal{E} \vDash \Gamma$.

Similarly, we can define what it means for both an environment $\mathcal{E}$ and memory μ together to be compatible with a typing environment Γ. Because the typing environment does not distinguish initialized and uninitialized memory cells, we cannot say that the all values associated in μ with a valid location in Γ have values of the corresponding type, as it could also hold the distinguished value $\perp$. Instead, we quantify over a subset of the locations that must hold values of an appropriate type.

Definition 4.4.2. A location $a[i_1, \ldots, i_n]$ is *well-typed* with type τ for a typing environment Γ in a runtime environment $\mathcal{E}$ if there is a binding $a : \tau[\iota_1 \times \cdots \times \iota_n]$ in Γ such that $[\![\iota_i]\!]_{\mathcal{E}} = n_i \in \mathbb{Z}$ and $0 \leqslant i_i < n_i$ for all $1 \leqslant i \leqslant n$.

$$\textsc{T-Var}\ \frac{}{\vdash \Gamma, x : \tau}\qquad \frac{}{x,\ \tau :\ \vdash x : \tau}$$

$$\textsc{T-Array}\ \frac{a\ :\ \tau[\iota'_1, \ldots, \iota'_n] \in \Gamma \qquad \forall 1 \leqslant i \leqslant n,\ \Gamma \vdash \iota_i : \mathbb{A}}{\Gamma \vdash_a a[\iota_1, \ldots, \iota_n] : \tau}$$

$$\textsc{T-Tensor}\ \frac{A \in \mathcal{S} \qquad \forall 1 \leqslant i \leqslant n_A,\ \Gamma \vdash \iota_i : \mathbb{A}}{\Gamma \vdash_A A(\iota_1, \ldots, \iota_{n_A}) : \tau_A}$$

$$\textsc{T-Bool}\ \frac{b \in \{\text{true}, \text{false}\}}{\Gamma \vdash b : \mathbb{B}}$$

$$\textsc{T-Int}\ \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \mathbb{A}}$$

$$\textsc{T-Call}\ \frac{\forall 1 \leqslant i \leqslant n, k_i \in \{k, \emptyset\} \Rightarrow \Gamma \vdash_{k_i} e_i : \tau_i \qquad k \in \{a, A\} \qquad f \in \mathcal{F} \qquad \tau_f = \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash_k f(e_1, \ldots, e_n) : \tau}$$

$$\textsc{T-Select}\ \frac{k \in \{a, A, \emptyset\} \qquad \Gamma \vdash \iota_1 : \mathbb{B} \qquad \Gamma, e_1 \vdash_k e_2 : \tau \qquad \Gamma, \neg e_1 \vdash_k e_3 : \tau}{\Gamma \vdash_k \text{select}(\iota_1, e_2, e_3)}$$

$$\textsc{T-Let}\ \frac{k \in \{a, A, \emptyset\} \qquad \Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A} \vdash_k e : \tau}{\Gamma \vdash_k \text{let } x = \iota \text{ in } e : \tau}$$

$$\textsc{T-Add}\ \frac{\Gamma \vdash \iota_1 : \mathbb{A} \qquad \Gamma \vdash \iota_2 : \mathbb{A}}{\Gamma \vdash \iota_1 + \iota_2 : \mathbb{A}}\qquad \textsc{T-Mul}\ \frac{\Gamma \vdash \iota : \mathbb{A}}{\Gamma \vdash n \cdot e : \mathbb{A}}\qquad \textsc{T-Div}\ \frac{\Gamma \vdash \iota : \mathbb{A} \qquad n > 0}{\Gamma \vdash \lfloor \iota / n \rfloor : \mathbb{A}}$$

$$\textsc{T-Mod}\ \frac{\Gamma \vdash \iota : \mathbb{A} \qquad n > 0}{\Gamma \vdash \iota \bmod n : \mathbb{A}}\qquad \textsc{T-Cmp}\ \frac{\Gamma \vdash \iota_1 : \mathbb{A} \qquad \Gamma \vdash \iota_2 : \mathbb{A} \qquad \odot \in \{=, \leqslant\}}{\Gamma \vdash \iota_1 \odot \iota_2 : \mathbb{B}}$$

$$\textsc{T-And}\ \frac{\Gamma \vdash \iota_1 : \mathbb{B} \qquad \Gamma \vdash \iota_2 : \mathbb{B}}{\Gamma \vdash \iota_1\ \&\&\ \iota_2 : \mathbb{B}}\qquad \textsc{T-Not}\ \frac{\Gamma \vdash \iota : \mathbb{B}}{\Gamma \vdash !\iota : \mathbb{B}}$$

Figure 4.7: Typing rules for SCHED expressions

A memory μ is *well-typed* over a set of locations ρ for a typing environment Γ in a runtime environment $\mathcal{E}$, written $\mathcal{E} \vDash \mu(\rho) : \Gamma$, if all locations in ρ are well-typed for Γ in $\mathcal{E}$ and for any well-typed location $\ell \in \rho$ with type τ, there is a binding $\ell \mapsto v$ in μ for some $v \in [\![\tau]\!]$.

A memory μ is *compatible* with a typing environment Γ in a runtime environment $\mathcal{E}$, denoted $\mathcal{E}; \mu \vDash \Gamma$, if the domain of μ is exactly the set of well-typed locations of Γ in $\mathcal{E}$.

We can now prove soundness theorems for our typing judgements, stating that when an expression is well-typed in typing environment Γ, it evaluates to a value of the appropriate type in compatible runtime environment and memories.

Theorem 4.4.1 (Type Soundness for expressions)**.** *If an expression ι is affine (resp. an affine constraint) in context Γ (i.e. $\Gamma \vDash \iota : \mathbb{A}$ (resp. $\Gamma \vDash \iota : \mathbb{B}$)), then the evaluation of ι in any environment $\mathcal{E}$ compatible Γ is an integer (resp. a Boolean).*

If an expression e has type τ in context Γ (i.e. $\Gamma \vdash_a e : \tau$ holds), then the evaluation of e in any environment $\mathcal{E}$ compatible for Γ and memory μ well-typed over $\mathrm{rd}_{\mathcal{E};\mu}(e)$ for Γ in $\mathcal{E}$ (i.e. $\mathcal{E} \vDash \Gamma$ and $\mathcal{E} \vDash \mu(\rho) : \Gamma$) is a value of type τ (i.e. $[\![e]\!]_{\mathcal{E};\mu} \in [\![\tau]\!]$).

Proof. For the first case, the proof proceeds by induction on the judgement $\Gamma \vdash \iota : \tau$ after generalizing over $\tau = \mathbb{A}$ or $\tau = \mathbb{B}$.

For the second case, the proof proceeds by induction on the judgement $\Gamma \vdash_a e : \tau$, remarking that if μ is well-typed over ρ for Γ in $\mathcal{E}$, it is also well-typed over any subset of ρ. Because we require μ to be well-typed over $\mathrm{rd}_{\mathcal{E};\mu}(e)$, we ensure that the value read in an array access is defined and of the correct type. □

We can also state and prove type soundness for prophetic expressions similarly:

Theorem 4.4.2 (Type Soundness for Prophetic Expressions)**.** *If e has prophetic type τ in context Γ (i.e. $\Gamma \vdash_A e : \tau$) then for any environment $\mathcal{E}$ compatible with Γ and well-typed model $\mathcal{M}$ the evaluation of e in $\mathcal{E}$ and $\mathcal{M}$ has type τ (i.e. $[\![e]\!]_{\mathcal{E};\mathcal{M}} \in [\![\tau]\!]$).*

Further, we note that affine expressions and constraints do not contain array reads:

Lemma 4.4.3. *If e is an affine expression or constraint in Γ (i.e. $\Gamma \vdash \iota : \mathbb{A}$ or $\Gamma \vdash \iota : \mathbb{B}$) then for any environment $\mathcal{E}$ compatible with Γ, we have $\mathrm{rd}_{\mathcal{E};\emptyset}(e) = \emptyset$.*

In particular, the conditions of Theorem 4.2.3 apply to affine expressions and constraints.

Since $\Gamma \vdash_a e : \tau$ ensures that all indices are affine, we also have:

Corollary 4.4.4. *If e has type τ in Γ, then for any $\mathcal{E}$ compatible with Γ, $\mathrm{rd}_{\mathcal{E};\emptyset}(e) = \rho$ is defined.*

Verifying a tensor compiler 5

The techniques and tools describe in chapter 2 are all used in production systems today, targeting a range of heterogeneous hardware. These systems are focused on practical applications and have different track records when it comes to their correctness. A recent study by Shen et al. [99] of three deep learning compilers (TVM, nGraph and Glow) found that about a quarter of deep learning compilers bug result in the compiler generating incorrect code. The authors of the study note that the high-level optimization are a source of more bugs than the low-level optimization, but they admit that this is partly due to their study focusing on frameworks that delegate low-level optimization to underlying libraries and toolkits. The generation of incorrect code is a concerning bug, and one that is challenging to design proper test cases for. The authors of the study note that more attention should be paid to design effective testing methods for wrong code bugs from both academia and industry.

By building upon the implementation language SCHED presented in the previous chapter, this chapter explores the design of a practical translation-validation tool to allow the formal verification of the code generated by a tensor compiler for a given operator. Formal verification guarantees correctness for any input and any (possibly conditional) input shapes.

The content of this chapter is based on the work presented in "End-to-end translation validation for the halide language" [27].

5.1 Verification conditions

Our technique can be understood as a generator of *verification conditions*, i.e. assertions that, if true, entail the desired property — here that the implementation matches the specification. By far the most common presentation of verification condition generators such as Why3 [19] or Dafny [66] is based on the concept of predicate transformers[33] in a Hoare-style logic, usually a "weakest precondition" predicate transformer. These tools use logical assertions, pre- and post-conditions for functions, and loop invariants to relate the code to a formal specification. The prophetic annotations we have introduced in Chapter 1 can be seen as assertions, as in this naive implementation of the matrix product from Chapter 1:

```
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    r[] := 0
    assert (r[] = 0)
    for k = 0 to P - 1 do
      r[] := r[] + a[i, k] * b[k, j]
      assert (r[] = R(i, j, k))
    c[i, j] := r[]
    assert (c[i, j] = C(i, j))
```

On its own, these assertions are not enough to prove the equivalence: loop invariants are required to propagate information across loop iterations, such as the value of `r[]` at line 6 that is needed to prove the assertion at line 7. In this case, we need to infer the invariant for the loop on `k` that we used in Chapter 1:

```
    for k = 0 to P - 1 do
      invariant { r[] = if k = 0 then 0 else R(i,j,k-1) }
      ...
```

In the general case, if `r[]` was an array, the invariant would also need to specify the values at the indices which are not written by the loop.

The process we use to generate invariants, presented in this chapter, proceeds as follows. We abstract the memory state of the program using a symbolic

heap h, and we abstract the behavior of a statement s by a symbolic heap $\Delta h(s)$ which represents the prophetic writes performed by s. We note $\triangleright$ the (associative) update combinator on symbolic heaps, such that the prophetic evaluation of statement s in any abstract heap h is $h \triangleright \Delta h(s)$. For a loop `for i = 0 to N do s`, after $0 \leqslant i \leqslant N$ iterations starting in any heap h, we end up in $h \triangleright \Delta h(s[i := 0]) \triangleright \cdots \triangleright \Delta h(s[i := i])$. If the concrete evaluation agrees with the prophetic evaluation, this must be a loop invariant when taking h to be the result of the prophetic evaluation up to that point. Because we verify the equality of concrete and prophetic evaluation on each assignment, this inferred invariant is correct by construction and does not need to be checked.

In the remainder of this chapter, we assume that a signature $\mathcal{S}$ is given, with a specification as a SARE S over $\mathcal{S}$. We formulate the verification condition generator as a symbolic evaluator, using symbolic heaps whose locations are array names indexed by affine expressions of the outer variables. The values in the symbolic heaps are not referring to any mutable state, only to outer loop iterators and specification tensors, and can be considered a form of ghost state. The definition of symbolic heaps is given in the next section.

The specification deals with possibly infinite domains, hence the implementation can only implement a subset of the specification. We want to express that evaluating the implementation in a memory where the input arrays match a subset of the input tensors results in a new memory where the output arrays match a corresponding subset of the output tensors.

We will occasionally write $\mathbf{e}$ for a sequence of expressions $e_1, \ldots, e_n$. When meaningful, different sequences in the same expression can have different lengths. The length of the sequence is written $|\mathbf{e}|$.

5.2 Symbolic Values and Heaps

The commands of Sched are imperative, and do not have a type; instead, they can be described using a well-formedness judgement. In addition, this well-formedness judgement computes the prophetic evaluation of the command, as described in Chapter 1. This judgement is presented in Section 5.3 and depends on a symbolic representation of heaps and values that we now present.

Symbolic heaps are a symbolic representation of memory states. We represent symbolic heaps using the Presburger relations and operations presented in Chapter 3. The symbolic heaps can be thought of as mapping symbolic locations to symbolic values, which are presented first.

Symbolic location A *symbolic location* $\hat{\ell}$ is represented as a single-valued Presburger set containing tuples with a space of the form a/n where a is an array name. As a Presburger set, the symbolic value $\hat{\ell}$ can be evaluated in a local environment $\mathcal{E}$ to a set of integer tuples. These integer tuples can be interpreted as locations with zero or one element, denoted $[\![\ell]\!]_{\mathcal{E}}$. By abusing notations, this evaluation is defined on the space decomposition of $\hat{\ell}$:

$$[\![\bigcup_i \{a_i\langle \mathbf{x}\rangle \mid \phi_i\}]\!]_{\mathcal{E}} = \biguplus_i \{a_i[\mathbf{j}] \mid [\![\phi_i]\!]_{env+\mathbf{x}\mapsto\mathbf{j}}\}$$

Because $\hat{\ell}$ must be single-valued, at most one of the sets on the right-hand side is nonempty.

Symbolic Values A *symbolic value* $\hat{v}$ is represented as a Presburger set containing tuples with a space of the form E/n where E is an expression context with n holes. As a Presburger set, the symbolic value $\hat{v}$ can be evaluated in a local environment $\mathcal{E}$ as a set of integer tuples, where the identifiers are expression contexts. These integer tuples can be interpreted as expressions by performing the corresponding substitution. By abusing notations, we can define the evaluation of a symbolic value $\hat{v}$ in environment $\mathcal{E}$ to a set of expressions on the space decomposition of $\hat{v}$ (recall that $E[\mathbf{j}]$ denotes the substitution of the holes in E by $j_1, \ldots, j_n$):

$$[\![\bigcup_i \{E_i\langle \mathbf{x}\rangle \mid \phi_i\}]\!]_{\mathcal{E}} = \bigcup_i \{E_i[\mathbf{j}] \mid [\![\phi_i]\!]_{\mathcal{E},\mathbf{x}\mapsto\mathbf{j}}\}$$

The obtained set of expressions can further be evaluated into a set of values (which may include $\perp$) by evaluating the expressions in a model M (or memory

μ):

$$[\![\bigcup_i \{E_i\langle \mathbf{x}\rangle \mid \phi_i\}]\!]_{\mathcal{E},M} = \bigcup_i \{[\![E_i[\mathbf{j}]]\!]_{\emptyset;M} \mid [\![\phi_i]\!]_{\mathcal{E},\mathbf{x}\mapsto\mathbf{j}}\}$$
$$[\![\bigcup_i \{E_i\langle \mathbf{x}\rangle \mid \phi_i\}]\!]_{\mathcal{E},\mu} = \bigcup_i \{[\![E_i[\mathbf{j}]]\!]_{\emptyset;\mu} \mid [\![\phi_i]\!]_{\mathcal{E},\mathbf{x}\mapsto\mathbf{j}}\}$$

Unlike symbolic locations, we do not require symbolic values to be single-valued Presburger sets. This is because the same value could be represented by different expressions: for instance,

$$\{(A(\square_1) + B(\square_2))\langle 0,0\rangle; (B(\square_1) + A(\square_2))\langle 0,0\rangle\} \tag{5.1}$$

is not single-valued as a Presburger set, but its evaluation in any model is a singleton (assuming that + is a commutative operation).

Computations with symbolic values can often be easier to perform if we can assume that they are single-valued, because they can then be represented using piece-wise quasi-affine expressions. Thus, we define the function υ that takes as argument a symbolic value $\hat{v}$ and returns a single-valued symbolic value $\hat{v'}$ by arbitrarily restricting all but one of the formulas in $\hat{v}$ in case of conflicts (in particular, $\upsilon(\hat{v}) = \hat{v}$ if $\hat{v}$ is single-valued). For instance, if $\hat{v}$ denotes the symbolic value Equation (5.1), $\upsilon(\hat{v})$ can be either $\{(A(\square_1) + B(\square_2))\langle 0,0\rangle\}$ or $\{(B(\square_1) + A(\square_2))\langle 0,0\rangle\}$ (which exactly it is does not matter and is left unspecified).

In any environment where the original symbolic value *was* a singleton, the two symbolic values evaluate to the same set:

Theorem 5.2.1. *If $\hat{v}$ is a symbolic value that evaluates to a singleton $\{v\}$ in environment $\mathcal{E}$ and model M (resp. environment $\mathcal{E}$ and memory μ) then $\upsilon(\hat{v})$ also evaluates to $\{v\}$ in environment $\mathcal{E}$ and model M (resp. environment $\mathcal{E}$ and memory μ).*

If e is an expression, we can define the function decompose(e) that decomposes e as a pair $E\langle \iota_1, \ldots, \iota_n\rangle$ where E is a context with n holes and no free variables, and $\iota_1, \ldots, \iota_n$ are affine expressions following the structure of the typing judgement.

We say that a symbolic value $\hat{v}$ is *well-formed with respect to a specification* S if it evaluates to a set with zero or one element for any environment $\mathcal{E}$ and model M of the specification S. Unless otherwise stated, all symbolic values in this manuscript are assumed to be well-formed with respect to an ambient specification. In these conditions, by abuse of notation, we denote by $[\![\hat{v}]\!]_{\mathcal{E};M}$ either its single element (when it has one) or the distinguished constant $\perp$ when it is empty.

Symbolic Lifting Any construct on expressions can be lifted to symbolic values by applying the construct to the underlying expression contexts, intersecting the domains when applicable. For instance, we can compute $\hat{v_1} + \hat{v_2}$ by considering the space decompositions $\hat{v_1} = \bigcup_i \{E_i^1\langle \mathbf{e_i}^1\rangle \mid \phi_i^1\}$ and $\hat{v_2} = \bigcup_j \{E_j^2\langle \mathbf{e_j}^2\rangle \mid \phi_j^2\}$, assuming that n_i^1 and n_j^2 are the numbers of holes in E_i^1 and E_j^2, respectively:

$$\hat{v_1} \quad + \quad \hat{v_2} \qquad = \qquad \bigcup_{i,j} \left\{ \left(\begin{array}{l} E_i^1[\square_1, \ldots, \square_{n_i^1}] + \\ E_i^2[\square_{n_i^1+1}, \ldots, \square_{n_i^1+n_j^2}] \end{array} \right) \langle \mathbf{e_i}^1, \mathbf{e_j}^2 \rangle \;\middle|\; \phi_i^1 \wedge \phi_j^2 \right\}$$

In the expression $E_i^1[\square_1, \ldots, \square_{n_i^1}] + E_i^2[\square_{n_i^1+1}, \ldots, \square_{n_i^1+n_j^2}]$, the indices represent the holes of the outer expression. In other words, the list of holes of the resulting expression correspond to the holes of E_i^1 followed by those of E_i^2. In practice, it is possible to fuse the holes that can be proven equal when $\phi_i^1 \wedge \phi_j^2$ holds to get a simpler representation.

Lifting a deterministic construct to symbolic values evaluates to the set of values obtained by applying the construct to all possible combinations of values for the arguments, for instance

$$[\![\hat{v_1} + \hat{v_2}]\!]_{\mathcal{E};M} = \{v_1 + v_2 \mid v_1 \in [\![\hat{v_1}]\!]_{\mathcal{E};M} \wedge v_2 \in [\![\hat{v_2}]\!]_{\mathcal{E};M}\}$$

In particular, if all the arguments are singletons in a given environment, the result is also a singleton in said environment.

Moreover, lifting a deterministic construct to single-valued symbolic values always returns a single-valued symbolic value.

Symbolic Heaps Where Presburger sets are used to represent values, i.e. the result of a single expression, Presburger relations can be used to represent the state of the program memory, i.e. representations of mappings from locations to values. A *symbolic heap*, ranged over by $\hat{h}$, is a Presburger relation whose domain contains tuples with space a/n where a is an array name and whose range contains tuples with space E/m where E is an expression context with m holes. Multiple relations with the same domain space but different range spaces can be present in the same symbolic heap. For instance, the following symbolic heap maps even indices to the context E_0 and odd indices to the context E_1:

$$\begin{array}{llll} \{ & a\langle x\rangle & \mapsto E_0\langle x-1\rangle & \mid x \bmod 2 = 0 \quad ; \\ & a\langle x\rangle & \mapsto E_1\langle x\rangle & \mid x \bmod 2 = 1 \quad \} \end{array}$$

Symbolic locations are singleton sets, hence, the application of a symbolic heap $\hat{h}$ to a symbolic location $\hat{\ell}$ is a symbolic value, representing the value at the corresponding position in the heap. The application $\hat{h}(\hat{\ell})$ is empty if either $\hat{\ell}$ is empty, or the corresponding location is not present in the symbolic heap.

In the same way that we can evaluate a symbolic value to a set of expressions, we can evaluate a symbolic heap $\hat{h}$ to a set of (location, expression) pairs $[\![\hat{h}]\!]_{\mathcal{E}}$ in environment $\mathcal{E}$:

$$[\![\{a\langle \mathbf{x}\rangle \rightarrow E\langle \mathbf{y}\rangle \mid \phi\}]\!]_{\mathcal{E}} = \{(a\langle \mathbf{i}\rangle, E[\mathbf{j}]) \mid [\![\phi]\!]_{\mathcal{E}+\mathbf{x}\mapsto\mathbf{i}+\mathbf{y}\mapsto\mathbf{j}}\}$$

We can then evaluate the expressions in the pairs in a model M or memory μ as for symbolic values, yielding sets of (location, value) pairs for $[\![\hat{h}]\!]_{\mathcal{E};M}$ and $[\![\hat{h}]\!]_{\mathcal{E};\mu}$.

If the resulting evaluation $[\![\hat{h}]\!]_{\mathcal{E};M}$ or $[\![\hat{h}]\!]_{\mathcal{E};\mu}$ is functional (i.e. each location is associated with at most one value), we identify the result with the corresponding partial memory. We can extend the concept of well-typed to symbolic heaps by making sure that h only contains well-typed expressions:

Definition 5.2.1. A symbolic heap h is *well-typed* with respect to an environment Γ if the following conditions hold:

- For each

$$\{a\langle x_1, \ldots, x_n\rangle \to E\langle \iota_1, \ldots, \iota_m\rangle \mid \phi\}$$

in h, there is a binding $a : \tau[\iota'_1, \ldots, \iota'_m]$ in Γ such that

$$\Gamma, x_1 : \mathbb{A}, \ldots, x_n : \mathbb{A} \vdash_A E[\iota_1, \ldots, \iota_m] : \tau$$

holds

- The implication $\phi \implies 0 \leqslant x_i < \iota_i$ holds in any environment $\Gamma, x_1 : \mathbb{A}, \ldots, x_n : \mathbb{A}$ for all $1 \leqslant i \leqslant n$.

Well-typed symbolic heaps are well-typed memories when they evaluate to a functional relation.

Lemma 5.2.2. *If h is well-typed in Γ, then for any environment $\mathcal{E}$ compatible with Γ such that $[\![h]\!]_{\mathcal{E};M}$ is functional, $[\![h]\!]_{\mathcal{E};M}$ is well-typed for Γ in $\mathcal{E}$.*

Proof. The proof is mechanical by checking that the typing conditions for a symbolic heap are the symbolic versions of the typing conditions for a memory. □

We can also evaluate an expression in a symbolic heap, replacing array accesses with the corresponding expression stored in the symbolic heap. We denote the evaluation of expression e in symbolic heap $\hat{h}$ $[\![e]\!]_{\hat{h}}$. $[\![e]\!]_{\hat{h}}$ is a symbolic value, defined inductively on the structure of e in Figure 5.1.

The evaluation of an expression in a symbolic heap preserves types; however, said evaluation may be empty (e.g. if e accesses a location that is not defined in h).

Lemma 5.2.3. *If h is a well-typed symbolic heap in environment Γ and e has type τ in Γ, then for any component $\{E\langle \iota_1, \ldots, \iota_n\rangle \mid \phi\}$ of $[\![e]\!]_h$, $E[\iota_1, \ldots, \iota_n]$ has type τ in environment Γ, ϕ.*

$$
\begin{aligned}
[\![x]\!]_h &= \{\square\langle x\rangle\} \\
[\![n]\!]_h &= \{\square\langle n\rangle\} \\
[\![l]\!]_h &= \{l\langle\rangle\} \\
[\![a[\iota_1,\ldots,\iota_n]]\!]_h &= h(\{a\langle[\![\iota_1]\!]_h,\ldots,[\![\iota_n]\!]_h\rangle\}) \\
[\![f(e_1,\ldots,e_n)]\!]_h &= f([\![e_1]\!]_h,\ldots,[\![e_n]\!]_h)
\end{aligned}
$$

Figure 5.1: Evaluation in a symbolic heap

$$
\begin{aligned}
\mathrm{reads}(x) &= \emptyset \\
\mathrm{reads}(n) &= \emptyset \\
\mathrm{reads}(l) &= \emptyset \\
\mathrm{reads}(a[\iota_1,\ldots,\iota_n]) &= \{[a]\iota_1,\ldots,\iota_n\} \\
\mathrm{reads}(f(e_1,\ldots,e_n)) &= \mathrm{reads}(e_1)\cup\cdots\cup\mathrm{reads}(e_n)
\end{aligned}
$$

Figure 5.2: Evaluation in a symbolic heap

To ensure that $[\![e]\!]_h$ is defined, we define the function reads(e) to compute the set of array accesses in e interpreted as a symbolic set of locations. reads(e), defined on the structure of e in Figure 5.2, is the symbolic counterpart to the dynamic set of reads $\mathrm{rd}_{e;\mathcal{E}}(\emptyset)$:

Lemma 5.2.4. *If* $\Gamma \vdash_a e : \tau$ *holds and* $\mathcal{E}$ *is compatible with* Γ *and* $\mathrm{rd}_{\mathcal{E};\emptyset}(e)$ *is defined, then* $[\![\mathrm{reads}(e)]\!]_{\mathcal{E}}$ *is equal to* $\mathrm{rd}_{\mathcal{E};\emptyset}(e)$.

Note that $[\![e]\!]_h$ and reads(e) are only properly defined for well-typed expressions; in particular, they return bogus values for expressions containing nested array accesses (e.g. a[b[i]]). However, when expressions are well-typed, if the set of read locations are defined in h, then the evaluation of e in h is nonempty:

Lemma 5.2.5. *If* e *is well-typed in* Γ*, then* reads(e) *is a symbolic set expressed in*

Presburger arithmetic and if the inclusion of symbolic sets $\mathrm{reads}(e) \subseteq \mathrm{dom}(h)$ *is valid in* Γ*, then* $\neg\,\mathrm{empty}(\llbracket e \rrbracket_h)$ *is also valid in* Γ*.*

In particular, $\llbracket \llbracket e \rrbracket_h \rrbracket_{\mathcal{E}}$ *is never empty for any environment* $\mathcal{E}$ *compatible with* Γ*.*

Proof. The proof follows by induction on the structure of e, noting that since $\Gamma \vdash_a e : \tau$ holds, any array access is indexed by affine index expressions. □

Finally, if the evaluation of h in an environment $\mathcal{E}$ and model M is functional, evaluating the symbolic value $\llbracket e \rrbracket_h$ is the same as evaluating e in $\llbracket h \rrbracket_{\mathcal{E};M}$ directly:

Lemma 5.2.6. *If* e *is well-typed in* Γ *and* $\llbracket h \rrbracket_{\mathcal{E};M}$ *is functional for an environment* $\mathcal{E}$ *compatible with* Γ*, the following set inclusion holds:*

$$\llbracket \llbracket e \rrbracket_h \rrbracket_{\mathcal{E};M} \subseteq \{ \llbracket e \rrbracket_{\mathcal{E};\llbracket h \rrbracket_{\mathcal{E};M}} \}$$

In particular, if $\llbracket \llbracket e \rrbracket_h \rrbracket_{\mathcal{E}}$ *is nonempty,* $\llbracket \llbracket e \rrbracket_h \rrbracket_{\mathcal{E};M}$ *is equal to the singleton* $\{ \llbracket e \rrbracket_{\mathcal{E};\llbracket h \rrbracket_{\mathcal{E};M}} \}$*.*

Symbolic Update The update operator $\triangleright$ and its iterated counterpart $\blacktriangleright$ can be defined on symbolic heaps.

Definition 5.2.2. For two symbolic heaps $\hat{h}_1$ and $\hat{h}_2$, the *update* of $\hat{h}_1$ with $\hat{h}_2$ is defined as:

$$\begin{aligned} \hat{h}_1 \triangleright \hat{h}_2 &= \hat{h}_2 \cup (\hat{h}_1 - \mathrm{dom}(\hat{h}_2)) \\ &= \{ \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in \hat{h}_2 \vee \mathbf{s} \to \mathbf{t} \in \hat{h}_1 \wedge \mathbf{s} \notin \mathrm{dom}(\hat{h}_2) \} \end{aligned}$$

Remember that the update operator $\triangleright$ is defined on (partial) mappings from locations to values, where missing values are ignored. We can then prove the following adequacy lemma:

Lemma 5.2.7 (Symbolic Update)**.** *If $\hat{\mathsf{h}}_1$ and $\hat{\mathsf{h}}_2$ are symbolic heaps, then for any environment $\mathcal{E}$ and model M where the evaluations of $\hat{\mathsf{h}}_1$ and $\hat{\mathsf{h}}_2$ are functional, the evaluation of $\hat{\mathsf{h}}_1 \triangleright \hat{\mathsf{h}}_2$ in $\mathcal{E}$ and M is functional and equal to:*

$$[\![\hat{\mathsf{h}}_1 \triangleright \hat{\mathsf{h}}_2]\!]_{\mathcal{E};\mathsf{M}} = [\![\hat{\mathsf{h}}_1]\!]_{\mathcal{E};\mathsf{M}} \triangleright [\![\hat{\mathsf{h}}_2]\!]_{\mathcal{E};\mathsf{M}}$$

To define an iterated version of the symbolic update, we need to compute, for each location, the last value assigned to that location. This can be done by substituting the iterated variable with its largest value writing to the location; hence, we need to define the substitution of a variable (or variable tuple) with a relation.

Let me give an intuition of the substitution operator. Consider for simplicity a homogenous relation $\mathsf{R}_1 = \{\mathbf{t} \to \mathbf{s} : \phi\}$ with a free variable x. In the simplest case, we may want to substitute x with a value that depends on the free variables, expressed as a singleton set $S = \{[x] : x = e\}$ where x is not free and e is piecewise affine. The substitution can then be expressed as in the λ caculus as $(\lambda[x].\ \mathsf{R}_1)(S)$, except that we need to unwrap the resulting set to obtain a relation with the same shape as R_1. Unfolding the definition of relation application, we obtain:

$$\mathrm{unwrap}(\mathrm{ran}((\lambda[x].\ \mathsf{R}_1) \cap S)) = \{\mathbf{t} \to \mathbf{s} : \exists x.\ \phi \wedge x = e\}$$

This construction is defined when S is not a singleton, although it is harder to justify calling it a substitution in that case.

If the domain of the relation R_1 represents memory location or other indexing, it makes sense for the value substituted for x to depend on the location. Hence, we allow S to be a relation $\{\mathbf{t} \to [x] : x = e\}$ instead, so that the expression e can depend on the memory location $\mathbf{t}$. This requires some bookkeeping using curry and wrap to properly align the tuples: we can use $\mathrm{uncurry}(\lambda[x].\ \mathsf{R}_1)$ to obtain a domain of shape $[[x], \mathbf{t}]$ and intersect it with $\mathrm{wrap}(\mathsf{R}_2^{-1})$ that has the same shape, then curry it back to a relation of shape $\{[x] : [\mathbf{t}, \mathbf{s}]\}$. Hence, we obtain the following definition for the substitution operator:

Definition 5.2.3 (Substitution)**.** If R_1 and R_2 are symbolic relations and $\mathbf{x}$ is a variable tuple, we define the substitution $\mathsf{R}_1[\mathbf{x} := \mathsf{R}_2]$ as:

$$\mathsf{R}_1[\mathbf{x} := \mathsf{R}_2] = \mathrm{unwrap}(\mathrm{ran}(\mathrm{curry}(\mathrm{uncurry}(\lambda x.\ \mathsf{R}_1) \cap \mathrm{wrap}(\mathsf{R}_2^{-1}))))$$

Note that we define the substitution of a variable tuple with an (implicitly single-valued) relation, and not with a piece-wise expression. There is no theoretical reason for this; it just was simpler and more efficient in practice to implement it that way using the operations provided by the `isl` library. Recall that piece-wise expressions can be converted to relations, and we allow the substituted relation R_2 to be a piece-wise expression instead, by first converting it to a single-valued relation. We also allow for R_2 to be a singleton set (in which case the domain dimensions are implicitly added), or a tuple $\mathbf{x}$ of free variables interpreted as the singleton set $\{\mathbf{y} : \mathbf{y} = \mathbf{x}\}$.

The definition of the substitution using Presburger sets operations is opaque, but we can check that it satisfies the expected properties of a substitution.

Theorem 5.2.8. *If* R_1 *and* R_2 *are symbolic relations and* $\mathbf{x}$ *is a variable tuple, then in any environment* $\mathcal{E}$ *the pair of integer tuples* $\mathbf{i} \longrightarrow \mathbf{j}$ *is in* $[\![\mathrm{R}_1[\mathbf{x} := \mathrm{R}_2]]\!]_{\mathcal{E}}$ *if, and only if, there exists an integer tuple* $\mathbf{k}$ *such that* $\mathbf{i} \longrightarrow \mathbf{k}$ *is in* $[\![\mathrm{R}_2]\!]_{\mathcal{E}}$ *and* $\mathbf{i} \longrightarrow \mathbf{j}$ *is in* $[\![\mathrm{R}_1]\!]_{\mathcal{E}+\mathbf{x}\mapsto\mathbf{k}}$.

Proof. Assume that $\mathbf{i} \longrightarrow \mathbf{j}$ is in $[\![\mathrm{R}_1[\mathbf{x} := \mathrm{R}_2]]\!]_{\mathcal{E}}$. By definition of unwrap, ran and curry, this holds iff there exists a tuple $\mathbf{k}$ such that $[\mathbf{k}, \mathbf{i}] \longrightarrow \mathbf{j}$ is in $[\![\mathrm{uncurry}(\lambda x.\ \mathrm{R}_1) \cap \mathrm{wrap}(\mathrm{R}_2^{-1})]\!]_{\mathcal{E}}$. This again holds iff $\mathbf{k} \longrightarrow [\mathbf{i}, \mathbf{j}]$ is in $[\![\lambda x.\ \mathrm{R}_1]\!]_{\mathcal{E}}$ and $\mathbf{i} \longrightarrow \mathbf{k}$ is in $[\![\mathrm{R}_2]\!]_{\mathcal{E}}$, from which we conclude. □

We can now properly define the iterated update.

Definition 5.2.4 (Iterated update)**.** For a symbolic heap $\hat{\mathrm{h}}$, variable x and affine expression ι, the *iterated update* of $\hat{\mathrm{h}}$ over x up to ι is defined as:

$$\bigrhd_{0 \leqslant x < \iota} \hat{\mathrm{h}} = \hat{\mathrm{h}}[x := \mathrm{lexmax}(\mathrm{bind}_{0 \leqslant x < \iota} \left(\mathrm{dom}(\hat{\mathrm{h}})\right))]$$

In other words, $\bigrhd_{0 \leqslant x < \iota} \hat{\mathrm{h}}$ is $\hat{\mathrm{h}}$ where x is substituted, for each location, with the latest value of x at that location such that there is an expression associated with that location and value of x.

This definition of $\bigrhd_{0 \leqslant x < \iota} \hat{\mathrm{h}}$ is quite abstract, and can be better understood through a few examples.

Example 8. When there is at most one value of x that affects a given location, the lexmax computes the single value and the iterated update is simply the iterated union. Assume that $\hat{h} = \{a\langle x\rangle \rightarrow A(\square)\langle x+2\rangle : 0 \leqslant x < N\}$, which, given that x is in the context, expands to:

$$\hat{h} = \{a\langle i\rangle \rightarrow A(\square)\langle j\rangle : 0 \leqslant x < N \wedge i = x \wedge j = x+2\}$$

Hence, we have:

$$\begin{aligned} \mathrm{dom}(\hat{h}) &= \{a\langle i\rangle \mid 0 \leqslant x < N \wedge i = x\} \\ \mathrm{bind}_{0\leqslant x<4}\left(\mathrm{dom}(\hat{h})\right) &= \{a\langle i\rangle \rightarrow [x] : 0 \leqslant x < \min(4, N) \wedge i = x\} \end{aligned}$$

Now, the lexicographic maximum eliminates the variable x and computes its last value for each location:

$$\mathrm{lexmax}(\mathrm{bind}_{0\leqslant x<4}\left(\mathrm{dom}(\hat{h})\right)) = \{a\langle i\rangle \mapsto [i] : 0 \leqslant i < \min(4, N)\}$$

Hence, we finally substitute x with i at location $a\langle i\rangle$ where $0 \leqslant i < \min(4, N)$ holds:

$$\blacktriangleright_{0\leqslant x<4} \hat{h} = \{a\langle i\rangle \rightarrow A(\square)\langle i+2\rangle : 0 \leqslant i < \min(4, N)\}$$

Assuming that $N \geqslant 4$, this represents the heap:

$$\left\{ \begin{array}{ll} a[0] & \mapsto A(2) \\ a[1] & \mapsto A(3) \\ a[2] & \mapsto A(4) \\ a[3] & \mapsto A(5) \end{array} \right\}$$

Example 9. When multiple values of x touch the same location, the lexmax computes the last value of x affecting that location. Assume now that $\hat{h} =$

$\{a\langle x \bmod 4\rangle \to A(\square)\langle x+3\rangle$. This is an *unbounded* symbolic heap. If we expand the notation to make the index explicit, we get:

$$\hat{h} = \{a\langle i\rangle \to A(\square)\langle j\rangle : j = x + 3 \wedge i = x \bmod 4\}$$

We can again compute the domain then bind x:

$$\mathrm{dom}(\hat{h}) = \{a\langle i\rangle : i = x \bmod 4\}$$
$$\mathrm{bind}_{0\leqslant x<N}\left(\mathrm{dom}(\hat{h})\right) = \{a\langle i\rangle \to [x] : i = x \bmod 4 \wedge 0 \leqslant x < N\}$$

The lexicographic maximum gives us the greatest value of x associated with a given location, which can be computed using `isl`:

$$\mathrm{lexmax}(\mathrm{bind}_{0\leqslant x<N}\left(\mathrm{dom}(\hat{h})\right) = \{a\langle i\rangle \mapsto [N-1-(N-1+3i) \bmod 4] : 0 \leqslant i \leqslant \min(N,4)\}$$

Finally, we substitute x with this expression as a function of the location to obtain:

$$\blacktriangleright_{0\leqslant x<N} \hat{h} = \{a\langle i\rangle \to A(\square)\langle N+2-(N-1+3i) \bmod 4\rangle : 0 \leqslant i \leqslant \min(N,4)\}$$

Assuming that $N \geqslant 4$, this represents the heap:

$$\left\{\begin{array}{ll} a[0] & \mapsto A(N+2-(N+3) \bmod 4) \\ a[1] & \mapsto A(N+2-(N+2) \bmod 4) \\ a[2] & \mapsto A(N+2-(N+1) \bmod 4) \\ a[3] & \mapsto A(N+2-N \bmod 4) \end{array}\right\}$$

Again, we can prove the adequacy lemma :

Lemma 5.2.9. *If $\hat{h}$ is a symbolic heap and ι is an expression, then in any environment $\mathcal{E}$ and model M where $[\![\iota]\!]_{\mathcal{E}} = n \in \mathbb{Z}$ and $[\![\hat{h}]\!]_{\mathcal{E}+x\mapsto i;M}$ is functional for all $0 \leqslant i < n$, then $[\![\blacktriangleright_{0\leqslant x<\iota} \hat{h}]\!]_{\mathcal{E};M}$ is functional and equal to:*

$$[\![\blacktriangleright_{0\leqslant \iota<e} \hat{h}]\!]_{\mathcal{E};M} = \blacktriangleright_{0\leqslant i<[\![\iota]\!]_{\mathcal{E}}} [\![\hat{h}]\!]_{\mathcal{E},x\mapsto i;M}$$

Proof. $\triangleright_{0 \leqslant x < \iota} \hat{h}$ is defined as $\hat{h}[x := \text{lexmax}(\text{bind}_{0 \leqslant x < \iota} \left(\text{dom}(\hat{h})\right))]$.

Let us first make sense of $[\![\text{lexmax}(\text{bind}_{0 \leqslant x < \iota} \left(\text{dom}(\hat{h})\right))]\!]_{\mathcal{E}}$. By definition of lexmax, this is a mapping between locations ℓ and values i such that i is the largest $0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}$ such that $\ell \in [\![\text{dom}(\hat{h})]\!]_{\mathcal{E}+x \mapsto i}$.

Let us now consider an arbitrary location ℓ. If ℓ is associated with a value in $\triangleright_{0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}} [\![\hat{h}]\!]_{\mathcal{E},x \mapsto i;M}$, this value must come from the largest $0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}$ such that $\ell \in \text{dom}([\![\hat{h}]\!]_{\mathcal{E}+x \mapsto i;M})$, because for any larger i, ℓ is not in the domain of the map.

This is exactly the definition of the lexmax above, hence, if ℓ is associated with a value in $\triangleright_{0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}} [\![\hat{h}]\!]_{\mathcal{E},x \mapsto i;M}$, it is associated with the same value in $[\![\hat{h}]\!]_{\mathcal{E}+x \mapsto i;M}$ where $i = [\![\text{lexmax}(\text{bind}_{0 \leqslant x < \iota} \left(\text{dom}(\hat{h})\right))]\!]_{\mathcal{E}}(\ell)$, and hence in $\triangleright_{0 \leqslant x < \iota} \hat{h}$.

On the other hand, if ℓ is not in the domain of $\triangleright_{0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}} [\![\hat{h}]\!]_{\mathcal{E},x \mapsto i;M}$, it means that it is not associated with a value in $[\![\hat{h}]\!]_{\mathcal{E},x \mapsto i;M}$ for any of the $0 \leqslant i < [\![\iota]\!]_{\mathcal{E}}$, i.e. ℓ is not in any of the $[\![\text{dom}(\hat{h})]\!]_{\mathcal{E}+x \mapsto i;M}$.

Hence, ℓ is not in the domain of $[\![\text{bind}_{0 \leqslant x < \iota} \left(\text{dom}(\hat{h})\right)]\!]_{\mathcal{E}}$, hence neither is it in the domain of its lexmax, hence neither is it in the substitution, from which we conclude ℓ is not in $[\![\triangleright_{0 \leqslant x < \iota} \hat{h}]\!]\mathcal{E}$. □

5.3 Prophetic Evaluation

Commands are imperative, and operate by performing effects (i.e. reads and writes) on the program memory. Hence, we extend the type system of Chapter 4 with a type and effect system for commands that precisely capture the effect of the command on the program memory. This type and effect system is dubbed *prophetic evaluation* since it captures the writes to the memory expressed in terms of the specification by exploiting the prophetic annotations.

Prophetic evaluation is expressed as a judgement $\Gamma \vdash c : \Delta h$ described in Fig. 5.3, where Δh is a symbolic heap representing the set of prophetic updates performed by c. Recall that our goal is to first perform an evaluation of c by assuming that the prophetic annotations are correct, and to use its result to define a symbolic evaluator (in Section 5.4) that generates verification conditions ensuring that the original assumption (i.e. that the prophetic annotations are correct) was sound.

The prophetic evaluation of a program disregards the right-hand side of assignments, and assumes that the value computed by all assignments corresponds to the prophetic annotation on that assignment. In particular, prophetic evaluation does not depend on a program memory assigning values to implementation arrays, but only on a model assigning values to specification tensors. As an example, let us consider the specification for a matrix product:

$$\begin{aligned} R(i,j,-1) &= 0 \\ R(i,j,k) &= R(i,j,k-1) + A(i,k) \times B(k,j) \quad 0 \leqslant k < P \end{aligned}$$

and an mostly unoptimized implementation of that specification:

```
par i = 0 to N - 1 do
  for j = 0 to M - 1 do
    c[i, j] {R(i,j,−1)} := 0
    for k = 0 to P - 1 do
      c[i, j] {R(i,j,k)} := c[i, j] + a[i, k] * b[k, j]
```

This specification uses a parallel loop for the outer loop on `i` in order to demonstrate the T-ParLoop rule.

When computing the prophetic evaluation of that implementation, each statement will be associated with an application of a corresponding prophetic evaluation rule. Even for a simple program such as this, representing the full tree of the prophetic evaluation can quickly get large; instead, we can annotate each statement with the resulting symbolic heap Δh obtained after computing the prophetic evaluation of that statement. In addition, we also represent the context changes introduced by the non-leaf rules such as T-SeqLoop or T-ParLoop.

Note that while rules T-SeqLoop and T-ParLoop have exactly the same effect in this case, T-SeqLoop computes an iterated update while T-ParLoop computes

T-Allocate
$$\frac{\Gamma, a : \tau[e_1 \times \cdots \times e_n] \vdash c : \Delta h \qquad \forall 1 \leqslant i \leqslant n,\ \Gamma \vdash e_i : \mathbb{A}}{\Gamma \vdash \text{allocate } a : \tau[e_1 \times \cdots \times e_n \text{ in } c : \Delta h \backslash a}$$

T-Skip
$$\frac{}{\Gamma \vdash \text{skip} : \emptyset}$$

T-Seq
$$\frac{\Gamma \vdash c_1 : \Delta h_1 \qquad \Gamma \vdash c_2 : \Delta h_2}{\Gamma \vdash c_1 \ ; c_2 : \Delta h_1 \triangleright \Delta h_2}$$

T-If
$$\frac{\Gamma \vdash e : \mathbb{B} \qquad \Gamma, e \vdash c_1 : \Delta h_1 \qquad \Gamma, \neg e \vdash c_2 : \Delta h_2}{\Gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : (\Delta h_1 \cap e) \uplus (\Delta h_2 \cap \neg e)}$$

T-SeqLoop
$$\frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < e \vdash c : \Delta h}{\Gamma \vdash \text{for } x < e; \text{do } c : \blacktriangleright_{0 \leqslant x < e} \Delta h}$$

T-ParLoop
$$\frac{\Gamma \vdash e : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < e \vdash c : \Delta h \qquad \upsilon\left(\bigcup_{0 \leqslant x < e} \Delta h\right) = \Delta h'}{\Gamma \vdash \text{par } x < e; \text{do } c : \Delta h'}$$

T-Assign
$$\frac{\begin{array}{c}\Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \\ E\langle \iota''_1, \ldots, \iota''_m \rangle = \text{decompose}(t) \qquad a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \\ \Gamma \vdash \iota_i : \mathbb{A} \text{ for all } 1 \leqslant i \leqslant n \qquad \Gamma \vdash 0 \leqslant \iota_i < \iota'_i \text{ for all } 1 \leqslant i \leqslant n\end{array}}{\Gamma \vdash a[\iota_1, \ldots, \iota_n]\ \{t\} := e : \{a\langle \iota_1, \ldots, \iota_n \rangle \to E\langle \iota''_1, \ldots, \iota''_m \rangle\}}$$

Figure 5.3: Prophetic Evaluation of Statements

an iterated union, then removes duplicate locations. The effect is the same for matrix multiplication because each iteration of the i loop write to a distinct set of locations; however, note that this is *not* enforced by the T-ParLoop rule. This, and other race conditions, are prevented by the symbolic evaluation rules described in the next section.

Also recall that when we write a relation such as $\{c[i,j] \longrightarrow R(i,j,-1) : \}$ when i and j are in the context, i and j are the variables from the context and the relation contains a single tuple. Once we leave the corresponding loop say for j, a relation such as $\{c[i,j] \longrightarrow R(i,j,P-1) : 0 \leqslant j < M\}$ introduces a fresh local variable j, and the relation spans M different rows.

```
// Γ ← a[N × P] : float, b[P × M] : float, c[N × M] : float
par i = 0 to N - 1 do
  // T-ParLoop(i): Γ ← Γ, i : 𝔸, 0 ⩽ i < N
  for j = 0 to M - 1 do
    // T-SeqLoop(j): Γ ← Γ, j : 𝔸, 0 ⩽ j < M
    c[i, j] {R(i,j,-1)} := 0
    // T-Assign: Γ ⊢ 0 ⩽ i < N ∧ 0 ⩽ j < M
    // T-Assign: ⇒ {c[i,j] ⟶ R(i,j,-1) : }
    for k = 0 to P - 1 do
      // T-SeqLoop(k): Γ ← Γ, k : 𝔸, 0 ⩽ k < P
      c[i, j] {R(i,j,k)} := c[i, j] + a[i, k] * b[k, j]
      // T-Assign: Γ ⊢ 0 ⩽ i < N ∧ 0 ⩽ j < M
      // T-Assign: ⇒ {c[i,j] ⟶ R(i,j,k) : }
    // T-SeqLoop(k): ⇒ {c[i,j] ⟶ R(i,j',P-1) : P > 0}
    // T-Seq: ⇒ {c[i,j] ⟶ R(i,j,-1) : P ⩽ 0 ;
    //           c[i,j] ⟶ R(i,j,P-1) : P > 0}
  // T-SeqLoop(j): ⇒ {c[i,j] ⟶ R(i,j,-1) : 0 ⩽ j < M ∧ P ⩽ 0 ;
  //                  c[i,j] ⟶ R(i,j,P-1) : 0 ⩽ j < M ∧ P > 0}
// T-ParLoop(i): ⇒ {c[i,j] ⟶ R(i,j,-1) : 0 ⩽ i < N ∧ 0 ⩽ j < M ∧ P ⩽ 0 ;
//                  c[i,j] ⟶ R(i,j,P-1) : 0 ⩽ i < N ∧ 0 ⩽ j < M ∧ P > 0}
```

The typing rules for the prophetic evaluation enforce restrictions on the programs that can be expressed in Sched: conditionals, loop iterators, and array indices can only be built from affine combinations of outer loop iterators and program parameters, through the use of the affine typing judgement. On the other hand, they are overly permissive: because they are not checking that array reads are within bounds, and because out-of-bounds reads are

unexpected runtime errors, they fail to guarantee the basic property that "well-typed programs do not go wrong". This is, in a way, by design: a type system is not enough to ensure these properties. The inclusion of the verification condition generator based on symbolic evaluation presented in the next section will ensure that these erroneous behaviors do not happen. Because the type system is not strong enough to ensure the existence of a non-erroneous execution, we can only prove the following weak soundness theorem:

Theorem 5.3.1. *If* $\Gamma \vdash c : \Delta h$ *holds let* $\mathcal{E}$ *be an environment such that* $\mathcal{E} \vDash \Gamma$ *and* μ *a memory such that:*

- *The evaluation of* c *is defined, i.e. there is some* $\delta\mu$, ω *and* ρ *such that* $\mathcal{E}; \mu \vdash c \Downarrow_u \langle \delta\mu; \rho \rangle$ *holds, and*
- *In the evaluation of* c *the prophetic annotations hold in some common model* M *for all assignments*

Then, $\delta\mu$ *is equal to the evaluation of* Δh *in* $\mathcal{E}$ *and* M*, i.e.* $\delta\mu = [\![\Delta h]\!]_{\mathcal{E};M}$.

This theorem is concerned with showing that the compositional behaviors of dynamic and prophetic evaluation match: it states that the condition $\delta\mu = [\![\Delta h]\!]_{\mathcal{E};M}$ is preserved by structural induction on a program that has both a dynamic and prophetic semantic. The second hypothesis requires that this equality hold for all the base cases (i.e. the assignments) that are encountered during the dynamic evaluation, and the conclusion states that this implies the equality also hold for the full program.

Let us illustrate the theorem on the same implementation of matrix multiplication as above:

```
par i = 0 to N - 1 do
  for j = 0 to M - 1 do
    c[i, j] {R(i,j,−1)} := 0
    for k = 0 to P - 1 do
      c[i, j] {R(i,j,k)} := c[i, j] + a[i, k] * b[k, j]
```

The theorem requires that a prophetic evaluation $\Gamma \vdash c : \Delta h$ for this program exists, and that a dynamic evaluation for that program exists in memory μ

and environment $\mathcal{E} \vDash \Gamma$ with resulting updates $\delta\mu$. Moreover, the prophetic annotations must hold in the same model $\mathcal{M}$ for all the assignments. This means that there must be a shared model $\mathcal{M}$ such that for each instance of rule U-ASSIGN in the dynamic evaluation

U-ASSIGN

$$\frac{[\![e]\!]_{\mathcal{E};\mu} = v \qquad \forall 1 \leqslant i \leqslant n,\ [\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{Z} \qquad \ell = a[n_1, \ldots, n_n] \in \mathrm{dom}(\mu)}{\mathcal{E}; \mu \vdash a[\iota_1, \ldots, \iota_n]\ \{t\} := e \Downarrow_u \langle \{\ell \mapsto v\}; \mathrm{rd}_{\mathcal{E};\mu}(e) \cup \bigcup_{1 \leqslant i \leqslant n} \mathrm{rd}_{\mathcal{E};\mu}(\iota_i) \rangle}$$

the assigned value v must be equal to $[\![t]\!]_{\mathcal{E};M}$ for the shared model $\mathcal{M}$.

In the case of matrix multiplication, this means that $R(i, j, -1)$ must be 0 for all $0 \leqslant i < \mathcal{E}(N)$ and $0 \leqslant j < \mathcal{E}(M)$, and `c[i, j] + a[i, k] * b[k, j]` must evaluate to the same value as $R(i, j, k)$ for each execution of the update assignment in the dynamic evaluation.

If this is true, then we can see that the rules of the prophetic evaluation preserve the proper value associated with each location: for instance, if each iteration of the loop over k writes the value of $R(i, j, k)$ in cell, `c[i, j]`, once the loop over k is over, the value in `c[i, j]` is the last value written by the loop, i.e. $R(i, j, P - 1)$ if $P > 0$. If $P \leqslant 0$, the loop is never executed, and the value is the value of $0 = R(i, j, -1)$ that was written by the previous statement.

It should be noted that the theorem requires only the existence of some model $\mathcal{M}$ that satisfies the hypotheses without requiring any relationship between the model and the implementation. In particular, the theorem does not require any sort of general equivalence between the model $\mathcal{M}$ and the annotations in the implementation, as a different model can be selected for different values of the input parameters in $\mathcal{E}$ and/or input data in μ for the implementation.

Proof of Theorem 5.3.1. By induction on the structure of c.

Case c = skip We have $\emptyset = [\![\emptyset]\!]_{\mathcal{E};M}$ for any $\mathcal{E}$ and M

Case $c = c_1 ; c_2$ By induction hypothesis c_1 and c_2 evaluate to differential memories $\delta\mu_1$ and $\delta\mu_2$ using derivations that satisfy the prophetic anno-

tations in M. Hence, we have $\delta\mu_1 = [\![\Delta h_1]\!]_{\mathcal{E};M}$ and $\delta\mu_2 = [\![\Delta h_2]\!]_{\mathcal{E};M}$. We conclude using Lemma 5.2.7.

Case $c = a[\iota_1, \ldots, \iota_n]\ \{e'\} := e$ Since $\Gamma \vdash \iota_i : \mathbb{A}$, using Lemma 4.4.3 and Lemma 4.2.3, we have $[\![\iota_i]\!]_{\mathcal{E};\mu} = [\![\iota_i]\!]_{\mathcal{E};M} \in \mathbb{Z}$. Moreover, by hypothesis, the runtime evaluation of the right-hand side in μ matches the evaluation of the prophetic annotation in M, i.e. $[\![e]\!]_{\mathcal{E};\mu} = [\![e']\!]_{\mathcal{E};M}$. Hence we have:

$$[\![\{a[\iota_1, \ldots, \iota_n] \mapsto t\}]\!]_{\mathcal{E};M} = \{a[[\![\iota_1]\!]_{\mathcal{E};\mu}, \ldots, [\![\iota_n]\!]_{\mathcal{E};\mu}] \mapsto [\![e]\!]_{\mathcal{E};\mu}\}$$

from which we conclude.

Case $c = \text{if } \iota \text{ then } c_1 \text{ else } c_2$ Since $\Gamma \vdash \iota : \mathbb{A}$, using Lemma 4.2.3, we have $[\![\iota]\!]_{\mathcal{E};\mu} = [\![\iota]\!]_{\mathcal{E};M}$. Assuming that $[\![\iota]\!]_{\mathcal{E};\mu} = \text{true}$ (resp. false), we have $\delta\mu = [\![\Delta h_1]\!]_{\mathcal{E};M}$ (resp. $[\![\Delta h_2]\!]_{\mathcal{E};M}$) by induction hypothesis. Moreover, $[\![\Delta h_1\big|_\iota]\!]_{\mathcal{E};M} = [\![\Delta h_1]\!]_{\mathcal{E};M}$ (resp. $\emptyset$) and $[\![\Delta h_2\big|_\iota]\!]_{\mathcal{E};M} = \emptyset$ (resp. $[\![\Delta h_2]\!]_{\mathcal{E};M}$), and the result follows from rule U-If-True (resp. U-If-False).

Case $c = \text{let } x = \iota \text{ in } c'$ Since $\Gamma \vdash \iota : \mathbb{A}$, using Lemma 4.2.3, we have $[\![\iota]\!]_{\mathcal{E};\mu} = [\![\iota]\!]_{\mathcal{E};M}$ and conclude by induction hypothesis.

Case $c = \text{allocate } a : \tau[\iota_1 \times \cdots \times \iota_n] \text{ in } c'$ By induction hypothesis, we have $\delta\mu = [\![\Delta h]\!]_{\mathcal{E};M}$ hence the equality still holds when removing locations in a.

Case $c = \text{for } x < \iota; \text{do } c'$ By induction hypothesis, we have $\mu'_i = [\![\Delta h]\!]_{\mathcal{E},x\mapsto i;M}$ for all $0 \leqslant i < [\![\iota]\!]_{\mathcal{E};\mu} = [\![\iota]\!]_{\mathcal{E};M}$ since $\Gamma \vdash \iota : \mathbb{A}$. We conclude using Lemma 5.2.9.

Case $c = \text{par } x < \iota; \text{do } c'$ By induction hypothesis, the evaluations for each parallel iteration match; moreover, since $\Gamma \vdash \iota : \mathbb{A}$, the evaluations $[\![\iota]\!]_{\mathcal{E};\mu}$ and $[\![\iota]\!]_{\mathcal{E};M}$ are equal. Rule U-Par ensures that whenever the domains of $\delta\mu_i$ and $\delta\mu_j$ intersect, the corresponding values are equal. Hence, even though $\bigcup_{0\leqslant x<\iota} \Delta h$ may not be single-valued as a Presburger relation (i.e. there might be distinct *expression contexts* associated with a given location), $[\![\bigcup_{0\leqslant x<\iota} \Delta h]\!]_{\mathcal{E};M} = \bigcup_{0\leqslant i<[\![\iota]\!]_{\mathcal{E}}} [\![\Delta h]\!]_{\mathcal{E}+x\mapsto i\mapsto;M}$ is functional. In particular, whichever value is selected by υ in case of conflict still satisfies the equation.

□

5.4 Symbolic Evaluation

We introduced a type and effect system called *prophetic evaluation* for SCHED commands in the previous section that computes the expected (or asserted) evaluation of the program. We will now define a symbolic evaluator for SCHED that computes a symbolic evaluation of the program using the right-hand side of the assignments, assuming that the prophetic evaluation holds in order to break cycles introduced by sequential loops. The symbolic evaluator generates verification conditions that ensure the prophetic evaluation is correct: if the verification conditions are correct, then the hypotheses of Theorem 5.3.1 hold and the dynamic execution matches the prophetic evaluation.

The judgement $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ is presented in Fig. 5.4 and follows the dynamic evaluation rules of SCHED. The typing environment Γ and the input heap h are symbolic representations of the dynamic environment $\langle \mathcal{E}; \mu \rangle$. The pair $\langle \Delta h; R \rangle$ is a symbolic representation of the dynamic state $\langle \delta\mu; \rho \rangle$. h and Δh are represented using symbolic heaps implemented using Presburger relations as described in Section 5.2; R is represented as a Presburger set of locations. C is also implemented using a Presburger relation and represents a set of constraints (i.e. verification conditions) that must be satisfied for the prophetic evaluation to be correct, as explained below.

The symbolic evaluation makes use of some auxiliary definitions and rules, explained below.

We define the construct rw-safe(x, ι, W, R) that ensures the disjointness of the locations W written by thread x and the locations $R[x := y]$ read by a different thread y, thereby ensuring the absence of read-write races. rw-safe(x, ι, W, R) returns a Presburger formula and is defined as follows, where y is a fresh variable:

$$\text{rw-safe}(x, \iota, W, R) = \nexists \Big(\bigcup_{0 \leqslant x < \iota} \bigcup_{0 \leqslant y < \iota} (W \cap R[x := y] \cap \{x \neq y\}) \Big)$$

S-Allocate

$$\frac{\Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]; h \vdash C \implies c : \langle \Delta h; R \rangle \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{for all } 1 \leqslant i \leqslant n}{\Gamma; h \vdash C \implies \text{allocate } a : \tau[\iota_1 \times \cdots \times \iota_n] \text{ in } c : \langle \Delta h \backslash a; R \backslash a \rangle}$$

S-Skip

$$\frac{}{\Gamma; h \vdash \emptyset \implies \text{skip} : \langle \emptyset; \emptyset \rangle}$$

S-Seq

$$\frac{\Gamma; h \vdash C_1 \implies c_1 : \langle \Delta h_1; R_1 \rangle \qquad \Gamma; h \triangleright \Delta h_1 \vdash C_2 \implies c_2 : \langle \Delta h_2; R_2 \rangle}{\Gamma; h \vdash C_1 \cup C_2 \implies c_1 \,;\, c_2 : \langle \Delta h_1 \triangleright \Delta h_2; R_1 \cup R_2 \rangle}$$

S-Let

$$\frac{\Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, x = \iota; h \vdash C \implies c : S}{\Gamma; h \vdash C[x := \iota] \implies \text{let } x = \iota \text{ in } c : S[x := \iota]}$$

S-SeqLoop

$$\frac{\begin{array}{c} z \text{ fresh} \\ \Gamma \vdash \iota : \mathbb{A} \qquad \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h \triangleright \bigtriangleup_{0 \leqslant z < x} \Delta h[x := z] \vdash C \implies c : \langle \Delta h; R \rangle \end{array}}{\Gamma; h \vdash \bigcup_{0 \leqslant x < \iota} C \implies \text{for } x < \iota \text{; do } c : \langle \bigtriangleup_{0 \leqslant x < \iota} \Delta h; \bigcup_{0 \leqslant x < \iota} R \rangle}$$

S-ParLoop

$$\frac{\begin{array}{c} \Gamma \vdash \iota : \mathbb{A} \\ \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h \vdash C \implies c : \langle \Delta h; R \rangle \qquad \Gamma \vdash \text{rw-safe}(x, \iota, \text{dom}(\Delta h), R) \\ \text{ww-covered}(\Gamma, x, \iota, \Delta h) = C' \qquad R' = \bigcup_{0 \leqslant x < \iota} R \qquad \upsilon\left(\bigcup_{0 \leqslant x < \iota} \Delta h\right) = \Delta h' \end{array}}{\Gamma; h \vdash C' \cup \bigcup_{0 \leqslant x < \iota} C \implies \text{par } x < \iota \text{; do } c : \langle \Delta h'; R' \rangle}$$

S-If

$$\frac{\begin{array}{c} \Gamma \vdash \iota : \mathbb{B} \qquad \Gamma, \iota; h \vdash C_1 \implies c_1 : \Delta h_1 R_1 \\ \Gamma, \neg\iota; h \vdash C_2 \implies c_2 : \Delta h_2 R_2 \qquad C = (C_1 \cap \iota) \uplus (C_2 \cap \neg\iota) \\ \Delta h = (\Delta h_1 \cap \iota) \uplus (\Delta h_2 \cap \neg\iota) \qquad R = (R_1 \cap \iota) \uplus (R_2 \cap \neg\iota) \end{array}}{\Gamma; h \vdash C \implies \text{if } \iota \text{ then } c_1 \text{ else } c_2 : \langle \Delta h; R \rangle}$$

S-Assign

$$\frac{\begin{array}{c} a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \\ \Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{for all } 1 \leqslant i \leqslant n \qquad \hat{\ell} = a\langle \iota_1, \ldots, \iota_n \rangle \\ \Gamma \vdash \{\hat{\ell}\} \subseteq \{a\langle x_1, \ldots, x_n \rangle \mid 0 \leqslant x_1 < \iota'_1, \ldots, 0 \leqslant x_n < \iota'_n\} \qquad \Gamma \vdash \text{reads}(e) \subseteq \text{dom}(h) \\ \hat{C} = [\![e]\!]_h = \{\text{decompose}(t)\} \qquad \Delta h = \{\hat{\ell} \to \text{decompose}(t)\} \end{array}}{\Gamma; h \vdash \hat{C} \implies a[\iota_1, \ldots, \iota_n]\,\{t\} := e : \langle \Delta h; \text{reads}(e) \rangle}$$

Figure 5.4: Symbolic Evaluator

Lemma 5.4.1. *If* $\Gamma \vdash \text{rw-safe}(x, \iota, W, R)$ *holds with* $\Gamma \vdash \iota : \mathbb{A}$*, then for any environment* $\mathcal{E}$ *compatible with* Γ *and any distinct integers* i *and* j *such that* $0 \leqslant i \neq j < [\![\iota]\!]_{\mathcal{E}}$*,* $[\![W]\!]_{\mathcal{E}+x\mapsto i}$ *and* $[\![R]\!]_{\mathcal{E}+x\mapsto j}$ *are disjoint.*

To ensure that all write-write races are benign, we must ensure that conflicting writes from disjoint threads can only write the same value. To do so, we first define $\text{conflicts}(x, \iota, W)$ that computes the set of locations written to by multiple distinct threads and is defined as follows, where y is a fresh variable:

$$\text{conflicts}(x, \iota, W) = \bigcup_{0\leqslant x<\iota} \bigcup_{0\leqslant y<\iota} (\text{fst}(W) \cap \text{snd}(W)[x := y] \cap \{x \neq y\})$$

We can then express that locations that are written to by distinct threads must have a single associated value (where W is a symbolic heap) by checking that

$$\text{sv-conflicts}(x, \iota, W) = \text{sv}((\left(\bigcup_{0\leqslant x<\iota} W\right) \cap \text{conflicts}(x, \iota, W))$$

holds. This would however prevent the verification of concurrent writes of the same value using different expressions, e.g. $A(i) + B(j)$ and $B(j) + A(i)$. While we do not expect such races in practice, we can capture them theoretically using the υ function:

$$\text{conflicts-ok}(x, \iota, W) = \upsilon(\left(\bigcup_{0\leqslant x<\iota} W\right) \cap \text{conflicts}(x, \iota, W))$$

In practice, the condition $\text{conflicts-ok}(x, \iota, W)$ can be unnecessarily slow to compute. The implementation first checks whether $\bigcup_{0\leqslant x<\iota} W$ is single-valued, and then checks $\text{sv-conflicts}(x, \iota, W)$ before resorting to the computation of $\text{conflicts-ok}(x, \iota, W)$. The first check is almost always enough, because threads tend to use local arrays (with the allocate construct) for intermediate computations, and only write once to global arrays when the final result is computed. Since the local arrays are removed from the write-set when they go out of scope, they do not appear in the conditions for the outer loops.

We thus define ww-covered(Γ, x, ι, W) as a function returning a set of constraints C ensuring that when true the write-write races in W are benign:

$$\text{ww-covered}(x, \iota, W) = \begin{cases} \emptyset & \text{if } \Gamma \vdash \text{sv}(\bigcup_{0 \leqslant x < \iota} W) \\ \emptyset & \text{if } \Gamma \vdash \text{sv-conflicts}(x, \iota, W) \\ \text{snd}(\text{conflicts-ok}(x, \iota, W)) & \text{otherwise} \end{cases}$$

Lemma 5.4.2. *If* ww-covered$(\Gamma, x, \iota, W) = \hat{C}$ *with* $\Gamma \vdash \iota : \mathbb{A}$*, then for any environment* $\mathcal{E}$ *compatible with* Γ *such that* $[\![\hat{C}]\!]_{\mathcal{E};M}$ *holds and for any distinct integers* i *and* j *such that* $0 \leqslant i \neq j < [\![\iota]\!]_{\mathcal{E}}$*, then* $[\![W]\!]_{\mathcal{E}+x\mapsto i}$ *and* $[\![W]\!]_{\mathcal{E}+x\mapsto j}$ *are compatible memories.*

In rule S-Assign, the constraint $\hat{C} = [\![e]\!]_h = \{\text{decompose}(t)\}$ is a symbolic value expressed as an union of equalities. For instance, when evaluating the assignment $a[3 \cdot i]$ $\{A(3 \cdot i)\} := b[i] + c[i]$ in heap $\{b\langle i\rangle \to (B(\square))\langle i\rangle, c\langle i\rangle \to (C(\square))\langle 2{\cdot}i\rangle\}$, $[\![e]\!]_h$ is $\{(B(\square_0)+C(\square_1))\langle i, 2{\cdot}i\rangle\}$ and decompose(t) is $\{(A(\square))\langle 3{\cdot}i\rangle\}$, hence $\hat{C}$ is $\{(B(\square_0) + C(\square_1) = A(\square_2))\langle i, 2 \cdot i, 3 \cdot i\rangle\}$.

The rules for the symbolic evaluator are mostly straightforward symbolic adaptations of the dynamic evaluation rules. Furthermore, except for rule S-SeqLoop, the rules are algorithmic: the outputs Δh, R and C never appear as inputs of the inductive applications of the predicate. Thus, except for rule S-SeqLoop, the rules form an algorithm that can be computed structurally on a given program. In the case of rule S-SeqLoop, the output Δh appears as input to the recursive call inside the iterated update, which would require inventing the summary Δh of a single iteration of the loop. However, we have designed prophetic evaluation to solve this issue. If we examine the rules for symbolic evaluation, we can see that the output Δh only depends on the prophetic expressions, and ignores the right-hand side of all assignments. Furthermore, it is constructed exactly as in the prophetic evaluation:

Lemma 5.4.3. *If* $\Gamma; h \vdash C \implies c : \langle \Delta h; R\rangle$ *holds, then* $\Gamma \vdash c : \Delta h$ *also holds.*

Proof. The proof follows by induction and remarking that the rules of $\Gamma \vdash c : \Delta h$ are exactly those of $\Gamma; h \vdash C \implies c : \langle \Delta h; R\rangle$ with the premises involving h, C and R removed. □

Hence, the following strategy for the evaluation of sequential loops: first, we compute the iteration summary using the $\Gamma \vdash c : \Delta h$ judgement which does not require a precise description of the symbolic heap. Then, we plug the resulting Δh in the input heap $h \triangleright \blacktriangleright_{0 \leqslant z < x} \Delta h[x := z]$ of the symbolic evaluation judgement to compute the C, W and R. By using $\Gamma \vdash c : \Delta h$, we rely on the fact that the concrete evaluation will follow the prophetic evaluation. The constraints C are the price we pay for that: they keep track of the equalities that must hold for this property to be true, tying the knot and ensuring the well-foundedness of our approach.

The symbolic evaluation of realistic programs is hard to read manually due to the amount of annotations they provide. As an example, let us consider the same unoptimized implementation of the matrix product used to demonstrate the prophetic evaluation, whose symbolic evaluation is shown in Fig. 5.5. Recall that the specification is as follows:

$$\begin{aligned} R(i, j, -1) &= 0 \\ R(i, j, k) &= R(i, j, k-1) + A(i, k) \times B(k, j) \quad 0 \leqslant k < P \end{aligned}$$

while the implementation is:

```
par i = 0 to N - 1 do
  for j = 0 to M - 1 do
    c[i, j] {R(i,j,−1)} := 0
    for k = 0 to P - 1 do
      c[i, j] {R(i,j,k)} := c[i, j] + a[i, k] * b[k, j]
```

When computing the symbolic evaluation of that implementation, each statement will be associated with an application of a corresponding symbolic evaluation rule. This is represented by annotating each statement using comments. There are two types of comments: comments starting with a => arrow indicate the context changes performed by a rule such as S-PARLOOP when examining the body of the statement, while comments starting with a <= arrow indicate the result of the rule, including both relevant side-conditions and return values such as the set of constraints C, the differential updates Δh, and the set of read locations R.

To keep the symbolic execution readable, some side conditions are omitted, such as the conditions in S-ASSIGN that the read locations are defined in h (i.e. $\Gamma \vdash \text{reads}(e) \subseteq \text{dom}(h)$) an that the written location is well-formed in Γ are

omitted to avoid too much clutter. In addition, we (ab)use the fact that $k - 1$ is -1 when $k = 0$, and similarly that $\max(P, 0) - 1$ is $P - 1$ when $P > 0$ and -1 otherwise, in order to write the conditions more compactly. The resulting differential heaps Δh are otherwise identical as in the prophetic evaluation.

On the other hand, for demonstration purposes, the ww-covered condition in rule S-ParLoop is fully expanded to its definition using conflicts-ok, even though in this example the relation $\{c[i, j] \to R(i, j, \max(P, 0) - 1) : 0 \leqslant i < N \wedge 0 \leqslant j < M\}$ is single-valued.

The verification conditions are stored as a Presburger relation representing a set of equalities. This set of equality can then be converted to the following set of actual equalities and sent to a SMT solver such as Z3 by replacing the bound variables of the Presburger relation with universally bound quantifiers:

$$\forall 0 \leqslant i < N,\ 0 \leqslant j < M,\ R(i, j, -1) = 0$$
$$\forall 0 \leqslant i < N,\ 0 \leqslant j < M,\ 0 \leqslant k < P,\ R(i, j, k) = R(i, j, k - 1) + A(i, k) \times B(k, j)$$
$$\forall 0 \leqslant i, i' < N,\ 0 \leqslant j < M,\ i = i' \wedge j = j \Rightarrow R(i, j, \max(P, 0) - 1) = R(i', j, \max(P, 0) - 1)$$

The astute reader will remark that the verification conditions obtained here are not complete: we also need to ensure that the resulting differential heap Δh corresponds to the expected differential heap, here $\{c[i, j] \to C(i, j) : 0 \leqslant i < N \wedge 0 \leqslant j < M\}$. This can be expressed separately using the compatibility operator $\supset$, resulting in the following additional verification condition:

$$\forall 0 \leqslant i < N,\ 0 \leqslant j < M,\ C(i, j) = R(i, j, \max(P, 0) - 1)$$

In practice, as is often the case in verification tools, we obtain this extra verification condition by adding the following code at the end of the implementation depending on the sizes provided by the user:

```
...
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    __discard {C(i,j)} := c[i, j]
```

```
par i = 0 to N - 1 do
  // => S-ParLoop(i): Γ ← Γ, i : 𝔸, 0 ⩽ i < N
  for j = 0 to M - 1 do
    // => S-SeqLoop(j): Γ ← Γ, j : 𝔸, 0 ⩽ j < M
    //                  h ← h ▷ {c[i,j'] → R(i,j',max(P,0) − 1) : 0 ⩽ j' < j}
    c[i, j] {R(i,j,−1)} := 0
    // <= S-Assign: C ← {R(i,j,−1) = 0}
    //              Δh ← {c[i,j] → R(i,j,−1) : }
    //              R ← ∅
    // => S-Seq: h ← h ▷ {c[i,j] → R(i,j,−1) : }
    for k = 0 to P - 1 do
      // => S-SeqLoop(k): Γ ← Γ, k : 𝔸, 0 ⩽ k < P
      //                  h ← h ▷ {c[i,j] → R(i,j,k − 1) : k > 0}
      c[i, j] {R(i,j,k)} := c[i, j] + a[i, k] * b[k, j]
      // <= S-Assign: ⟦c[i,j]⟧h = R(i,j,k − 1)
      //              ⟦a[i,k]⟧h = A(i,k)
      //              ⟦b[k,j]⟧h = B(k,j)
      //              C ← {R(i,j,k) = R(i,j,k − 1) + A(i,k) × B(k,j)}
      //              Δh ← {c[i,j] → R(i,j,k) : }
      //              R ← {c[i,j]; a[i,k]; b[k,j]}
    // <= S-SeqLoop(k):
    //      C ← {R(i,j,k) = R(i,j,k − 1) + A(i,k) × B(k,j) : 0 ⩽ k < P}
    //      Δh ← {c[i,j] → R(i,j,P − 1) : P > 0}
    //      R ← {c[i,j]; a[i,k] : 0 ⩽ k < P; b[k,j] : 0 ⩽ k < P}
    // <= S-Seq: C ← {R(i,j,−1) = 0 : }}
    //             ∪ {R(i,j,k) = R(i,j,k − 1) + A(i,k) × B(k,j) : 0 ⩽ k < P}
    //           Δh ← {c[i,j] → R(i,j,max(P,0) − 1) : }
    //           R ← {c[i,j]; a[i,k] : 0 ⩽ k < P; b[k,j] : 0 ⩽ k < P}
  // <= S-SeqLoop(j):
  //      C ← {R(i,j,k) = R(i,j,k − 1) + A(i,k) × B(k,j) : 0 ⩽ j < M ∧ 0 ⩽ k < P}
  //        ∪ {R(i,j,−1) = 0 : 0 ⩽ j < M}
  //      Δh ← {c[i,j] → R(i,j,max(P,0) − 1) : 0 ⩽ j < M}
  //      R ← {c[i,j] : 0 ⩽ j < M; a[i,k] : 0 ⩽ k < P; b[k,j] : 0 ⩽ j < M ∧ 0 ⩽ k < P}
// <= S-ParLoop(j):
//      Γ, i' : 𝔸, 0 ⩽ i' < N ∧ i' ≠ i ⊢ {c[i,j] : 0 ⩽ j < M} # R[i := i']
//      C ← {R(i,j,k) = R(i,j,k − 1) + A(i,k) × B(k,j) : 0 ⩽ i < N ∧ 0 ⩽ j < M ∧ 0 ⩽ k < P}
//        ∪ {R(i,j,−1) = 0 : 0 ⩽ i < N ∧ 0 ⩽ j < M}
//        ∪ {R(i,j,max(P,0) − 1) = R(i',j,max(P,0) − 1) : 0 ⩽ i,i' < N ∧ 0 ⩽ j < M}
//      Δh ← {c[i,j] → R(i,j,max(P,0) − 1 : 0 ⩽ i < N ∧ 0 ⩽ j < M}
//      R ← {c[i,j] : 0 ⩽ i < N ∧ 0 ⩽ j < M; a[i,k] : 0 ⩽ i < N ∧ 0 ⩽ k < P;
//           b[k,j] : 0 ⩽ j < M ∧ 0 ⩽ k < P}
```

Figure 5.5: Symbolic Evaluation of a Matrix Product

This allows reusing the same mechanism as for the rest of the verification condition generation instead of having a special case for the outputs of the program.

5.5 Correctness proof

Let us now prove that our symbolic evaluator correctly captures the dynamic semantics of Sched as defined in Chapter 4. Fundamentally, we wnt to state that if $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ holds, then c evaluates in big-step to $\langle [\![\Delta h]\!]_{\mathcal{E};M}; [\![R]\!]_{\mathcal{E}} \rangle$ in $\mathcal{E}$ and $[\![h]\!]_{\mathcal{E};M}$ – provided that the constraints in C hold, i.e. false $\notin [\![C]\!]_{\mathcal{E};M}$.

$[\![\Delta h]\!]_{\mathcal{E};M}$ and $[\![h]\!]_{\mathcal{E};M}$ are, in general, *relations* that do not have to be functional (i.e. there might be multiple values associated with the same location), and hence the result cannot be stated in this form. We can easily prove that $[\![\Delta h]\!]_{\mathcal{E};M}$ is always functional, however we must take the functionality of $[\![h]\!]_{\mathcal{E};M}$ as an additional hypothesis.

Lemma 5.5.1. *If* $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ *holds, then* $[\![\Delta h]\!]_{\mathcal{E};M}$ *as a relation is functional in any environment* $\mathcal{E}$ *compatible with* Γ *and such that* false $\notin [\![C]\!]_{\mathcal{E};M}$.

Proof. By induction on $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$, S-Assign introduces a singleton Δh hence $[\![\Delta h]\!]_{\mathcal{E};M}$ is functional by construction, and all rules except S-ParLoop immediately preserve the functionality of the relation.

For S-ParLoop, we must have false $\notin [\![\text{ww-covered}(\Gamma, x, \iota, \Delta h)]\!]_{\mathcal{E};M}$, hence $[\![\bigcup_{0 \leqslant x < \iota} \Delta h]\!]_{\mathcal{E};M}$ is functional by Lemma 5.4.2. □

We can again easily prove that the output Δh of a symbolic evaluation is always a well-typed symbolic heap.

Lemma 5.5.2. *If* $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ *holds then* Δh *is well-typed with respect to* Γ.

By combining Lemma 5.5.1, Lemma 5.5.2 and Lemma 5.2.2, we get that the evaluation of Δh is always a well-typed memory:

Corollary 5.5.3. *If* $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ *holds, then the evaluation of* Δh *in any environment* $\mathcal{E}$ *compatible with* Γ *is a memory that is well-typed over its domain for* Γ *in* $\mathcal{E}$.

Let us now prove that our symbolic evaluator correctly captures the dynamic semantics of SCHED as defined in Chapter 4. More precisely, we are interested in the following theorem:

Theorem 5.5.4. *If* $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$ *holds where* h *is well-typed in* Γ*, then for all environments* $\mathcal{E}$ *and memory* μ *such that:*

- $\mathcal{E}$ *is compatible with* Γ
- μ *is compatible with* Γ *in* $\mathcal{E}$
- μ *contains* $[\![h]\!]_{\mathcal{E};M}$ *that hence must be functional*
- $\mathrm{false} \notin [\![C]\!]_{\mathcal{E};M}$ *(i.e. the constraints are satisfied)*

then c *evaluates in big-step to* $\langle [\![\Delta h]\!]_{\mathcal{E};M}; [\![R]\!]_{\mathcal{E}} \rangle$ *in* $\mathcal{E}$ *and* μ.

Proof of Theorem 5.5.4. The proof proceeds by structural induction on the judgement $\Gamma; h \vdash C \implies c : \langle \Delta h; R \rangle$.

S-Assign $\mathcal{E}$ is compatible with Γ hence we get $[\![\iota_i]\!]_{\mathcal{E}} = i_i \in \mathbb{Z}$ and $[\![\iota'_i]\!]_{\mathcal{E}} = n_i \in \mathbb{Z}$ for $1 \leqslant i \leqslant n$ from Theorem 4.4.1. Moreover, we get $0 \leqslant i_i < n_i$ for $1 \leqslant i \leqslant n$ from the condition $\Gamma \vdash \{\hat{\ell}\} \subseteq \{a\langle x_1, \ldots, x_n \rangle \mid 0 \leqslant x_1 < \iota'_1 \wedge \cdots \wedge 0 \leqslant x_n < \iota'_n\}$ and we get $\ell \in \mathrm{dom}(\mu)$ because μ is compatible with Γ in $\mathcal{E}$.

Since $\Gamma \vdash_a e : \tau$ holds and $\mathcal{E}$ is compatible with Γ, $\mathrm{rd}_{\mathcal{E};\emptyset}(e) = \rho$ is defined by Corollary 4.4.4, hence we also have $[\![\mathrm{reads}(e)]\!]_{\mathcal{E}} = \rho$ by Lemma 5.2.4. Moreover, since the indices ι_i are affine, we have $\mathrm{rd}_{\mathcal{E};\mu}(\iota_i) = \emptyset$ by Lemma 4.4.3.

It remains to show that $[\![e]\!]_{\mathcal{E};\mu}$ is equal to a value v, and that $[\![\{\hat{\ell} \to \text{decompose}(t)\}]\!]_{\mathcal{E};M}$ is equal to $\{\ell \mapsto v\}$, which amounts to proving that $[\![e]\!]_{\mathcal{E};\mu}$ and $[\![t]\!]_{\mathcal{E};M}$ evaluate to the same value v.

$[\![h]\!]_{\mathcal{E};M}$ is a functional relation, hence by Lemma 5.2.5 and Lemma 5.2.6 $[\![[\![e]\!]_h]\!]_{\mathcal{E};M}$ is equal to the singleton $\{[\![e]\!]_{\mathcal{E};[\![h]\!]_{\mathcal{E};M}}\}$.

We must rule out the case $[\![e]\!]_{\mathcal{E};[\![h]\!]_{\mathcal{E};M}} = \bot$. Since h is well-typed in Γ and $\Gamma \vdash e : \tau$,

Moreover, h is well-typed in Γ and e has type τ in Γ hence, by Lemma 5.2.3, any component of $[\![[\![e]\!]_h]\!]_{\mathcal{E}}$ is a well-typed prophetic expression of type τ, hence $[\![[\![e]\!]_h]\!]_{\mathcal{E};M}$ is a subset of $[\![\tau]\!]$ by Theorem 4.4.2, hence we get that $[\![e]\!]_{\mathcal{E};[\![h]\!]_{\mathcal{E};M}}$ is a value v of type τ.

Moreover, by Theorem 4.4.2, $[\![t]\!]_{\mathcal{E};M}$ is also a value v' of type τ, and the condition false $\notin [\![[\![e]\!]_h = \{\text{decompose}(t)\}]\!]_{\mathcal{E};M}$ ensures that $v = v'$, i.e. $[\![e]\!]_{\mathcal{E};[\![h]\!]_{\mathcal{E};M}} = [\![t]\!]_{\mathcal{E};M}$.

Since the evaluation of e in $[\![h]\!]_{\mathcal{E};M}$ is defined, it is unchanged in the larger memory μ, from which we conclude.

S-Skip skip evaluates to $\langle \emptyset; \emptyset \rangle$ in all environments, hence the properly holds trivially.

S-Seq All the conditions of the theorem are preserved on smaller arguments, hence by induction hypothesis we get that c_1 evaluates to $\langle [\![\Delta h_1]\!]_{\mathcal{E};M}; [\![R_1]\!]_{\mathcal{E}} \rangle$ in $\mathcal{E}$ and μ.

By Lemma 5.5.2, Δh_1 is well-typed in Γ, hence $h \triangleright \Delta h_1$ stays well-typed in Γ. In particular, the domain of $[\![\Delta h_1]\!]_{\mathcal{E};M}$ contains only valid locations in Γ, hence is a subset of the domain of μ, and $\mu \triangleright [\![\Delta h_1]\!]_{\mathcal{E};M}$ stays compatible with Γ. By Lemma 5.5.1, $[\![\Delta h_1]\!]_{\mathcal{E};M}$ is functional, hence $[\![h \triangleright \Delta h_1]\!]_{\mathcal{E};M}$ stays functional.

We conclude after applying the induction hypothesis to get the evaluation of c_2 in $\mathcal{E}$ and $\mu \triangleright [\![\Delta h_1]\!]_{\mathcal{E};M}$ into $\langle [\![\Delta h_2]\!]_{\mathcal{E};M}; [\![R_2]\!]_{\mathcal{E}} \rangle$.

S-SeqLoop By Theorem 4.4.1, we get that $[\![\iota]\!]_{\mathcal{E}} = n \in \mathbb{Z}$, and $\mathcal{E} + x \mapsto i$ is

compatible with $\Gamma, x : \mathbb{A}, 0 \leqslant x < \iota$ for $0 \leqslant i < n$.

Moreover, $[\![\Delta h]\!]_{\mathcal{E}+x\mapsto i;M}$ is functional and well-typed with respect to $\Gamma, x : \mathbb{A}, 0 \leqslant x < \iota$ for each $0 \leqslant i < n$ by Lemma 5.5.1 and Lemma 5.5.2, hence each of the $[\![h \triangleright \bigtriangleup_{0\leqslant z<x} \Delta h[x := z]]\!]_{\mathcal{E}+x\mapsto i;M}$ are functional and well-typed with respect to $\Gamma, x : \mathbb{A}, 0 \leqslant x < \iota$; moreover, the intermediate evaluations do not introduce new locations by Lemma 5.5.2.

We conclude after applying the induction hypothesis to get the evaluation of c in $\mathcal{E}$ and $\mu \triangleright [\![\bigtriangleup_{0\leqslant z<x} \Delta h[x := z]]\!]_{\mathcal{E}+x\mapsto i;M}$ into $\langle [\![\Delta h]\!]_{\mathcal{E}+x\mapsto i;M}; [\![R]\!]_{\mathcal{E}+x\mapsto i;M}\rangle$ for $0 \leqslant i < n$.

S-Let The result follows from Theorem 4.4.1 and the remark that $[\![C[x := \iota]]\!]_{\mathcal{E};M} = [\![C]\!]_{x+[\![\iota]\!]_{\mathcal{E}}\mapsto;M}$ as well as the corresponding equalities for Δh and R.

S-If From Theorem 4.4.1, we have $[\![\iota]\!]_{\mathcal{E}} \in \{\text{true}, \text{false}\}$. If $[\![\iota]\!]_{\mathcal{E}}$ is true (resp. false), $[\![(C_1 \cap \iota) \uplus (C_2 \cap \neg\iota)]\!]_{\mathcal{E};M}$ is equal to $[\![C_1]\!]_{\mathcal{E};M}$ (resp. $[\![C_2]\!]_{\mathcal{E};M}$) and $\mathcal{E}$ and μ stay compatible with Γ, ι (resp. $\Gamma, \neg\iota$).

Moreover, $[\![(\Delta h_1 \cap \iota) \triangleright (\Delta h_2 \cap \neg\iota)]\!]_{\mathcal{E};M}$ is equal to $[\![\Delta h_1]\!]_{\mathcal{E};M}$ (resp. $[\![\Delta h_2]\!]_{\mathcal{E};M}$) and $[\![(R_1 \cap \iota) \uplus (R_2 \cap \neg\iota)]\!]_{\mathcal{E}}$ is equal to $[\![R_1]\!]_{\mathcal{E}}$ (resp. $[\![R_2]\!]_{\mathcal{E}}$), hence we conclude by induction hypothesis and U-IF-TRUE (resp. U-IF-FALSE).

S-ParLoop By Theorem 4.4.1, we get that $[\![\iota]\!]_{\mathcal{E}} = n \in \mathbb{Z}$, hence $\mathcal{E} + x \mapsto i$ and μ are compatible with $\Gamma, x : \mathbb{A}, 0 \leqslant x < \iota.$ for $0 \leqslant i < n$. h stays well-typed in the extended environments since no new arrays are added, and stays functional since x does not appear in h.

Hence, by induction hypothesis, c evaluates to $\langle [\![\Delta h]\!]_{\mathcal{E}+x\mapsto i;M}; [\![R]\!]_{\mathcal{E}+x\mapsto i;M}\rangle$ in $\mathcal{E} + x \mapsto i$ and μ for all $0 \leqslant i < n$.

The side conditions of rule U-PARLOOP follows from Lemma 5.4.1 and Lemma 5.4.2, and since $[\![\bigcup_{0\leqslant x<\mathrm{iota}\, a} \Delta h]\!]_{\mathcal{E};M}$ is functional, it is equal to $[\![\Delta h']\!]_{\mathcal{E};M}$ by Theorem 5.2.1.

S-Allocate h contains no locations associated with array a, hence h stays well-typed in $\Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]$; moreover, we have $[\![\iota_i]\!]_{\mathcal{E};\mu} = [\![\iota_i]\!]_{\mathcal{E}} = n_i \in \mathbb{Z}$

by Theorem 4.4.1, hence $(\mu \backslash a) \triangleright \mu_a$ has exactly the appropriate locations associated with a to be compatible with $\Gamma, a : \tau[\iota_1 \times \cdots \times \iota_n]$.

By induction hypothesis, c evaluates to in $(\mu \backslash a) \triangleright \mu_a$ to $\langle [\![\Delta h]\!]_{\mathcal{E};M}; [\![R]\!]_{\mathcal{E}} \rangle$ and we conclude by removing the locations associated with array a.

□

5.6 Generation of prophetic expressions

We have proposed an intermediate language for a tensor compiler that is a simple imperative language with arrays and concurrent loops. This language requires the tensor compiler to output annotations, called *prophetic expressions*, that indicate, for each array write, an expression in the specification that corresponds to the value written at that location. We will show in the next chapter that this information is enough to be able to validate the output of the compiler, but is it a reasonable expectation for the compiler authors to preserve? If we want this intermediate language to be of practical use for compiler writers, it needs to be. We will see in chapter 6 that it was fairly easy to modify the Halide compiler to preserve this transformation across most compilation passes, but that is only a specific example. I argue that, for a tensor compiler that relies on a pointful specification language such as that of Halide, Tensor Comprehensions, or a SARE derivative, preserving this information does not impose undue burden on the compiler writer.

Assume that we are using a compiler for a tensor specification language. The specification is composed of a set of tensor equations $A(\iota_1, \ldots, \iota_n) = e$, and compiled down to an imperative language similar to Sched but without prophetic annotations. Further assume, for the sake of simplicity, that the compiler does not introduce intermediate storage except for values originally defined as a tensor in the specification; in other words, when the compiler generates an array assignment, the right-hand side of the assignment is derived from the right-hand side of one of the original equations. The expression can have been arbitrarily transformed through the use of algebraic rewritings, simplifications, inlining, replacement of tensor accesses by array accesses, etc. but it was originally the right-hand side of *some* tensor definition. We can

build a derived specification where we introduce a new abstract function f_A (abstract meaning here that the compiler is forced to treat f_A as a black-box without known semantic content) for each tensor A in the original specification. Each equation $A(\iota_1, \ldots, \iota_n) = e$ in the original specification is replaced with an equation:

$$A(\iota_1, \ldots, \iota_n) = f_A(e, \iota_1, \ldots, \iota_n)$$

in the derived specification.

If we implement each of the f_A as the function that returns its first argument, this new specification is semantically equivalent to the original specification. However, when compiling the modified specification with our tensor compiler, the tensor compiler does not know about the semantics of f_A, and is required to preserve the call $f_A(e, \iota_1, \ldots, \iota_n)$ in full. Hence, in the generated code, each assignment (that might be writing to an array whose dimensionality and layout has nothing to do with the original tensor A) must have as its right-hand side an $f_A(e', \iota'_1, \ldots, \iota'_n)$ obtained through arbitrary transformations from some $f_A(e, \iota_1, \ldots, \iota_n)$. Since the original $f_A(e, \iota_1, \ldots, \iota_n)$ originally appeared as the right-hand side of the defining equation for $A(\iota_1, \ldots, \iota_n)$ by construction, we can claim that if the compilation is correct, at that point in the program, the evaluation of e' must be equal to the evaluation of $A(\iota'_1, \ldots, \iota'_n)$ in the original specification. This corresponds to the prophetic expression we were looking for.

By introducing an appropriate uninterpreted function, we have shown that compilers for pointful tensor computations must already have the necessary underlying machinery to be instrumented to produce prophetic annotations. In particular, this is true of any polyhedral compiler. However, it should be noted that this does not show that those annotations can be generated through all the transformations performed by the compiler: some optimizations that were available when compiling the original specification may be prevented by the presence of the uninterpreted functions representing the tensor definitions. This is notably true if the compiler performs inlining; thus, it would be advised to keep track of the original tensor indices in an auxiliary structure for each assignment, and treat that auxiliary structure as if part of the right-hand side whenever transforming the code but ignore it if the right-hand side is inlined into another expression. This can also be an issue when dealing with vectorizing transformations as an uninterpreted function typically cannot be

vectorized, so there is still some additional work for the compiler writer to do. However, because the main mechanism for keeping track of the annotations through loop transformations is necessarily already present, the amount of work required can be expected to be fairly reasonable.

We verify that claim experimentally in the case of the Halide compiler in chapter 6 by using the approach described here. Without any prior experience with the Halide compiler source code, the author of this manuscript was able to successfully instrument the compiler and verify multiple examples from the official Halide benchmarks using the approach described in the next chapter.

Experimental evaluation 6

This section discusses the implementation of the approach described in Chapter 5 in OCaml, using bindings to the `isl` library [112] to represent affine expressions, and the Z3 SMT solver [74] to discharge the generated verification conditions.

6.1 Generation of Sched from Halide

The Halide compiler is a parameterized code generator: the schedule guides the generation of imperative code from the specification. We instrumented the Halide compiler to add prophetic annotations to the generated code, as described below. We also altered the compiler to produce a textual representation of the specification which can be parsed with our tool without having to interpret the C++ DSL.

Halide starts by generating an imperative loop nest where the arrays live in the specification index space, shifted to start at 0. We thus annotate each assignment with the stage, reduction variables, and a copy of the original tensor indices. Since transformations must preserve the semantics of the right-hand side of the assignment, subsequent transformations are applied to the annotations as if they were part of the right-hand side. We note that this approach can be applied to any compiler which generates a loop structure from the specification before possibly applying structure-preserving transformations. In particular, this is the case for polyhedral compilers.

Multidimensional arrays are eventually flattened into buffers in linear memory. We annotate accesses to the flat buffer with the original multidimensional

array indices, so that we are able to recover the multidimensional affine program. Linearized and multidimensional indices are both kept, making the linearization step independently verifiable, as discussed in Section 9.7.

The bound_small_allocations pass is disabled. This pass transforms allocations where the product of the extents in all dimensions is provably smaller than 128 bytes into an allocation of 128 bytes, the minimum allocation size of the Halide memory allocator at runtime. The resulting allocation is always one-dimensional, whereas the original allocation has the dimensionality of the original tensor. Hence, disabling this transformation makes the conversion of the output of the instrumented Halide compiler to SCHED easier.

Most of the technical difficulties in the instrumentation of the Halide compiler comes from its handling of vectorized loops, which I now explain. Consider the following specification that computes tensor B by doubling the values in tensor A:

$$B(i,j) = 2 \times A(i,j)$$

With the schedule `B.vector(j, 4)`, and assuming that $M \bmod 4 = 0$ for simplicity, this is first lowered internally by the non-instrumented version of the Halide compiler to the following intermediate representation (where vector is a special case of parallel loops representing a vectorized loop):

```
for i = 0 to N do
  for j0 = 0 to M/4 do
    vector j1 = 0 to 3 do
      let j = j0 * 4 + j1 in
      b[i * M + j] := 2 * a[i * M + j]
```

The vector loop is then transformed into a ramp intrinsic encoding all the indices of the vectorized loop in a single assignment, leading to the following intermediate representation:

```
for i = 0 to N do
  for j0 = 0 to M/4 do
    b[ramp(i * M + j0 * 4, 1, 4)] :=
        2 * a[ramp(i * M + j0 * 4, 1, 4)]
```

$\text{ramp}(b, s, n)$ represents the list of indices $b + x \cdot s$ where $0 \leqslant x < n$ is an implicit

variable representing the lane index. Halide relies on LLVM to transform this intrinsic into hardware vector instructions when possible.

In the instrumented version of the compiler, we keep track of the *multidimensional* array indices in addition to the linearized indices used by the Halide compiler. Hence, the intermediate representation using the vector loop looks as follows (the original linearized indices are omitted for readability):

```
for i = 0 to N do
  for j0 = 0 to M/4 do
    vector j1 = 0 to 3 do
      let j = j0 * 4 + j1 in
      b[i, j] := 2 * a[i, j]
```

When transforming vector into ramp, some care needs to be taken to properly handle the multidimensional indices to obtain the correct program below, where `x4(e)` is a shorthand for `replicate(e, 0, 4)`:

```
for i = 0 to N do
  for j0 = 0 to M/4 do
    b[x4(i), ramp(j0 * 4, 1, 4)] :=
        2 * a[x4(i), ramp(j0 * 4, 1, 4)]
```

When reading the output of the Halide compiler, the verifier then translates *back* the ramp representation of the Halide compiler into a Sched program using par loops, by explicitly re-introducing the implicit lane index around assignments involving ramp or xN constructs:

```
for i = 0 to N do
  for j0 = 0 to M/4 do
    par lane = 0 to 3 do
      b[i, j0 * 4 + 1 * lane] :=
          2 * a[i, j0 * 4 + 1 * lane]
```

ramps can also appear in the prophetic annotations (not represented here), and must be handled there in a similar manner. Properly keeping track of the implicit lane indices when performing transformations involving ramps is more complex than just performing substitutions, because it involves adjusting the ramps in non-obvious ways. Most of the non-trivial issues encountered in properly threading the annotations through the compiler pipeline were due to proper

handling of ramp-related transformations, in particular when shuffling (reordering the indices in a vector, possibly changing the total size if some indices are missing or repeated) is involved. As a result, the rewrite_interleavings and flatten_nested_ramps transformation passes are disabled in the experiments.

6.2 OCaml prototype

Our tool takes the annotated output generated from the Halide compiler and rebuilds a Halide algorithm and a SCHED candidate implementation from that output.

We first convert all the indices and statement-level conditonals into piece-wise quasi-affine functions represented using isl, and simplify them based on the context. `ramp`-based vectors are transformed into parallel loops with an explicit index. The code generated by Halide often features two "accidentally non-affine" constructs, which we convert into an equivalent affine representation. First, when multiple dimensions of sizes $e_1, \ldots, e_n$ are parallelized, Halide uses a single parallel loop with size $e_1 \times \cdots \times e_n$, which we recognize and split into nested parallel loops of affine sizes. Second, Halide can generate expressions of the form $\lfloor (e \times e' - 1)/e \rfloor$, which is always equal to $e' - \mathrm{sgn}(e)$ provided e is non-zero. Since Halide's simplifier fails to do so, we recognize and simplify this pattern.

After this initial conversion phase, our tool implements the algorithms described in Chapter 5. We rely on isl to represent symbolic heaps, domains, and ranges. The coalescing operation provided by `isl` has proven effective to keep simple representations of the symbolic heaps as they get updated. `isl` uses parametric integer programming [36] to perform efficient quantifier elimination for the union and lexicographic maximum operators. Symbolic operations and simplifications on Presburger arithmetic is a unique asset of `isl` — tailored to the needs of polyhedral compilation; these operations would be much more cumbersome and inefficient to reproduce using Z3 (or any feasibility-focused solver).

We note two optimizations that we make in the representation to improve efficiency. In order to represent symbolic sets and heaps where the right-hand

side expression does not have to be affine, we transform every such expression into a template where each index of a tensor is replaced with an affine hole. The templates are then de-duplicated. Similarly, to represent the set of constraints C, we make the observation that in a correct compilation, the indices in the right-hand side (the implementation expression) of the equalities generated by rule S-Assign must be deducible only from the indices in the left-hand side (the specification expression). Hence, we compute an expression of the former as a piece-wise quasi-affine function of the latter, and only store the set of specification indices, thereby reducing the dimensionality of the set.

Once the toplevel set of constraints C has been computed, we generate Z3 queries to prove the corresponding equalities. The translation to Z3 is mostly straightforward. We first encode the Halide algorithm into a SARE, which are then encoded using the define-funs-rec facility [12]. The well-formedness constraint on Halide algorithms ensures that these mutually recursive functions are well-defined, but it is not checked. Z3 has specialized support for such recursive functions which is typically more efficient than a direct encoding using quantifiers. Each constraint in C is a set of equalities which can then be expressed directly in Z3's logic. Note that thanks to the coalescing and simplifications performed by `isl`, there are typically fewer constraints to be verified than assignments in the program, because unrolled statements yield the same constraint. In addition, we observe that the domain of the formula is often, but not always, simple. It might be worth investigating whether the coalescing logic of `isl` can be improved to better coalesce the sets produced by our tool. In the outer product example of Chapter 1, the constraint is exactly $\forall 0 \leqslant i < N, 0 \leqslant j < M. A(i) \times B(j) = B(j) \times A(i)$.

Finally, we perform additional processing before sending the generated equalities to Z3, leveraging polyhedral checks on indices using `isl` in a best-effort strategy. Namely, before sending an equality $e_s = e_i$ where e_s comes from a prophetic annotation and e_i was inferred from the implementation:

- We unfold the tensors in e_s which do not appear in the transitive dependences of the tensors in e_i, excluding cycles.

- For each access a_s in e_s and a_i in e_i to the same tensor, if the indices are equal, we replace both accesses with the same let-bound variable to either a_s or a_i.

In most cases, after these simplifications, the equality contains the same accesses syntactically on both sides of the equality, helping Z3 focus on value-level reasoning rather than on resolving tensor indexings.

6.3 Benchmark selection

Halide has a large benchmark suite in the `benchmarks/` subdirectory of its repository. Some are out of the scope of this thesis due to containing non-affine specifications, including those with data-dependent accesses and histograms. Others use unsupported features, e.g. assigning an undefined value to simulate in-place input updates. We have run our tool on the remaining benchmarks, using the schedule provided in the original benchmark suite. This covers about 25% of the Halide benchmark suite, not counting the variations that some benchmarks have (e.g. transposed matrix multiplication). The benchmarks have implicit assumptions on the required input sizes (e.g. that some dimensions are multiples of 16 or 32), implied by the scheduling directives used. They do not appear in the specification, but Halide generates runtime checks for them. These assumptions are given as contextual axioms to the verifier.

The benchmarks can be roughly separated into two application domains: linear algebra, including the convolution operators of deep learning, and image processing.

From the linear algebra domain, we consider the following benchmarks:

- sdot is a simple dot product manually implemented with a tiling factor of 8. This is expected to be easy to verify.

- sgemm is a general matrix-matrix multiply on floats, from the linear_algebra Halide application. It uses an optimized CPU schedule and specializations for small and large matrices. As a compute-intensive program, matrix-matrix multiply requires precise optimizations to get good performance: as such, this benchmark features heavy loop transformations and is a good stress test of our verifier as far as linear algebra benchmarks go. The sgemmTA and sgemmTB are variants where one of the input matrices is transposed. The Halide specification has a bug in these cases,

and assumes that matrix A is square. As such, we expect the verification to fail.

- cmm1024 is another matrix-matrix multiplication implementation, tuned for GPUs. This Halide benchmark is written for 1024×1024 square matrices, and we verify it for this concrete size only.

- conv_layer is a 2D convolution layer.

- dsc is a depthwise separable convolution. The specification contains non-affine indices, hence we use a constant grouping factor of 3, eliminating the non-affine component.

From the image processing domain, we consider the following benchmarks:

- blur is a two-dimensional blur filter, performing an average of three neighbors in each dimension. This example features storage folding: the schedule only stores four lines of the intermediate tensor at once in a rolling buffer.

- sc is a chain of large stencils of width 25 (i.e. each stage reads from 25 distinct neighbors from the previous stage). The default depth (i.e. number of stages) for the benchmark is 32, which is reported as sc32; we also include a variant sc1 with a single stage.

- harris, unsharp, and nl_means are implementations of image processing algorithms, namely the Harris corner detector, unsharp masking, and non-local means.

In most cases, the Halide compiler applies algebraic transformations such as associativity and commutativity to the expressions in the specification. We represent signed and unsigned integer types using Z3's native representation based on bit-vectors. For floating-point numbers, we provide three possible encodings, discussed in Section 9.4. In most cases, we simply encode floats as real to capture the transformations which Halide makes under "fast-math" assumptions. For harris, unsharp and nl_means, Halide performs constant propagation in the floating point domain, which we cannot validate when

interpreting floats as reals. As such, we run Halide in "strict float" mode for these benchmarks, disabling floating point optimizations.

6.4 Evaluation

Table 6.1 shows that our translation validation system succeeds on 14 of the 17 examples, with running times from one second to 5 minutes for the successful examples. Two of the remaining examples are expected to fail due to a bug in the specification. The last example reaches a timeout of 15 minutes. To put these results in perspective, we ran the same examples through the ISA tool from Verdoolaege, Janssens, and Bruynooghe [116]. ISA is only able to verify 7 of the examples, and times out on 8 including one of the incorrect examples. Of the three remaining examples, one is correctly shown to be incorrect, while the others cannot be proven by ISA due to using simplifications beyond associativity and commutativity.

Unlike our approach, ISA is fully automatic, and does not rely on annotations. ISA is able to handle associative and commutative operators, but it is optional as it increases its runtime. We indicate with superscripts when ISA was allowed to use associativity (A) or commutativity (C) of an operator.

ISA takes C programs as inputs, which we generate from SCHED by erasing the annotations. Halide can directly generate C code with linearized (non-affine) accesses and using vectorization primitives not supported by ISA. We also convert the specification to a C program, by creating a different loop nest for each assignment. A simple data-flow analysis is used to infer the bounds. Alternatively, we could compare the optimized Halide schedule with Halide's default schedule.

ISA takes the form of three command-line tools. `c2pdg` converts the C program into a polyhedral representation. To disable overflow checks, we pass the option `--pet-signed-overflow=ignore` to `c2pdg`. `da` performs a data-flow analysis on the polyhedral representation. `eqv` performs the equivalence checking on the `da` output of two programs. We report the run-times of the different tools separately.

The results of our experimental evaluation are summarized in Table 6.1. The benchmarks have been run on a machine with an Intel® Core™ i9 – 9900 CPU, with a timeout of 15 minutes. For isa, the timeout applies separately to each step. For each benchmark, we indicate the run-time of each tool in seconds, as well as whether the equivalence was successfully proved. For isa, the `c2pdg` and `da` timings include the sum of times for both the implementation and specification.

Table 6.1: Results of the experimental evaluation (times in seconds)

	isa				Ours	
	c2pdg	da	eqv			
blur	<1	<1	$1.1^{\mathrm{AC+}}$	✓	<1	✓
cmm1024	<1	1.2	10	✓	2.6	✓
sgemm1024	<1	2.5	$27.3^{\mathrm{C\times}}$	✓	1	✓
sqsgemm	2.5	4min34	$>15\mathrm{min}^{\mathrm{C\times}}$	?	15.1	✓
bigsqsgemm	1.6	17.5	$8\mathrm{min}12^{\mathrm{C\times}}$	✓	2.7	✓
sgemm	7.	>15min	N/A	?	3min24	✓
bigsgemm	2.8	1min19	>15min	?	12.1	✓
sc1	3.44	4m20	3.3	✗	12.5	✓
sc32	1min41	>15min	N/A	?	4min53	✓
dsc	10.2	13min49	>15min	?	1min43	✓
conv	2.	12.2	27.9^{Cmax}	✓	2min12	✓
sdot	<1	<1	<1	✓	<1	✓
harris*	7.9	31.5	1min8	✓	44.8	✓
unsharp*	1.3	6.1	13min44	✗	6.1	✓
nl_means*	>15min	N/A	N/A	?	>15min	?
sgemmTA†	21.5	>15min	N/A	?	10.4	✗
sgemmTB†	21.1	N/A	N/A	✗	34.7	✗

*No floating-point optimizations

†Expected to fail

When both our tool and ISA successfully validate the implementation, our tool has comparable or better performance, except on the conv benchmark. This is because for the conv benchmark, the affine conditions for the equalities sent to Z3 are quite complex. The conversion from the internal representation of isl to Z3 for sets with many disjuncts is a bottleneck of our approach, and in this specific case we suffer from a suboptimal implementation which

does multiple conversion to and from the internal representation of isl. In addition, our tool is able to prove more cases than ISA. In the case of parametric matrix multiplications (sqsgemm and sgemm), ISA reached a timeout of 15 minutes. Parts of these kernels are specialized, with different implementations depending on the values of the parameters. The bigsqsgemm and bigsgemm entries present results when the size of the matrices are larger than 512: by allowing the pruning of some branches, this allows ISA to complete the verification. Note that this also drastically decreases runtime of our own algorithm.

Finally, let us mention the nl_means benchmark, which we fail to verify due to running out of time. Like in conv, Halide generates complex affine conditions involving many minimums and maximums: in consequence the `isl` representation of symbolic heaps contains many disjuncts, hurting the performance. In fact, our current implementation times out during the initial simplification of affine expressions. This could be mitigated by including more contextual information during the simplification: manual experiments on some expressions extracted from that benchmark indicate that it could result in up to an order of magnitude less disjuncts. Another avenue to explore would be to find independent piece-wise expressions that can be abstracted and factored out to reduce the number of disjuncts.

7 Verifying reductions

A reduction is the iterated application of a binary function or operator on the elements of a sequence of values, ultimately *reducing* them down to a single value. The most common example of reductions is that of the summation operator $\sum$ iterating the addition operator + over a sequence of values. As mathematicians know, the result of a summation does not depend on the order in which it is performed, and the summation can be split off arbitrarily into partial sums. These properties follow from the associativity and commutativity of +, and can be exploited in various ways to simplify computations — for instance to reveal cancelling pairs or to factor out repeated terms.

A tensor compiler rarely performs the type of simplifications mathematicians do. They do, however, exploit the ability to change the order in which a reduction is computed in order to better optimize for cache locality or to reveal hidden parallelism [49]. When implemented sequentially, reductions can be the bottleneck of otherwise well-optimized programs: for instance, the stopping criterion of an algorithm iterating until convergence in a vector space is obtained through a summation (to compute a tensor norm). The detection, modeling, and optimization of reductions in array programs has thus been a long-lasting topic of research since the early days of parallel computing, and is still an important component in a tensor compiler's toolkit.

The re-ordering of computation within a reduction performed by a tensor compiler cannot be expressed directly in the validation framework using prophetic annotations introduced in chapter 5, because that framework assumes a sequential representation of reductions. Expressing the new intermediate values computed by a re-ordered reduction cannot be done using the specification language without introducing new equations in the specification.

Polyhedral compilers have specific representations of reduction statements

that interact differently than regular statements with the framework in terms of ordering. Similarly, in this chapter I propose an extension to the SCHED language and the validator of chapter 5 that introduce explicit reduction operators in the specification, and augment the prophetic annotations to validate re-orderings of the underlying computations. The new annotations work by mapping accumulating assignments to the corresponding position in the original reduction.

The extensions presented here have not been implemented in the validator of chapter 6.

7.1 Parallel Implementations of Reductions

Let us start by looking at the parallel implementation of reductions that can be performed by tensor compilers, in order to have a better grasp of the patterns that our validator must be able to handle. To the imperative programmer, a reduction is simply a sequential for loop that repeatedly applies the reduced operator. For instance, the following imperative program computes the sum $\sum_{0 \leqslant i < N} a[i]$ of array a into the variable r used as an accumulator:

```
r = 0
for i = 0 to N - 1 do
  r = r + a[i]
```

This program, as written, is inherently sequential. The value of r at one iteration of the loop explicitly depends on its value at the previous iteration of the loop. A deeper look at the program reveals parallelization opportunities when + is associative. To understand why, let us consider the computation performed by the program when N = 8: it is the leftward summation

+ is not associative, famously, for floating-point numbers — and tensor operations often operate on floating-point numbers. We ignore the issue for now and discuss it in section 9.4.

$$r = ((((((a_0 + a_1) + a_2) + a_3) + a_4) + a_5) + a_6) + a_7$$

where the initial addition with 0 has been omitted for the sake of the argument.

In this expression, since + is associative, we can change the nesting of the parentheses without changing the result. A balanced nesting naturally corresponds

to a "divide-and-conquer" algorithm: both left- and right- hand sides at each level can be computed independently, then combined back into the result

$$r = ((a_0 + a_1) + (a_2 + a_3)) + ((a_4 + a_5) + (a_6 + a_7))$$

There is a time-memory tradeoff: distinct memory locations must be used to store partial sums computed concurrently.

Actual hardware having a finite number of processors, performance-sensitive code is tuned to an efficient balance of computation between the number of available processors. On a machine with 4 processors, for instance, the following implementation assigns a quarter of the computation to each processor:

```
par i = 0 to 3 do
  tmp[i] = 0
  for j = 0 to min(floor(N / 4),
                   N - i * floor(N / 4)) - 1 do
    tmp[i] = tmp[i] + a[i * floor(N / 4) + j]
result = (tmp[0] + tmp[1]) + (tmp[2] + tmp[3])
```

In general, this implementation has complexity $O(n/p + p)$ where n is the size of the array and p the number of processors.

Unfortunately, this implementation is not an affine program because of the $i\lfloor N/4 \rfloor$ terms that cannot be expressed in Presburger arithmetic. Hence, this implementation is out of reach both of our verifier and of affine compilation techniques such as polyhedral compilers. Yet, the polyhedral model is able to parallelize reductions: the key insight is that if computation is assigned to processors in a round-robin fashion instead of by consecutive chunks, we assign a strided segment of the original array to each processor, which can be expressed using a modulo condition. The following implementation is indeed (quasi-)affine:

```
par i = 0 to 3 do
  tmp[i] = 0
  for j = 0 to floor((N - i - 1)/4) do
    tmp[i] = tmp[i] + a[4 * j + i]
result = (tmp[0] + tmp[1]) + (tmp[2] + tmp[3])
```

Assigning the computation to processors in round-robin fashion requires the underlying operator to be commutative: we have effectively re-ordered the computation so that for $N = 8$ we are now computing

$$r = ((a_0 + a_4) + (a_1 + a_5) + (a_2 + a_6) + (a_3 + a_7))$$

The most common reduction operators in scientific computing are addition, maximum, and minimum — all of which are both associative and commutative. Polyhedral compilers can only perform this second type of transformation using strided instead of consecutive chunks. However, non-polyhedral compilers such as Halide can still perform the original transformation, which cannot be represented using affine constructs. This is a particular case of loop tiling, discussed in section 9.6; in the rest of this chapter, we will assume that the compiler transforms reductions using strided chunks.

The IEEE 754-1985 and IEEE 754-2008 specifications allow for the result of $\min(0, -0)$ *and* $\max(0, -0)$ *to be either* 0 *or* -0 *at the choice of the implementation, because* 0 *and* -0 *compare equal. The recent IEEE 754-2019 specification requires that* -0 *be considered less than* 0 *for the purpose of* min *and* max*, making both operations well-defined, associative and commutative on non-NaN values.*

One final remark is that when the reduction operator is both associative and commutative, arbitrary re-ordering and re-parenthesizing of the computation is allowed, making the result depend only on the (multi)set of values in the reduction. This creates a new parallelization opportunity: the reduction loop for an associative-commutative reduction can be implemented with a parallel loop provided that the reduction operator is implemented using atomic update operations (typically using primitive atomic operations such as an atomic fetch-and-add instruction, although a lock-based implementation could also be used). This can be combined with the previous transformations; for instance, partial sums can be assigned to different compute units that then use atomic operations to implement the final stage of recombining the results.

Parallel reductions in the polyhedral model In the polyhedral community, early approaches to handle reductions were ad-hoc: after scheduling the program without taking reductions into consideration, approaches such as that of Jouvelot and Dehbonei [55] or [86] are applied *after* polyhedral scheduling to detect reductions. Reductions are then optimized separately using program rewriting techniques when applicable. Pugh and Wonnacott [80] introduced *reduction dependences* to model reductions: a reduction dependence between statement instances u and v indicate that u and v access the same memory location but can be freely reordered. To accommodate this possible reordering, instances that depend on the final value of the reductions must depend on all previous statement instances taking part in the reduction instead. This approach has been implemented in production compilers such as that of Doerfert et al. [35]. While most approaches involve pattern-matching techniques of various complexities to detect the reductions in the first place, Reddy, Kruse, and Cohen [85] propose an extension to the PENCIL language of Baghdadi et al. [6] with explicit annotations indicating the initialization and accumulation statements

in a reduction, allowing to express reductions using arbitrary operators and to focus on scheduling rather than detection of reductions. Reductions are modelled using a virtual "merge" statement that depends on all instances of the update statement.

Parallel Reductions and SAREs Redon and Feautrier [87, 88] extend the SARE representation with explicit reductions and scans (i.e. reductions that keep all intermediate results) operators. Reductions are scheduled atomically in the PRAM model: the whole reduction is represented as a single statement with all its dependencies, assuming a sufficiently large amount of parallelism to compute the reduction in one step. Gupta, Rajopadhye, and Quinton [45] extend that work by assigning a variable duration to the reduction operation depending on the ratio of the reduction size and the number of available processors, taking into account the multiple steps necessary to compute the reduction.

Parallel Reductions and Halide Halide has no syntactic support for reductions in the specification language other than the syntaxic sugar for operator assignment such as +=. Reductions are defined like other sequential constructs using recurrence variables in an update definition. If Halide can prove that the update can be expressed using an associative and commutative operator, the iteration order of the recurrence variable can be changed arbitrarily using the same `split` and `reorder` that are used for pure variables. If the update is made atomic, the corresponding loop can also be parallelized. Finally, Halide supports the partial parallelization of associative-commutative reductions using the `rfactor` primitive. Unlike other scheduling directives, however, `rfactor` modifies the *algorithm* directly, explicitly introducing new tensors to hold the intermediate results.

The handling of parallel reductions in Halide is described by Suriana, Adams, and Kamil [103].

Parallel Reductions and Combinators Combinator-based frameworks are fairly different from the type of imperative representation we are considering in our validator. Their functional nature makes the representation of reductions

fairly easy: all approaches incorporate a primitive reduce operator that is expected to be used to write reductions. Transformations on reductions are expressed similarly to any other transformations using appropriate rewrite rules, and do not necessitate special treatment.

7.2 List Homomorphisms

In the previous section, we made the remark that re-parenthesizing expressions involving associative operators correspond to parallelizing their computation. This remark is well-known in the parallel programming community since its early days, and has been formalized using the notion of *list homomorphism* [18].

Fold can naturally be defined on many recursive data structures other than lists, but that is out of scope of the current discussion.

To understand list homomorphisms, let us first go back to the fold or reduce of functional programming. fold is a function on lists which comes in two variants, the *left fold* foldl and the *right fold* foldr. Both iterate over the list and accumulate an operator over the elements of the list. If $l = [e_1, \ldots, e_n]$ is a list, then:

$$\text{foldl}(\odot_l, a, l) = ((a \odot_l e_1) \odot_l \ldots) \odot_l e_n$$
$$\text{foldr}(\odot_r, a, l) = e_1 \odot_r (\cdots \odot_r (e_n \odot_r a))$$

The recursive definitions of both foldl and foldr can be given as follows, here in OCaml:

```
let rec foldl op a = function
  | [] -> a
  | x :: xs -> foldl op (op a x) xs

let rec foldr op a = function
  | [] -> a
  | x :: xs -> op x (foldr op a xs)
```

If we denote ++ the concatenation operator on lists, a function f is called a *left fold* if it can be implemented using the foldl function with a given initial value and operator, and a *right fold* if it can be implemented using the foldr function with a given initial value and operator. Left and right folds can be

characterized equationally: a function f is a left fold if, and only if, it satisfies the following equations for some initial value a and binary operator $\odot_l$

$$f([]) = a$$
$$f(x +\!\!+ [y]) = f(x) \odot_l y$$

and it is a right fold if, and only if, it satisfies the following equations for some initial value a and binary operator $\odot_r$

$$f([]) = a$$
$$f([x] +\!\!+ y) = x \odot_r f(y)$$

In imperative terms, a left fold is a function that can be implemented as a forward sequential loop over a list (or array), and a right fold is a function that can be implemented as a backwards sequential loop over the list (or array). In a left or right fold, there is an implicit order that forces the evaluation in one direction or the other. A reduction should not possess that forced order; instead, we can define a list homomorphism h to be a function on lists such that there exists an operator $\odot$ making the following equation true, for all pairs of lists l_1 and l_2:

$$h(l_1 +\!\!+ l_2 = h(l_1) \odot h(l_2)$$

Since $+\!\!+$ is associative, and $[]$ is a neutral element for $+\!\!+$, $\odot$ is necessarily associative and admits $h([])$ as a neutral element. It is easy to verify that a list homomorphism is both a left fold and a right fold; perhaps more surprisingly, any function that is both a left and a right fold is a list homomorphism.

Some authors do not require h to be defined on empty lists; for the sake of simplicity, we will assume that list homomorphisms are total functions.

Finally, when the operator of a list homomorphism h is commutative, we have the equality $h(x +\!\!+ y) = h(y +\!\!+ x)$ for all lists x, y. This makes the proper data structure to represent reductions with a commutative operator that of a multiset: sets with multiplicity, or equivalently, lists without order. Let us call such a list homomorphism a *multiset homomorphism*.

An interesting remark is that if h is a multiset homomorphism, it can be seen as a set homomorphism, where sets are equipped with the partial operator of

disjoint union: it holds that $h(x \uplus y) = h(x) \odot h(y)$ where $\uplus$ denotes the disjoint union.

There is a natural hierarchy between folds, list homomorphisms, and multiset homomorphisms: any multiset homomorphism is also a list homomorphism, and any list homomorphism is also a fold. In each case, the restrictions come with additional properties on the parallelization opportunities. In the rest of this chapter, I will focus on multiset homomorphism, i.e. associative and commutative reductions: they are both the most common type of recurrences in scientific code, and the ones that enable the most parallelization opportunities.

7.3 Implementing Reductions

Let us revisit the summation example from Section 7.1. Equipped with our understanding of reductions as list homomorphisms, we know that the summation is not only a left fold as originally written but also a list homomorphism, because + is associative — and even a (multi)set homomorphism, because + is also commutative. In order to be able to incorporate the validation of the transformations presented in Section 7.1 into the framework of Chapter 5, we must make one final remark: namely, that the reduction can not only be expressed as a (multi)set homomorphism on the multiset of reduced values, but also as a *set homomorphism* on the *set of reduced indices*, by defining the set homomorphism h such that:

$$\begin{aligned} h(\{\}) &= 0 \\ h(\{i\}) &= A(i) \\ h(x \uplus y) &= h(x) + h(y) \end{aligned}$$

This is true because we are using a single iteration over the index i to compute the reduction. This is true for reductions defined using Halide's recurrence variables (where each value in the reduction domain is computed exactly once), or the reductions computed using variadic operators such as $\sum$: ultimately, the reduction is defined over a set of indices and accumulates exactly once for each of

those indices. Note that the reduced set of indices is not necessarily directly used as index in an array or tensor: for instance, the sum $\sum_{0 \leqslant i < N} A(\lfloor i/2 \rfloor)$ iterates over the set of indices $\{0 \leqslant i < N\}$ but reads each of the $\{A(i) \mid 0 \leqslant i < \lfloor N/2 \rfloor\}$ exactly twice (if N is even).

In order to represent reductions, I propose to reconstruct a mapping from the accumulations performed in the imperative code to the reduced indices in the specification. This is in line with the rest of the proposed verification method: for regular assignments, we make explicit a mapping from the loop iterators to the tensor indices; for reductions, we make explicit a mapping from the loop iterators to the tensor indices *and* to the reduction indices. A similar idea is proposed in Iooss, Alias, and Rajopadhye [53] discussed further in chapter 8. However we will not try to infer this mapping but rely on compiler annotations instead. Moreover, we will assume that the compiler does not drastically transform the reduction domain: for instance, the sum $\sum_{0 \leqslant i < 2N} A(\lfloor i/2 \rfloor)$ could be transformed into $\sum_{0 \leqslant i < N} 2A(i)$ with a different iteration domain, but we do not intend to support such transformations.

7.3.1 Reductions as Nested Computations

If we see the reduction as a function of a set of indices, as suggested above, the computation of the reduction can be understood as the construction of an array collecting the indices that have already been included in the computation. For instance, if we consider the following pseudo-program, assuming that `{}` builds a sparse table, the value of the variable `result` when entering an iteration of the loop is always equal to the sum of the values stored in the table `result'`:

```
result = 0
result' = {}
for i = 0 to N - 1 do
  result += a[i]
  result'[i] = a[i]
```

Because of this, we can recompute the value stored in `result` from the values stored in the table `result'`. `result'` does not have to be introduced in the code, as its only purpose is for the verification — it is effectively a *ghost variable*

whose value is enough to re-compute the value of the variable `result`. Instead of keeping track of the value stored in `result`, it is thus enough in the prophetic evaluator and in the symbolic evaluator to keep track of the values stored in `result'`. By doing so, we reduce the problem of tracking accumulation using the accumulating assignment += to the solved problem of tracking regular assignments with `:=`, that can be mostly handled by the same techniques described in chapter 5. When program transformations are applied to the loop, the assignment could become multiple separate assignments, and the order in which the accumulations are performed can change. However, for the program to be correct, we expect that each of the `result'[i]` will have been written with the corresponding value exactly one.

Following this idea, accumulating assignments should have a different annotation compared to regular assignments, because we also need to keep track of the index in the original reduction that is currently being accumulated. We propose to annotate update assignments as follows:

$$a[\iota_1, \ldots, \iota_n] \ \{t\} \ +\{\iota'_1, \ldots, \iota'_n\}= e$$

The prophetic annotation t should denote a reduction over operator + (see next section), and this indicates that we are writing to $a[\iota_1, \ldots, \iota_n]$ the value e which must be equal to the reduced element at position $\iota'_1, \ldots, \iota'_n$ in the reduction denoted by t. For instance, the following program is implementing the sum $B() = \sum_{0 \leqslant i < N} A(i)$

```
b[] { 0 } := 0 ;
for i = 0 to N - 1 do
  b[] { B() } +{N - i - 1}= a[N - i - 1]
```

We can argue, like we did in section 5.6, that a tensor compiler can be annotated to produce these annotations, provided that the compiler does not merge or split separate iterations of the reduction. Fortunately, this type of "across-iteration" optimization is often out of scope for tensor compilers: not only it does not fit nicely in their per-iteration model, the traditional compiler algorithms that consume the intermediate language produced by the tensor compilers are quite good at performing these optimizations opportunistically. In the Halide representation, for instance, the extra information required correspond to the value of the recurrence variables of the current tensor, an information that is readily available in the compiler infrastructure.

Before using the value resulting from the accumulation in another expression, we need to ensure that it is equal to the original value of the reduction. This is why we require the annotation to contain a specification expression representing the whole reduction: this ensures that we know the set of indices that we are reducing over in the specification ($\{[i] \mid 0 \leqslant i < N\}$ in the example), and we can check that the indices accumulated by the implementation ($\{[N - i - 1] \mid 0 \leqslant i < N\}$ in the example) are a permutation of these.

Our proposed encoding uses *sets* to represent the set of indices that are being reduced over. This means that we must forbid accumulating into an array when a value for the annotated indices has already been accumulated. Since the reductions in the specifications iterate over a set (not a multiset) of indices, this could only occur in case of a miscompilation.

It is a good thing that we can ignore the issue, because associating an integral count to each value of the reduced indices could result in a count expressed as a polynomial in the presence of parametric loops, which leaves the Presburger arithmetic language we have been working with. Representing such multisets symbolically would be possible by using techniques such as Barvinok counting [14], and deferring to SMT solvers to check the resulting multiset equality, but would not have much practical interest since we do not expect the situation to happen.

7.3.2 Initialization

In the previous section, we have devised (informally) a scheme to handle the case where an array is only updated using accumulating operations. However, when implementing a reduction, the reduced array is first initialized using a regular assignment operation. The simplest case occurs when the initialization is performed using the neutral element of the reduction operator:

```
b[] := 0
for i = 0 to N - 1 do
  b[] += b[i]
```

This simple case can easily be integrated into the proposed representation: if regular assignments are restricted to the neutral element of the reduction, we

can have a special representation to indicate that the cell is initialized. When updating with a cell that is initialized, the existing reduced indices should not be taken into consideration: if Δh has a reduction associated with a cell and $\Delta h'$ re-initializes that same cell, the reduced indices in Δh should not be included in $\Delta h \triangleright \Delta h'$.

It is not entirely obvious that it is possible to extend the symbolic evaluator to handle this construct as it is not monotone in the set of defined indices. We present a formal justification for doing so in Section 7.5.

In the general case, the reduced variable can be initialized with an arbitrary value, or even not be initialized at all if the reduction is performed in-place on an existing array. For instance, an implementation of the general matrix multiply $D = \alpha AB + \beta C$ might start by assigning $\beta C(i, j)$ to $d[i, j]$. Thus, we need to keep track of the initial value of the reduction as a separate piece of information from the set of reduced indices. In fact, we need three pieces of information for each location:

- A specification expression that indicates the reduction that is being implemented (in particular, this indicates the reduction operator so that we are not mixing up reductions with different operators),
- The "initial value" for the location, i.e. the last value written to the location by a regular assignment (or the original value of the location if it is an input array)
- The set of reduced indices accumulated since the last regular assignment, if any

7.3.3 Partial Reductions

Using the representation of arrays taking part in a reduction proposed above, we can represent reductions performed out-of-order compared to their specification, but we cannot split a complete reduction into multiple partial reductions that are then recombined. In order to be able to validate this transformation, two components are needed: we need to keep track of the partial accumulations stored in temporary variables, and we need to ensure that when the partial accumulations are used, all the indices of the full reductions are covered. Keeping track of the partial accumulations is no different from keeping track of the full accumulation and can be performed easily in our representation;

however, some additional thought need to be given to the re-combining step.

To see why, let us consider the example of the sum $C() = A() + \sum_{0 \leqslant i < N} B(i)$ implemented by the following program, assuming $N = 4k$ for simplicity:

```
c[] {A()} := a[]
allocate tmp[4] in
par j = 0 to 3 do
  tmp[j] {0} := 0
  for i = 0 to N / 4 do
    tmp[j] {∑_{0⩽i<N} B(i)} +{4i+j}= b[4 * i + j]
for j = 0 to 3 do
  c[] += tmp[j]
```

The assignments to `tmp[j]` can be handled using the method described in the previous sections; however, there is no good annotation that can be written for the accumulating assignment into `c[]`. This assignment conceptually contributes many indices of the reductions: the set $\{i \mid 0 \leqslant i < N \wedge i \bmod 4 = j\}$. It is not clear that a compiler would have a straightforward way of reconstructing that set of indices. As such, it does not seem reasonable to require an annotation for that set.

In such cases, one would usually not expect the variable `c[]` to be used *during* the accumulation, as the intermediate value holds no meaningful value in the original computation. And once the accumulation is over, we expect the stored value to be equal to the complete reduction from the specification. In this situation, we can use the prophetic evaluation of the partial reduction to infer the set of indices accumulated into the final reduction. This works even in the case of a reduction that has been split recursively into many partial reductions, such as below:

```
out[] = 0
for i = 0 to 3 do
  tmpi[i] = 0
  for j = 0 to 3 do
    tmpj[j] = 0
    for k = 0 to 3 do
      tmpj[j] { C() } +{(i, j, k)}= b[i, j, k]
    tmpi[i] { C () } += tmpj[j]
  out[] { C () } += tmpi[i]
```

Recalling that the prophetic evaluation can only "see" the writes that are textually before the current statement, it is correct to infer that we are writing the indices $\{(\mathtt{i}, \mathtt{j}, \mathtt{k}) \mid 0 \leqslant \mathtt{k} < 4\}$ into `tmpi[i]` and the indices $\{(\mathtt{i}, \mathtt{j}, \mathtt{k}) \mid 0 \leqslant \mathtt{j} < 4 \wedge 0 \leqslant \mathtt{k} < 4\}$ into `out[]`. In general, this is not correct because it is possible that the last write is textually after the current statement (it could have been performed by a previous iteration of an enclosing loop); hence, for correctness, we must *check* during the symbolic evaluation that the equality inferred during the prophetic evaluation holds.

It would also be possible to infer the set of indices written using a multi-pass approach by applying a lexicographic maximum at each sequential loop level to capture writes performed after the current statement textually. However, I will not consider that possibility in this presentation, as I believe that it would not be needed in practice for reductions.

7.3.4 Consecutive Reductions

One final issue with the proposed representation remains to be tackled: the same variable can be used for multiple consecutive reductions. For instance, consider the following specification:

$$\mathrm{R}_0() = \sum_{0 \leqslant \mathtt{i} < \mathrm{N}} A(\mathtt{i})$$
$$\mathrm{R}_1() = \mathrm{R}_0() \times \prod_{0 \leqslant \mathtt{i} < \mathrm{N}, 0 \leqslant \mathtt{j} < \mathrm{M}} B(\mathtt{i}, \mathtt{j})$$

that can be implemented by the following program:

```
r[] {0} := 0
for i = 0 to N - 1 do
  r[] {R₀()} +i}= a[i]
for i = 0 to N - 1 do
  for j = 0 to N - 1 do
    r[] {R₁()} *{i,j}= b[i, j]
```

It is unclear how to properly handle the verification of such a program: the array `r[]` is accumulated into using *both* addition and multiplication, where

we have until now assumed that a given variable would be accumulated into using a single reduction operator.

The issue here is that at some point we need to "switch" from considering `r[]` as a sum to considering it as a product whose initial value is the result of the first sum. Dynamically, this happens when the first accumulation into `r[]` using $*=$ occurs, and ideally we would record this information in the prophetic evaluation. Doing so would require keeping track of the multiple nested reductions: assuming for the sake of the argument that the sequencing operator is right-associative (i.e. $s_1 \; ; \; s_2 \; ; \; s_3$ is $s_1 \; ; (s_2 \; ; \; s_3)$), the semantics of the code above is the function $\lambda x.(x + \sum_{0 \leqslant i < N} R_0()@(i)) \times \prod_{0 \leqslant i < N, 0 \leqslant j < M} R_1()@(i,j)$. As reductions are stacked, we would end up building more and more complex expressions. These complex expressions then have to be checked on use to be equal to their more succinct form (here, $R_1()$) in order to ensure simpler formulas are given to the final SMT check.

An "obvious" solution would be to require the generated code to use different variables for each of the reductions, with an additional assignment to change the variable after the first reduction. This approach fails when the reduced array is not zero-dimensional, because the additional move now has a computational cost. Properly applying this solution to arrays would require being able to rename arrays or aliasing array names, which we do not currently support.

While this approach seems feasible, it is simpler to introduce some way to know when one of the "steps" of the reduction is over, and to assert at that point that the reduction must be equal to the corresponding specification expression (here, $R_0()$ or $R_1()$), in a similar spirit to the "merge" node used by Reddy, Kruse, and Cohen [85] in their representation of reductions. Keeping in mind that any additional annotation need to be designed to ensure that it is reasonably straightforward to extract the required information from a tensor compiler, instead of requiring the tensor compiler to create a new virtual "merge" statement to be scheduled, I propose to use a lexically-scoped accumulate a in c indicating the scope within which a reduction is performed. This scope is a hint that a reduction is taking place on the array a within the scope: upon entering the scope, an accumulating representation of a should be used, and upon exiting the scope, the stored accumulation — if any has taken place — should be replaced with a regular symbolic heap. The previous program can then be written as follows:

```
r[] {0} := 0
accumulate r in
  for i = 0 to N - 1 do
    r[] {R0()} +{i}= a[i]
accumulate r in
  for i = 0 to N - 1 do
    for j = 0 to N - 1 do
      r[] {R1()} *{i,j}= b[i, j]
```

Using these annotations, the accumulating scopes for `r[]` are well-defined and there is no risk of confusion between the two reductions.

The use of an explicitly scoped statement is appropriate because multiple reductions are performed in stages, but not interleaved (distinct operators usually do not commute). The tensor compiler knows typically knows the range of the reduction, but the proper scopes could also be inferred by finding the maximal regions that only contain a given type of accumulating assignment for an array or location. We will assume that annotations are given indicating the region within which a reduction is performed per array; this is appropriate when the array variables come from distinct tensors in the specification, but fails if multiple arrays are combined into one. Compilers often merge all buffers stored in the "shared memory" of GPUs into a single buffer of static size, which could interfere with these annotations. Dealing with this mismatch is left to future research.

7.3.5 Differential memories

The semantics of an accumulating assignment such as `x += e` cannot be captured by a resulting memory: the semantics depends on the current value of `x`. To represent such semantics at runtime, we introduce *differential memories*. A differential memory $\delta\mu$ is a curried function from a location and an (old) value to a (new) value.

Memories can be represented by constant differential memories, i.e. if μ is a memory it can be understood as the differential memory $\ell \mapsto \nu \mapsto \mu(\ell)$ that simply ignores the old value. The sequencing of differential memories is

defined in terms of function composition:

$$(\delta\mu_1 \triangleright \delta\mu_2)(\ell) = \delta\mu_2(\ell) \circ \delta\mu_1(\ell)$$

Note that a partial memory μ can be represented as a differential memory by returning the corresponding value, i.e. the differential memory $\mathrm{diff}(\mu)$ defined as follows represents the memory μ.

$$\mathrm{diff}(\mu)(\ell)(v) = \begin{cases} \mu(\ell) & \text{if } \ell \in \mathrm{dom}(\mu) \\ v & \text{otherwise} \end{cases}$$

In that case, the implementation of $\triangleright$ on memories and differential memories ensures that $\mathrm{diff}(\mu_1 \triangleright \mu_2) = \mathrm{diff}(\mu_1) \triangleright \mathrm{diff}(\mu_2)$.

A differential memory $\delta\mu$ can be applied to a memory μ to obtain a new memory by applying the stored function to each location:

$$\delta\mu(\mu)(\ell) = \delta\mu(\ell)(\mu(\ell))$$

7.3.6 Reduction-Aware Dynamic Semantics

The dynamic semantics of Figure 4.5 can be adapted to use differential memories instead of memories as the output $\delta\mu$ in the state triple. We must also replace rules U-Seq, U-For and U-Assign with the following adapted rules.

$$\textsc{Red-U-Seq}\quad \frac{\mathcal{E};\mu \vdash s_1 \Downarrow_u \langle \delta\mu_1;\rho_1\rangle \qquad \mathcal{E};\delta\mu_1(\mu) \vdash s_2 \Downarrow_u \langle \delta\mu_2;\rho_2\rangle}{\mathcal{E};\mu \vdash s_1\ ;\ s_2 \Downarrow_u \langle \delta\mu_1 \triangleright \delta\mu_2;\rho_1 \cup \rho_2\rangle}$$

$$\textsc{Red-U-For}\quad \frac{[\![e]\!]_{\mathcal{E};\mu} = n \in \mathbb{Z} \qquad \forall 0 \leqslant i < n,\ \mathcal{E} + x \mapsto i;\left(\blacktriangleright_{0\leqslant j<i} \delta\mu_j\right)(\mu) \vdash c \Downarrow_u \langle \delta\mu_i;\rho_i\rangle}{\mathcal{E};\mu \vdash \text{for } x < e;\text{do } c \Downarrow_u \langle \blacktriangleright_{0\leqslant i<n} \delta\mu_i; rd_{\mathcal{E};\mu}(e) \cup \bigcup_{0\leqslant i<n} \rangle \rho_i}$$

$$\textsc{Red-U-Assign}\quad \frac{\begin{array}{c}[\![e]\!]_{\mathcal{E};\mu} = v \qquad [\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{Z} \quad \text{for all } 1 \leqslant i \leqslant n \\ \ell = a[n_1,\dots,n_n] \qquad \ell \in \mathrm{dom}(\mu) \qquad \rho = rd_{\mathcal{E};\mu}(e) \cup \bigcup_{1\leqslant i\leqslant n} rd_{\mathcal{E};\mu}(\iota_i)\end{array}}{\mathcal{E};\mu \vdash a[\iota_1,\dots,\iota_n]\ \{t\} := e \Downarrow_u \langle \{\ell \mapsto \lambda\_.v\};\rho\rangle}$$

The new rules apply the differential memories to the current memory instead of using the update operator $\triangleright$. Moreover, rule RED-U-ASSIGN returns the constant differential memory ignoring the pre-existing value at the location.

The definition of the compatibility $\delta\mu_i \frown \delta\mu_j$ used in rule U-PAR must also be updated, because $\delta\mu_i$ and $\delta\mu_j$ are now differential memories.

Definition 7.3.1. Two differential memories $\delta\mu_1$ and $\delta\mu_2$ are *compatible* if they are equal to constant functions on their shared domain, i.e. for all $\ell \mapsto f_1$ in $\delta\mu_1$ and $\ell \mapsto f_2$ in $\delta\mu_2$, and for all value v, $f_1(v) = f_2(v)$.

Finally, we introduce the rule RED-U-OPASSIGN for the update assignment $a[\iota_1,\dots,\iota_n]\ \{t\}\ \odot(\iota'_1,\dots,\iota'_m) = e$:

$$\textsc{Red-U-OpAssign}\quad \frac{\begin{array}{c}[\![e]\!]_{\mathcal{E};\mu} = v \qquad [\![\iota_i]\!]_{\mathcal{E};\mu} = n_i \in \mathbb{Z} \quad \text{for all } 1 \leqslant i \leqslant n \qquad \ell = a[n_1,\dots,n_n] \\ \ell \in \mathrm{dom}(\mu) \qquad \delta\mu = \{\ell \mapsto \lambda x.x \odot v\} \qquad \rho = rd_{\mathcal{E};\mu}(e) \cup \bigcup_{1\leqslant i\leqslant n} rd_{\mathcal{E};\mu}(\iota_i)\end{array}}{\mathcal{E};\mu \vdash a[\iota_1,\dots,\iota_n]\ \{t\}\ \odot(\iota'_1,\dots,\iota'_m) = e \Downarrow_u \langle \delta\mu;\rho\rangle}$$

Rule Red-U-OpAssign adds the updated location to both $\delta\mu$ and ρ, ensuring that there are no benign races involving a reduction. The semantics could be improved to allow races between atomic accumulations using e.g. atomic fetch-and-add instructions, as discussed below; this would require changing the compatibility relation to require $f_1 \circ f_2 = f_2 \circ f_1$ instead, and having an atomic version of Red-U-OpAssign that does not add the location to the read-set.

Reflection on Races When no reductions are involved, there are two types of accesses to a memory location: read accesses and write accesses. With the addition of reductions, there is also an *accumulating* or reducing access. Such accumulating access can be implemented using a sequence of read and write operations, or using an atomic operation, whether implemented with a lock or an atomic fetch-and-accumulate instruction provided by the hardware. Hence, we have three types of accesses: read (R), write (W) and accumulate (A); and conflicts whenever between concurrent accesses, at least one of which is either a write or an accumulation.

RW and RA conflicts No thread can read from a location that is concurrently being either written to or accumulated into, as that would cause non-determinism. This is enforced by the $\mathrm{dom}(\delta\mu_i) \# \rho_j$ constraint.

WW conflicts As previously, two concurrent writes are allowed only if they are both writing the same value. This is enforced by the $\delta\mu_i \asymp \delta\mu_j$ constraint.

WA conflicts Concurrent write and accumulating accesses must be forbidden, because depending on the order the accumulation may or may not be taken into consideration in the final value. Since accumulation is considered as both a read and a write, this is forbidden by the $\mathrm{dom}(\delta\mu_i) \# \rho_j$ constraint. In the presence of atomic accumulations that are not added to the set of read locations, these conflicts are prevented by the extended compatibility constraint $\delta\mu_i \asymp \delta\mu_j$ because a constant function only commutes with itself and the identity function.

AA conflicts Accumulating acccesses are treated as both reads and writes hence conflicts between two accumulating accesses are disallowed by the $\mathrm{dom}(\delta\mu_i) \# \rho_j$ constraint. In the presence of atomic accumulations that are not added to the set of read locations, concurrent atomic accesses using

the same commutative operator are allowed as they commute. Concurrent atomic and non-atomic accumulations are not allowed, because the atomic accumulation writes to the location and the non-atomic accumulation reads from the location.

7.4 Specification of Reductions

In order to represent reductions in the specification, we extend SAREs with a parametric reduction construct. In addition to the existing construct, a SARE expression with reductions can also include expressions of the form:

$$\bigodot_{\{j|\phi\}} e$$

The set $\{\mathbf{j} \mid \phi\}$ represents the indices over which the reduction is performed, and the variables $\mathbf{j}$ are bound in the expression e. The set $\{\mathbf{j} \mid \phi\}$ must be bounded, i.e. it must be contained within a multidimensional rectangle defined as a function of the existing variables in the context. Most often, reductions are implemented over a full interval, in which case we write $\bigodot_{a \leqslant j < b} e$ for $\bigodot_{\{j|a \leqslant j < b\}} e$.

This representation of reductions is different from the usual presentation of reduction in SAREs, such as that used by Iooss, Alias, and Rajopadhye [53], that use a projection π from the domain of the inner expression to the outer expression: if e is an expression with domain $\mathcal{D}$, then $\bigodot_{\pi} e$ has domain $\pi(\mathcal{D})$. The two formulations are equivalent: a projection π can be represented using the set $\{\mathbf{j} \mid \mathbf{i} = \pi(\mathbf{i}, \mathbf{j})\}$ where $\mathbf{i}$ denotes the variables in scope; conversely, a set $\{\mathbf{j} \mid \phi\}$ can be represented using the projection $\pi(\mathbf{i}, \mathbf{j}) = \mathbf{j}$ and restricting the domain of the inner expression to $\{(\mathbf{i}, \mathbf{j}) \mid \phi\}$.

The semantics of the parametric $\odot$ operator is to apply $\odot$ repeatedly to the expression e evaluated at each point in the domain in lexicographic order. If $\odot$ is associative and commutative, the evaluation order does not matter and $\bigodot$ is a set homomorphism; however, evaluating $\bigodot$ in lexicographic order ensures that the semantics is well-defined even when it is not a set

homomorphism. In particular, by enforcing a specific order, we are able to represent list homomorphisms and associative operators that are not necessarily commutative.

In order to verify an implementation of the reduction, it is not enough to be able to talk about the final result: intermediate assertions must mention partial reductions. Hence, we extend the specification language with a construct to denote the value at a specific position in the reduction. We will use the @ operator for this, intuitively, we want the following equation to hold (recall that the variadic operator $\bigodot$ is a *specification-only* operator):

$$[\![\left(\bigodot_{\{j_1,\ldots,j_n|\phi\}} e\right)@(\iota_1,\ldots,\iota_n)]\!] = [\![e[j_1 := \iota_1,\ldots,j_n := \iota_n]]\!]$$

The dynamic semantics of a reduction operator $\bigodot$ must contain the set of reduced values and the corresponding indices so that the semantics of the @ operator can be defined. Thus, we can evaluate a reduction to a pair $\langle h, V\rangle$ where h is a set homomorphism representing the reduction being performed and V is the domain over which the reduction is performed. The evaluation of $\bigodot_{\{\mathbf{x}|\phi\}} e$ is a pair $\langle h, V\rangle$ such that:

- V is the evaluation of $\{\mathbf{x} \mid \phi\}$ as a Presburger set in $\mathcal{E}$, and
- h is the unique set homomorphism for $\odot$ such that for an integer tuple $\mathbf{i}$, $h(\mathbf{i})$ is $[\![e]\!]_{\mathcal{E}+\mathbf{x}\mapsto\mathbf{i};M}$ if $\mathbf{i} \in V$ and $h(\emptyset)$ otherwise

Recalling that we require that the set of indices of a reduction must be bounded, the pair $\langle h, V\rangle$ can be obtained by simply applying the set homomorphism h to its full domain V. This conversion can be performed automatically by the semantics, except when storing the value to an array and when applying the @ operator.

The semantics for the @ operator are now easy to define using this representation: it corresponds to applying the h homomorphism to the corresponding argument.

$$\frac{[\![e]\!]_{\mathcal{E};M} = \langle h, V\rangle \qquad [\![\iota_i]\!]_{\mathcal{E};M} = n_i \in \mathbb{Z} \quad \text{for } 1 \leqslant i \leqslant n}{[\![e@(\iota_1,\ldots,\iota_n)]\!]_{\mathcal{E};M} = h(\{[n_1,\ldots,n_n]\})}$$

Note that when the indices fall outside the range of the reduced domain, the @ operator returns the neutral element $h(\emptyset)$ for the associative operator $\odot$. Also note that this dynamic representation of reductions can be adapted without issues to a non-commutative operator $\odot$ by using lists and list homomorphisms instead of sets and set homomorphisms.

There is an issue with this representation. Indeed, consider the following Halide algorithm featuring an update definition:

```
RDom x(0, N);
B() = 7;
B() += A(x);
```

that is represented by the following SARE with reductions:

$$B_0() = 7$$
$$B_1() = B_0() + \sum_{0 \leqslant x < N} A(x)$$

We would like to be able to annotate the accumulating assignments to the variable that represents tensor B using annotations such as $B_1()@(x)$. Hence, the evaluation of $B_0 + \sum_{0 \leqslant x < N} A(x)$ must also encode the homomorphism h and set of indices V, but include the initial value $B_0()$ and information about the $+$ operator that can be applied to the initial value. Hence, we augment the semantics for a reduction operator to a quadruple $\langle 0_\odot, \odot, h, V \rangle$ where h and V are defined as before, $\odot$ is the reduction operator, and $0_\odot$ is a distinguished constant representing the neutral element for $\odot$.

More generally, we define a *reduction value* r using the following grammar, where v denotes the original values of the language:

$$r ::= v \mid 0_\odot \mid \langle r, \odot, h, V \rangle$$

We can evaluate a reduction value r to a primitive value v using the concretization operator $\Downarrow r$:

$$\Downarrow v = v$$
$$\Downarrow 0_\odot = [\![0_\odot]\!]$$
$$\Downarrow \langle r, \odot, h, V \rangle = \Downarrow r \odot h(V)$$

where $[\![0_\odot]\!]$ is the neutral element for $\odot$.

The application of an operator $\odot$ to a reduction value r and a value v or a neutral element $0_\otimes$ for an operator $\otimes \neq \odot$ forces the concretization of the reduction value:

$$r \odot v = \Downarrow r \odot v$$
$$r \odot 0_\otimes = \Downarrow r \odot \Downarrow 0_\otimes$$

The application of $\odot$ to a reduction value r and the neutral element $0_\odot$ for that operator is a no-op:

$$r \odot 0_\odot = r$$

The application of an operator $\odot$ to two reduction values where the second has operator $\odot$ does not force any concretization and instead incorporates the first reduction value into the initial value of the second reduction value:

$$r \odot \langle r', \odot, h, V \rangle = \langle r \odot r', \odot, h, V \rangle$$

If the second argument is a reduction value with a different operator, this forces concretization:

$$r \odot \langle r', \otimes, h, V \rangle = \Downarrow r \odot (\Downarrow r' \otimes h(V))$$

This simplification process is coherent in the sense that it commutes with concretization:

Theorem 7.4.1. *If r and r' are two reduction values and $\odot$ an associative operator, the concretization of $r \odot r'$ is equal to the application of $\odot$ to the concretizations of r and r'.*

Proof. By structural induction on r'.

Case $r' = v$ Since r' is a value, we have $r' = \Downarrow r'$; moreover, by definition, $r \odot r' = \Downarrow r \odot r'$ is also a value. Hence we have $\Downarrow(r \odot r') = \Downarrow r \odot r' = \Downarrow r \odot \Downarrow r'$.

Case $r' = 0_\odot$ By definition, $r \odot 0_\odot = r$; moreover $\Downarrow 0_\odot$ is the neutral element for $\odot$, hence $\Downarrow(r \odot 0_\odot) = \Downarrow r = \Downarrow r \odot \Downarrow 0_\odot$.

Case $r' = 0_\otimes$ By definition, $r \odot r' = \Downarrow r \odot \Downarrow r'$ is a value, hence $\Downarrow(r \odot r') = \Downarrow r \odot \Downarrow r'$.

Case $r' = \langle r'', \odot, h, V \rangle$ If the operator of r' is the reduced operator, we have:

$$r \odot r' = \langle r \odot r'', \odot, h, V \rangle$$

By induction hypothesis, we have $\Downarrow(r \odot r'') = \Downarrow r \odot \Downarrow r''$. Hence:

$$\begin{aligned} \Downarrow(r \odot r') &= \Downarrow(r \odot r'') \odot h(V) \\ &= (\Downarrow r \odot \Downarrow r'') \odot h(V) \\ &= \Downarrow r \odot (\Downarrow r'' \odot h(V)) \\ &= \Downarrow r \odot \Downarrow r' \end{aligned}$$

Case $r' = \langle r'', \otimes, h, V \rangle$ If the operator of r' is not $\odot$, then $r \odot r' = \Downarrow r \odot \Downarrow r'$ by definition, which is a value.

□

Whenever a reduction tuple needs to be evaluated in the semantics (i.e. except when applied to the @ operator or applied as the right-hand side of the corresponding reduction operator), it has to be concretized. Because concretization commutes with the application of operators, we have the same semantics as previously for an expression that does not use @.

We need to ensure that the @ operator is only applied to terms that have a semantics as the appropriate reduction tuple. We can rely on the type system for this; in addition, we will record the operator of the reduction in the type system, which is useful when annotating reduction assignments with prophetic expressions in the next section. If τ is a type, $\odot$ an associative and commutative binary operator over T and n a non-negative integer, we define the type $\odot^n\tau$ to be the type of reductions using $\odot$ over a n-dimensional space. $\odot^n\tau$ is

We restrict ourselves to n-tuples as reduction spaces for simplicity.

introduced by the following rule:

$$\frac{\Gamma, j_1 : \mathbb{A}, \ldots, j_n : \mathbb{A}, \phi \vdash_A e : \tau \quad \odot \text{ is an associative operator for values of type } \tau}{\Gamma \vdash_A \bigodot_{\{(j_1,\ldots,j_n)|\phi\}} e : \odot^n \tau}$$

and can be propagated using the "initial value" rule:

$$\frac{\Gamma \vdash_A e_1 : \tau \qquad \Gamma \vdash_A e_2 : \odot^n \tau}{\Gamma \vdash_A e_1 \odot e_2 : \odot^n \tau}$$

There are two elimination rules for type $\odot^n \tau$: either a value of type $\odot^n \tau$ can be used whenever a value of type T is expected using an implicit conversion (this corresponds to converting $\langle r, \odot, h, V \rangle$ to $h(V)$ in the dynamic semantics), or a specific iteration of the reduction can be accessed using the @ operator.

$$\frac{\Gamma \vdash_A e : \odot^n \tau}{\Gamma \vdash_A e : \tau} \qquad \frac{\Gamma \vdash_A e : \odot^n \tau \qquad \Gamma \vdash \iota_i : \mathbb{A} \text{ for } 0 \leqslant i < n}{\Gamma \vdash_A e@(\iota_1, \ldots, \iota_n) : \tau}$$

Note that we do not allow applying the $\odot$ operator to a sequence of values of type $\odot^n \tau$ without concretizing them: we do not allow the compiler to merge distinct reductions from the specification. This simplifies the formalism; adding support for this would simply require allowing more complex spaces in the reduction (namely, the iteration space would be a *disjoint* tuple of the combined reduction spaces).

7.5 Validation of Programs with Reductions

Let us now formalize the treatment of reductions in the implementation proposed in section 7.3. This formalization builds upon the reduction-free approach presented in chapter 5, and represents reduction tuples by increasing the expressiveness of symbolic heaps. We first describe the concept of *accumulating* symbolic heaps to represent the value of a reduced variable mid-way through the reduction and adapt the existing semantics of Sched to use accumulating symbolic heaps as appropriate.

Symbolic Accumulation In order to represent the accumulating updates performed by a reduction, we need to represent "accumulating" symbolic values. Recall that a symbolic value is represented by a Presburger set whose tuples are named with expression contexts:

$$\hat{v} = \bigcup_i \{E_i\langle \mathbf{x}^i \rangle : \phi_i\}$$

To represent a variable that can be accumulated into to represent a reduction in the specification, and following the intuition of reductions as nested (array) computations, we can think of representing *accumulating* symbolic values using Presburger relations instead, mapping positions in the reduction space to the corresponding reduced value.

Conceptually, an accumulating symbolic value should represent the result of repeated application of the accumulating operator $\odot =$. Note that this is different from evaluating the result of the partial reduction: for instance, consider the program for $i < 4; \text{do } x += a[i]$ for some variable x. Its behavior is not to compute $\sum_{0 \leqslant i < 4} A(i)$, but to *add* that value to the pre-existing value of x. In particular the resulting heap after evaluating the program depends on the initial value of x. Hence, accumulating symbolic values should be thought of as *functions* from values to values, taking the old value of a memory cell and returning the new value for that memory cell after applying the accumulation. This will be made more precise with differential heaps below.

In order to represent the accumulating updates performed by a reduction, we define an *accumulating symbolic value* $\hat{a}$ as a pair $\langle O, R \rangle$. O is a singleton Presburger set containing tuples of the shape $\odot/0$, where $\odot$ is a reduction operator. R is a single-valued Presburger relation *with bounded domain* mapping flat anonymous tuples to expression tuples of shape E_i/m_i, where E_i is an expression context with m_i holes. Effectively, an accumulating symbolic value maps iteration in the reduction space to the corresponding reduced value. We write $\text{op}(\hat{a}) = O$ and $\text{red}(\hat{a}) = R$. We further require that whenever R is nonempty, O must be nonempty, i.e. if there are reduced indices, there must be a reduction operator.

We can evaluate an accumulating symbolic value by applying the operator to the mapping:

Definition 7.5.1. If $\hat{a} = \langle O, R \rangle$ is an accumulating symbolic value, then the

evaluation of $\hat{a}$ in environment $\mathcal{E}$ and model M is:

$$[\![\hat{a}]\!]_{\mathcal{E},M}(v) = v \odot \bigodot_{[\mathbf{j}] \to E\langle \mathbf{i} \rangle \in [\![R]\!]_{\mathcal{E}}} E[\mathbf{i}]$$

when $[\![O]\!]_{\mathcal{E}} = \{\odot\}$, and

$$[\![\hat{a}]\!]_{\mathcal{E},M}(v) = v$$

otherwise.

Note that the evaluation of $\hat{a}$ is well-defined due to the requirement that R must have a bounded domain.

The update combinator can be defined on accumulating symbolic values as follows. Since an accumulating symbolic value represents an accumulation as a function, the evaluation of the update must correspond to function composition: indeed, if $x+ = e_1$ is represented by f_1 and $x+ = e_2$ is represented by f_2, then $x+ = e_1 \; ; \; x+ = e_2$ should be represented by $f_2 \circ f_1$. This means that if $\hat{a}_1$ and $\hat{a}_2$ are accumulating symbolic values with the same operator $\odot$, then $\hat{a}_1 \triangleright \hat{a}_2$ should be such that:

$$[\![\hat{a}_1 \triangleright \hat{a}_2]\!]_{\mathcal{E};M} = [\![\hat{a}_2]\!]_{\mathcal{E};M} \circ [\![\hat{a}_1]\!]_{\mathcal{E};M}$$

Defining $\triangleright$ as a total function would require being able to associate *multiple* expressions to each position in the reduction space, making the relation R no longer single-valued. Moreover, keeping track of a count associated with each position would make the representation no longer representable using Presburger arithmetic in the present of parametric loops. As discussed in subsection 7.3.1, we can sidestep the issue by assuming that we only combine accumulating values with disjoint domains, and define $\hat{a}_1 \triangleright \hat{a}_2$ as a partial operator:

$$\langle \odot, R_1 \rangle \triangleright \langle \odot, R_2 \rangle = \begin{cases} \langle \odot, R_1 \cup R_2 \rangle & \text{if } \mathrm{empty}(\mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\langle \odot, R_1 \rangle \triangleright \langle \otimes, R_2 \rangle = \begin{cases} \text{undefined} & \text{if } \odot \neq \otimes \end{cases}$$

Lemma 7.5.1. *For accumulating symbolic values* $\hat{a}_1 = \langle O_1, R_1 \rangle$ *and* $\hat{a}_2 = \langle O_2, R_2 \rangle$, *environment* $\mathcal{E}$ *and model* M *such that* $[\![\mathrm{dom}(R_1) \cap \mathrm{dom}(R_2)]\!]_{\mathcal{E}}$ *is empty and* $[\![O_1]\!]_{\mathcal{E}} =$

$[\![O_2]\!]_{\mathcal{E}}$, *then the following holds:*

$$\begin{aligned}[\![\hat{a}_1 \triangleright \hat{a}_2]\!]_{\mathcal{E};M} &= [\![\hat{a}_1]\!]_{\mathcal{E};M} \triangleright [\![\hat{a}_2]\!]_{\mathcal{E};M} \\ &= [\![\hat{a}_2]\!]_{\mathcal{E};M} \circ [\![\hat{a}_1]\!]_{\mathcal{E};M}\end{aligned}$$

In the same way, the iterated update can be defined by taking the union of the underlying sets, provided that the domains are disjoint — which can be expressed using the single-valuedness of the bound maps $\text{bind}_{0\leqslant x<e}(\text{dom}(R))$ and $\text{bind}_{0\leqslant x<e}(O)$:

$$\bigtriangleright_{0\leqslant x<e} \langle O, R\rangle = \langle \bigcup_{0\leqslant x<e} O, \bigcup_{0\leqslant x<e} R\rangle$$

if $\text{bind}_{0\leqslant x<e}(\text{dom}(R))$ and $\text{bind}_{0\leqslant x<e}(O)$ are single-valued, and is undefined otherwise.

Lemma 7.5.2. *For an accumulating symbolic value* $\hat{a} = \langle O, R\rangle$, *environment* $\mathcal{E}$ *and model* M *such that* $\text{bind}_{0\leqslant x<\iota}(\text{dom}(R))$ *and* $\text{bind}_{0\leqslant x<\iota}(O)$ *are single-valued in* $\mathcal{E}$, *and* $[\![\iota]\!]_{\mathcal{E}} = n \in \mathbb{Z}$, *then the following holds:*

$$[\![\bigtriangleright_{0\leqslant x<\iota} \hat{a}]\!]_{\mathcal{E};M} = \bigtriangleright_{0\leqslant i<n} [\![\hat{a}]\!]_{\mathcal{E}+x\mapsto i;M}$$

Accumulating heaps In the same way symbolic heaps are defined on top of symbolic values, accumulating symbolic heaps are defined on top of accumulating symbolic values. An accumulating symbolic heap is represented as a pair $\langle O, R\rangle$ of Presburger relations where:

- O is a single-valued Presburger relation from locations to reduction operators, i.e. O maps tuples in space a/n where a is an array name to tuples in space $\odot/0$ where $\odot$ is a reduction operator. There is at most one operator for each location.

- R is a Presburger relation mapping tuples in space a/n to tuples in space $\epsilon/m \times E/p$ where E is an expression context with p holes. The first component of the pair represents the position in the reduction space, while the second component represents the value that was written at that position in the reduction space.

- uncurry(R) is single-valued, i.e. there is at most one expression context associated for a given reduction position at a given location.

- Each location that has associated reduced values must have a reduction operator, i.e. dom(R) ⊆ dom(O) must hold.

Note that these restriction do not prevent having different operators associate with different locations of the same array. Although this situation rarely occurs in programs written by humans, it can occur when a compiler merges multiple arrays into a single one. For instance, Halide merges all arrays stored on the so-called shared memory on GPUs into a single statically-sized array. In this situation the code on the left might be transformed into the code on the right:

```
allocate x[] in
allocate y[] in
x[] := 0 ;
y[] := 1 ;
for i = 0 to 31 do
  x[] += a[i] ;
  y[] *= b[i] ;
done
```

```
allocate xy[2] in
xy[0] := 0 ;
xy[1] := 1 ;
par i = 0 to 31 do
  xy[0] += a[i] ;
  xy[1] *= b[i] ;
done
```

If $a[\mathbf{i}]$ is a location, then $O(a\langle\mathbf{i}\rangle)$ is an operator $\odot$ and $R(a\langle\mathbf{i}\rangle)$ is a set of pairs, which can be unwrapped. Together, they can be used to represent an accumulating symbolic value, which we note $\langle O, R\rangle(a\langle\mathbf{i}\rangle)$.

Accumulating symbolic heaps can be interpreted as differential memories performing the corresponding accumulation:

$$[\![\langle O, R\rangle]\!]_{\mathcal{E},M}(a[\mathbf{i}])(v) = \begin{cases} [\![\langle \odot, \mathrm{unwrap}(R(a\langle\mathbf{i}\rangle))\rangle]\!]_{\mathcal{E},M}(v) & \text{if } [\![O(a\langle\mathbf{i}\rangle)]\!]_{\mathcal{E}} = \{\odot\} \\ v & \text{otherwise} \end{cases}$$

The sequencing operator and its iterated counterpart can be defined as for accumulating symbolic values, using the union of Presburger relations. The following conditions are necessary for the correctness of this representation:

Lemma 7.5.3. *If* $\langle O_1, R_1\rangle$ *and* $\langle O_2, R_2\rangle$ *are accumulating symbolic heaps, then in any environment* $\mathcal{E}$ *and model* M *in which* $O_1 \cup O_2$ *is functional (i.e. locations appearing in both heaps have the same operator) and* $\mathrm{dom}(\mathrm{uncurry}(R_1)) \cap \mathrm{dom}(\mathrm{uncurry}(R_2))$ *is empty (i.e. no index is reduced twice), we have* $[\![\delta h_1 \triangleright \delta h_2]\!]_{\mathcal{E},M} = [\![\delta h_1]\!]_{\mathcal{E},M} \triangleright [\![\delta h_2]\!]_{\mathcal{E},M}$.

Lemma 7.5.4. *If* $\langle O, R\rangle$ *is an accumulating symbolic heap, then in any environment* $\mathcal{E}$ *and model* M *in which* $\bigcup_{0\leqslant x<\iota}(O)$ *is functional (i.e. no location is associated with different operators in different iterations) and* $\mathrm{bind}_{0\leqslant x<\iota}\left(\mathrm{dom}(\mathrm{uncurry}(R_1))\right)$ *is functional (i.e. no index is reduced by distinct iterations), we have*

$$[\![\bigtriangleright_{0\leqslant x<e} \delta h]\!]_{\mathcal{E},M} = \bigtriangleright_{0\leqslant i<[\![e]\!]_{\mathcal{E}}} [\![\delta h]\!]_{\mathcal{E},x\mapsto i,M}$$

Reduction Heaps In order to represent both regular and accumulating assignments, we add a regular symbolic heap to this pair. A reduction heap Δh is thus a tuple $\langle h, \langle O, R\rangle\rangle$ where h is a regular symbolic heap as described in chapter 5, and the $\langle O, R\rangle$ pair is an accumulating symbolic heap as described in the previous section. Redution heaps evaluate to differential memories by applying the accumulating heap to the regular heap:

$$[\![\langle h, \langle O, R\rangle\rangle]\!]_{\mathcal{E},M} = [\![h]\!]_{\mathcal{E},M} \triangleright [\![\langle O, R\rangle]\!]_{\mathcal{E};M}$$

This evaluation first overwrites any existing value with the value present in h, if any, then uses the result to initialize the reduction O to the values in R.

When combining reduction heaps, the presence of an overwrite in the second reduction heap must erase any accumulation to the corresponding location in the first reduction heap. The update is thus defined as follows:

$$\langle h, \langle O, R\rangle\rangle \triangleright \langle h', \langle O', R'\rangle\rangle = \langle h \triangleright h', \langle (O \setminus \mathrm{dom}(h')) \cup O', (R \setminus \mathrm{dom}(h')) \cup R\rangle$$

In order for this symbolic update to be correct, the single-valuedness invariants on the accumulating component must be respected. They can be expressed as a Presburger formula or unit set, called update-ok$(\langle h, \langle O, R\rangle\rangle, \langle h', \langle O, R\rangle\rangle)$, that states:

- The disjointness of $\mathrm{dom}(\mathrm{uncurry}(R \setminus \mathrm{dom}(h')))$ and $\mathrm{dom}(\mathrm{uncurry}(R'))$, ensuring that we are not counting reduction indices twice

- The single-valuedness of $(O \setminus \mathrm{dom}(h')) \cup O'$, ensuring that each array cell is associated with a single reduction operator

If the reduction operator associated with a cell in O is non-commutative, we must also ensure that the reduced indices associated with that cell in the second argument are lexicographically larger than the reduced indices in the first argument, to ensure order is preserved. If we define $O'' = O \setminus \mathrm{dom}(h')$ and $R'' = R \setminus \mathrm{dom}(h')$ and denote by $\mathrm{nc}(O)$ the subset of the domain of O that is associated with a non-commutative operator, this can be expressed as the Presburger formula:

$$\mathrm{fst}(R'') \cap \mathrm{nc}(O \cup O') \prec \mathrm{fst}(R) \cap \mathrm{nc}(O \cup O')$$

Recall that $\mathrm{fst}(R)$ denotes the first component in the range of the relation, i.e. $\mathrm{fst}(R)$ maps each cell to its reduced indices.

We can now state a soundness lemma:

Lemma 7.5.5. *If* Δh *and* $\Delta h'$ *are reductions heaps such that* update-ok$(\Delta h, \Delta h')$ *holds in environment* $\mathcal{E}$ *and memory* M*, the following holds:*

$$[\![\Delta h \triangleright \Delta h']\!]_{\mathcal{E};M} = [\![\Delta h]\!]_{\mathcal{E};M} \triangleright [\![\Delta h']\!]_{\mathcal{E};M}$$

To represent an iterated update, we must apply the same ideas and first remove the accumulations that are performed before the last overwrite. As in the definition of the iterated update for a regular heap, we can use $\mathrm{lexmax}(\mathrm{bind}_{0 \leqslant x < \iota}(\mathrm{dom}(h)))$ to construct a mapping, denoted W, from each location to the last iteration that writes to that location. We can also use $\mathrm{bind}_{0 \leqslant x < \iota}(\mathrm{wrap}(R))$ to construct a mapping, denoted A, from tuples (location, (reduced indices, expression)) to the iterations that accumulate the expression into the location. If we denote by $\tilde{W}$ the relation obtained by adding the full possible set of reduced indices and expressions to each location in W, then $A \prec \tilde{W}$ is the set of (locations, (reduced indices, expressions)) tuples that are overwritten by the last write and must be removed from A. This set imposes conditions on x, so finally $\mathrm{unwrap}(\mathrm{dom}(A \setminus (A \prec \tilde{W})))$ is the set of mappings from locations to (reduced indices, expression) pairs that are not overwritten in the iterated update. The result of $\blacktriangleright_{0 \leqslant x < \iota} \langle h, \langle O, R \rangle \rangle$ can thus be obtained as

the reduction heap $\langle h', \langle O', R' \rangle \rangle$ where:

$$h' = \bigtriangleup_{0 \leqslant x < \iota} h$$
$$R' = \text{unwrap}(\text{dom}(A \backslash (A \prec \tilde{W})))$$
$$O' = \bigcup_{0 \leqslant x < \iota} O$$

In order for this to be correct, we must ensure that each (location, reduced indices) pair is only written to by at most one iteration. This can be expressed as the condition iupdate-ok(R') that states the single-valuedness of snd(curry($A \backslash (A \prec \tilde{W})$)), i.e. the mapping from (location, reduced indices) to the iterations that write to it. When non-commutative operators are present, we must additionally ensure that this mapping is increasing.

Symbolic summarization At some point (i.e. when exiting the accumulate blocks presented in Section 7.3.4), we must convert the accumulating value into a regular symbolic value that represents the reduction. For instance, the accumulating heap

$$\{a\langle i \rangle \rightarrow [k, E\langle (i, k) \rangle] : 0 \leqslant k < i\}$$

become the symbolic heap (assuming that the operator associated with $a[i]$ is $+$)

$$\{a\langle i \rangle \rightarrow \left(\sum_{0 \leqslant k < \square} E[\square, k] \right) \langle i, i \rangle\}$$

The most intuitive way to do this would be to perform this transformation on the space decomposition of the accumulating value. However, if different expression contexts are used in the decomposition, we want to keep the order in which the accumulations have been performed, such as in this even-odd accumulation:

$$\begin{array}{llll} \{ & \odot\langle x \rangle & \mapsto E_0\langle x \rangle & | \; x \mod 2 = 0 \wedge 0 \leqslant x < N \; ; \\ & \odot\langle x \rangle & \mapsto E_1\langle x \rangle & | \; x \mod 2 = 1 \wedge 0 \leqslant x < N \; \} \end{array}$$

We want to preserve the order in such cases for two reasons. The first is that ultimately we will need to generate an equality comparing the accumulation performed by the implementation with a reduction in the specification, where our goal is to have the indices match in order to alleviate the solver from having to invent a permutation. The second reason is that preserving the order is the only correct approach for non-commutative reductions, which we will consider shortly.

In order to ensure that we keep the order of the accumulation, we only define this *summarization* for homogeneous accumulating symbolic values. For non-homogeneous accumulating symbolic values, we merge the different possible expressions into a single one using by lifting a summarization process on symbolic values described below, before applying the summarization for homogenous accumulating heaps.

A symbolic value has a piece-wise definition following its space decomposition. A symbolic value can be converted to a homogeneous symbolic set using the summarize function, which embeds the piece-wise definition into the expression context. For a symbolic value which is already a homogeneous symbolic set, summarize simply removes the piece-wise condition if it exists:

$$\text{summarize}(\{\mathsf{E}\langle \hat{\mathfrak{i}_1}, \ldots, \hat{\mathfrak{i}_n}\rangle : \phi\}) = \{\mathsf{E}\langle \hat{\mathfrak{i}_1}, \ldots, \hat{\mathfrak{i}_n}\rangle\}$$

For a symbolic value $\hat{v}$ which is not homogeneous, let $(\mathsf{E}_i/n_i)_i$ be the space decomposition of $\hat{v}$, and let $\{\mathsf{E}_0\langle \mathbf{e}\rangle \mid \phi\}$ be the set associated with space E_0/n_0. ϕ is an affine expression and can be represented using a concrete expression e_ϕ; let $e_\phi = \mathsf{E}_\phi[\mathbf{e}']$ be a context decomposition of e_ϕ. Finally, let $\{\mathsf{E}\langle \mathbf{f}\rangle\}$ be the result of summarizing the rest of $\hat{v}$. Then, summarize($\hat{v}$) can be defined recursively as:

$$\text{summarize}(\hat{v}) = \{\text{select}(\mathsf{E}_\phi, \mathsf{E}_0, \mathsf{E})\langle \mathbf{e}', \mathbf{e}, \mathbf{f}\rangle\}$$

Since select implements a conditional at the expression level and $[\![\{\mathsf{E}_\phi\langle \mathbf{e}'\rangle\}]\!]_{\mathcal{E},\mathcal{M}}$ is $[\![e_\phi]\!]_{\mathcal{M}} = [\![\phi]\!]_{\mathcal{E}}$, we have the following lemma:

Lemma 7.5.6. *For any symbolic value $\hat{v}$, environment $\mathcal{E}$ and model $\mathcal{M}$, the following*

value inclusion holds:

$$[\![\text{summarize}(\hat{v})]\!]_{\mathcal{E},\mathcal{M}} \sqsupseteq [\![\hat{v}]\!]_{\mathcal{E},\mathcal{M}}$$

In particular, if $[\![\hat{v}]\!]_{\mathcal{E},\mathcal{M}} \in \mathcal{V}$*, we have* $[\![\text{summarize}(\hat{v})]\!]_{\mathcal{E},\mathcal{M}} = [\![\hat{v}]\!]_{\mathcal{E},\mathcal{M}}$.

The definition of summarize($\hat{v}$) is not unique as it depends on the order in which the space decomposition of $\hat{v}$ is performed, and it also depends on the decomposition $\mathsf{E}_\phi[\mathbf{e}']$ which is used to represent the affine condition ϕ. However, the above lemma guarantees that whichever choices are made in the definition of summarize, the semantics evaluation is preserved. The exact representation used can have performance implications, notably because it controls whether merging can be performed.

As a unary operator on symbolic sets, the summarize function can be extended to single-valued symbolic relations by defining it on the range decomposition of the relation.

For a homogenous accumulating symbolic value $\mathsf{R} = \{[\mathbf{x}] \to \mathsf{E}\langle\iota\rangle \mid \phi\}$, we first decompose $\phi = \psi \wedge \psi'$ where the variables in $\mathbf{x}$ do not appear in ψ'. We then evaluate ϕ and ι (which are affine expressions) to the explicit expressions e_ϕ and ι' in order to build the expression $\bigodot_{\{\mathbf{x}|e_\phi\}} \mathsf{E}[\mathbf{e}']$. This expression can then be decomposed as a context $\mathsf{E}_\odot[\mathbf{f}]$ with no free variable, and we can define :

$$\text{asummarize}(\langle \mathsf{O}, \mathsf{R}\rangle) = \{\mathsf{E}_\odot[\mathbf{f}] \mid \psi'\}$$

Note that if R is empty (i.e. the reduction is applied to the empty set), the summarization is empty: this represents the identity function, which is equivalent to applying the reduced operator to its neutral element.

Prophetic evaluation We are now ready to extend the prophetic evaluator and the symbolic evaluator with support for reductions. The prophetic evaluation rules with reductions are given in Figure 7.1. In rule P-SEQ and P-SEQLOOP, we use the conditions update-ok and iupdate-ok to ensure that each cell has at most one associated reduction operator and that no reduced index is counted

P-Skip

$$\frac{}{\langle \Gamma \rangle \vdash \text{skip} \Downarrow \langle \emptyset, \emptyset \rangle}$$

P-Seq

$$\frac{\langle \Gamma \rangle \vdash c_1 \Downarrow \Delta h_1 \qquad \langle \Gamma \rangle \vdash c_2 \Downarrow \Delta h_2 \qquad \Gamma \vdash \text{update-ok}(\Delta h_1, \Delta h_2)}{\langle \Gamma \rangle \vdash c_1 \ ; \ c_2 \Downarrow \Delta h_1 \triangleright \Delta h_2}$$

P-SeqLoop

$$\frac{\Gamma \vdash \iota : \mathbb{A} \qquad \langle \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota \rangle \vdash c \Downarrow \Delta h \qquad \Gamma \vdash \text{iupdate-ok}(x, \iota, \Delta h)}{\langle \Gamma \rangle \vdash \text{for } x < \iota; \text{do } c \Downarrow \bigtriangleright_{0 \leqslant x < \iota} \Delta h}$$

P-Assign

$$\frac{\begin{array}{c}\Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \qquad E\langle \iota''_1, \ldots, \iota''_m \rangle = \text{decompose}(t) \\ a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \qquad \Gamma \vdash \iota_i : \mathbb{A} \text{ for all } 1 \leqslant i \leqslant n \\ \Gamma \vdash 0 \leqslant \iota_i < \iota'_i \text{ for all } 1 \leqslant i \leqslant n \hat{h} = \{a[\iota_1, \ldots, \iota_n] \mapsto E\langle \iota''_1, \ldots, \iota''_m \rangle\}\end{array}}{\langle \Gamma \rangle \vdash a[\iota_1, \ldots, \iota_n] \ \{t\} := e \Downarrow \langle \hat{h}, \emptyset \rangle}$$

P-OpAssign

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \qquad \Gamma \vdash_A t : \tau \\ \mathbf{t} = \text{decompose}(t) \qquad a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \qquad \Gamma \vdash \iota_i : \mathbb{A} \text{ for all } 1 \leqslant i \leqslant n \\ \Gamma \vdash 0 \leqslant \iota_i < \iota'_i \text{ for all } 1 \leqslant i \leqslant n \qquad \Gamma \vdash \iota''_i : \mathbb{A} \text{ for all } 1 \leqslant i \leqslant p \\ \ell = a\langle \iota_1, \ldots, \iota_n \rangle \qquad \delta h = \langle \{\ell \to \odot\langle\rangle\}, \{\ell \to [[\iota''_1, \ldots, \iota''_m], \mathbf{t}]\} \rangle\end{array}}{\langle \Gamma \rangle \vdash a[\iota_1, \ldots, \iota_n] \ \{t\} \ \odot(\iota''_1, \ldots, \iota''_p) = e \Downarrow \langle \emptyset, \delta h \rangle}$$

P-ParLoop

$$\frac{\Gamma \vdash e : \mathbb{A} \qquad \langle \Gamma, x : \mathbb{A}, 0 \leqslant x < \hat{\iota} \rangle \vdash c \Downarrow \Delta h \qquad \upsilon\left(\bigcup_{0 \leqslant x < e} \Delta h\right) = \langle \Delta h', C \rangle}{\langle \Gamma \rangle \vdash \text{par } x < e; \text{do } c \Downarrow \bigcup_{0 \leqslant x < \hat{\iota}} \Delta h}$$

P-Accumulate

$$\frac{\langle \Gamma \rangle \vdash c \Downarrow \Delta h \qquad \Delta h' = (\Delta h \backslash a) \uplus \text{asummarize}(\Delta h \cap a)}{\langle \Gamma \rangle \vdash \text{accumulate } a \text{ in } c \Downarrow \Delta h'}$$

Figure 7.1: Prophetic Evaluation

twice, as explained above. Rule P-ACCUMULATE is new, and simply summarizes the reduction into a regular heap when exiting an accumulate block.

A symbolic state S represents the result of the execution of a program. It is represented by a triple $S = \langle h, r, C \rangle$ where:

- h = updates(S) is the set of *updates* performed by the state; it is represented by a reduction heap which models both regular writes using `:=` and accumulating writes.

- r = reads(S) is the set of *reads* performed by the statement; it is represented as a symbolic set of locations. The evaluation of a statement only depends on the value of the read locations, which is used to ensure the absence of race in parallel loops.

- C = constraints(S) is a set of *constraints*, or assertions, that must be satisfied. The assertions are represented as a symbolic set of tensor expressions. All the assertions in the set must be satisfied for the rest of the state to be valid. An assertion $\{[x_1, \ldots, x_n] \to \hat{e}\langle y_1, \ldots, y_m \rangle \mid \hat{b}\}$ represents the following formula, in a context assigning values to the $x_1, \ldots, x_n$:

$$\forall y_1, \ldots, y_n, \hat{b} \Rightarrow \hat{e}[y_1, \ldots, y_m]$$

Intuitively, a symbolic state represents a program whose behavior is captured by updates(S) provided that constraints(S) hold. Thus, the operations on symbolic states are performed by applying them to the updates component and the verification conditions are added to the constraints component.

Because accumulating assignments are considered as both a write and a read of the corresponding locations, the rw-safe constraint ensures that there are no conflicts between reduction accesses and any other type of access, including another reduction access.

Red-S-Skip

$$\frac{}{\langle \Gamma; h\rangle \vdash \text{skip} \Downarrow \langle \emptyset; \emptyset; \emptyset\rangle}$$

Red-S-If

$$\frac{\Gamma \vdash \iota : \mathbb{B} \qquad \langle \Gamma, \iota; h\rangle \vdash c_1 \Downarrow s_1 \qquad \langle \Gamma, \neg\iota; h\rangle \vdash c_2 \Downarrow c_2}{\langle \Gamma; h\rangle \vdash c \Downarrow (s_1 \cap \iota) \cup (s_2 \cap \neg\iota)}$$

Red-S-Let

$$\frac{\Gamma \vdash \iota : \mathbb{A} \qquad \langle \Gamma, x : \mathbb{A}, x = \iota; h\rangle \vdash c \Downarrow s}{\langle \Gamma; h\rangle \vdash \text{let } x = \iota \text{ in } c \Downarrow s[x := \iota]}$$

Red-S-Seq

$$\frac{\langle \Gamma; h\rangle \vdash c_1 \Downarrow s_1 \qquad \langle \Gamma; h \triangleright \text{updates}(s_1)\rangle \vdash c_2 \Downarrow s_2}{\langle \Gamma; h\rangle \vdash c_1 \,;\, c_2 \Downarrow s_1 \triangleright s_2}$$

Red-S-SeqLoop

$$\frac{\Gamma \vdash \iota : \mathbb{A} \qquad \langle \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h \triangleright \bigtriangledown_{0\leqslant z<x} \Delta h[x := z]\rangle \vdash c \Downarrow \langle \Delta h, R, C\rangle}{\langle \Gamma; h\rangle \vdash \text{for } x < \iota; \text{do } c \Downarrow \bigtriangledown_{0\leqslant x<\iota} \langle \Delta h, R, C\rangle}$$

Red-S-ParLoop

$$\frac{\begin{array}{c}\langle \Gamma, x : \mathbb{A}, 0 \leqslant x < \iota; h\rangle \vdash c \Downarrow \langle\langle h', \langle O, R_h\rangle\rangle; R; C\rangle \\ W = \text{dom}(h') \cup \text{dom}(R_h) \\ \Gamma \vdash \text{rw-safe}(x, \iota, W, R) \qquad \Gamma \vdash \iota : \mathbb{A} \qquad \text{ww-covered}(\Gamma, x, \iota, h') = C' \\ \upsilon\left(\bigcup_{0\leqslant x<\iota} h'\right) = h'' \qquad \Gamma \vdash \text{sv}(\text{bind}_{0\leqslant x<\iota}\left(\text{fst}(R'_h)\right)) \\ R' = \bigcup_{0\leqslant x<\iota} R \qquad s' = \langle\langle h'', \langle \bigcup_{0\leqslant x<\iota} O, \bigcup_{0\leqslant x<\iota} R'_h\rangle\rangle; R'; C' \cup \bigcup_{0\leqslant x<\iota} C\rangle\end{array}}{\langle \Gamma; h\rangle \vdash \text{par } x < \iota; \text{do } c \Downarrow s'}$$

Red-S-Assign

$$\frac{\begin{array}{c}a : \tau[\iota'_1 \times \cdots \times \iota'_n] \in \Gamma \qquad \Gamma \vdash \text{reads}(e) \subseteq \text{dom}(h) \\ \Gamma \vdash_a e : \tau \qquad \Gamma \vdash_A t : \tau \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{for all } 1 \leqslant i \leqslant n \\ \hat{\ell} = a\langle \iota_1, \ldots, \iota_n\rangle \qquad \Gamma \vdash \{\hat{\ell}\} \subseteq \{a\langle x_1, \ldots, x_n\rangle \mid 0 \leqslant x_1 < \iota'_1, \ldots, 0 \leqslant x_n < \iota'_n\} \\ \hat{C} = [\![e]\!]_h = \{\text{decompose}(t)\} \qquad \hat{h} = \{\hat{\ell} \longrightarrow \text{decompose}(t)\}\end{array}}{\langle \Gamma; h\rangle \vdash a[\iota_1, \ldots, \iota_n]\ \{t\} := e \Downarrow \langle\langle \hat{h}; \langle \emptyset; \emptyset\rangle\rangle; \hat{r}; \hat{C}\rangle}$$

Red-S-OpAssign

$$\frac{\begin{array}{c}a : \tau[\iota''_1 \times \cdots \times \iota''_n] \in \Gamma \qquad \Gamma \vdash \text{reads}(e) \subseteq \text{dom}(h) \qquad \Gamma \vdash_a e : \tau \\ \Gamma \vdash_A t : \tau \qquad \Gamma \vdash \iota_i : \mathbb{A} \quad \text{for all } 1 \leqslant i \leqslant n \qquad \Gamma \vdash \iota'_i : \mathbb{A} \quad \text{for all } 1 \leqslant i \leqslant m \\ \hat{\ell} = a\langle \iota_1, \ldots, \iota_n\rangle \qquad \Gamma \vdash \{\hat{\ell}\} \subseteq \{a\langle x_1, \ldots, x_n\rangle \mid 0 \leqslant x_1 < \iota''_1, \ldots, 0 \leqslant x_n < \iota''_n\} \\ \hat{C} = [\![e]\!]_h = \{\text{decompose}(t)\} \\ \delta\hat{h} = \{\hat{\ell} \longrightarrow [[\iota'_1, \ldots, \iota'_m],\ \text{decompose}(t)]\}\end{array}}{\langle \Gamma; h\rangle \vdash a[\iota_1, \ldots, \iota_n]\ \{t\}\ \odot(\iota'_1, \ldots, \iota'_m) = e \Downarrow \langle\langle \emptyset; \langle\{\odot\}; \delta\hat{h}\rangle; \hat{r}; \hat{C}\rangle}$$

Figure 7.2: Symbolic Evaluation with Reductions

Related work 8

8.1 Translation Validation

Translation validation is part of a family of verification techniques that can be described as *instance verification*. Where program verification is concerned about proving properties that hold of any run of a program (under certain conditions), instance verification is about checking properties of a *specific run of a program on a specific input*. Translation validation is instance verification applied to a compiler: the goal is to check *a posteriori* that the output of a compiler, or compiler pass, has a semantics that is compatible with those allowed by the source program. This contrasts to the formal verification or certification of a compiler such as CompCert [67] whereby the compilation process itself is proven *a priori* to never generate incorrect code.

Some optimization passes of the CompCert compiler use translation validation approaches, with untrusted code performing program transformations and a verified checker. If the verified checker fails to validate the transformations, the compiler either skips the optimization altogether, or aborts the compilation.

Tristan's PhD thesis [107] provides a good overview of translation validation works up to 2009, as well as presenting several validators for optimization passes in the CompCert compiler, and was an oft-referred source for the redaction of this section.

Origins of Translation-Validation The first instance of translation validation, in spirit if not in name, is probably found in Samet's Ph. D. thesis [95] in 1975. The dissertation presents a validator for an optimizing compiler from a subset of Lisp 1.6 to an assembly language for the PDP-10. Twenty year later, Pnueli, Siegel, and Singerman [77] re-introduced the concept under the name of *translation validation* that is now popular. The authors consider the compilation of the synchronous language Signal to C, and their approach works by encoding both source and target programs into state transition systems. By making syntactic assumptions about the shape of the C code generated from a given

Signal program, they are able to generate a refinement mapping [2] between the two systems. The conditions for the refinement to be correct are expressed in a general-purpose logic and discharged using an automated theorem prover. The authors remark that their approach seems to work "in all cases that the source and the target program each consist of a repeated execution of a single loop body, and the correspondence between the executions is such that a single loop iteration in the source corresponds to a single iteration in the target" — hinting at the fact that many translation validation approaches designed since would struggle with transformations that deeply modify the structure of the code.

Credible Compilation Around the same time, Rinard and Marinov [90] propose a similar idea with the name of *credible compilation*, applied here to imperative programs with pointers that are represented as control flow graphs. Where Pnueli, Siegel and Singerman use syntactic methods and assumptions on the shape of the code generated by the compiler, the credible compiler of Rinard and Marinov is designed to produce alongside the generated code an explicit proof, in an ad-hoc logic, that it simulates the source behavior — a possibility mentioned but not explored by Pnueli, Siegel and Singerman. The proofs generated by the credible compiler contain two types of invariants: *standard invariants* apply to the original program and are used to validate the results of compiler analyses such as points-to analysis that transformations can rely upon, and *simulation invariants* that relate the values of variables between the source and target programs at various execution points. Both kinds of invariants can refer to a finite set of local variables and pointers, and are used for the verification of standard compiler optimizations such as dead code elimination and loop unrolling.

Translation-Validation of Optimizing Compilers The credible compilation framework of Rinard and Marinov is able to verify some compiler optimizations, at the cost of making the compiler generate explicit proofs of those optimizations. The work of Necula [76] is the first to apply translation validation to a pre-existing production-grade optimizing compiler, and provides a strong case for the practicality of the non-proof-generating version of translation validation even in the presence of optimization. Necula's validator works on the IL intermediate language used by the GNU C compiler, and — based on the

remark that validating a single type of transformation at once is simpler than validating an arbitrary combination thereof — is applied between each optimization pass used by the compiler. Necula's validator uses symbolic evaluation to infer a simulation relation, similar to those used by Rinard and Marinov [90], and a custom automated solver to prove its correctness. Basic blocks are used as synchronization points, where memories in both programs must match except on a finite set of locations.

Translation-Validation of Loop Transformations The validators of Necula [76] or Rival [93] are able to handle loop transformations that mostly preserve the execution order of instructions such as loop unrolling by Necula. Other special-purpose validators for specific transformations have been developed, such as the validators for software pipelining of Tristan and Leroy [109] and the validator for loop-invariant code motion of Tristan, Govereau, and Morrisett [108], both implemented within the CompCert verified compiler, or the validator for loop-peeling and induction variable strength reduction of Tate et al. [104]. On the other hand, loop transformations such as loop permutation, loop fusion and loop tiling fundamentally change the structure of the program and are out of reach of these techniques based on simulation relations.

Zuck, Pnueli, and Leviathan [123] propose a translation validation framework that distinguishes between *structure-preserving transformations*, validated using simulation techniques, and *structure-altering transformations* for which a simulation relation does not necessarily exist and for which different techniques must be developed. For structure-preserving transformations, the TVoc compiler described by Zuck, Pnueli and Leviathan produces annotations relating the nodes in the control-flow graphs of the input and output programs. Later work [43] mentions that this requirement can be relaxed as long as at least one node has an annotation within each cycle of the control-flow graph, as that is enough to infer the remaining relations using symbolic evaluation. For structure-altering transformations, on the other hand, a series of pattern-matching "meta rules" are proposed to handle loop transformations such as loop tiling, loop fusion, and loop distribution. The TVoc compiler generates an auxiliary file along with the generated code that indicates which loop transformations were performed — an information that is not readily available in schedule-based compilers but can sometimes be reconstructed [8, 122]. Later work [124] consolidates the meta rules into a single "Permute" rule, and propose to *guess* the sequence

of loop transformations that were (or could have been) applied by the compiler, generating a sequence of intermediate programs that are each proved equivalent with the previous one using the Permute rule.

Product Programs Translation validation is instance verification applied to a compiler, and somewhat blurs the line between instance verification and program verification: a compiler is a program that transforms programs, and translation validation is about checking properties of the output of the compiler, which is itself a program. As such, translation validation approaches share common techniques with the field of automated program verification and analysis. One line of research on translation validation constructs a "product program" embedding the semantics of both the original and transformed program: the question of validation can then be expressed as a *single-program* property on the product program, opening the path to program analysis techniques designed for single-program verification. This idea was introduced by Zaks and Pnueli [120] who apply it to the validation of transformations performed by the LLVM compiler. Their verifier relies on an underlying invariant generation algorithm to build loop invariant, proving their equivalence through bisimulations.

To make the product program approach tractable, some sort of synchronization points must be found where the states of both programs mostly align. Modern approaches to find those synchronization points such as the work of Churchill et al. [26] and Gupta, Rose, and Bansal [46] use combinations of brute-force search and concrete executions to guide the search of appropriate synchronization points and invariants leading to a provable bisimulation.

Equality Saturation and Value Graphs Equality saturation [104] is a technique for reasoning about program equivalence. It works by first converting programs into *Program Expression Graphs* (PEGs for short), a purely functional representation of programs as value graphs. Equality saturation extends PEGs to E-PEGs, able to represent *equivalence classes* of PEGs in a compact way, by repeatedly augmenting the equivalence classes through equality axioms (i.e. rewrite rules) until saturation is reached. This representation of equivalence classes means equality saturation can explore all the possible application orders in which rewrite rules can be applied in a work-efficient way. E-PEGs are inspired by the

E-graphs used in SMT solvers, but are specialized for the purpose of representing programs that may contain cyclic graphs in the presence of loops. Initially designed for the purpose of code optimization in conjunction with heuristics to pick the best representative within an equivalence class, equality saturation can be used for translation validation using a product program approach, by converting both the source and target program to a single E-PEG with shared nodes and checking whether the output of both programs end up in the same equivalence class. Equality saturation has been applied to LLVM by Stepp, Tate, and Lerner [100], who were able to validate many optimizations including dead code elimination, global value numbering, but also loop-invariant code motion, and loop unswitching on about 80% of the SPEC 2006 C benchmarks. Instead of E-PEGs, Tristan, Govereau, and Morrisett [108] use a similar approach, except that normalization is used instead of saturation and uses Gated SSA, another value graph representation of programs. By using normalization instead of saturation, only a single series of axiom applications is considered instead of all possible series of applications, resulting in a better runtime at the cost of introducing a reliance on the order in which the normalization axioms are applied in a non-confluent system. Still, by selecting an appropriate application order, the authors obtain comparable results on the SPEC 2006 as the equality saturation approach, with runtimes that are an order of magnitude faster. These approaches based on a value graph representation are able to handle some structure-preserving loop transformations such as loop-invariant code motion and loop unswitching. [108] mentions successful preliminary experiments with loop fusion and loop fission but it is unclear how to integrate transformations such as loop interchange and loop tiling in these frameworks.

Some transformations, such as loop-invariant code motion, are invisible in a value-graph transformation, and do not need any rewrite rules: the value graph representations of a program before and after loop-invariant code motion is applied are always identical.

Special-Purpose Translation Validators Many approaches to translation validation take a "kitchen sink" approach and build general-purpose validators relying on generic techniques such as symbolic evaluation, automated theorem provers, model-checking and abstract interpretation. These validator directly benefit of improvements in the underlying techniques, but the undecidable nature of the equivalence problem is a double-edged sword: while it would be in theory possible to validate a large number of transformations, these validators can fail to verify transformations that are actually correct and lack a formal characterization of their applicability. On the other hand, it is possible to design special-purpose validators focusing on specific families of optimizations. Such special-purpose validators can exploit the limited range of code

transformations performed by the compiler, and formal completeness results can be obtained. This type of special-purpose validators is particularly suited to be incorporated to the design of formally verified compilers: a formally verified validator opens the gate to the use of optimization passes that would be hard to verify formally. Several special-purpose validators have been developed and formally verified using Coq, such as those developed by Tristan in his PhD dissertation [107], or the alias analysis of Robert and Leroy [94] that has been integrated into CompCert.

Modern Translation-Validation Approaches To this day, translation validation is seen as a valuable and promising avenue of research to find bugs and increase trust in compilers, with projects such as Crellvm [57] adapting credible compilation to the LLVM compiler infrastructure. The authors instrumented the LLVM toolchain to produce correctness proofs using an extensible variant of relational Hoare logic specifically designed for the LLVM IR. The logic relies on the alignment of the source and target programs and cannot express structure-modifying transformations. Crellvm is implemented and formally verified using Coq a validator for the correctness proofs and apply their approach to two major optimizations in LLVM, namely register promotion and global value numbering, exposing four new miscompilation bugs in the process. The validator also enables the compiler to produce partial proofs and integrates with external inference programs that can complete the proofs before performing the validation.

More recently, and still in the context of the LLVM compiler toolchain, Lopes et al. [70] introduced *bounded* translation validation. Bounded translation validation, like bounded model checking, works by unrolling loops up to a given size, abandoning any hope for completeness (bugs that require more iterations than the unroll factor cannot be found). In exchange, after unrolling the control-flow graph is acyclic, and the approach is theoretically able to find bugs in loop transformations without requiring inductive reasoning. A second contribution of the work is to incorporate the *undefined behavior* semantics of LLVM into the validator. LLVM depends on undefined behavior to perform certain optimizations, and a proper validator for these optimizations must be aware of undefined behavior semantics to allow certain transformations that would otherwise be invalid.

Invariant Translation Rival [93] applies translation validation to the non-optimizing compilation of C down to assembly as a one-shot transformation. The approach uses symbolic transfer functions and abstract interpretation to establish a common semantics interpretation of the C and assembly languages and generate verification conditions discharged by a first-order theorem prover. The result of the translation validation is used to implement *invariant translation*: whereas translation validation is concerned with the preservation of semantics between the original and transformed program, invariant translation further requires that global or local invariants proven on the source program be preserved on the target program. Examples of such invariants can be found in prior work [92] and include the absence of division by 0 or of overflowing computations.

TV for Compiler Construction Kanade, Sanyal, and Khedker [56] propose a different approach to translation validation. Primitive transformations on the control-flow graph are developed, and executable soundness conditions are proven independently. Soundness conditions for high-level transformations built on top of the primitive transformations can be obtained by replacing the application of the primitive transformation by its soundness condition. When applying a transformation to a program, the soundness conditions can be executed on the control-flow graph of the program to ensure the correctness of the transformation. Their system is implemented on top of the PVS proof assistant. Another similar work is that of Glesner [42]. In that work, the compiler optimizes SSA graphs using rewrite rules, and produces a trace of the instantiated rules used in the optimization. A separate verifier uses the trace to re-play the optimization while checking the applicability of the rewrite rules. Finally, the verifier validates that the resulting CFG is identical to the CFG obtained by the compiler.

Kundu, Tatlock, and Lerner [64] provide a different application for translation validation techniques. By combining the symbolic evaluation approach of Necula [76] for structure-preserving transformations and the permute rule of Zuck et al. for structure-modifying transformations, they design a validator for the transformation of parameterized programs or program sketches. The validator is then used to prove correct compiler optimizations expressed as rewrite rules in a domain-specific language. The proven-correct optimizations can then be used as components of a certified compiler without needing to use

translation validation at runtime.

8.2 Affine Program Equivalence

Relevant related work on polyhedral compilation as a representation of programs is presented extensively with references in section 2.1. While there is ample literature on the use of polyhedral compilation as an optimization tool, it is not directly relevant to the topic of this thesis and hence not mentioned here. Related work using the polyhedral model for translation validation is called affine program equivalence checking, and described in this section.

In scientific computing, the structure-modifying transformations were increasingly becoming crucial to obtain good performance on computationally heavy programs. These structure-modifying transformations can be performed by polyhedral compilers (that are often implemented as source-to-source transformations for C or FORTRAN), but are also often performed by hand by performance engineers, especially in the domain of embedded computing. Banerjee and Karfa [9] provide a short survey of the area.

Early work tackling the verification of structure modifying transformation by Samsom et al. [96] proposed an approach based on pattern-matching of the right-hand side of assignments on the original and optimized programs, then proving that the loop nests for each pair of matched assignments iterate over the same domains. This approach is restricted to code without recurrences, and the pattern-matching rules can only handle the most basic algebraic transformations.

In this context, Shashidhar et al. [98] propose a translation validation approach for affine programs by converting both the original and transformed statement into the polyhedral model. By assuming that the program is expressed in dynamic single-assignment form (i.e. each array cell is written once), and restricting the allowed transformations to the introduction of caches and the reorganisation of the loop structure of the program without modifying the right-hand side of assignments (except for array indices, as appropriate), it is possible to identify matching statements in both programs. The authors then devise polyhedral checks that the dependencies involving matching statements

are preserved by the transformation, whose correctness is checked by the Omega tool and implies the equivalence of both programs.

Shashidhar et al. [97] introduce Array Data-flow Dependence Graphs (ADDG) to overcome the syntaxic restriction of this previous work. Instead of relying on array names, ADDG express the computation as a graph of operators by eliminating any internal array names, so that only the input and output arrays remain. The method assumes that the source and target programs are in dynamic single assignment form, and that no algebraic transformations have been performed on the computed expressions. The verifier proceeds backwards from the output arrays to build sufficient equalities between array cells in the both programs. In order to recover the matching between two programs across recurrences, the authors rely on an approximation of the transitive closure operation that cannot handle all recurrences. The authors propose to extend the method to handle a finite amount of associative and commutative rewritings by trying all the possible re-orderings.

Verdoolaege, Janssens, and Bruynooghe [115, 116] further improve upon the ADDG method and lift the requirement that the code be in dynamic single assignment form by incorporating dataflow analysis into the method. The new technique still operates on an ADDG, but is implemented as a two-pass approach. In the first pass, equalities between the two programs are inferred based on a backwards pass similar to that of Shashidhar et al. [97]. The backwards pass differs, however, in their treatment of recurrences: instead of approximating the transitive closure of the dependences, the new method optimisticaly computes the affine hull of the equalities obtained by unrolling the loop over a few iterations, an instance of the widening technique from abstract interpretation. From this first pass, the verifier infers a set of "needed" equalities for the proof of equivalence to hold. The second pass is a forward pass that computes the subset of the needed equalities that can actually be proven by saturation from the input equalities. The soundness of the method does not depend on the result of the first pass, and only the second pass actually needs to be trusted. The method uses a similar approach to associative and commutative rewritings as the original ADDG method.

In a different line of research, Karfa et al. [59] extend the ADDG method to handle more algebraic transformations by computing a normal form of the program, essentially inlining all array definitions. Recurrences are not supported, and Banerjee, Mandal, and Sarkar [10] proposes an extension to

re-introduce support for recurrences by trying to match the bodies of the loop, effectively preventing structure-modifying transformation from being applied to recurrences.

The same year as Shashidhar et al. [98], Barthou, Feautrier, and Redon [13] proposed a method for checking the equivalence of systems of affine recurrence equations. This is the same problem seen under the lense of SAREs instead of affine programs with loops. The authors give a simple proof that the equivalence problem is undecidable, even in the absence of algebraic transformations on the data. They propose semi-decision algorithms by reducing the equivalence problem to reachability queries on a memory state automaton, and use (over)approximations of the transitive closure operation to handle recurrences. The resulting algorithm is very similar to that of Shashidhar et al. [97], but to the best of my knowledge, no formal comparison exist.

Iooss, Alias, and Rajopadhye [53] extend the method of Barthou, Feautrier, and Redon [13] to handle *parametric* associative-commutative reductions in the manner of those described in chapter 7. The equivalence problem is cast as a parametric perfect matching problem, and a semi-decision algorithm for the problem based on the augmenting path method used for non-parametric perfect matching is proposed. Due to the parametric nature of the problem, only augmenting paths of non-parametric length can be discovered, which limits the applicability of the method in non-obvious ways. The authors also note that their technique should be applicable to the widening-based method of Verdoolaege, Janssens, and Bruynooghe [116].

All the approaches avoid depend on some sort of backwards pass on the two programs, following the dependencies from the outputs to the inputs, in order to infer a constraint on the input for the outputs to be equivalent. This backwards pass is sometimes followed by a forward pass to check that *a posteriori* the correctness of the inferred equivalences. This limits the transformations on expressions that can be supported to those that can be traversed backwards, possibly with additional branching as in the case of commutativity. On the other hand, our approach only performs a forward pass guided by compiler-generated annotations.

Karfa et al. [58] directly encode the program equivalence problem as a formula which is fed to an SMT solver. They show that the formulas this creates are too complex for SMT solvers to handle in practice. We avoid the issue by using

prophetic expressions as natural stopgaps to generate multiple, simpler queries to the SMT solver.

Finally, Bao et al. [11] propose a dynamic approach, dubbed PolyCheck, to the problem. It exploits the structure of affine program control and data-flow to build a checker with the same structure as the transformed program. If successful, it ensures the validity of all executions for a given problem size.

8.3 Other Approaches

Abstract Interpretation Journault and Miné [54] propose abstract domains to represent and infer properties about matrix manipulating program. They successfully apply their approach in presence of loop tiling, as performed by the Pluto polyhedral compiler. Unlike ours, their approach does not rely on annotations but relies instead on a library of patterns to match assignments with a corresponding semantic predicate. It is not clear how well this library of patterns would scale to arbitrary code transformations.

Specification of Tensor and Array Optimizations The previous approaches prove optimizations on intermediate representations and their transformations. Unfortunately, most tensor compilers do not provide a formal semantics or type system to reason about, or for that matter to prove their correctness w.r.t. some functional specification. TeIL is one significant effort in this direction [91], but its semantics based on combinators is not at the appropriate abstraction level to easily express the iterator-based specifications of most tensor programming languages. We rely on a simple equational language to capture the semantics of these specifications, while demonstrating the translation validation of a tensor compiler independently of the intermediate representations encountered along the flow.

Reinking, Bernstein, and Ragan-Kelley [89] proposes a formal semantics for the Halide compiler. They give an imperative semantics to Halide specifications and implementations. These semantics are used to describe the code generation procedure using Halides core scheduling primitives. This procedure depends on a bounds inference algorithm to find the appropriate loop bounds, after

scheduling, so that the appropriate subsets of the tensors are computed. Reinking et al. introduce holes in their implementation language, and formalize the bounds inference step as a program synthesis problem to fill these holes. To pose this program synthesis problem, their implementation language features "compute" annotations that are somewhat reminiscent of our prophetic annotations. These take the form of scopes such that, after exiting the scope, the values in a rectangular region of the array are equal to the values in the same region in the tensor. At first glance, these annotations seem more expressive than our prophetic annotations, because a prophetic annotation could be expressed using a one-element scope around an assignment. However, Reinking et al's "compute" annotations are generated at a coarser granularity which is not sufficient for our verification procedure, and do not include the value of recurrence variables.

Formal Verification of Tensor Compilers Liu et al. [69] have implemented in Coq the ATL language introduced by Bernstein et al. [17] to represent tensor computations using map and reduce combinators. They prove "reduction rules" as theorems stating parameterized equalities between ATL expressions, using a formal semantics of ATL developed in the Coq proof assistant, and provide a framework for developing "schedules" using the proof assistant's tactic mechanism. Program optimization is performed by the user within the proof assistant, using a combination of Coq's primitive tactics and the framework's framework. The user can also implement arbitrary new optimizations, provided they can be proven correct using the proof assistant's logic. This system is able to represent sufficiently complex transformations to be competitive with Halide on the classic two-dimensional blur example.

In a different direction, Courant and Leroy [29] verify an implementation in Coq of a polyhedral code generation algorithm based on Bastoul [15]'s version of Quilleré's algorithm. The implementation led to the design of an intermediate language, PolyLoop, to represent the intermediate stages of the code generation process, and has uncovered a possible error case in Quilleré's algorithm that however does not seem to happen in practice. The formal proof ensures that the generated code evaluates statements in an order compatible with the given schedule, without verifying that the schedule respects the dependencies of the original program.

9 Conclusion

9.1 Summary of My Approach and Results

This dissertation on the translation validation of tensor compilers follows a reflection around the formal guarantees provided by compilers for low-level tensor specifications as used for image processing and deep learning applications. These compilers manipulate tensors (or, equivalently, multidimensional arrays) and mostly focus on *structure-modifying transformations*, making their verification out of reach of traditional verification techniques based on bisimulations — and yet, as compilers, their correctness is paramount to the trust in the software infrastructure that uses them.

To address the issue, we can turn to two wide categories of approaches to guarantee the correctness of compilers: in formal verification, the goal is to build a compiler with a machine-checked proof that it can only produce correct output; while in translation validation, a separate validation tool is developed to accompany the compiler. This validation tool can check that the compilation is correct for a given input program and compiler output. While formally verified compilers for tensor languages such as the one of Liu et al. [69] have been developed, formal verification is difficult to apply to an existing compiler and essentially requires rewriting the compiler from scratch. Formal verification also imposes a high maintenance burden: any modification to the compiler now requires adapting the correctness proof, becoming more costly and potentially out of reach of existing compiler developers that have not been trained in formal methods. Hence, we rather turn to translation validation: the validator can be developed (or formally verified itself) separately from the compiler, and can be shared across multiple versions of the compilers or even across different compilers, making both integration and maintenance simpler.

For this to be true, translation validation has one critical requirement: the validator must be powerful enough to actually establish equivalence of the input and output programs, possibly relying on annotations from the compiler. At the beginning of this thesis, it was not clear whether this could be the case for tensor compilers. Some approaches to translation validation for structure modifying transformations require the compiler to output a sequence of the loop transformations it performed, or rely on knowledge of the pass ordering to guess the transformations applied — both approaches sounding inappropriate for tensor compilers implemented as code generators. Another family of techniques, known as *affine program equivalence checking*, applies translation validation to loop transformations by using a polyhedral program representation. These rely on delicate syntactic correspondences between the original and transformed programs and hence cannot verify some of the more complex algebraic transformations performed by tensor compilers.

The solution explored in this thesis relies on the observation that many tensor compilers work as *code generators*: roughly speaking, the compiler builds an imperfect loop nest around the tensor definitions in the specification, while backing up the tensors by arrays of possibly smaller domains. Arguing that this makes it reasonable for these compilers to provide an explicit correspondence between array writes in the generated code and tensor definitions in the input specification, we could then devise and formally specify both a language annotated with this correspondence and a verification condition generator for programs written in this language. Assuming affine control flow, the verification condition generator first abstracts the program behavior using a symbolic representation that represents an optimistic, or prophetic, evaluation, as if all annotations held. This symbolic representation can be checked against the expected output from the specification using an SMT solver. The output of this optimistic evaluation is also used to build loop invariants and verification conditions that can also be checked using an SMT solver and ensure that concrete evaluations of the program indeed match this optimistic evaluation — tying the knot, so to speak.

With the goal of being as widely applicable as possible, the validation algorithm expects a specification expressed as a SARE, an intermediate representation of polyhedral programs, decoupling the verifier itself from the details of the specification language. The implementation language developed in this manuscript and called Sched is also an annotated subset of the Stmt language originating in the Halide compiler and that is used as the output of other low-

level tensor compilers such as TVM, Tensor Comprehensions, and Tiramisu.

To validate the assertion that these annotations are effectively enough for the validator to succeed on the transformations performed by industrial tensor compilers, I have implemented the validation algorithm in OCaml, using the `isl` library, and applied it to the Halide compiler, instrumented to generate annotated Stmt which is readily converted into Sched and fed into the validator. On a limited but realistic set of benchmarks extracted from the Halide repository, the validator has been shown to be a viable candidate for the parametric verification of important tensor primitives such as matrix multiplication, and compares favorably in terms of performance with state-of-the-art affine program equivalence tools when applicable.

Finally, I have developed extensions to the core verification algorithm to be able to verify transformations involving *reductions*, an important primitive representing the iterated application of an associative and commutative operator.

By providing theoretical foundations and a prototype implementation for a core algorithm devoted to the task, anecdotal evidence that tensor compilers can be instrumented to produce the required annotations, and experimental evidence that the algorithm successfully establishes correctness in practice, this thesis makes a strong case for the viability of translation validation applied to low-level tensor compilers. It also leaves many questions unanswered, notably regarding the applicability of the approach outside strictly affine specifications and transformations: some of these questions are explored in the following sections.

9.2 Ecosystem Integration

The work presented in this thesis applies to low-level tensor compilers deriving imperative kernels from pointful specifications. These imperative kernels are then lowered to low-level representations such as the LLVM IR and fed into traditional compilers that ultimately emit assembly code. The validation techniques developed here can thus be combined with existing translation validation or compiler verification approaches for traditional compilers to obtain formal guarantees down to the binary produced. However, this would

require formally describing the lowering process from SCHED to the underlying low-level representation: in the presence of parallel loops, this requires some care because consecutive non-communicating parallel loops of SCHED are typically lowered to a single loop with synchronizations instead, as discussed in section 4.3.

On the other side of the pipeline, in the domain of deep learning, users typically interact with higher level libraries such as TensorFlow, PyTorch, or JAX at a different abstraction level: operations in these language take and return tensors, and represent computations using (static or dynamic) graphs of operators that can be transformed using high-level rules such as the commutativity of *matrix* multiplication. These high-level transformations operate, for us, at the specification level and are out of scope of the approaches described in this thesis. On the other hand, the operators themselves are usually implemented by either delegating to lower-level specialized libraries (when applicable), or using handwritten low-level code. The validation approach presented in this manuscript can integrate with these frameworks to help guarantee the correctness of the operators, which can be considered primitive building blocks for the rest of the framework.

9.3 Sparse Arrays

Some applications domains such as computational chemistry use sparse arrays. TACO [62] is a tensor compiler that can express optimizations on both sparse and dense arrays. Unlike dense arrays, sparse arrays do not provide random access to their elements. Instead, nonzero elements can be iterated over in order; when multiple sparse arrays are involved, they can be iterated over jointly by comparing the next nonzero indices of both arrays. The validation methods proposed in this dissertation should generally be applicable to implementations involving sparse arrays, provided an appropriate representation for the iteration on sparse arrays can be found. A simple idea is that if an iteration is skipped due to a missing element in a sparse array, any cell that would have been written in that iteration must already hold the value that would have been written had the iteration been executed with a value of 0 instead. While this is enough to represent loops on sparse arrays, more research is needed to understand how to properly represent the result of transformations such as tiling applied to the

sparse iterators.

9.4 Floating-Point Arithmetic

This manuscript mostly ignores the difficulties occurring in the verification of computations involving floating-point numbers. And yet, in applications such as linear algebra, image processing, and deep learning, computations on theoretically continuous values are often expressed using floating-point numbers (or simply *floats*), usually represented not only following the IEEE Standard for Floating-Point Arithmetic (IEEE 754) but also sometimes using alternate representations such as the `bfloat16` format introduced by TensorFlow. We will now explore these difficulties in the context of compilers and consider how they fit with the validation approach proposed here.

Floats, independently of their representation, are a particularly ill-behaved approximation of real numbers, for many reasons:

- To represent overflow, floats use two distinguished values representing respectively $+\infty$ and $-\infty$, which are not real numbers.
- As a consequence, floats also distinguish between *positive* and *negative* zeroes, values that must compare equal yet have different inverses ($+\infty$ and $-\infty$, respectively).
- To represent erroneous values such as the result of dividing zero by itself, floats use distinguished "not-a-number", or NaN, values. NaNs have the peculiar property that it should compare different (and unordered) to any value, including itself. Two different kinds of NaNs exist, *signaling* and *quiet*, with different propagation rules. Signaling NaNs may interrupt the normal flow of execution when consumed by an operation.
- Even ignoring the presence of infinities and NaNs, most of the reasoning rules mathematicians are used to on real numbers, such as associativity and distributivity laws or the equality $\frac{a}{b} = a \cdot \frac{1}{b}$.

Until the recent IEEE 754-2019 version of the standard, the specification of the min *and* max *operations allowed values such as* $\max(+0, -0)$ *to be either* $+0$ *or* -0 *at the discretion of the implementation.*

Because of these properties, compilers are very restricted in their ability to

optimize computations involving floating-point numbers, but programmers rarely rely on all of these properties, and often do not care about the exact order in which computations are performed: there is no reason for the programmer writing an image processing pipeline to compute the luminance of a pixel as $\alpha_r r + (\alpha_g g + \alpha_b b)$ rather than $(\alpha_r r + \alpha_g g) + \alpha_b b$, or for the engineer writing a deep learning primitive to compute the sum $\sum_{0 \leqslant i < N} A(i)$ forward rather than backward. For this reason, modern compilers provide flags to enable so-called "fast-math" optimizations that try to respect the spirit, rather than the letter, of the program. Unlike traditional compilers, and in line with the habits of their target communities, tensor compilers such as Halide or Tensor Comprehensions enable "fast-math" optimizations by default.

Halide provides a both a global "strict float" mode and local annotations to disable "fast-math" optimizations.

The "fast-math" optimizations can be roughly separated into three categories, ordered by the challenges they pose to formal verification:

- The least controversial "fast-math" optimization simply assumes the absence of signaling NaNs, ensuring that floating point computations do not disrupt the control flow.

- Another class of "fast-math" optimizations are conditionally sound: these are the optimizations that are valid provided that no NaNs, infinities, and/or negative zeroes occur during the execution of the program, whether as inputs or in intermediate computations.

- The last class of "fast-math" optimizations are *fundamentally unsound*: these correspond to the application of many algebraic rules (e.g. associativity) valid on the reals, but not on the floating-point numbers,

In the context of formal verification, the absence of signaling NaNs is reasonably safe to assume: the program can be configured to make all floating-point operations non-signaling (i.e. no operation ever returns a signaling NaN), in which case the condition reduces to ensuring the absence of signaling NaNs in the input data. This is an assumption that is often made (sometimes implicitly) by verification tools targeting floats, because it means that floating-point operations can be modelled by pure functions. The formalization presented in this dissertation, and the tool implemented in chapter 6, implicitly make this assumption already.

The second category — the conditionally sound optimizations — can be

handled by assuming, possibly through annotations, appropriate magnitudes for the values in the input tensors and arrays. We can then generate auxiliary verification conditions ensuring the absence of infinities, NaNs, and negative zeroes in the resulting computations. This approach has been used successfully for instance in the Astrée static program analyzer [30] and by Menendez, Nagarakatte, and Gupta [73] for the verification of floating-point optimizations in LLVM.

These two categories of transformations can be handled by using a native or axiomatic representation of floats in the underlying verification condition solver. The implementation presented in chapter 6 can be configured to use the native Z3 type for floating-point numbers, and checks bit-wise equality of the original and transformed expressions. This representation is trivially compatible with constant propagation performed by the compiler using floating-point arithmetic.

On the other hand, the third category of "fast-math" optimizations is harder to handle properly: the equality $a + (b + c) = (a + b) + c$ is *false* on floats, and cannot be simply added as an axiom without making the whole system inconsistent. One alternative would be to use traditional approaches to floating-point verifications, e.g. using interval analysis, to ensure that the result of the transformed computation is "not too far" from the original computation — but because "not too far" is not a transitive property (i.e. if v_1 and v_2 are within distance ϵ of each other, and v_2 and v_3 are also within ϵ of each other, v_1 and v_3 are not necessarily within ϵ of each other), it is not clear how to compose these approaches with the two-step approach of prophetic evaluation that relies heavily on transitivity.

Another alternative is to follow common practice in numerical computing and to accept program transformations that are valid on the reals. Following this practice, the default behavior of the verification tool presented in chapter 6 is to (incorrectly) represent floats using Z3's built-in type for real numbers (although a custom axiomatization of reals as an abstract type equipped with associative and distributive operators is an alternative). While this provides no formal guarantees on the output of the program when ran using floats, it can still increase the trust in the correctness of the compilation (it is not "obviously incorrect"), especially when the goal is to prevent accidental bugs rather than defeat an adversarial compiler. Unfortunately, while mostly appropriate for the algebraic specifications encountered in linear algebra or deep learning, it

Consider for instance the replacement of x with $(x + f_M) - f_M$ where f_M is the largest representable float. This is correct on the reals, but the second expression evaluates to 0 for any finite value of x!)

causes issues with the frequent presence of constant computations in image processing pipelines: the compiler is performing constant propagation using floating-point math, which fails to validate under the validator's use of real numbers. Verifying constant propagation using this approach would require the compiler to use exact rational math for floating-point simplifications, which does not make much sense.

An idea for a third alternative that tries to bridge the gap would be to represent floats using algebraic expressions. For instance, if x and y are two floats, $x + y$ is represented by itself as an expression tree. Expression trees can be equipped with *directional* rewrite rules encoding both floating-point equalities (for constant propagation) and acceptable "fast-math" rules (but, crucially, *not* constant propagation in the reals). Instead of stating that the value computed by the implementation is equal to the value computed by the specification, we can state that implementation value must be accessible from the specification: if location `a[i]` maps to an expression t in a symbolic heap, it means that at runtime, `a[i]` contains a value accessible from t by following the directional rewrite rules. Reduction is transitive (if e_1 reduces to e_2 and e_2 to e_3, e_1 also reduces to e_3 by concatenation) and hence compatible with prophetic evaluation. The use of directional rewrite rules is enough to remove at least the obvious sources of inconsistencies: for instance, we can have floating point values a, b and c such that $(a + b) + c = 1$ and $a + (b + c) = 0$, which invalidates the equality $(a + b) + c = a + (b + c)$ because it entails $0 = 1$. On the other hand, if $(a + b) + c$ reduces to 1 and $a + (b + c)$ reduces to 0 using unidirectional reductions, we can apply the associativity rule before reducing to see that both expressions can evaluate to either 0 or 1, but it does not introduce inconsistencies because it does not imply a reduction between 0 and 1. This approach might be able to give a formal definition to the transformations performed by the compiler, although the ability for the compiler to duplicate expressions and compute them differently makes every use of floating point variables nondeterministic (e.g. X() − X() could be nonzero if the compiler decides to replicate the computation of X() and applies different optimizations to both copies). On the other hand, it is unclear if it could be efficiently implemented in automated solvers, and whether the formal guarantees provided would be useful to users of the compiler.

One final remark is that if we were to allow conversions from floats to indices when lifting the affine restrictions (see Section 9.6), special care would be required because "fast-math" transformations could lead the conversion to

result in a different index, possibly creating out-of-bounds accesses.

9.5 Overflow Checking

Another topic that has been absent of the discussion so far is integer overflows, that have simply been ignored by modelling all computations on array indices using unbounded integers. Technically, this is a threat to the formal soundness of our results when running the generated code using machine integers instead. There are two general approaches to dealing with this threat.

The first approach is to check the absence of integer overflow separately. Since we require a strict separation between indices and values, we can express the absence of integer overflow and underflow for a given width as a condition on the range of the values for each index computation, which can be reduced to a condition on the program parameters by eliminating the intermediate variables. This is similar to the approach of Cuervo Parrino et al. [32], and should compose well with the rest of the validator (if no overflow in the computation using machine integers, the result is the same as the computation using unbounded integers).

The second approach requires to explicitly model overflowing computations during the validation by assuming some fixed well-defined behavior of signed integer overflow and using a piece-wise or modulo expression to symbolically represent index computations. This would increase the runtime of the algorithm, potentially drastically, by introducing disjunctions and/or auxiliary variables in the `isl` representation. This is the case in the ISA tool I compare to in chapter 6, hence the overflow checking has been disabled in the experiments for a fair comparison.

Finally, it should be noted that this discussion only considers overflow in *index computations*. Integers used as values stored in arrays are value types represented by machine integers of the appropriate width in Z3, and follow standard overflow rules, assuming wrapping arithmetic. In the case of signed integers, Halide can perform simplifications that are only valid under the assumptions that no signed overflow occurs in the subset of the specification that is being computed, and these simplifications are currently rejected by the

validator. It should be possible to allow these simplifications by modifications by adding additional assertions that no overflow occurs in the appropriate subset of the specification, using dependence analysis. An additional question is how to express this constraint explicitly on the input arrays.

9.6 Non-Affine Specifications and Schedules

In this dissertation, we have made the assumption that specifications and schedules are affine (including piece-wise affine). While this captures a wide variety of both specifications and schedules, there are exceptions, some of which are discussed here. Non-affine expressions involve both non-linear arithmetic (e.g. terms involving a multiplication between two variables) and data-dependent expressions (e.g. specifications where an index is computed as an array access, as in $A(B(i))$). We will discuss non-affine schedules, in particular *tiling*, specifications with non-affine reads (i.e. non-affine indices on the right-hand side of a specification), and non-affine writes.

9.6.1 Non-Affine Reads

The first source of non-affine expressions in a tensor compiler is when a non-affine expression appears on the right-hand side of an assignment. For instance, a strided convolution with filter F can be expressed as $B(i) = \sum_{0 \leqslant r < R} A(i + r \cdot S) \times F(r)$ which contains the non-affine index $A(i + r \cdot S)$ (it is not affine due to the multiplication $r \cdot S$).

Such non-affine expressions in indices make the expression no longer expressible as a SARE; however, as long as these accesses are read accesses on the right-hand side of assignments, this can be integrated as an extension, provided that a non-linear solver is used to check that the non-linear index expressions are within the domain of the accessed tensor. This check can be performed using a modern SMT solver such as Z3, that have decent support for non-linear arithmetic, especially simple uses thereof. When tensors have infinite domains, such as when considering a Halide specification, no extra check is necessary at this level.

At the implementation level, this specification can be implemented "trivially" by the following program:

```
for i = 0 to N - 1 do
  b[i] {0} := 0
  for r = 0 to R - 1 do
    b[i] {B(i)@(r)} := b[i] + a[i + r * S] * f[r]
```

In this program, non-affine expression only occur as indices to arrays on the right-hand side of assignments. Since prophetic evaluation only looks at the left-hand side of assignments and at prophetic expressions, we only need to take care of these non-affine accesses in the symbolic evaluation. The access `a[i + r * S]` is non-affine, but we know the value of the array `a` at that point in the program, hence we can obtain a symbolic expression representing the value of `a[j]` for an arbitrary `j` by summarization. We can then substitute `j` with `i + r * S` in that expression, to obtain a symbolic expression representing the value in `a[i + r * S]`. We also must generate a verification condition that `i + r * S` is within the defined bounds of `a`, a check that must be done using Z3 (whereas we usually use `isl` to verify that indices are within bounds).

If the non-affine read occurs within a parallel loop, we must either approximate the read as potentially reading *all* the cells in the array, or keep enough non-linear information in the set of reads ρ to generate a verification condition using Z3 instead of `isl` to check the absence of read-write races.

Note that this approach requires to find proper expression contexts within the non-affine expression: for instance, in the example, the symbolic value representing `b` could be:

$$\{b\langle i\rangle \longrightarrow (B(\square_1) + A(\square_2 + \square_3 \cdot \square_4) \times F(\square_5))\langle i, i, r, S, r\rangle\}$$

9.6.2 Non-Affine Specializations

A specialization is a duplication of the code used when a specific condition is true, often in order to better tune performance. For instance, the CUDA matrix multiplication used in chapter 6 has different specializations for small and big matrices that make scheduling decisions appropriate for the volume

of computation available. Specializations are of this type (i.e. simple case analysis on the size of the program parameters) are amongst the most common specializations and can be expressed using affine constraints and handled by the existing validator algorithm. On the other hand, some specializations are non-affine, often because they are data dependent.

Let us first consider the case of a specialization that is used to simplify the expressions inside a non-affine select conditional. For instance, consider the following Halide specification, that selects the rows of matrix C, except for columns that are a multiple of T (a parameter), in which case the row from matrix B is selected instead:

```
D(i, j) = select(i mod T == 0, B(i, j), C(i, j))
```

While a naive implementation of the specification can look like the first program below the left, a specialized implementation that avoids repeating the test can look like the second program below.

This transformation is known as loop unswitching.

```
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    d[i, j] := select(i mod T == 0, b[i, j], c[i, j])
```

```
for i = 0 to N - 1 do
  if i mod T == 0
    for j = 0 to M - 1 do
      d[i, j] := b[i, j]
  else
    for j = 0 to M - 1 do
      d[i, j] := c[i, j]
```

Even though the conditional uses a non-affine condition, both branches of the "if" always write to the same set of locations. In this case, we can perform a simple transformation when treating the if statement, by wrapping the symbolic expressions associated with each branch within the corresponding non-affine select statement. Here, the condition $i \bmod T = 0$ does not contain any array read and can be evaluated prophetically; if the computation is data dependent (e.g. if the condition was `a[i]` instead), an extra annotation is needed indicating a prophetic expression (e.g. $A(i)$). In that case, a verification condition must be generated by the symbolic evaluator ensuring that the asserted prophetic expression is equal to the runtime expression.

This approach of reconstructing a select expression should work for most cases of specialization with non-affine expressions, with the possibility of an additional annotation for data-dependent specializations. However, in some cases, one of the branch of the specialization is a no-op and can be simply removed by the compiler.

Consider for instance the general matrix multiplication (GEMM) $D = \alpha AB + \beta C$. If β is 0, the GEMM is degenerate and becomes a simple matrix multiplication: there is no need to consider the values in matrix C, because they are nullified by the multiplication with 0. GEMM implementations often have a "fast path" specialization for that case, avoiding loads of C entirely. A simple specialized implementation might look like this (assuming that D is specified in two stages D_0 computing αAB and D_1 computing $D_0 + \beta C$):

```
for i = 0 to N - 1 do
  for j = 0 to M - 1 do
    d[i, j] := 0
    for k = 0 to P - 1 do
      d[i, j] {D0(i,j)} += alpha * a[i, k] * b[k, j]

// Fast path when beta = 0
if (beta != 0)
  for i = 0 to N - 1 do
    for j = 0 to M - 1 do
      d[i, j] {D1(i,j)} += beta * c[i, j]
```

Since `beta` is of a value type (e.g. a float), the test `beta != 0` is not affine, and the method presented in this dissertation fails at representing this code. However, the following symbolic heap could represent the final value of the `d` array:

$$\{\, d\langle i, j\rangle \to \text{select}(\beta \neq 0, D_1(i,j), D_0(i,j)) \,\} \tag{9.1}$$

Computing this symbolic heap cannot be done prophetically, because we cannot give a prophetic evaluation to the if statement: the prophetic expression `d[i, j]` after evaluating the if statement is unknown when the condition is false. In this case (i.e. the specialization occurs at top-level — and more generally, when the specialization does not occur within a sequential loop), we can use the current state of the prophetic evaluation to find the value $D_0(i,j)$ and the heap above. For non-affine ifs that we expect come from a specialization,

it is reasonably safe to assume in such cases that the prophetic annotations within are still valid within the elided branch. This is a sound (but incomplete) over-approximation that we can expect to be enough to handle the non-affine patterns obtained through specialization.

A specialization occurring within a parallel loop may lead to over-approximating the set of locations read and written, possibly leading to false positives in the detection of read-write races. To handle this, we can attach the non-affine condition to the written location and use Z3 instead of `isl` for the tests when evaluating a parallel loop. It is unclear whether the added complexity and verification time would be worth it, as compilers are unlikely to exploit non-affine or data-dependent conditions to enable parallelism.

The contextual information from the non-affine conditional should be kept as an additional path condition to be given as hypothesis to any verification condition generated within the conditional, including the bounds checking for data-dependent reads. If the non-affine conditional uses nonlinear arithmetic but no data-dependent indices, it may be worth generating bounds checks within the conditional using Z3 even for the affine case because the non-affine conditional can prevent out-of-bounds situations for an affine index (e.g. the condition $\mathtt{i} \times \mathtt{i} \leqslant 4$ is non-affine but forces the affine condition $-2 \leqslant \mathtt{i} \leqslant 2$).

Some comparison should be made between the approach described here and the method of Verdoolaege et al. [117] to handle non-affine control that is present in both the specification and the implementation. Their handling of such non-affine control is similar to the capabilities of our approach using a non-affine select operator: when encountering a non-affine conditional, their approach ensures that the same array elements are written in both branches of the if, and the if is then transformed into expression-level ternary expressions equivalent to the select operator. This is the same transformation we propose to use, but our approach is better equipped to handle ifs containing an elided branch by either exploiting the prophetic annotations in the non-elided branch or exploiting the prophetic evaluation of a prefix program.

9.6.3 Non-Affine Writes and Histograms

To handle non-affine or data-dependent writes, Verdoolaege et al. [117] proposes to reduce them to non-affine conditionals. More precisely, an assignment `a[f(i)] = g(b[i])` is treated as if it was the following program:

```
for j = n to m do
  a[j] := select(j = f(i), g(b[i]), a[j])
```

where $f(i)$ is known to be bounded below by n and above by m.

In our case, a non-affine write must first be represented in the specification. To properly understand how to do this, we can first consider the simpler case of non-affine writes: histograms. A histogram computes the number of times each value occur in an array or tensor. A histogram specification in Halide can be written as follows:

```
Var i;
RDom r(0, R)
H(i) = 0;
H(A(r)) += 1;
```

This can be represented equationally (as an "extended" SARE that is no longer affine) using a reduction over the set of indices r such that $A(r)$ is equal to the current position in H:

$$\begin{aligned} H_0(i) &= 0 \\ H_1(j) &= \sum_{0 \leqslant r < R \wedge j = A(r)} 1 \end{aligned}$$

An implementation for this specification can look as follows, using explicit annotations for the prophetic value of `a[r]`:

```
for i = 0 to N - 1 do
  h[i] {H0(i)} := 0
for r = 0 to R - 1 do
  h[a[r] {A(r)}] {H1(A(r))@(r)} += 1
```

If $A(r)$ was an affine expression, using the proposed symbolic heap representation for reductions, we would want to compute the symbolic heap within the reduction loop as:

$$\{h\langle A(r)\rangle \to [[r], (H_1(\square_1)@(\square_2))\langle A(r), r\rangle]\}$$

Since $A(r)$ is not an affine expression, we can instead introduce a fresh variable j within the range of array h, similar to the encoding of Verdoolaege et al. [117] mentioned above, as well as an additional component to the tuple to represent the non-affine condition $j = A(r)$:

$$\{h\langle j\rangle \to [[[r], [(\square_1 = A(\square_2))]jr], [(H_1(\square_1)@(\square_2))]j, r] : 0 \leqslant j < N\}$$

This represents a heap where `h[j]` is associated with $\sum_{r'=r \wedge j=A(r')} H_1(j)@(r')$ which is equal to 0 when $j \neq A(r)$ and $H_1(A(r))@(r)$ otherwise.

Because we are dealing with a reduction with an associative and commutative operator, we can compute the symbolic evaluation of the loop by taking the union of these representations. We must check that the values appearing as index to the reduction are distinct, because we do not know how to represent multiple values associated with a reduction index, as explained in chapter 7; fortunately, the index to the reduction is r, which is an affine expression, and we can check the disjointness using `isl` as usual. If the underlying operator is only associative, but not commutative, we can also check that the reduced indices are increasing along the loop normally. In both cases, we end up with the following symbolic heap to represent h after the loop:

$$\{h\langle j\rangle \to [[[r], [(\square_1 = A(\square_2))]j, r], [(H_1(\square_1)@(\square_2))]j, r] : 0 \leqslant j < N \wedge 0 \leqslant r < R\}$$

This symbolic heap represents the equalities, for $0 \leqslant j < N$:

$$h[j] = \sum_{0 \leqslant r < N \wedge j = A(r)} H_1(j)@(r)$$

This can be summarized to $H_1(j)$ for $0 \leqslant j < N$, by checking that when the condition $j = A(r)$ and the condition in the definition of H_1 are both true, then the body $H_1(j)@(r)$ is equal to the body in the definition of H_1, and when only one of the conditions is true, the corresponding element is equal to the neutral element of the reduction, if it exists.

We can now go back to the treatment of non-affine or data-dependent writes that are not histograms. Although we have encoded assignments by adding additional tensor indices, we can remark that the "assignment" operator $\odot$ defined by $x \odot y = y$ is associative (but not commutative). An assignment within a recurrence can thus be modelled as a non-commutative reduction using this operator, and the modelling described above for reductions can be applied to regular assignments. Compared to an arbitrary non-commutative operator, $\odot$ has the additional property that the value of the reduction is the value of the last defined index. This permits relaxing some restrictions when combining reductions, which may be useful here.

9.6.4 Parametric Tiling

An important program transformation performed by tensor compilers is *tiling*, already mentioned in this dissertation. Some forms of tiling, namely tiling of the low bits of an index by a constant factor, can be expressed as an affine transformation, and modelled directly using `isl`'s Presburger sets and relations. However, other forms of tiling, and notably tiling where the tiling factor is non-constant, cannot be represented this way. This is another form of non-affine use case that is nontrivial to handle, because we must now have non-affine expressions (e.g. $\lceil \frac{N}{T} \rceil$) in loop bounds.

Let us consider a simple case of parametric tiling for a one-dimensional array copy $B(i) = A(i)$. A tiled program with tile size T may look as follows:

```
for i0 = 0 to ceil(N / T) - 1 do
  let m = min(N - i0 * T, T) in
  for i1 = 0 to m - 1 do
    let i = i0 * T + i1 in
    b[i] {B(i)} := a[i]
```

This is an issue for our verifier because the expressions `i0 * T + i1` and `ceil(N / T) - 1` are non-affine. The first one could possibly be handled as a non-affine write (although it wouldn't be entirely satisfactory, as we would have to treat the write as possibly touching every cell of the `b` array), but the second one is involved in a loop bound, and it is unclear how it can be integrated in our representation.

In their work on monoparametric tiling, Iooss, Alias, and Rajopadhye [52] show that in the case where all parametric tilings occurring in an expression are multiples of some base tiling factor, the parametric construct is expressible as an affine construction by decomposing every variable i involved in the tiling into a "block" part i_b and a "local" part i_l using euclidean division such that $0 \leqslant i_l < T$ and $i = i_b \times T + i_l$. If we introduce the monoparametric parameters N_b and N_l such that $N = N_b \times T + N_l$ and $0 \leqslant N_l < T$, substitute for N, and perform the appropriate simplifications we can express this implementation using a "view" of array `b` that is indexed using pairs (i_0, i_1) where $0 \leqslant i_0 < Nb$ and $0 \leqslant i_1 < T$ such that $0 \leqslant i_0 \times T + i_1 < N$:

```
for i0 = 0 to select(Nl = 0, Nb - 1, Nb) do
  let m = select(i0 < Nb, T - 1, Nl - 1) in
  for i1 = 0 to m do
    let i = i0 * T + i1 in
    b[(i0, i1)] {B(i)} := a[i]
```

Since there is an order-preserving (when the tuples (i_0, i_1) are ordered using the lexicographic order) bijection between the original index space of `b` and the new index space of `b`, we can perform the verification using this tiled view of `b`. Obviously, the equality $N = N_b \times T + N_l$ cannot be used in affine contexts, and non-tiled accesses such as `a[i]` become non-affine accesses and need to be handled using the techniques for non-affine accesses described earlier in the section; alternatively, we can also envision introducing a tiled view of array `a`.

Since we have a tiled representation of array `b`, any use of `b` that is not tiled with the tiling parameter T is now non-affine and needs to be handled with care. In practice, we can expect one of two scenarios: either array `b` is used locally within a single tiling (possibly parametric), in which case only the tiled view needs to be considered, or the tiling for array `b` is used locally, then forgotten (for instance, the writes to `b` may be tiled with factor T only for one of its defining stages). In this case, it is useful to transform back from the tiled view to a non-tiled view in order to avoid non-affine accesses. For instance, we want to transform the tiled symbolic heap:

$$\left\{ b\langle i_0, i_1\rangle \rightarrow B(\square_1 \times T + \square_2)\langle i_0, i_1\rangle \;\middle|\; \begin{array}{l} (0 \leqslant i_0 < N_b \wedge 0 \leqslant i_1 < T) \vee \\ (i_0 = N_b \wedge 0 \leqslant i_1 < N_l) \end{array} \right\}$$

into the untiled symbolic heap:

$$\{\, b\langle i\rangle \rightarrow B(\square)\langle i\rangle \mid 0 \leqslant i < N \,\}$$

This transformation can be performed by recognizing the following equalities between boxes, that hold under the conditions $N = N_b T + N_l$, $0 \leqslant N_l < T$, $0 \leqslant i_1 < T$, and $0 \leqslant m \leqslant p \leqslant T$:

$$\alpha N_b + n \leqslant i_b < \beta N_b + k \Leftrightarrow \alpha(N - N_l) + nT \leqslant i < \beta(N - N_l) + kT$$
$$i_b = \beta N_b + k \wedge m \leqslant i_l < p \Leftrightarrow \beta(N - N_l) + kT + m \leqslant i < \beta(N - N_l) + kT + p$$

We can compute the condition for the interval over i_l to be full, i.e. when the set removed from the interval is empty. Over that set, we can eliminate i_l by finding α, n, β and k such that $\alpha N_b + n \leqslant i_b < \beta N_b + k$ is full (typically, $\alpha = n = k = 0$ and $\beta = 1$). We then replace this with the condition $\alpha(N - N_l) + nT \leqslant i < \beta(N - N_l) + kT$. Then, we decompose the remaining sets into equalities $i_b = \alpha N_b + k$ (typically, $\alpha = 1$ and $k = 0$), we compute appropriate n and m so that the interval on i_l is full (typically, $n = 0$ and $m = N_l$), and we get the $\alpha(N - N_l) + kT + n \leqslant i < \alpha(N - N_l) + kT + m$. If we take $\alpha = 0$, $\beta = 1$ and $m = N_l$, we obtain $0 \leqslant i < N - N_l + N_l$ and we can eliminate N_l.

This "untiling" transformation should be guided using annotations provided by the compiler similar to the accumulate construct for reductions, which should be relatively easy for the compiler to insert around the scope the tiling is performed in. Only information about *where* the untiling should be performed are needed. If the transformation fails, we can either report an error, or keep using the tiled version of the array using non-affine accesses. Outside of monoparametric tiling, if an appropriate monotonous bijection is given, this same approach should generalize to more complex schemes such as the sum decomposition $N = N_1 \times T_1 + N_2 \times T_2$ used by Tollenaere et al. [106].

9.7 Array linearization

The focus of this presentation has been on multidimensional tensors and arrays, because it is the way specifications are written. Ultimately, tensor compilers transform multidimensional arrays into flat buffers in linear memory, often creating non-affine indices due to parametric array sizes. For instance, the

array access `a[i, j]` might be transformed into a buffer access `a[i * M + j]` or `a[j * N + i]` depending on the layout.

Reconstructing affine multidimensional indices from the linearized indices is possible, but difficult and implementations often resort to runtime checks to ensure the reconstructed indices are correct [34]. Fortunately, if we know both the multidimensional and linearized indices, it is easier to check that the same injective linearization function is used for all accesses using a solver such as Z3, because it does not involve reconstructing any expression. This is a fairly simple non-affine condition that can often be expressed as a piece-wise polynomial equality. I performed preliminary experiments with Z3, which was able to prove that the all the linearized accesses to the same array use the same linearization function for all the accesses involved in the benchmarks described in chapter 6.

This verification is not merely syntaxic, because the compiler can perform additional simplifications in the linearized access. For instance, consider the access `a[i + 2, M - i]` to array `a` of dimension $N \times M$. Applying the linearization function naively results in the expression $\texttt{a}[(\texttt{i}+2)\times M + M - \texttt{i}]$, while the actual buffer access found in the code might be $\texttt{a}[\texttt{i}\times(M-1)+3M-4]$ or $\texttt{a}[(\texttt{i}+3)\times M - \texttt{i} - 4]$.

To cover verification in full, we need to check that the linearized expression does not introduce overflowing computation: even if we have proved that all of the multidimensional indices fit the integer type used to represent array indices, the computation of the linearized index may involve intermediate expressions that overflow that type. However, if only wrapping (or unsigned) arithmetic is used to compute the linearized index, it is enough to check that the buffer size fits the appropriate integer type: if we have separately proven that the multidimensional indices are within bounds, the linearized index (as an unbounded integer) is necessarily less than the buffer size, and hence also fits that integer type.

9.8 Array Aliasing and Overlapping Arrays

The work presented in this thesis makes the implicit assumption that no two allocated array cells are identical, i.e. each cell is uniquely identified by its array name and indices. There are two ways in which this assumption can fail. The most obvious is that multiple input and/or output arrays can overlap, such as when performing an in-place update, but there can also be "overlap" within a single array. As explained in section 9.7, multidimensional arrays are ultimately implemented by buffers in linear memory. This is typically done by computing the dot product of the index vector with a (parametrically) constant *stride vector*, so that the linear index in the buffer is $\sum_{0 \leqslant j < N} i_j \times s_j$, where i_j is the index in the j-th dimension and s_j is the stride for the j-th dimension. Usually, the strides are chosen such that this linearization step is injective, but some frameworks such as Halide do not treat this as a hard requirement and allow a non-injective linearization step. The main use case is to set some strides to zero to emulate a broadcast.

Typically, dimensions are ordered in some way, and the stride is the product of the sizes of the previous dimensions.

To make things clear, we will say that an array is an *input array* if the initial value of (at least one of) its cells is read by the implementation, and an *output array* if the implementation writes to at least one of its cells. Some arrays are both input and output arrays, for instance when performing an in-place update. We can determine output arrays by simple inspection of the source code: they are the non-local arrays that are written to. The arrays that are read from is a sound over-approximation of the input arrays, but there may be non-local arrays where all reads are from a previous write during the execution of the program . We can determine input arrays more precisely by computing the set of cells that have been written to at each program point (in the same way as in the symbolic evaluation) and computing the cells that are read from without having been written to previously. When the implementation is affine, we can be more precise and compute the *set* of input and output locations.

It is common for the user to provide "workspace" or "scratchpad" arrays that are re-used across consecutive calls, avoiding the cost of repeated allocation and deallocation.

The possibility of overlapping arrays introduce two challenges. The first challenge is one of soundness: we must ensure that after a cell has been written to, we never rely on the old value of any cell it *could* overlap with. The second one is about completeness: the compiler may make use of precise overlapping information to perform certain optimizations (e.g. there can be a specialized path in the generated code when two input arrays are identical).

Soundness Let us first focus on soundness. It should be clear that as far as soundness is concerned, overlap is only an issue when one of the arrays involved is an output array. Moreover, overlap between two output arrays or within a single output array can be a soundness issue, but it is also fairly useless as it would lead to computing the same output value twice and can be safely prevented by a runtime assertion or contextual analysis. Hence, for soundness, we can only consider overlap between an input and an output array, where both may be the same array. There are two cases to consider: the overlap can be arbitrary or constrained (e.g. an output buffer is allowed to start, but not end, within an input buffer, as in a "copy" implementation forcing a specific iteration order).

When we want to allow arbitrary overlaps between the two arrays, all the writes to the output array must occur after the reads to the input array. This can be integrated into our symbolic representation by considering that all writes to the output array also virtually write an undetermined constant $\perp$ to all the cells in the input array (here, $\perp$ represents either the old value in that cell, or the value that was just written to the output array).

When the overlap is constrained, writing to the output array should only write $\perp$ to the cells of the input array it can overlap with. For this analysis to be precise, the desired overlap can be represented as a Presburger relation provided as an annotation; however it may be possible to infer an over-approximation of the legal overlaps in some cases. Presburger relations can represent the common instance where a buffer is prevented from either starting or ending within the range of another buffer. The constrained case includes the case where the two arrays can be identical, in which case each index in one array is related to the index in the other array. These overlap relations behave like "may-write" relations in polyhedral dependence analysis. If instead of merely allowing certain overlap patterns one wish to *force* a specific overlap, a relation similar to the "must-write" relation of dependence analysis should be used: writing to one cell must be treated as also being a write to all related cells.

Completeness Allowing overlap in arrays can be done soundly, as mentioned above. However, when the compiler is aware of some necessary overlap between input arrays, it may exploit this overlap. For instance, it is not uncommon in Halide to specialize for the case where the stride of an input array is zero, and

to exploit the resulting dimensionality collapse to avoid unnecessary repeated loads. To verify an implementation that exploits this ability, we first need to detect the dynamic stride check, and translate it to the appropriate overlapping relationship between array cells. We also need to assert the corresponding equalities between tensor indices when using Z3 to check verification conditions occurring within the specialization. For instance, if `a[i, j]` maps to $A(i,j)$ and we are within a specialization where the stride of the first dimension is 0, we need to assert that $A(i_1,j) = A(i_2,j)$ whenever i_1, i_2 and j are within the bounds of `a`.

One additional remark is that when the compiler exploits such information, extra care will have to be taken within the compiler so that the presence of prophetic annotations do not prevent optimizations. Taking again the previous example, two accesses `a[0 * s_1 + 7 * s_2]` and `a[1 * s_1 + 7 * s_2]` both simplify to `a[7 * s_2]` within a specialization `s_1 == 0`. But if there are annotations tracking back the indices to the original multidimensional indices, after simplification, we get `a[7 * s_2]` $\{A(0,7)\}$ and `a[7 * s_2]` $\{A(1,7)\}$. Hence, either the prophetic annotations must be simplified as appropriate by the compiler, or ignored by compiler phases such as common sub-expression elimination.

9.9 Garbage Writes

In some cases, the assumption that the tensor compiler knows what tensor definition backs up a given assignment breaks down, because some results are thrown out and ignored, such as with Halide's "round up" tail strategy. Typically, this gives the compiler the ability to fully unroll a tiled loop without having to emit a prologue or epilogue, performing computation using uninitialized array cells for the "extra" iterations. The resulting garbage cells are ultimately not used in the output, but because we need to perform eager verification of the prophetic annotations, it would still cause verification to fail (if only due to the reads from uninitialized memory). While it is not clear whether it would be feasible to track this information in the compiler itself, the prophetic evaluation in the validator can be adapted directly: when an assignment reads from uninitialized memory, instead of raising an error, we can treat this assignment as writing the undefined value $\perp$, and do the

same for any assignment that reads from a $\perp$ value, iteratively "fixing" the prophetic annotations. Because this only makes sense for the compiler to do for independent assignments (otherwise the later garbage assignments would overwrite the earlier semantically meaningful ones), we can expect the iterative process to terminate fairly quickly, which I confirmed in preliminary experiments with Halide's `camera_pipe` benchmark.

An alternative approach that I did not experiment with is to compute from the specification the set of intermediate tensor indices that are semantically meaningful for the output (either directly or using the compiler's default schedule), and treat prophetic annotations involving tensor indices outside that set as writing $\perp$ instead.

Finally, I will mention that it is possible to request the compiler precise annotations of these garbage writes, such as the "compute bounds" used by Reinking, Bernstein, and Ragan-Kelley [89] in their formalization of the Halide code generation algorithm. These annotations seem to be a byproduct of scheduling in the existing Halide implementation and do not appear in the generated code, but going by that paper could be added to the code generation algorithm naturally. However, it is not clear how well they would fit other compilers — in particular compilers based on the polyhedral model that may use non-rectangular regions.

9.10 Formal Verification

Formal verification can be used to increase confidence in both the proofs of the theorems presented in this manuscript (notably Theorems 4.3.3, 4.3.7 and 5.5.4), and in the implementation of the verifier itself. Since the goal is to have an automatic verifier relying on Z3 to check the verification conditions and on `isl` for an efficient representation of Presburger sets and relations, the mechanization would have to depend on an axiomatization of the `isl` primitive to prove that the verification conditions that are generated ensure the correctness of the implementation.

In fact, I have used the Coq proof assistant [105] to perform preliminary exploration of this. I have formalized and proven the existence and determinism

theorems for the small-step semantics with respect to the big-step semantics (Theorems 4.3.3 and 4.3.7), in a restricted language featuring sequential and parallel composition for no loops for simplicity. Separately, I have formalized and verified the soundness theorem Theorem 5.5.4 relating symbolic and concrete evaluations using an early version of the prophetic evaluation rules, expressed using code rather than inference rules, not presented in this thesis and expressed using an algebraic representation of symbolic heaps.

The proof steps involving reduction rules SeqLoop and ParLoop require many technicalities due to the use of lists.

While these are promising first steps, further work is needed to reach a formal verification of the full system, following the proofs presented in this manuscript. The resulting code could then be extracted to OCaml and used directly with the OCaml bindings to Z3 and `isl`.

Finally, it could be interesting to plug the Sched language to a verified compiler such as CompCert. Tensor compilers typically delegate the low-level compilation to a readily available compiler such as LLVM, that could theoretically re-introduce bugs. By directly plugging Sched to a verified compiler such as CompCert instead, we can get a formal proof in Coq that if the verification conditions computed with `isl` are correct (as checked by Z3), then the assembly generated by CompCert faithfully implement the original specification.

Bibliography

[1] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (visited on 05/31/2022).

[2] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, July 1988, pp. 165–175. DOI: 10.1109/LICS.1988.5115. URL: https://doi.org/10.1109/LICS.1988.5115.

[3] Andrew Adams et al. "Learning to optimize halide with tree search and random programs". In: *ACM Trans. Graph.* 38.4 (2019), 121:1–121:12. DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

[4] Corinne Ancourt and François Irigoin. "Scanning Polyhedra with DO Loops". In: *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Williamsburg, Virginia, USA, April 21-24, 1991*. Ed. by David S. Wise. ACM, Apr. 1, 1991, pp. 39–50. DOI: 10.1145/109625.109631. URL: https://doi.org/10.1145/109625.109631.

[5] Luke Anderson et al. "Learning to Schedule Halide Pipelines for the GPU". In: *CoRR* abs/2012.07145 (Dec. 13, 2020). arXiv: 2012.07145. URL: https://arxiv.org/abs/2012.07145.

[6] Riyadh Baghdadi et al. "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming". In: *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. IEEE Computer Society, 2015, pp. 138–149. DOI: 10.1109/PACT.2015.17. URL: https://doi.org/10.1109/PACT.2015.17.

[7] Riyadh Baghdadi et al. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *CoRR* abs/1804.10694 (Dec. 20, 2018). arXiv: 1804.10694. URL: http://arxiv.org/abs/1804.10694.

[8] Lénaïc Bagnères et al. "Opening polyhedral compiler's black box". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by Björn Franke, Youfeng Wu, and Fabrice Rastello. ACM, 2016, pp. 128–138. DOI: 10.1145/2854038.2854048. URL: https://doi.org/10.1145/2854038.2854048.

[9] Kunal Banerjee and Chandan Karfa. "A Quick Introduction to Functional Verification of Array-Intensive Programs". In: *CoRR* abs/1905.09137 (May 22, 2019). arXiv: 1905.09137. URL: http://arxiv.org/abs/1905.09137.

[10] Kunal Banerjee, Chittaranjan A. Mandal, and Dipankar Sarkar. "Translation validation of loop and arithmetic transformations in the presence of recurrences". In: *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, Santa Barbara, CA, USA, June 13 - 14, 2016*. Ed. by Tei-Wei Kuo and David B. Whalley. ACM, Aug. 2016, pp. 31–40. DOI: 10.1145/2907950.2907954. URL: https://doi.org/10.1145/2907950.2907954.

[11] Wenlei Bao et al. "PolyCheck: dynamic verification of iteration space transformations on affine programs". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 539–554. DOI: 10.1145/2837614.2837656. URL: https://doi.org/10.1145/2837614.2837656.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010.

[13] Denis Barthou, Paul Feautrier, and Xavier Redon. "On the Equivalence of Two Systems of Affine Recurrence Equations". In: *Euro-Par 2002 Parallel Processing*. Ed. by Burkhard Monien and Rainer Feldmann. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2400. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 309–313. ISBN: 978-3-540-44049-9 978-3-540-45706-0. DOI: 10.1007/3-540-45706-2_40.

URL: http://link.springer.com/10.1007/3-540-45706-2_40 (visited on 04/21/2021).

[14] Alexander I. Barvinok. "A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension Is Fixed". In: *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*. IEEE Computer Society, Nov. 1993, pp. 566–572. DOI: 10.1109/SFCS.1993.366830. URL: https://doi.org/10.1109/SFCS.1993.366830.

[15] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*. IEEE Computer Society, Sept. 29, 2004, pp. 7–16. DOI: 10.1109/PACT.2004.10018. URL: http://doi.ieeecomputersociety.org/10.1109/PACT.2004.10018.

[16] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Rajiv Gupta. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 283–303. DOI: 10.1007/978-3-642-11970-5_16. URL: https://doi.org/10.1007/978-3-642-11970-5_16.

[17] Gilbert Bernstein et al. "Differentiating a Tensor Language". In: *CoRR* abs/2008.11256 (Aug. 25, 2020). arXiv: 2008.11256. URL: https://arxiv.org/abs/2008.11256.

[18] R. S. Bird. "An Introduction to the Theory of Lists". In: *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*. Berlin, Heidelberg: Springer-Verlag, June 1, 1987, pp. 5–42. ISBN: 978-0-387-18003-8.

[19] François Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, Aug. 2011, pp. 53–64.

[20] Hans-Juergen Boehm. "How to Miscompile Programs with "Benign" Data Races". In: *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011*. Ed. by Michael McCool and Mendel Rosenblum. USENIX Association, May 26, 2011. URL:

https://www.usenix.org/conference/hotpar-11/how-miscompile-programs-benign-data-races.

[21] Uday Bondhugula. "Compiling affine loop nests for distributed-memory parallel architectures". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. Ed. by William Gropp and Satoshi Matsuoka. ACM, Nov. 17, 2013, 33:1–33:12. DOI: 10.1145/2503210.2503289. URL: https://doi.org/10.1145/2503210.2503289.

[22] Pierre Boulet and Paul Feautrier. "Scanning Polyhedra without Do-loops". In: *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998*. IEEE Computer Society, Oct. 1998, pp. 4–11. DOI: 10.1109/PACT.1998.727127. URL: https://doi.org/10.1109/PACT.1998.727127.

[23] Sylvain Boulmé et al. "The Verified Polyhedron Library: an Overview". In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*. IEEE, Sept. 2018, pp. 9–17. DOI: 10.1109/SYNASC.2018.00014. URL: https://doi.org/10.1109/SYNASC.2018.00014.

[24] Chun Chen. "Polyhedra scanning revisited". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, June 11, 2012, pp. 499–508. DOI: 10.1145/2254064.2254123. URL: https://doi.org/10.1145/2254064.2254123.

[25] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, Feb. 12, 2018, pp. 578–594. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[26] Berkeley R. Churchill et al. "Semantic program alignment for equivalence checking". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, June 8, 2019, pp. 1027–1040. DOI: 10.1145/3314221.3314596. URL: https://doi.org/10.1145/3314221.3314596.

[27] Basile Clément and Albert Cohen. "End-to-end translation validation for the halide language". In: *Proc. ACM Program. Lang.* 6.OOPSLA (Dec. 2022), pp. 1–30. DOI: 10.1145/3527328. URL: https://doi.org/10.1145/3527328.

[28] Jean-François Collard and Martin Griebl. "Array Dataflow Analysis for Explicitly Parallel Programs". In: *Euro-Par'96 Parallel Processing*. Ed. by Luc Bougé et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 406–413. ISBN: 978-3-540-70633-5. DOI: 10.1007/3-540-61626-8_54.

[29] Nathanaël Courant and Xavier Leroy. "Verified code generation for the polyhedral model". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 4, 2021), pp. 1–24. DOI: 10.1145/3434321. URL: https://doi.org/10.1145/3434321.

[30] Patrick Cousot et al. "The ASTREÉ Analyzer". In: *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Shmuel Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3. URL: https://doi.org/10.1007/978-3-540-31987-0%5C_3.

[31] *CTAN: Package Knowledge*. URL: https://ctan.org/pkg/knowledge?lang=en (visited on 04/25/2022).

[32] Bruno Cuervo Parrino et al. "Dealing with Arithmetic Overflows in the Polyhedral Model". In: *IMPACT 2012 - 2nd International Workshop on Polyhedral Compilation Techniques*. Ed. by Uday Bondhugula and Vincent Loechner. Louis-Noel Pouchet. Paris, France, Jan. 2012. URL: https://hal.inria.fr/hal-00655485.

[33] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975. URL: https://doi.org/10.1145/360933.360975.

[34] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. "Optimistic loop optimization". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, 2017, pp. 292–304. URL: http://dl.acm.org/citation.cfm?id=3049864.

[35] Johannes Doerfert et al. "Polly's Polyhedral Scheduling in the Presence of Reductions". In: *CoRR* abs/1505.07716 (May 28, 2015). arXiv: 1505.07716. URL: http://arxiv.org/abs/1505.07716.

[36] P. Feautrier. "Parametric Integer Programming". In: *RAIRO Recherche Opérationnelle* 22.3 (1988), pp. 243–268.

[37] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *Int. J. Parallel Program.* 20.1 (Feb. 1, 1991), pp. 23–53. DOI: 10.1007/BF01407931. URL: https://doi.org/10.1007/BF01407931.

[38] Paul Feautrier. "Array Expansion". In: *Proceedings of the International Conference on Supercomputing* (Aug. 5, 1996). DOI: 10.1145/55364.55406.

[39] Paul Feautrier. "The Power of Polynomials". In: *Proceedings of 5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)* (Jan. 19, 2015).

[40] Paul Feautrier and Christian Lengauer. "Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by David A. Padua. Springer, Sept. 1, 2011, pp. 1581–1592. DOI: 10.1007/978-0-387-09766-4_502. URL: https://doi.org/10.1007/978-0-387-09766-4_502.

[41] Marc Le Fur. "Scanning parameterized polyhedron using Fourier-Motzkin elimination". In: *Concurr. Pract. Exp.* 8.6 (1996), pp. 445–460. DOI: 10.1002/(SICI)1096-9128(199607)8:6%3C445::AID-CPE253%3E3.0.CO;2-G. URL: https://doi.org/10.1002/(SICI)1096-9128(199607)8:6%3C445::AID-CPE253%3E3.0.CO;2-G.

[42] Sabine Glesner. "Using Program Checking to Ensure the Correctness of Compiler Implementations". In: *J. Univers. Comput. Sci.* 9.3 (2003), pp. 191–222. DOI: 10.3217/jucs-009-03-0191. URL: https://doi.org/10.3217/jucs-009-03-0191.

[43] Benjamin Goldberg, Lenore D. Zuck, and Clark W. Barrett. "Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers". In: *Electron. Notes Theor. Comput. Sci.* 132.1 (May 30, 2005), pp. 53–71. DOI: 10.1016/j.entcs.2005.01.030. URL: https://doi.org/10.1016/j.entcs.2005.01.030.

[44] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. "Polyhedral AST Generation Is More Than Scanning Polyhedra". In: *ACM Trans. Program. Lang. Syst.* 37.4 (Aug. 13, 2015), 12:1–12:50. DOI: 10.1145/2743016. URL: https://doi.org/10.1145/2743016.

[45] Gautam Gupta, Sanjay V. Rajopadhye, and Patrice Quinton. "Scheduling reductions on realistic machines". In: *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, Winnipeg, Manitoba, Canada, August 11-13, 2002*. Ed. by Arnold L. Rosenberg and Bruce M. Maggs. ACM, Aug. 10, 2002, pp. 117–126. DOI: 10.1145/564870.564888. URL: https://doi.org/10.1145/564870.564888.

[46] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Counterexample-guided correlation algorithm for translation validation". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 13, 2020), 221:1–221:29. DOI: 10.1145/3428289. URL: https://doi.org/10.1145/3428289.

[47] Bastian Hagedorn et al. "High performance stencil code generation with lift". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. Ed. by Jens Knoop et al. ACM, 2018, pp. 100–112. DOI: 10.1145/3168824. URL: https://doi.org/10.1145/3168824.

[48] Bastian Hagedorn et al. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2, 2020), 92:1–92:29. DOI: 10.1145/3408974. URL: https://doi.org/10.1145/3408974.

[49] Mark Harris. "Optimizing Parallel Reduction in CUDA". 2008. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[50] Troels Henriksen et al. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 556–571. DOI: 10.1145/3062341.3062354. URL: https://doi.org/10.1145/3062341.3062354.

[51] Pieter Hijma, Rob V van Nieuwpoort, and Henri E Bal. "Programming Many-Cores on Different Levels of Abstraction". In: *HotPar '13 Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism* (July 2013), p. 7. URL: https://staff.fnwi.uva.nl/h.p.hijma/papers/Hijma2013Programming.pdf.

[52] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. "Monoparametric tiling of polyhedral programs". In: *International Journal of Parallel Programming* 49.3 (2021), pp. 376–409.

[53] Guillaume Iooss, Christophe Alias, and Sanjay V. Rajopadhye. "On Program Equivalence with Reductions". In: *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*. Ed. by Markus Müller-Olm and Helmut Seidl. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 168–183. DOI: 10.1007/978-3-319-10936-7_11. URL: https://doi.org/10.1007/978-3-319-10936-7_11.

[54] Matthieu Journault and Antoine Miné. "Inferring functional properties of matrix manipulating programs by abstract interpretation". In: *Formal Methods Syst. Des.* 53.2 (Feb. 2018), pp. 221–258. DOI: 10.1007/s10703-017-0311-x. URL: https://doi.org/10.1007/s10703-017-0311-x.

[55] Pierre Jouvelot and Babak Dehbonei. "A unified semantic approach for the vectorization and parallelization of generalized reductions". In: *Proceedings of the 3rd international conference on Supercomputing, ICS 1989, Heraklion, Crete, Greece, June 5-9, 1989*. Ed. by George Paul et al. ACM, 1989, pp. 186–194. DOI: 10.1145/318789.318810. URL: https://doi.org/10.1145/318789.318810.

[56] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. "A PVS Based Framework for Validating Compiler Optimizations". In: *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*. IEEE Computer Society, Sept. 2006, pp. 108–117. DOI: 10.1109/SEFM.2006.4. URL: https://doi.org/10.1109/SEFM.2006.4.

[57] Jeehoon Kang et al. "Crellvm: verified credible compilation for LLVM". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, June 11, 2018, pp. 631–645. DOI: 10.1145/3192366.3192377. URL: https://doi.org/10.1145/3192366.3192377.

[58] Chandan Karfa et al. "Experimentation with SMT Solvers and Theorem Provers for Verification of Loop and Arithmetic Transformations". In: ACM International Conference Proceeding Series. Oct. 17, 2013. DOI: 10.1145/2528228.2528231.

[59] Chandan Karfa et al. “Verification of Loop and Arithmetic Transformations of Array-Intensive Behaviors”. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 32.11 (Nov. 1, 2013), pp. 1787–1800. DOI: 10.1109/TCAD.2013.2272536. URL: https://doi.org/10.1109/TCAD.2013.2272536.

[60] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. “The Organization of Computations for Uniform Recurrence Equations”. In: *J. ACM* 14.3 (July 1967), pp. 563–590. DOI: 10.1145/321406.321418. URL: https://doi.org/10.1145/321406.321418.

[61] W. Kelly, W. Pugh, and E. Rosser. “Code Generation for Multiple Mappings”. In: *The Fifth Symposium on the Frontiers of Massively Parallel Computation Proceedings Frontiers ’95*. The Fifth Symposium on the Frontiers of Massively Parallel Computation Proceedings Frontiers ’95. Feb. 1995, pp. 332–341. DOI: 10.1109/FMPC.1995.380437.

[62] Fredrik Kjolstad et al. “The tensor algebra compiler”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 12, 2017), 77:1–77:29. DOI: 10.1145/3133901. URL: https://doi.org/10.1145/3133901.

[63] Kensuke Kojima, Akifumi Imanishi, and Atsushi Igarashi. “Automated Verification of Functional Correctness of Race-Free GPU Programs”. In: *J. Autom. Reason.* 60.3 (Mar. 1, 2018), pp. 279–298. DOI: 10.1007/s10817-017-9428-2. URL: https://doi.org/10.1007/s10817-017-9428-2.

[64] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. “Proving optimizations correct using parameterized program equivalence”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by Michael Hind and Amer Diwan. ACM, June 15, 2009, pp. 327–337. DOI: 10.1145/1542476.1542513. URL: https://doi.org/10.1145/1542476.1542513.

[65] Leslie Lamport. “The Parallel Execution of DO Loops”. In: *Commun. ACM* 17.2 (Feb. 1, 1974), pp. 83–93. DOI: 10.1145/360827.360844. URL: https://doi.org/10.1145/360827.360844.

[66] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, Apr. 25, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://doi.org/10.1007/978-3-642-17511-4_20.

[67] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[68] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. "An affine partitioning algorithm to maximize parallelism and minimize communication". In: *Proceedings of the 13th international conference on Supercomputing, ICS 1999, Rhodes, Greece, June 20-25, 1999*. Ed. by Theodore S. Papatheodorou et al. ACM, May 1, 1999, pp. 228–237. DOI: 10.1145/305138.305197. URL: https://doi.org/10.1145/305138.305197.

[69] Amanda Liu et al. "Verified tensor-program optimization via high-level scheduling rewrites". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 16, 2022), pp. 1–28. DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[70] Nuno P. Lopes et al. "Alive2: bounded translation validation for LLVM". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, June 19, 2021, pp. 65–79. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[71] Lee-Chung Lu. "A Unified Framework for Systematic Loop Transformations". In: *ACM SIGPLAN Notices* 26.7 (Apr. 1, 1991), pp. 28–38. ISSN: 0362-1340. DOI: 10.1145/109626.109630. URL: https://doi.org/10.1145/109626.109630 (visited on 04/01/2022).

[72] Trevor L. McDonell et al. "Optimising purely functional GPU programs". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, Sept. 25, 2013, pp. 49–60. DOI: 10.1145/2500365.2500595. URL: https://doi.org/10.1145/2500365.2500595.

[73] David Menendez, Santosh Nagarakatte, and Aarti Gupta. "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM". In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 317–337. DOI: 10.1007/978-3-662-53413-7_16. URL: https://doi.org/10.1007/978-3-662-53413-7_16.

[74] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *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*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[75] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*. Ed. by Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas. ACM, 2015, pp. 429–443. DOI: 10.1145/2694344.2694364. URL: https://doi.org/10.1145/2694344.2694364.

[76] George C. Necula. "Translation validation for an optimizing compiler". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. Ed. by Monica S. Lam. ACM, 2000, pp. 83–94. DOI: 10.1145/349299.349314. URL: https://doi.org/10.1145/349299.349314.

[77] A. Pnueli, M. Siegel, and E. Singerman. "Translation Validation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1384. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166. ISBN: 978-3-540-64356-2 978-3-540-69753-4. DOI: 10.1007/BFb0054170. URL: http://link.springer.com/10.1007/BFb0054170 (visited on 03/15/2022).

[78] M. Presburger. "Über Die Vollständigkeit Eines Gewissen Systems Der Arithmetik Ganzer Zahlen, in Welchem Die Addition Als Einzige Operation Hervortritt". In: *Comptes Rendus Du Premier Congrès de Mathématiciens Des Pays Slaves*. Warsaw, Poland, 1929, pp. 92–101.

[79] William W. Pugh. "Uniform techniques for loop optimization". In: *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*. Ed. by Edward S. Davidson and Friedel Hossfeld. ACM, June 1, 1991, pp. 341–352. DOI: 10.1145/109025.109108. URL: https://doi.org/10.1145/109025.109108.

[80] William W. Pugh and David Wonnacott. "Static Analysis of Upper and Lower Bounds on Dependences and Parallelism". In: *ACM Trans. Program. Lang. Syst.* 16.4 (July 1994), pp. 1248–1278. DOI: 10.1145/183432.183525. URL: https://doi.org/10.1145/183432.183525.

[81] Fabien Quilleré, Sanjay V. Rajopadhye, and Doran Wilde. "Generation of Efficient Nested Loops from Polyhedra". In: *Int. J. Parallel Program.* 28.5 (Oct. 1, 2000), pp. 469–498. DOI: 10.1023/A:1007554627716. URL: https://doi.org/10.1023/A:1007554627716.

[82] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, June 23, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[83] Jonathan Ragan-Kelley et al. "Halide: decoupling algorithms from schedules for high-performance image processing". In: *Commun. ACM* 61.1 (Dec. 27, 2017), pp. 106–115. DOI: 10.1145/3150211. URL: https://doi.org/10.1145/3150211.

[84] Harenome Razanajato, Vincent Loechner, and Cédric Bastoul. "Splitting Polyhedra to Generate More Efficient Code". In: IMPACT 2017, 7th International Workshop on Polyhedral Compilation Techniques. Jan. 23, 2017. URL: https://hal.inria.fr/hal-01505764 (visited on 04/01/2022).

[85] Chandan Reddy, Michael Kruse, and Albert Cohen. "Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*. Ed. by Ayal Zaks et al. ACM, Sept. 11, 2016, pp. 87–97. DOI: 10.1145/2967938.2967950. URL: https://doi.org/10.1145/2967938.2967950.

[86] Xavier Redon and Paul Feautrier. "Detection of Recurrences in Sequential Programs with Loops". In: *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, Munich, Germany, June 14-17, 1993, Proceedings*. Ed. by Arndt Bode, Mike Reeve, and Gottfried Wolf. Vol. 694. Lecture Notes in Computer Science. Springer, 1993, pp. 132–145. DOI: 10.1007/3-540-56891-3_11. URL: https://doi.org/10.1007/3-540-56891-3_11.

[87] Xavier Redon and Paul Feautrier. "Scheduling reductions". In: *Proceedings of the 8th international conference on Supercomputing, ICS 1994, Manchester, UK, July 11-15, 1994*. Ed. by John R. Gurd and William Jalby. ACM, July 16, 1994, pp. 117–125. DOI: 10.1145/181181.181319. URL: https://doi.org/10.1145/181181.181319.

[88] Xavier Redon and Paul Feautrier. "Detection of Scans". In: *Parallel Algorithms and Applications* 15.3-4 (Dec. 1, 2000), pp. 229–263. ISSN: 1063-7192. DOI: 10.1080/01495730008947357. URL: https://doi.org/10.1080/01495730008947357 (visited on 05/02/2022).

[89] Alex Reinking, Gilbert Bernstein, and Jonathan Ragan-Kelley. "Formal Semantics for the Halide Language". EECS Department, University of California, Berkeley, May 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-40.html.

[90] Martin C. Rinard and Darko Marinov. "Credible Compilation with Pointers". In: *In Proceedings of the Workshop on Run-Time Result Verification*. 1999.

[91] Norman A. Rink and Jerónimo Castrillón. "TeIL: a type-safe imperative tensor intermediate language". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*. Ed. by Jeremy Gibbons. ACM, 2019, pp. 57–68. DOI: 10.1145/3315454.3329959. URL: https://doi.org/10.1145/3315454.3329959.

[92] Xavier Rival. "Abstract Interpretation-Based Certification of Assembly Code". In: *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*. Ed. by Lenore D. Zuck et al. Vol. 2575. Lecture Notes in Computer Science. Springer, Jan. 9, 2002, pp. 41–55. DOI: 10.1007/3-540-36384-X_7. URL: https://doi.org/10.1007/3-540-36384-X_7.

[93] Xavier Rival. "Symbolic transfer function-based approaches to certified compilation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, Jan. 1, 2004, pp. 1–13. DOI: 10.1145/964001.964002. URL: https://doi.org/10.1145/964001.964002.

[94] Valentin Robert and Xavier Leroy. "A Formally-Verified Alias Analysis". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 11–26. DOI: 10.1007/978-3-642-35308-6_5. URL: https://doi.org/10.1007/978-3-642-35308-6_5.

[95] Hanan Samet. "Automatically Proving the Correctness of Translations Involving Optimized Code". PhD thesis. Stanford, CA, USA: Computer Science Department, Stanford University, May 1975. 222 pp. URL: http://www.cs.umd.edu/~hjs/pubs/compilers/CS-TR-75-498.pdf (visited on 04/22/2022).

[96] Hans Samsom et al. "System level verification of video and image processing specifications". In: *Proceedings of the 8th International Symposium on System Synthesis (ISSS 1995), September 13-15, 1995, Cannes, France*. Ed. by Pierre G. Paulin and Farhad Mavaddat. ACM, 1995, pp. 144–149. DOI: 10.1145/224486.224533. URL: https://doi.org/10.1145/224486.224533.

[97] K. C. Shashidhar et al. "Verification of Source Code Transformations by Program Equivalence Checking". In: *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Rastislav Bodík. Vol. 3443. Lecture Notes in Computer Science. Springer, 2005, pp. 221–236. DOI: 10.1007/978-3-540-31985-6_15. URL: https://doi.org/10.1007/978-3-540-31985-6_15.

[98] K.C. Shashidhar et al. "Geometric Model Checking". In: *Electronic Notes in Theoretical Computer Science* 65.2 (Apr. 2002), pp. 67–82. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)80397-9. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066104803979 (visited on 03/14/2022).

[99] Qingchao Shen et al. "A comprehensive study of deep learning compiler bugs". In: *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. Ed. by Diomidis Spinellis et al. ACM, Aug. 20, 2021, pp. 968–980. DOI: 10.1145/3468264.3468591. URL: https://doi.org/10.1145/3468264.3468591.

[100] Michael Stepp, Ross Tate, and Sorin Lerner. "Equality-Based Translation Validator for LLVM". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 737–742. DOI: 10.1007/978-3-642-22110-1_59. URL: https://doi.org/10.1007/978-3-642-22110-1_59.

[101] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation". In: *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, Oct. 1, 2016, 15:1–15:10. DOI: 10.1145/2968455.2968521. URL: https://doi.org/10.1145/2968455.2968521.

[102] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, Feb. 2017, pp. 74–85. DOI: 10.1109/cgo.2017.7863730. URL: http://dl.acm.org/citation.cfm?id=3049841.

[103] Patricia Suriana, Andrew Adams, and Shoaib Kamil. "Parallel associative reductions in halide". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. Ed. by Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang. ACM, Feb. 4, 2017, pp. 281–291. URL: http://dl.acm.org/citation.cfm?id=3049863.

[104] Ross Tate et al. "Equality Saturation: A New Approach to Optimization". In: *Log. Methods Comput. Sci.* 7.1 (Mar. 2011). DOI: 10.2168/LMCS-7(1:10)2011. URL: https://doi.org/10.2168/LMCS-7(1:10)2011.

[105] The Coq Development Team. *The Coq Proof Assistant*. Zenodo. Version 8.13. Zenodo, Jan. 2021. DOI: 10.5281/zenodo.4501022. URL: https://doi.org/10.5281/zenodo.4501022.

[106] Nicolas Tollenaere et al. "Efficient Convolution Optimisation by Composing Micro-Kernels". Apr. 2021. URL: https://hal.archives-ouvertes.fr/hal-03149553 (visited on 05/31/2022).

[107] Jean-Baptiste Tristan. "Formal verification of translation validators". PhD thesis. Paris Diderot University, France, Nov. 6, 2009. URL: https://tel.archives-ouvertes.fr/tel-00437582.

[108] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating value-graph translation validation for LLVM". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 295–305. DOI: 10.1145/1993498.1993533. URL: https://doi.org/10.1145/1993498.1993533.

[109] Jean-Baptiste Tristan and Xavier Leroy. "A simple, verified validator for software pipelining". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, Jan. 2010, pp. 83–92. DOI: 10.1145/1706299.1706311. URL: https://doi.org/10.1145/1706299.1706311.

[110] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. "Polyhedral Code Generation in the Real World". In: *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*. Ed. by Alan Mycroft and Andreas Zeller. Vol. 3923. Lecture Notes in Computer Science. Springer, 2006, pp. 185–201. DOI: 10.1007/11688839_16. URL: https://doi.org/10.1007/11688839_16.

[111] Nicolas Vasilache et al. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically". In: *ACM Trans. Archit. Code Optim.* 16.4 (Jan. 2020), 38:1–38:26. DOI: 10.1145/3355606. URL: https://doi.org/10.1145/3355606.

[112] Sven Verdoolaege. "Isl: An Integer Set Library for the Polyhedral Model." In: *ICMS*. Ed. by Komei Fukuda et al. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 299–302. ISBN: 978-3-642-15581-9. URL: http://dblp.uni-trier.de/db/conf/icms/icms2010.html#Verdoolaege10.

[113] Sven Verdoolaege. "Integer Set Coalescing". In: (2015). DOI: 10.13140/2.1.1313.6968. URL: http://rgdoi.net/10.13140/2.1.1313.6968 (visited on 04/25/2022).

[114] Sven Verdoolaege. "Presburger Formulas and Polyhedral Compilation". In: *Presburger formulas and polyhedral compilation* (2016), p. 174. URL: https://lirias.kuleuven.be/retrieve/361209.

[115] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. "Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 599–613. DOI: 10.1007/978-3-642-02658-4_44. URL: https://doi.org/10.1007/978-3-642-02658-4_44.

[116] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. "Equivalence checking of static affine programs using widening to handle recurrences". In: *ACM Trans. Program. Lang. Syst.* 34.3 (Nov. 5, 2012), 11:1–11:35. DOI: 10.1145/2362389.2362390. URL: https://doi.org/10.1145/2362389.2362390.

[117] Sven Verdoolaege et al. "Experience with Widening Based Equivalence Checking in Realistic Multimedia Systems". In: *J. Electron. Test.* 26.2 (Nov. 1, 2009), pp. 279–292. DOI: 10.1007/s10836-009-5140-4. URL: https://doi.org/10.1007/s10836-009-5140-4.

[118] Sven Verdoolaege et al. "Schedule Trees". In: 4th International Workshop on Polyhedral Compilation Techniques. Jan. 20, 2014. URL: https://www.research.ed.ac.uk/en/publications/schedule-trees (visited on 04/01/2022).

[119] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. "The ALPHA language and its use for the design of systolic arrays". In: *J. VLSI Signal Process.* 3.3 (1991), pp. 173–182. DOI: 10.1007/BF00925828. URL: https://doi.org/10.1007/BF00925828.

[120] Anna Zaks and Amir Pnueli. "CoVaC: Compiler Validation by Program Analysis of the Cross-Product". In: *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*. Ed. by Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere. Vol. 5014. Lecture Notes in Computer Science. Springer, 2008, pp. 35–51. DOI: 10.1007/978-3-540-68237-0_5. URL: https://doi.org/10.1007/978-3-540-68237-0_5.

[121] Jie Zhao and Albert Cohen. "Flextended Tiles: A Flexible Extension of Overlapped Tiles for Polyhedral Compilation". In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 17, 2019), 47:1–47:25. DOI: 10.1145/3369382. URL: https://doi.org/10.1145/3369382.

[122] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. "Visual Program Manipulation in the Polyhedral Model". In: *ACM Trans. Archit. Code Optim.* 15.1 (2018), 16:1–16:25. DOI: 10.1145/3177961. URL: https://doi.org/10.1145/3177961.

[123] L Zuck, A Pnueli, and R Leviathan. *Validation of Optimizing Compilers*. Computer Science Department, NYU, p. 13. URL: https://cs.nyu.edu/faculty/pnueli/ZPL01.pdf.

[124] Lenore D. Zuck et al. "Translation and Run-Time Validation of Loop Transformations". In: *Formal Methods Syst. Des.* 27.3 (Nov. 1, 2005), pp. 335–360. DOI: 10.1007/s10703-005-3402-z. URL: https://doi.org/10.1007/s10703-005-3402-z.

RÉSUMÉ

Les compilateurs de tenseurs sont utilisés dans des domaines comme le traitement d'image et l'apprentissage profond pour générer du code bas niveau efficace à partir de spécification de haut niveau sur des tenseurs multi-dimensionnels. Le code généré peut présenter une structure drastiquement différente de celle de la spécification suite à l'application de transformations de boucles et de simplifications algébriques. La vérification formelle des compilateurs de tenseurs est donc une tâche ardue, qui ne peut être traitée par les techniques standard à base de bisimulations. Je propose une nouvelle méthode pour la vérification de compilateurs de tenseurs en présence de transformations algébriques et de boucles. Cette méthode s'inspire des techniques polyédriques de représentation de programmes, et s'appuie sur une association de raffinement depuis les affectations dans le code bas niveau vers les définitions de tenseurs dans la spécification fournie par le compilateur. Chaque exécution du compilateur est vérifiée par un outil de vérification indépendant implanté en OCaml, faisant donc de la méthode un validateur de traduction. Cet outil de vérification est testé sur Halide, un compilateur de tenseurs de niveau industriel.

MOTS CLÉS

Validation de Traduction, Modèle Polyédrique, Compilateurs de Tenseurs, Vérification Formelle

ABSTRACT

Tensor compilers are used in domains such as image processing and deep learning to generate efficient low-level code from high-level specifications on multidimensional tensors. After the application of both loop transformations and algebraic simplifications to the specification, the resulting low-level code can have a drastically different structure. This makes the formal verification of tensor compilers an arduous task, unsuitable for standard bisimulation techniques. I propose a new method for the verification of tensor compilers in the presence of loop and algebraic transformations. This method draws inspiration from polyhedral techniques for program representation, and relies on a refinement mapping from assignments in the low-level code to tensor definition in the specifications provided by the tensor compiler. Each run of the compiler is verified by an independent verification tool implemented in OCaml, making the method an instance of translation validation. This verification tool is tested on Halide, an industrial-grade tensor compiler.

KEYWORDS

Translation Validation, Polyhedral Model, Tensor Compilers, Formal Verification