UNIVERSITÉ PARIS-EST

Pôle de recherche et d'enseignement supérieur

École doctorale MSTIC

MÉMOIRE

Habilitation à Diriger des Recherches

spécialité Traitement du Signal et des Images

25 juin 2012

OUTIL LOGICIEL POUR LE TRAITEMENT DES IMAGES
BIBLIOTHÈQUE, PARADIGMES, TYPES ET ALGORITHMES

THIERRY GÉRAUD

Jury :

Présidente :	Laurence DUCHIEN, Pr., Université Lille 1
Rapporteurs :	Pierre COINTE, Pr., École des Mines de Nantes Jacques-Olivier LACHAUD, Pr., Université de Savoie Sylvain LOMBARDY, Pr., Université Paris-Est
Examinateurs :	Henri MAÎTRE, Pr., Télécom ParisTech Lionel MOISAN, Pr., Université Paris Descartes Marc VAN DROOGENBROECK, Pr., Université de Liège, Belgique, Laurent NAJMAN, Pr., Université Paris-Est (directeur d'HDR)

Résumé

Un scientifique manipulant des données de nature variée est très souvent bloqué par son outil logiciel. En effet, l'implémentation d'algorithmes dans les bibliothèques logicielles restreint généralement leur utilisation à un petit nombre de types de données. Réussir à s'affranchir de ces limitations est non seulement intéressant en tant que tel mais présente également d'autres avantages. Les possibilités de l'outil sont décuplées en termes d'extensibilité et de ré-utilisabilité ; mieux, l'outil devient un facilitateur pour explorer de nouvelles pistes de recherche. Nous proposons une solution pour obtenir des bibliothèques logicielles scientifiques ayant de telles propriétés. Cette solution est multi-paradigmes, mêlant généricité, orienté-objet et déclaratif. Elle aboutit à un ensemble cohérent de types de données, d'outils et d'algorithmes, tout en préservant les performances attendues en calcul scientifique. Une réalisation effective dédiée au domaine du traitement d'images, la bibliothèque Milena, sera présentée. Elle nous servira à illustrer l'efficience de la solution proposée.

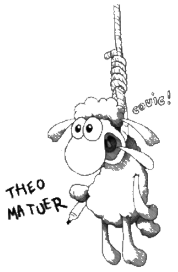


TABLE DES MATIÈRES

Première partie

LE DÉBUT

0 PRÉFACE

> The Analytical Engine weaves algebraic patterns, just as the Jacquard loom weaves flowers and leaves.
>
> *Notes on Menabrae's Memoir on the Analytic Engine, Ada Lovelace, 1843.*

Ce rapport a pour sujet la réalisation d'une bibliothèque logicielle dédiée au traitement d'images. Le choix de ce sujet a été relativement aisé puisqu'il représente l'activité de recherche que j'ai menée en "tâche de fond" depuis une dizaine d'année. Si au final ce sujet n'a pas réussi à remplir mon emploi du temps autant que je l'aurais souhaité, il s'est toutefois bien concrétisé sous la forme d'un outil effectif, MILENA, disponible sous forme de logiciel libre.

Contrairement à ce que l'on pourrait croire, puisqu'il s'agit d'un logiciel, l'aspect "développement" n'a représenté qu'une partie minime de mon travail sur cet outil. L'essentiel du temps a été consacré à de multiples problématiques de recherche, qui peuvent être classées en trois catégories : des sujets informatiques ; des sujets de traitement d'images ; un sujet à mi-chemin entre l'informatique et le traitement d'images. Les problématiques purement informatiques touchent aux constructions liées aux langages de programmation, au typage et au génie logiciel. Ces problématiques ne sont qu'évoquées dans le chapitre 3 de ce rapport mais les publications associées sont portées en annexe B. Les sujets de recherche en traitement d'images ont généralement été menés dans le cadre de projets de recherche en collaboration avec des partenaires industriels ou institutionnels. Ces sujets n'apparaissent que dans mon *curriculum vitæ*.

Pour ce rapport, j'ai choisi un sujet de recherche qui se situe à mi-chemin entre le domaine du traitement d'images et celui de l'informatique : *la traduction du domaine du traitement d'images en informatique*. Ce sujet, relativement peu traité dans la littérature [32, 3, 33, 8, 22, 21], a représenté à lui seul, en temps cumulé, de nombreux mois de réflexion. Curieusement ce sujet a une saveur "traitement d'images" beaucoup plus marquée que sa saveur informatique. La difficulté de ce sujet réside essentiellement dans la nécessité d'obtenir une réponse précise à la question ouverte "qu'est-ce qu'une image ?" Souvent, le traiteur d'images se place dans un contexte théorique très particulier pour lequel la réponse à cette question est un postulat servant de base à son travail. La

difficulté de notre travail de recherche consiste à ne rien présupposer : la réponse à cette question doit rester générale afin de convenir à n'importe quel traiteur d'images, quelque soit son approche.

Les résultats que nous présentons dans ce rapport ont nécessité un travail de longue haleine. Ce travail s'est traduit par une introspection du domaine du traitement d'images et par la confrontation de résultats intermédiaires à l'épreuve de l'implémentation d'opérateurs de traitement d'images. Cette partie implémentatoire s'est avérée nécessaire afin de valider nos résultats ou, quelquefois, de les invalider et de procéder à des corrections. *In fine* l'existence de la bibliothèque MILENA et son utilisabilité (Cf. l'annexe D) sont une sorte de "validation" empirique de notre travail.

Ce travail de recherche, de réflexion et de tests n'a pas été mené isolément. Il m'est difficile de ne pas remercier les personnes qui ont également travailler sur ce projet puisqu'il est ouvertement collaboratif [1]. Il s'agit des étudiants qui année après année ont dû me subir et se casser les dents sur des pistes ardues et bancales, ainsi que les contributeurs du LRDE, passés ou présents.

1. Votre aide est bienvenue.

1 INTRODUCTION

A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.

Alan Turing, 1948.

1.1 PRÉAMBULE

L'arrivée des ordinateurs dans les années 1960 a révolutionné le domaine du traitement d'images (TDI), le traitement optique a été remplacé massivement par le traitement numérique. Les liens qui existent entre informatique et traitement d'images ne sont pourtant pas si clairs dans les esprits. D'un côté, certains vont considérer que le traitement d'images est une discipline de l'informatique ; c'est le cas aussi bien pour Wikipedia [1] que pour certaines équipes de chercheurs en TDI appartenant à un laboratoire d'informatique. D'autres regroupent ces deux domaines indiquant ainsi qu'ils sont fortement liés ; par example, la section 61 du CNU s'intitule "Génie informatique, automatique et traitement du signal". Enfin, une perspective diamétralement opposée au premier cas de figure a également sa place ; ainsi, pour le GRETSI, le thème informatique "architectures matérielles et logicielles" n'est qu'une composante du domaine signal/image. Finalement, que doit-on penser ? Quelles sont les relations entre informatique et traitement d'images ? Et de quelle nature sont ces relations ?

Un point de vue assez consensuel est que *l'informatique est un outil pour les traiteurs d'images*. Cette assertion n'est bien sûr pas vraie pour tous les traiteurs d'images au sens où une minorité de chercheurs mène des travaux purement théoriques et laisse à d'autres la tâche d'appliquer, de mettre en pratique, leurs résultats sur des images à l'aide d'ordinateurs. Si la majorité des traiteurs d'images utilisent des ordinateurs pour réaliser leurs traitements et visualiser leurs résultats, soit directement, soit indirectement en délèguant ce travail, on ne peut cependant pas affirmer qu'ils "font de l'informatique". Comme l'a dit Edsger Dijkstra : *Computer science is no more*

1. http://fr.wikipedia.org/wiki/Traitement_d'images

about computers than astronomy is about telescopes. La recherche en informatique a sa propre spécificité, la plupart du temps disjointe de la recherche en TDI. En discutant avec des chercheurs en TDI, on entend effectivement très souvent cette affirmation : “je fais du traitement d’images, pas de l’informatique”. En effet, les sujets de recherche à l’intersection de ces deux domaines sont véritablement peu nombreux. Le plus emblématique est l’*algorithmique*, sous-domaine de l’informatique, incontournable pour un chercheur en TDI qui propose de nouveaux algorithmes de traitements. L’utilisation de grammaires en vision par ordinateur est également un emprunt du TDI à l’informatique. Un autre sous-domaine, déjà plus rare, est la définition d’architectures matérielles dédiées au TDI. Enfin, lorsqu’il s’agit de montrer *comment* faire supporter des traitements d’images à des puces de calcul généralistes, donc non dédiées à ce domaine, la notion d’*informatique outil*, ressurgit, ici volontairement mise en évidence par l’utilisation du mot “comment”.

Si l’on ne peut pas nier que l’informatique est un outil pour le traitement d’images, l’exacerbation de la différence entre l’*outil* (le moyen, le comment) et le *but* (le traitement d’images en soi) conduit un relecteur anonyme d’une de nos publications à dire : “les propositions restent à un niveau purement logiciel (informatique) donc la contribution de ce travail n’apparaît pas évidente.” Pourtant l’outil est essentiel à celui qui s’en sert et *contribuer* à l’amélioration de cet outil permet indirectement de contribuer aux réalisations que permet cet outil. Mieux, on peut même être en mesure d’attendre aujourd’hui qu’un outil “puissant”, non seulement facilite la recherche en TDI, mais encore favorise la découverte de résultats scientifique dans ce domaine. En fait, il devrait dépasser le stade de “simple outil” pour devenir un véritable “outil efficient”. C’est aussi lorsqu’on dispose d’un outil plus puissant que l’on peut alors imaginer, beaucoup plus facilement, de nouveaux horizons de recherche. L’outil devrait se caractériser par ses capacités à inciter à franchir certaines limites, à s’affranchir de certains murs.

Que l’informatique et le traitement d’images soient perçus comme opposés, duaux ou complémentaires, il subsiste une attitude paradoxale de la part de beaucoup de traiteurs d’images. Dire “je fais du traitement d’images, pas d’informatique” n’est qu’à moitié vrai car, aujourd’hui, beaucoup de traiteurs d’images *programment*. Effectivement leur sujet de recherche, leur préoccupation principale, réside dans leur domaine et ils ne font généralement pas de recherche en informatique pure. Malgré cela, il est faux de considérer que l’informatique n’est qu’un outil utilitaire et qu’ils ne font pas d’informatique ; en fait, dès l’instant où l’on programme des méthodes de traitement d’images, on fait de l’informatique ! La différence entre “utiliser un programme” et “programmer” est beaucoup plus importante qu’on peut l’imaginer. Dans le premier cas, il s’agit d’informatique-outil et peu de compétences en informatique sont requises ; l’utilisateur se contente d’être un “simple consommateur” d’informatique. Dans le second cas en revanche, puisqu’il y a écriture de programmes, on passe du statut de consommateur à celui de “producteur”. Et produire des programmes informatique, c’est bien sûr être créateur d’informatique donc... “faire de l’informatique”, même si l’on s’en défend ! Qui n’a pas entendu de la bouche d’un traiteur d’images “j’ai un bug”, “j’ai lancé une compilation” ou, tout simplement, “mon programme”.

L’attitude paradoxale du traiteur d’images programmant et qui consiste à nier “faire de l’informatique” peut malheureusement aboutir à une certaine inconscience qui se traduit par de multiples artéfacts. Le plus manifeste est de ne pas s’interroger sur son outil de travail informatique, souvent une bibliothèque logicielle ou un environnement de développement. Des questions à se poser sont

par example les suivantes. Est-ce que mon outil est de qualité ? Quels sont ses avantages et ses inconvénients ? Est-il vraiment adapté au travail à réaliser ? Et ai-je donc intérêt à en changer ? Obtenir des réponses n'est pas facile car il faut déjà accepter de s'immerger dans le monde particulier de l'informatique auquel l'outil appartient. Lorsque l'on connaît les deux communautés que sont les informaticiens et les traiteurs d'images, une différence frappante est que les premiers changent volontiers d'outils, tandis que les seconds sont plutôt adeptes du conservatisme (*better the devil you know*).

Mais, au final, l'essentiel reste que tout le monde s'accorde sur le fait que, meilleur est outil, meilleures sont ses réalisations.

1.2 CONTEXTE

Le contexte du travail présenté dans ce rapport se trouve à l'intersection de deux de mes centres d'intérêt : l'informatique et le traitement d'images. Ma thèse de doctorat avait été consacrée à la segmentation des structures internes du cerveau en IRM 3D [14]. La principale contribution de cette thèse avait été d'utiliser de la fusion floue dans un schéma itératif de vision par ordinateur. Parrallèlement à mon travail de recherche, j'avais dû écrire une bibliothèque logicielle de traitement d'images, TIVOLI [36]. Je devais en effet traiter des images tri-dimensionnelles et il manquait un outil logiciel permettant de gérer ce type d'images. D'une part, le laboratoire possédait deux BIBLIOTHÈQUES écrites en interne mais elles étaient limitées au TRAITEMENT des images bi-dimensionnelles ; d'autre part, la notion de logiciel en libre service sur Internet n'était alors pas encore très développée dans le domaine de l'imagerie, autant dire qu'aucun outil était disponible. A l'issue de ma thèse, j'avais pu juger des défauts de l'outil que j'avais contribué à créer.

À gros traits, ces défauts forment deux classes et touchent deux catégories de traiteurs d'images [2]. Pour ceux qui écrivent des algorithmes, le code est extrêmement redondant, trop verbeux et peu sûr. Pour les traiteurs non programmeurs, de simples utilisateurs donc, les opérateurs de traitements offerts par cette bibliothèque sont limités à quelques types de valeurs de pixels / voxels, et ce, sans raison apparente. Ces défauts sont à mettre en perspective par rapport aux idées du préambule ; on peut alors saisir l'absurdité de la situation. Le traiteur d'images qui veut bien programmer n'a pas la tâche facile d'un point de vue informatique (il a l'habitude de *souffrir* et celui qui ne veut pas programmer se heurte aux limitations de son outil informatique (il a l'habitude d'*échouer*). Le traiteur d'images est donc confronté à des difficultés liées à l'informatique, à des problèmes informatiques. Même s'il pense que l'informatique n'est pas son métier, l'outil informatique fait maintenant partie intégrante de son métier et, avec un peu de recul, on peut affirmer que cet outil est loin d'être parfait, loin d'être adapté au traiteur d'images. En fait, un traiteur d'image ne devrait ni souffrir ni échouer, mais il n'a pas vraiment conscience qu'il peut en être autrement.

Arrivant dans une école d'informatique, je voulais construire un outil logiciel permettant d'effectuer efficacement de la recherche en traitement d'images. Plus précisemment, j'étais à la recherche d'un outil dont le pré-requis était de ne pas être limitatif, donc non bloquant, mais aussi qui me permette de travailler vite, donc puissant tout en restant simple d'utilisation et maniable, et enfin, qui m'incite à explorer de nouvelles pistes de recherche.

2. Ces défauts seront détaillés dans la suite de ce rapport.

Cela n'a pas été facile. Autrefois, le menuisier fabriquait ses outils ; ce n'est plus le cas de nos jours car ces outils, pour être plus puissants, sont devenus plus complexes et fabriquant d'outils est devenu un métier à part entière. Le sujet de ce rapport est l'élaboration d'un outil pour le traiteur d'images. Cet outil dépasse de beaucoup les compétences que l'on est en mesure d'attendre de la part de n'importe quel traiteur d'images, y compris de ceux qui sont très à l'aise en informatique. En fait, nous nous sommes aperçu que la réalisation de cet outil relevait de la recherche actuelle en informatique. Le domaine de recherche principalement concerné est la "programmation générique" (*generic programming*). Ce domaine est souvent défini par son but : *permettre la construction d'algorithmes indépendants de l'implémentation des structures de données*. Cette définition, qui a l'avantage de la simplicité, est plutôt réductrice ; en effet, la notion de programmation générique dépasse le seul cadre de "façon de programmer" et s'étend aux problématiques de théorie des types, de paradigmes de programmation et de génie logiciel. Bref, c'est une notion multivoque.

Ce rapport va aborder les différents aspects, informatiques, de la réalisation d'un outil de traitement d'images. Puisque cet outil est destiné au traiteur d'images, notre choix s'est porté vers un niveau de langue informatique qui reste compréhensible pour le néophyte. Par cela, nous voulons sensibiliser le lecteur traiteur d'images à la bête "cachée" qu'il croise sans forcément la regarder avec attention, voire même sans la remarquer. Les axes de recherche et résultats présentés dans ce rapport seront donc détaillés du point de vue du traiteur d'images.

1.3 ORGANISATION DU RAPPORT

Ce rapport est organisé en plusieurs chapitres. Le chapitre 2 est une sorte d'avant-propos dont le but est double. Il s'agit d'expliquer au traiteur d'images quelles sont les problématiques informatiques de notre travail et par là-même de lui permettre de lire la suite du rapport en apportant les bases informatiques nécessaires à sa compréhension. En complément, dans le corps du texte, avant certains passages, des cadres rappelant quelques notions fondamentales ont été insérés.

Le cœur de notre travail de recherche est présenté dans les trois chapitres suivants. Le chapitre 3 traite de paradigmes de programmation dédiés au calcul scientifique intensif. Le but de ce chapitre est de présenter notre contribution sur l'obtention de traitements génériques et performants ; il est principalement illustré par le domaine applicatif du traitement d'images. Le chapitre 4 traite du passage du monde des images à celui de l'informatique. Il s'agit d'une introspection, au sens étymologique du terme, de ce que sont les images. Elle se traduit au final par une proposition de taxonomie des images, exploitable d'un point de vue informatique. La fin de ce chapitre est consacrée aux algorithmes ; elle se démarque du reste du rapport, plus orienté structures de données. Nous y validons *a posteriori* le multi-paradigme proposé, qui mélange inclusion, paramétrisation et déclaratif, en constatant qu'il permet d'obtenir de bonnes propriétés sur l'écriture de traitements.

Parmi les annexes, citons-en deux particulières. L'annexe D donne un aperçu de ce qu'est la bibliothèque MILENA. Et, pour ne pas fruster les informaticiens, la matière de notre travail qui relève de l'informatique "dure" est incluse dans ce rapport, bien que rejetée en annexe B.

2 UNE BIBLIOTHÈQUE GÉNÉRIQUE

Mais c'est impossible !

Une éminente chercheuse en traitement d'images à qui nous venions d'expliquer ce que nous faisions, 2009.

2.1 BIBLIOTHÈQUE LOGICIELLE

Ce chapitre traite de "bibliothèque générique" ; il peut être compris comme un avant-propos dont le premier but est de présenter les "pré-requis" à la lecture des chapitres suivants. Mais il va au-delà et présente déjà quelques résultats de réflexions et un certain nombre de conclusions concernant la notion de bibliothèque générique. Commençons donc par l'aspect "bibliothèque" avant d'aborder l'aspect "généricité".

2.1.1 *Avantages d'une bibliothèque*

Tout comme une bibliothèque propose des livres en libre service, une bibliothèque logicielle est une collection de fonctions que l'utilisateur peut appeler à sa guise. Cette collection ne constitue pas un programme car elle ne possède pas de fonction principale ; il revient alors à l'utilisateur d'écrire des programmes en s'appuyant sur les fonctions disponibles. Une bibliothèque dédiée au traitement d'images permet au traiteur de construire des solutions logicielles à ses problèmes. L'utilisation d'une bibliothèque logicielle présente un certain nombre d'avantages.

GAIN DE TEMPS. À l'échelle d'une unique personne, une bibliothèque lui permet d'utiliser des fonctions déjà programmées au lieu de devoir les écrire. Si l'on considère la phase d'apprentissage de l'outil bibliothèque comme étant terminée, il s'agit donc d'un énorme gain de temps. En effet, la recherche au sein de la bibliothèque de la fonction que l'utilisateur désire et la réalisation d'un appel correct à cette fonction restent deux étapes rapides, tandis que devoir programmer cette même

fonction l'est beaucoup moins. Éviter de re-programmer ce qui existe déjà revient à ne pas vouloir ré-inventer la roue et permet de gagner à la fois du temps et d'économiser de l'énergie.

FOCALISATION. En fait, le traiteur d'images devrait pouvoir se focaliser au mieux sur les problèmes qu'il cherche à résoudre. Idéalement il devrait y consacrer toute son énergie et s'épargner le travail préliminaire qui consiste à construire les outils de base dont il aura besoin. Le simple fait de passer beaucoup de temps—trop de temps—à programmer est un signe que la focalisation du traiteur d'images sur son métier n'est pas optimale.

FIABILITÉ. Programmer une fonction ne suffit généralement pas, il faut également la tester. Elle doit tout d'abord être valide : elle doit délivrer le résultat attendu. Ce résultat est soit défini par la théorie, soit par des spécifications, lorsque la théorie est limitée par des contraintes techniques. Toute fonction doit également être robuste : son comportement doit rester acceptable même lorsque une anomalie survient. Le temps nécessaire à rendre un outil fiable, c'est-à-dire à la fois valide et robuste, est beaucoup plus élevé que le temps initial passé à programmer cet outil. Une bibliothèque logicielle propose généralement des fonctions fiabilisées avec le temps, donc débarassées de défauts et d'erreurs. Par opposition, une fonction écrite en *one-shot* a relativement peu de garantie d'être fiable.

STABILITÉ. À un niveau de fiabilité équivalent, une bibliothèque présente l'avantage d'être de meilleure qualité qu'une production hétérogène d'outils. Cela est dû au fait que l'ensemble de fonctions au sein d'une bibliothèque repose sur un socle commun. Ce socle est constitué de "briques" de base, utilitaires, utilisées par les différentes fonctions. Il est unique, fondamental et largement éprouvé car indispensable au bon fonctionnement des fonctions. Élément essentiel, il est inévitablement maintenu en état de fonctionnement et il évolue avec son environnement, en particulier avec l'apparition de nouvelles versions de compilateurs. Ainsi ce socle garantie une bonne stabilité à l'ensemble de la bibliothèque. À l'inverse, dans le cas d'outils ou fonctions inhomogènes, il y a autant de socles que d'outils et la stabilité de cette multiplicité est alors beaucoup plus aléatoire.

QUALITÉ. La fiabilité et la stabilité ne sont les deux premiers objectifs des opérations de maintenance. Le troisième est souvent l'amélioration de l'outil, ce qui se traduit par une qualité croissante de cet outil avec le temps. Comme tout logiciel, le contenu d'une bibliothèque est intrinsèquement modelable—grâce à la nature "*soft*" du *software*. Avec son utilisation répétée, certains aspects de ce logiciel ont pu être révisés afin d'être bonifiés ou certaines caractéristiques sympathiques ont pu être ajoutées. À l'inverse, un logiciel qui n'a pas été écrit pour être utilisé de façon récurrente ne sera presque jamais l'objet de telles opérations d'amélioration. Au final, un utilisateur peut alors attendre de fonctions de bibliothèque une qualité supérieure à sa propre production.

PÉRENNITÉ. Les actions de maintenance, inéluctables pour une bibliothèque logicielle, induisent un avantage certain de ce type de logiciel par rapport à une collection disparate de logiciels

ou par rapport à des logiciels qui non pas été prévus pour être utilisés de façon répétée. Cet avantage, très important, est la pérennité de l'outil. Qui n'a pas recherché un logiciel qu'il avait utilisé mais qui ne sait plus le retrouver ? Avec une bibliothèque, cet outil est naturellement localisé et ne peut pas être perdu. Qui n'a pas repris un "vieux" logiciel pour s'apercevoir qu'il n'est plus utilisable instantanément et finit par l'abandonner ?

CAPITALISATION. Une bibliothèque est un conteneur à fonctions. Y ajouter une nouvelle fonction est possible et, même, naturel. En effet, une bibliothèque fournit une boîte à outils élémentaires facilitant l'ajout de fonctions, donc favorise voire suscite l'augmentation de ce capital de fonctions. Bénéficier d'une bibliothèque permet de capitaliser un nombre croissant de fonctions et tout utilisateur peut bénéficier de cet enrichissement, qu'il en soit l'auteur ou pas.

DISPONIBILITÉ. Une bibliothèque est synonyme d'un large ensemble de fonctions, généralement plus étendu que ce dont l'utilisateur a besoin. C'est donc un panel large de fonctions déjà disponibles qui est offert à l'utilisateur. Au delà du gain de temps réalisé en évitant le re-développement de fonctions existantes et au delà de sa faculté à capitaliser des fonctions, c'est avant tout sa disponibilité qui caractérise une bibliothèque. Et les notions de stabilité et de pérennité qu'elle garantie favorise cette disponibilité.

UNIFORMITÉ. Une bibliothèque force les fonctions à suivre un certain format. Ce cadre formatif est sein puisqu'il uniformise l'outil. Il y a alors capitalisation de l'apprentissage de la bibliothèque puisque les contraintes de forme facilitent la transposition des connaissances acquises d'une fonction à d'autres. Ce dernier point est à mettre en contraste par rapport à l'utilisation d'une collection d'outils hétérogènes, où chacun nécessite un apprentissage spécifique.

2.1.2 *Bénéfices d'une bibliothèque pour un groupe*

À l'échelle d'un groupe de personnes, des traiteurs d'images appartenant à un même laboratoire par exemple, les avantages donnés dans la section précédente se muent en véritables bénéfices.

TRAVAIL COLLABORATIF. Disposer d'une bibliothèque à l'échelle d'un groupe de personnes améliore le travail collaboratif. En effet, plusieurs personnes pourront travailler sur le même "projet", au sens du dépôt logiciel (à l'aide d'outils de gestion de versions décentralisée, comme GIT par exemple). L'intégration du travail de ces différentes personnes en est facilitée.

FACILITATEUR DE TRANSFERT. La reprise du travail d'une personne par quelqu'un autre est également facilitée. L'environnement (la bibliothèque) n'a pas changé : le "repreneur" n'a pas à ré-écrire dans son environnement à lui ce qu'on lui a donné, il peut donc se concentrer sur la matière à reprendre, et non sur comment il peut le reprendre.

ACCELÉRATEUR DE COMPÉTENCES. En corollaire, on peut aussi y trouver des avantages en termes de transfert de compétences d'une personne à une autre : la personne la plus faible du

point de vue de l'outil logiciel pourra être aidée par d'autres. De par notre expérience, nous avons constaté que la "montée en puissance" d'un nouveau venu est rapide. Non seulement, plusieurs personnes sont capables de l'aider mais cette aide est souvent efficace : il suffit souvent d'aller pointer, dans la bibliothèque, un exemple existant, pertinent pour servir d'apprentissage.

LANGAGE COMMUN. Au sein d'un groupe, tout le monde va parler le même *langage* de traitement d'images, celui de la bibliothèque. Cet avantage est à mettre en opposition avec le cas de deux personnes ne parlant pas la même langue et essayant de communiquer.

Pour tous ces bénéfices, avoir *une* bibliothèque pour tout un groupe est importante pour ce dernier.

2.1.3 *Types d'utilisateurs*

Nous avons identifiés quatre catégories distinctes d'acteurs liés à une bibliothèque de traitements d'images :

- les assembliers ;
- les créateurs-concepteurs ;
- les fournisseurs ;
- les architectes.

ASSEMBLIERS. Un assemblier se contente de manipuler les briques de base offertes par la bibliothèque pour composer une application. Il ne réalise que de l'assemblage de composants. Mis bout à bout ces derniers forment une solution. L'outil idéal pour un assemblier est un environnement de programmation visuelle. Sa tâche consiste à dessiner des diagrammes de flux de données. Chaque boîte d'un tel diagramme matérialise un traitement et les liens entre les boîtes matérialise le flux des données. Si un diagramme ressemble à une séquence d'appels à des fonctions et comporte des structures de contrôle, le résultat n'est pas, à proprement parlé, un algorithme mais une "chaîne de traitements". Grâce à l'interface graphique d'un environnement de programmation visuelle, l'assemblier n'a besoin d'aucune compétence particulière en informatique et, *a fortiori*, en "véritable" programmation. En l'absence d'un environnement, les enchaînements de traitements sont décrits à l'aide d'un langage de programmation textuel, le C par exemple. Mais, pour ce type de tâche, les connaissances requises en programmation restent d'un niveau relativement bas puisqu'il s'agit de ne réaliser que des appels aux fonctions.

CRÉATEURS. Un créateur cherche à concevoir un nouvel algorithme ou une nouvelle méthode de traitement d'images. Il ne peut pas se contenter d'assembler des fonctions existantes pour obtenir ce qu'il veut. Son but est de créer une nouvelle fonction, quelle soit plutôt élémentaire, comme un "simple" opérateur de traitement d'images, soit de plus haut niveau, comme une tâche applicative "complexe". Il a un réel besoin de programmation ; d'ailleurs, un environnement de programmation visuelle ne peut pas lui rendre le service qu'il attend. Les connaissances requises en programmation pour un créateur sont d'un niveau déjà supérieur à celui d'un assemblier, sans toutefois atteindre un niveau très élevé. La plupart des traiteurs d'images programmant se situent dans cette catégorie

d'utilisateurs. Cette catégorie est le sujet de la discussion du préambule, page 5 et suivantes : le traiteur d'images créateur est véritablement un concepteur de programmes. Pour cela, il doit être aidé par son outil, la bibliothèque, surtout si son niveau d'informatique reste relativement modeste. C'est à cette personne au comportement dual, créateur de traitements et concepteur de programmes, que s'adresse le travail décrit dans ce rapport.

FOURNISSEURS. Un fournisseur se distingue d'un créateur car il n'apporte pas des fonctions mais de la matière première. Cette matière se divide en deux catégories : les types de données et les utilitaires. Parmi les types de données se trouvent les structures de données servant à définir des images, par exemple, un type pour gérer une image bi-dimensionnelle en niveaux de gris, et des types destinés aux valeurs des pixels, par exemple le type de couleur correspondant à un espace colorimétrique donné comme RGB (*Red-Green-Blue*). Les types de données représentent la matière manipulée par les fonctions, cette matière formant le flux de données d'un programme de traitement. Définir des types de données nécessite des compétences déjà avancées en informatique. En effet il s'agit d'abord de comprendre comment ces types doivent être formés pour être compatibles, donc utilisables sans problème, avec la bibliothèque. Il en va de même pour les "petits" objets utilitaires qui doivent se marier parfaitement avec les types de données et les algorithmes. Contrairement aux deux catégories précédentes, assemblier et créateur, un fournisseur doit avoir l'âme d'un informaticien.

ARCHITECTES. Un architecte ne fait clairement pas de traitement d'images ; son travail est centré sur la bibliothèque en tant que logiciel. Il effectue de la maintenance : des mises à jour, des correctifs, des améliorations ; et peut-être même de la conception, du re-modelage de logiciel (*refactoring*), du génie informatique, de l'architecture logicielle. Pour cette dernière catégorie, le niveau informatique attendu est supérieur, voire nettement supérieur, à celui des autres catégories. En effet, un architecte doit pouvoir saisir les différentes problématiques liées à l'organisation complexe que peut être celle d'une bibliothèque et, pour cela, doit bien maîtriser l'outil informatique.

TABLE 1.: Catégories de clients d'une bibliothèque.

assemblier	plus de 90%
créateur	moins de 10%
fournisseur	moins de 1%
architecte	ϵ%

Pour se fixer les idées quant aux compétences requises par ces différents utilisateurs, on peut observer les artefacts suivants : un assembleur, s'il programme, écrit essentiellement des appels à des fonctions ; un créateur doit savoir écrire des boucles telles que "pour chaque point de l'image" ; un fournisseur connaît les mots-clefs struct et/ou class et les notions associées ; un architecte a déjà ouvert la norme du langage qu'il utilise. Pour donner une répartition de la population d'utilisateurs dans ces différentes catégories, il faut tenir compte qu'une même personne a souvent un rôle double. Le cas manifeste est celui du créateur qui, pour son travail, utilise des fonctions déjà toutes

faites et, par cela, est également partiellement un assembleur. Au final, une tentative de répartition approximativement est dans la table 1.

Cette répartition se comprend par temps passé dans une des catégories. Elle est statistique par personne, par temps passé et corrigée par le poids de code considéré. Un créateur, par exemple, lorsqu'il appelle une fonction existante ne passe que peu de temps à écrire cet appel, pourtant le poids, en nombre de lignes de code dépliées, appelées en cascade par cette fonction, est important ; le poids en tant qu'assemblier devient fort car la quantité de matière utilisée (et non créée) est très significative.

Une telle répartition peut aussi être rapportée à la fréquence des tâches : utiliser l'existant (assembler) est courant, créer l'est moins, et fournir un nouveau type de données ou un nouvel utilitaire est déjà très rare.

2.1.4 *Bibliothèque comme cœur d'un environnement*

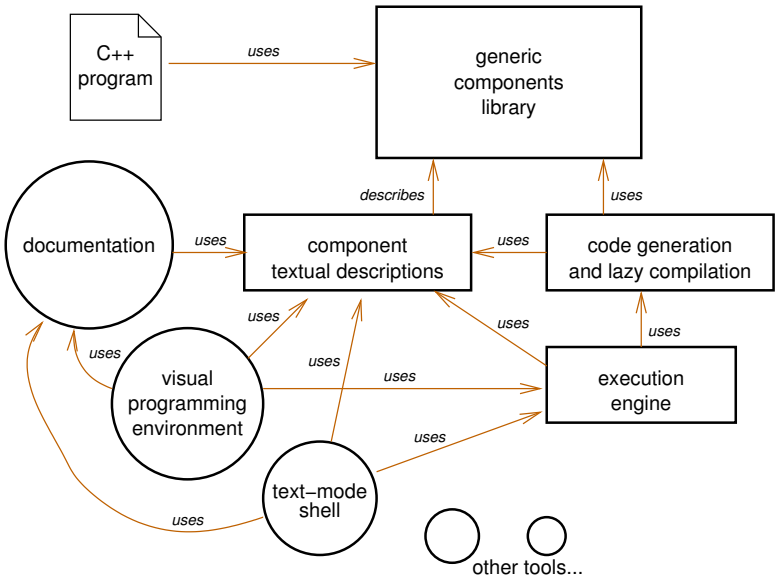


FIGURE 1.: Une architecture d'environnement.

Une bibliothèque n'est pas toujours un outil isolé ; elle peut être partie intégrante d'un environnement logiciel plus large. Une architecture possible pour ce dernier est donnée en figure 1.

Du côté de l'utilisateur, cet environnement peut comprendre des outils indépendants qui lui permettent de travailler sans pour autant devoir coder dans le langage de programmation de la bibliothèque. Citons par exemple :

- un jeu de commandes en ligne pour lancer des traitements à partir d'un invite de commandes (*shell*), ou pour pouvoir écrire des scripts
- un outil de programmation visuelle, comme CANTATA de KHOROS ;
- un logiciel de traitement d'images avec interface graphique, comme THE GIMP, dans lequel les traitements sont accessibles à partir de menus ;

– une interface écrite dans un langage de programmation tiers afin de pouvoir appeler les fonctions de la bibliothèque à partir d'un autre environnement, par exemple, en Java lorsque la bibliothèque est écrite en C++ ;
– etc.

Au final, ces outils utilisent les fonctionnalités offertes par la bibliothèque, même si l'utilisateur ne voit pas explicitement que celle-ci est utilisée, asservie, en interne. Les fonctions de traitement d'images qui sont appelées de tous les outils satellites imaginables sont véritablement celles de la bibliothèque. Autant dire qu'elle est donc finalement le cœur de l'environnement.

L'ensemble des fonctions de traitement d'images disponibles pour l'utilisateur correspond à celui que fournit la bibliothèque, ou à un sous-ensemble si toutes les fonctionnalités ne sont pas "traduites" vers les outils satellites, rendues accessibles à ces outils. Pour que l'utilisateur puisse profiter de tout ce qu'offre une bibliothèque, cette transmission de fonctionnalités vers les outils satellites doit est réalisée soit complètement automatiquement, soit sans effort supplémentaire. Pour cela, nous avons inclus dans l'architecture une description des fonctionnalités sous la forme de composants. Cette description, qui doit rester en phase avec le contenu de la bibliothèque, est exploitée comme donnée brute pour générer les ponts entre la bibliothèque et tous ses périphériques. Ainsi, l'accès aux fonctionnalités peut être garanti.

Les fonctionnalités des outils de l'environnement provenant de la bibliothèque, il faut bien comprendre que les services sont donc finalement rendus à l'utilisateur par la bibliothèque. Les outils satellites sont en fait des sur-couches, côté utilisateur, du cœur central qu'est la bibliothèque. Le périmètre des fonctionnalités de l'environnement est clairement défini par le contenu de la bibliothèque. Ces fonctionnalités sont internes, bien encloses dans la bibliothèque. En corollaire, une remarque cruciale émerge lorsque l'on considère l'extérieur de ce périmètre : tout se qui est impossible pour la bibliothèque restera obligatoirement impossible pour l'utilisateur final, quelque soit l'outil périphérique qu'il emploie. Avec la notion de périmètre vient celle de limite. Si la bibliothèque possède des limitations, celles-ci sont automatiquement transmises à l'environnement. Le périmètre d'utilisation d'une bibliothèque a donc tout intérêt à être le plus large possible. Cette conclusion semble assez évidente ; ce qui ne l'est pas, en revanche, est de considérer qu'*obtenir le plus large périmètre d'utilisation possible doit être une des priorités principales de la conception d'une bibliothèque*.

2.2 OBJECTIFS

2.2.1 *Contenu et public*

Pour comprendre les objectifs qui sous-tendent la réalisation d'une bibliothèque, il faut d'abord s'intéresser à son contenu. Il y a plusieurs façons de décrire ce contenu. Une première consiste à distinguer différentes natures d'entités définies dans une bibliothèque :

$\mathcal{A}$ les algorithmes ;
$\mathcal{T}$ les types de données ;
$\mathcal{U}$ les utilitaires.

Les algorithmes sont des fonctions élémentaires ; ce sont des traitements qui produisent des résultats à partir de données ou modifient les données qui leur sont passées en entrée. Du point de vue du traitement d'images, un algorithme peut être un filtre, un opérateur élémentaire de traitement, un calcul estimatif sur une image, etc. Les types de données sont définis pour pouvoir manipuler des données. Ils peuvent être relativement simples comme par exemple un type de couleur correspondant à un espace colorimétrique précis ou plus complexes comme un type d'images. Ce dernier est un type conteneur puisqu'il peut falloir stoker des valeurs pour chaque pixel de l'image. Les utilitaires sont des outils, généralement de taille modeste, qui rendent des services particuliers. Cette dernière catégorie d'entités est très hétérogène car elle est constitué d'utilitaires de nature de très diverse. Parmi les utilitaires rencontrés en le traitement d'images, on trouve la notion de point, de fenêtre, de voisinage ; on trouve aussi les fonctions mathématiques classiques, les transformations géométriques, etc.

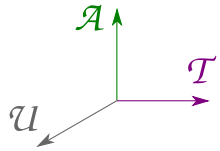


FIGURE 2.: Axes : $\mathcal{A}$lgorithmes / $\mathcal{T}$ypes / $\mathcal{U}$tilitaires.

Ces trois natures d'entités seront notées respectivement $\mathcal{A}$, $\mathcal{T}$ et $\mathcal{U}$ par la suite. Elles peuvent être vues comme trois axes "relativement" indépendants d'une bibliothèque, Cf. figure 2. Cette indépendance se reflète dans les livres par le fait que ces entités sont définies séparément, dans des chapitres différents. Les illustrations de cette indépendance sont nombreuses ; nous en donnons ici trois. 1. La plupart des algorithmes peuvent agir sur différents types de données ; rares sont ceux qui sont véritablement dédiés à un type précis de données. Le filtre de Sobel par exemple s'applique aussi bien à des images bidimensionnelles que tridimensionnelles. 2. Lorsqu'un type est défini, c'est souvent intrinsèquement, sans référence aux algorithmes qui peuvent lui être appliqués. Par exemple, tel espace colorimétrique est défini avant tout par un certain nombres de propriétés, pas par les traitements ou algorithmes qui lui sont applicables. D'ailleurs, soit ces traitements pré-existent, soit ils ne sont définis qu'*a posteriori*. 3. Enfin, les outils existent en tant que tels et se caractérisent par leur grande polyvalence. Ainsi un point (deux coordonnées en 2D) a une identité mathématique propre ; ce n'est qu'à titre d'accessoire qu'il sert à exprimer le domaine de définition d'une image bidimensionnelle. Le fait que l'on puisse voir une certaine indépendance entre algorithmes, types et utilitaires joue une importance particulière pour une bibliothèque. En effet, ces entités peuvent y être rangées dans des compartiments distincts.

Les différentes catégories d'utilisateurs données en section 2.1.3 peuvent maintenant être décrites vis-à-vis de ces entités :

- les assembleurs connaissent les axes $\mathcal{T}$ et $\mathcal{U}$ mais manipulent essentiellement l'axe $\mathcal{A}$;
- les créateurs étendent l'axe $\mathcal{A}$, en utilisant de façon accessoire l'axe $\mathcal{U}$;
- les fournisseurs définissent les entités des axes $\mathcal{T}$ et $\mathcal{U}$;
- quant aux architectes, ils maintiennent la cohérence entre les trois axes.

FIGURE 3.: Entités indépendantes mais cohérentes.

In fine ces entités doivent pouvoir être manipulées conjointement, voire elles doivent pouvoir collaborer. Dans le contexte d'une bibliothèque, ces entités doivent ainsi être cohérentes. Pour reprendre les exemples donnés ci-avant, il devra être possible d'appliquer un filtre de Sobel sur image bidimensionnelle en couleur, tout comme écrire un filtre dont le comportement s'adapte en chaque point, etc. Les trois axes "algorithme - types - utilitaires" doivent réussir à former un ensemble, un espace de travail. Leur indépendance est ainsi relative. Les notions d'indépendance et de cohérence ne sont pas incompatibles comme le montre la figure 3 et c'est le rôle de l'architecte de garantir le fonctionnement harmonieux de l'ensemble. Lors de la conception de la bibliothèque d'abord, puis tout au cours de la vie de ce logiciel.

2.2.2 *Objectifs mous*

Les objectifs que nous nous sommes fixés sont décrits par la suite. Nous les avons classés en deux catégories. Les objectifs "mous" dans un premier temps sont des objectifs relativement généraux. Les objectifs "durs" viennent ensuite ; plus pragmatiques, ils traduisent les objectifs mous sous le jour d'une incidence "technique" est forte. Ils se rapprochent plus du "comment".

GÉNÉRALISTE. En faisant le tour des bibliothèques disponibles en traitement d'images, on peut remarquer qu'elles sont décrites soit par leurs fonctionnalités applicatives, soit par les images qu'elles ciblent. Dans le premier cas, l'emphase a été mise sur un jeu d'algorithmes, donc l'axe $\mathcal{A}$, et ces bibliothèques se décrivent par la liste de leurs fonctions :

> CIMG *"contains useful image processing algorithms for image loading/saving, displaying, resizing/rotating, filtering, object drawing, etc."*

ou par le dénominateur commun de leurs fonctions :

> ITK *"an open-source software toolkit for performing registration and segmentation."*
>
> OPENCV *"example areas are object identification ; segmentation and recognition ; [...]"*
>
> MORPH-M *"contains most of the operators offered by mathematical morphology."*
>
> QGARLIB *"a set of C++ components implementing basic graphics analysis and recognition methods."*

Dans le second cas, en revanche, il s'agit de réussir à traiter un type d'images particulier ou une liste de types d'images ; l'axe caractéristique de ces "autres" bibliothèques est alors l'axe $\mathcal{T}$:

> IMLIB3D *"an open source C++ library for 3D (volumetric) image processing."*
>
> PANDORE *"regroupe des opérateurs traitant d'images 1D, 2D et 3D, en niveaux de gris, en couleurs et multi-spectrales."*

Enfin, la fusion de ces deux approches donne des bibliothèques où c'est le domaine métier, applicatif, qui est l'objectif premier :

> INVT *"is a comprehensive set of C++ classes for the development of neuromorphic models of vision"*
>
> ORFEO TOOLBOX *"offers particular functionalities for remote sensing image processing."*

Contrairement à ces approches, nous avons souhaité construire une bibliothèque généraliste. Cela signifie qu'elle n'est pas liée à un domaine applicatif particulier, qu'elle ne favorise pas telle saveur de fonctionnalités (filtrage, segmentation, recalage, etc.) et qu'elle ne se focalise pas non plus sur certains types de données. Garder cette généralité à l'esprit est un garde-fou pour ne pas "fermer" la bibliothèque en limitant ses possibilités.

NON LIMITATIVE. En voulant rester général, nous pouvons espérer obtenir un objectif encore plus fort : une bibliothèque qui ne soit pas limitative. Derrière l'idée de mettre à disposition une bibliothèque se cache une utopie flamboyante, pointée du doigt par cette remarque :

> *"Pourquoi voulez-vous écrire une bibliothèque ? Il y a trop d'algorithmes... vous ne pouvez pas tous les implémenter !"*—Josiane Zérubia, conversation privée, 1995.

D'un côté, cette assertation peut être facilement réfutée : ne pas pouvoir tout implémenter n'est pas une raison pour ne rien implémenter. D'un autre côté, le domaine du traitement d'images est effectivement riche ; les méthodes (ou algorithmes) sont effectivement en nombre plétorique dans la littérature, mais surtout, ce domaine recouvre une large diversité d'approches. Cette dernière constatation se traduit par le fait qu'un traiteur d'images à la recherche d'une fonction précise a relativement peu de chance de la trouver, déjà disponible, dans une bibliothèque. La multiplicité

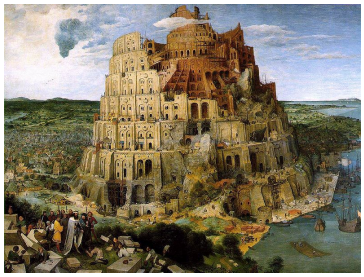


FIGURE 4.: Une autre bibliothèque de traitement d'images !

de sujets et de solutions en traitement d'images peut ainsi mettre en échec la pertinence d'une bibliothèque, comme dans le cas de la figure 4.

En fait, il existe une différence fondamentale entre la disponibilité des fonctionnalités et le potentiel de fonctionnalités. Pour une bibliothèque qui se veut généraliste, vouloir atteindre une disponibilité satisfaisante de fonctionnalités est voué à l'échec et cet objectif est utopique. En revanche, proposer un environnement dans lequel tout développement de fonctionnalités est possible semble un objectif atteignable. Il s'agit alors de privilégier les possibilités, bien plus que les "déjà possibles". Pour cela, il faut penser le contenu d'une bibliothèque comme n'étant pas (ou restant peu) limitatif.

EFFICACE. Un objectif particulier est l'efficacité. Ce terme est à prendre au sens large : tirer le meilleur parti des ressources informatiques disponibles. Il se justifie par la nécessité d'obtenir de bonnes performances en temps d'exécution, donc des temps d'exécution courts, et en termes de ressources matérielles utilisées, par exemple en n'étant pas dispendieux en mémoire vive (RAM). Tout le monde n'est pas intéressé par l'objectif d'efficacité. En revanche, lorsqu'il s'agit de traiter de grands volumes de données, cet objectif devient souvent une nécessité car il s'agit alors de réussir à obtenir des résultats dans des temps raisonnables et dans la limite des gigas de RAM supportée par les ordinateurs classiques. Ces volumes de données sont maintenant monnaie courante ; ce sont soit des images de (très) grande taille, soit de (très) gros jeux d'images. Pour réaliser des traitements, on peut ne pas vouloir ou pouvoir se contenter de prototypes lents. On peut par exemple souhaiter être capable de donner un temps de calcul estimatif à un industriel. De plus, les chaînes de traitements, pour être *a priori* de meilleure qualité, sont de plus en plus sophistiquées donc plus lourdes en traitements et plus gourmandes en ressources temps et matérielles. L'efficacité est synonyme d'amélioration sans pour autant sacrificer l'utilisabilité. Ne pas viser l'efficacité, c'est accepter de perdre du temps et s'infliger un facteur multiplicatif inutile en temps de calcul lorsqu'il est facilement évitable. À l'extrême inverse, la quête d'efficacité ne doit pas aller jusqu'à vouloir tout optimiser, ni optimiser finement certaines parties, car vouloir gratter des sur-performances ridicules a également un coût.

SIMPLE. Enfin, le dernier objectif mou est celui de la simplicité. Contrairement à une idée reçue, le complexité inhérente à l'obtention de fonctionnalités "avancées" n'est pas obligatoirement synonyme de difficultés pour l'utilisateur. L'interface, la partie visible des outils et accessible à

l'utilisateur, peut être simple bien que le code en interne, non exposé à l'utilisateur, soit complexe. Si les fonctionnalités offertes ne sont pas aisément compréhensibles et manipulables alors l'objectif de fournir des outils pour *aider* l'utilisateur n'est pas atteint. Les concepts et interfaces fournis par la bibliothèque doivent nécessairement rester simples.

2.2.3 *Objectifs durs*

Lors de leur mise en pratique, les objectifs décrits dans la section précédente—généraliste, non limitative, efficace et simple—se traduisent respectivement en riche, générique, performante et proche du langage naturel.

RICHE. Pour être généraliste, une bibliothèque doit avant tout être riche. Les axes $\mathcal{T}$ et $\mathcal{U}$, types de données et utilitaires, sont ceux que ne développera pas les clients majoritaires d'une bibliothèque (les assembleurs et les créateurs). Comme ils sont indispensables, ils doivent être présents, disponibles en amont. La richesse de ces entités permet le "possible". La disponibilité essentielle ne concerne pas les algorithmes (axe $\mathcal{A}$) mais le reste.

GÉNÉRIQUE. L'objectif de généricité est le sujet central de ce rapport. Une section à part entière de cet avant-propos, section 2.3, lui est consacré ; nous nous contentons ici d'illustrer la notion de généricité sur un exemple.

FIGURE 5.: Pas générique (à gauche) v. générique (à droite).

La figure 5 montre à gauche un lecteur de cartes mémoire photo ; comme il y a plusieurs types de cartes différentes, ce lecteur propose une prise pour chaque type. Le problème surviendra lorsqu'un nouveau format de carte sera défini : ce lecteur ne pourra pas le prendre en compte ; il atteindra alors ses limites. À droite, la figure montre un terminal (*hub*) USB. Contrairement au cas du lecteur de cartes, l'arrivée d'un nouveau périphérique ne posera pas de problème : les ports USB sont **génériques**, adaptés et adaptables à toute nouveauté. Dans la section précédente, nous avons posé l'objectif de pouvoir disposer d'une bibliothèque "non limitative." Comme dans le cas de l'USB, elle doit être générique, c'est-à-dire posséder le périmètre d'utilisation le plus large.

PERFORMANTE. Nous comptons vraiment traiter des images et, quelquefois dans le cadre de contrat industriel où l'on va nous demander d'évaluer les temps d'exécution. Il n'est donc pas question d'avoir des outils plus lents, moins efficaces, que nécessaires ; nous ne serions pas en

mesure d'estimer les temps de traitements de façon fiable. De plus, deux cas d'utilisation sont assez courants :

– les images de très grandes tailles ;
– les bases d'images très peuplées ;

avec, bien entendu, la combinaison de ces deux situations. Une bibliothèque logicielle doit rester effective dans ces contextes. Le volume des données va toujours croissant avec le temps ; pourquoi accepterions-nous des facteurs multiplicatifs en temps d'exécution dus à l'outil ? Si nous pouvons garantir la performance de notre outil, pourquoi s'en priver ?

PROCHE DU LANGAGE NATUREL. Le but d'une bibliothèque dédié à un métier en particulier est de parler le langage du domaine. Il n'est pas question que la bibliothèque ressemble à un alien du point de vue de son utilisateur. Elle doit donc reprendre les noms des entités du domaine et essayer de coller au plus prêt de ce qui est connu *a priori* des gens de sa communauté. Un second aspect important est qu'un algorithme devrait pouvoir s'exprimer comme dans les articles scientifiques. Son implémentation ne devrait donc pas être alourdie de "détails d'implémentation" ; puisqu'ils ne sont que détails, ils ne devraient pas apparaître. Au final, un bon critère pour savoir si une bibliothèque est proche du langage naturel de son utilisateur consiste à comparer le nombre de lignes de la description de l'algorithme et celui de son implémentation. Idéalement, ces deux nombres devraient rester très proches.

2.3 GÉNÉRICITÉ EN TDI

Dans cette section, nous abandonnons partiellement la seule notion de "bibliothèque" pour discuter de "bibliothèque générique". Et avant tout, il faut comprendre ce que "générique" signifie. Nous entrons donc dans le vif du sujet.

2.3.1 *Un exemple de non-généricité*

Nous allons commencer par donner un exemple de non-généricité avec la routine la plus simple que l'on retrouve dans toutes les bibliothèques de traitement d'images : le remplissage (fill) de tous les pixels d'une image avec une même valeur. En fait, la généricité touche toutes les parties d'un logiciel comme une bibliothèque, il n'est donc pas étonnant qu'une fonction, même aussi simple que fill, soit déjà entâchée de non-généricité !

```
void fill(image* ima, unsigned char v)
{
  unsigned short row, col;
  for (row = 0; row < ima->nrows; ++row)
    for (col = 0; col < ima->ncols; ++col)
      ima->data[row][col] = v;
}
```

Code source 2.1: Version non-générique de fill.

L'exemple de code donné en figure 2.1 est on ne peut plus classique, on le retrouve dans 9 bibliothèques sur 10, soit exactement sous cette forme, soit sous une forme très approchée. Cette routine de seulement 7 lignes comporte pas moins de 8 hypothèses implicites :

1. les valeurs des pixels sont des entiers non signés sur 8 bit (unsigned char, ligne 1) ;
2. les coordonnées des points sont encodées sur 16 bit (short, ligne 3) ;
3. les coordonnées des points sont toujours positives (unsigned, ligne 3) ;

pour les lignes 4 et 5 :

4. l'image est bidimensionnelle ;
5. plus précisemment, elle est rectangulaire ;
6. l'origine du domaine de définition de l'image est le point "en haut à gauche" de l'image (de coordonnées $(0,0)$) ;

enfin, ligne 6 :

7. les valeurs des pixels sont stockés en mémoire ;
8. et ce stockage est à double indexation.

Ces hypothèses sont implicites car elles n'apparaissent pas directement dans l'écriture de cette routine ; on est obligé de les déduire des lignes de code. Le caractère implicite de ces hypothèses est assez pernicieux car, à cause de ces non-dits, la fonctionnalité fill se révèle en fait être "fill pour des images 2D rectangulaires avec valeurs en RAM, etc." Ces hypothèses sont donc autant de contraintes qui limitent le champ d'application de cette routine.

Ces différentes hypothèses peuvent se lire sous la forme de limitations aboutissant à des impossibilités. Pour chaque hypothèse, nous donnons ici un exemple sortant de leur périmètre :

1. les pixels ne peuvent pas avoir des valeurs flottantes ;
2. l'image ne peut pas être de grande taille ;
3. le point de coordonnées $(-1,-1)$ ne peut pas faire partie du domaine de définition d'une image ;
4. l'image ne peut pas être tridimensionnelle ;
5. l'image (bidimensionnelle donc) ne peut pas être autre que rectangulaire ;
6. l'origine du domaine de définition de l'image ne peut pas être le point $(1,1)$;
7. même si l'image est uniforme (tous les pixels ont la même valeur), cette valeur est stockée autant de fois qu'il y a de pixels ;
8. les valeurs des pixels ne peuvent pas être calculées à la volée.

Au final, telle qu'écrite, la routine fill est limitative ; elle ne peut pas s'appliquer sur tous les types d'images qu'elle devrait pourtant pouvoir traiter. La présence de ces restrictions rend cette routine *non générique*. Des exemples de non-généricité ont déjà été donnés dans l'introduction en section 1.2. Durant notre thèse, les seules bibliothèques disponibles ne géraient pas les images tridimensionnelles donc elles n'étaient génériques. Nous avons donc développé la bibliothèque TIVOLI afin de pouvoir gérer, en plus des images bidimensionnelles, des images tridimensionnelles. Cette démarche est également limitée. Nous n'avons ajouté qu'un cas supplémentaire aux possibilités de traitements, celui des images tridimensionnelles ; en particulier, TIVOLI ne pouvait pas prendre en compte de nouveaux types de données, comme des images 3D+t par exemple. Ajouter la prise en compte d'un unique type supplémentaire, ou même un ensemble déterminé de plusieurs types, ne rend pas une bibliothèque générique pour autant. La situation est comparable à celle du lecteur

de cartes montré en figure 5 : les possibilités d'utilisation restent à terme limitées. TIVOLI était également non-générique.

2.3.2 *Besoin de généricité*

Les exemples de la section précédente mettent en lumière des restrictions qui, en fait, n'ont pas lieu d'être. L'implémentation présentée de la fonctionnalité de remplissage est sans doûte la plus utilisée statistiquement ; les images sont souvent bidimensionnelles, rectangulaires, avec des valeurs entières non signées sur 8 bit stockées en RAM, etc. Si l'utilisation de ce type d'images peut, à la limite, être considéré comme un cas général *statistiquement*, il ne revêt aucune autre forme de généralité. Pire, il s'agit d'un unique cas, donc d'un cas en particulier parmi de nombreux autres cas qui devraient être possibles. Vouloir être générique signifie considérer qu'un cas en particulier n'est pas un cas particulier mais une occurence du cas général.

Pour faire écho aux limitations listées précédemment, nous avons traité au cours de dix dernières années :

1. des images dont les pixels ont des valeurs flottantes ou ont pour valeurs des fonctions ;
2. des image de grande taille (plusieurs dizaines de milliers de pixels dans chaque dimension) ;
3. des images parfaitement définies en des points de coordonnées négatives ;
4. des signaux (images monodimensionnelles) et tridimensionnelle, des graphes, des complexes cellulaires ;
5. des images bidimensionnelles restreintes à une région du plan donc avec un domaine de définition quelconque ;
6. des images dont le domaine de définition est centré en $(0,0)$;
7. des images dont les valeurs de pixels sont stockées sur disque ou sont données par une expression mathématique ;
8. des images aux données éparses donc non accessibles aléatoirement.

De plus, il n'a pas été rare que des combinaisons de ces cas de figure se présentent.

À chaque fois que nous sommes sorti du cas classique "2D, rectangulaire, 8 bit...", nous avons considéré qu'il ne s'agissait que d'un cas à traiter, ni général, ni particulièrement particulier. Seulement un cas de plus, un cas quelquefois nouveau, lié au contexte du travail en cours. Se retrouver hors du périmètre des images "classiques" n'est pas une situation exceptionnelle. Donnons-en une illustration.

Dans le cadre du projet EFIGI, les données étaient des images d'astronomie et une image, véritablement astronomique, pesait plusieurs gigaoctets (Go). Dans ce type de situation, l'outil logiciel est souvent inadapté : soit les données ne rentrent tout simplement pas en mémoire et elles ne peuvent pas être traitées, soit l'ajout de mémoire vive rend les traitements possibles mais désespérément lents donc finalement impraticables. Deux solutions courantes existent. La première est de céder à la facilité, d'extraire une ou plusieurs sous-images de l'image originale et de ne traiter que ces sous-images. Avec cette solution, on accepte de ne traiter d'une partie d'une problème et, finalement, de baisser les bras face au problème complet. Les conséquences d'une telle attitude peuvent être assez négatives : on passe probablement à côté d'informations contenues dans les données laissées de côté (sur le bon ou mauvais fonctionnement des traitements par exemple) ; pire,

la méthode mise au point sur des bouts d'images peut ne pas s'appliquer *in extenso* lorsque les données sont complètes. Une seconde solution consiste à réaliser un partitionnement de l'image et à traiter l'image par parties. Dit autrement, cette solution revient à tordre le cou au problème pour le faire rentrer dans le périmètre de l'outil. Ici, le problème concerne la taille des données et il s'agit que ces dernières tiennent en RAM pour que l'outil logiciel soit utilisable. Aussi le traiteur d'images est contraint à déployer des efforts supplémentaires par rapport à son cœur de métier, le traitement d'images, et ce, afin de pallier une limitation purement informatique. De façon non surprenante, la solution élaborée est malheureusement souvent rapide, sale et temporaire.

Les exemples rigoureusement équivalents au précédent sont légion et, à chaque rencontre avec une limitation, l'alternative courante est de "faire moins bien" ou de "faire de l'informatique." Face aux limitations, opter pour l'une ou l'autre de ces solutions, de toute façon bancale, revient à accepter que son outil n'est pas adapté. Même lorsque l'on a conscience de cette situation, le pas mental suivant, souvent franchi, est de trouver "normal" que l'outil ne soit pas adapté, d'y être habitué et, finalement, de se résigner à cet état de fait. Parfois, l'utilisateur n'a même pas conscience que son outil n'est pas adapté parce qu'il pense que, de toute façon, aucun outil ne peut satisfaire à ses besoins. Dans son esprit, il ne s'agit donc pas d'une limitation de son outil mais d'une limite générale de l'outil informatique. Au final, il ne sait pas que tout ce qu'il fait pour contourner cette limitation est complètement évitable.

Il est sain de réaliser qu'un outil logiciel n'est pas toujours adapté et il est salvateur de savoir que des solutions à cet écueil existent. La généricité est le moyen de repousser les limites de l'outil.

2.3.3 *Généricité dans le langage*

Attention :
Pour faciliter la lecture d'exemples de programme, nous utiliserons un mini-langage dont la syntaxe est concise et sans ambiguïté. Cela permet également d'insister sur le fait que ce que nous présentons n'est pas lié à un langage en particulier. Ce mini-langage est décrit dans l'annexe A.

Derrière le terme 'généricité' se cachent plusieurs définitions. La plus connue est sans doûte la généricité en tant que caractéristique d'un langage de programmation. De ce point de vue, il s'agit d'un concept parmi ceux qui participent à la définition d'un langage. Cette généricité se traduit par l'introduction de paramètres dans la déclaration d'entités comme les classes et les procédures.

```
image2d_uint8 : type is
  class
  {
    - data : array of * uint8,
    + get : (row, col : int) -> uint8
    + set self : (row, col : int) -> ref uint8
    ...
  }
```

Code source 2.2: Classe non-générique d'images.

Considérons une classe d'images bidimensionnelles dont les valeurs des pixels sont explicitement d'un type donné, par exemple uint8, pour gérer le cas très courant de niveaux de gris sur 8 bit. Son code est donné en 2.2.

Cette classe est malheureusement *limitée* à ce type de valeurs. Si l'on souhaite manipuler des pixels à valeurs flottantes, il faudra ré-écrire une nouvelle classe dont le code sera extrêmement similaire au premier. En plus du problème de la limitation imposée sur le type de valeurs, cette duplication d'écriture est très maladroite. Non seulement, elle conduit à plus de travail de programmation, mais elle induit également plus d'erreurs potentielles, comme dans toute opération de "copié-collé-modifié" et, par conséquence, plus de maintenance car il y a plus de code à gérer et à garder cohérent.

DÉFINITION 1: Paramètre formel

Dans les langages de programmation, un paramètre formel est une variable spéciale, déclarée comme entrée d'une entité. Contrairement aux arguments, un paramètre fait intégralement partie de la définition d'une entité. Dans l'équivalent informatique de la fonction mathématique :

$$\forall a \in \mathbb{N},\ f_a : \begin{cases} \mathbb{R} \to \mathbb{R} \\ x \to \cos(a\,x) \end{cases}$$

x, variable de type float, est un argument de la fonction f_a, tandis que a, de type uint, est son paramètre formel.
La valeur d'un paramètre formel est appelée *paramètre*. Par exemple, f_7 est une fonction dont le paramètre est 7.

```
image2d : [V : type] -> type is
  class
  {
    - data : array of * V,
    + get : (row, col : int) -> V
    + set self : (row, col : int) -> ref V
    ...
  }
```

Code source 2.3: Classe d'images déjà un peu générique.

Une solution consiste à abstraire le type des valeurs grâce à un paramètre formel (⇝ définition page 25). En notant V le paramètre formel qui représente le type des valeurs, on aboutit à la définition générique donnée par le code 2.3.

La définition de la classe, donnée par les lignes à partir du mot-clef class, est identique à la version non-générique à une unique différence près : uint8 a été remplacé par V. En revanche, le

statut de ce bout a complètement changé. Au lieu de définir une seule classe, image2d_uint8, nous avons ici une fonction, image2d, permettant de disposer de plusieurs classes. Un appel à cette fonction permet de donner une valeur au paramètre formel V, donc de préciser le type de valeurs souhaité. Ainsi, nous pouvons alors re-créer le type d'images "classique" :

```
image2d_uint8 : type is !image2d[uint8]
```

Cette ligne de code déclare un "alias", c'est-à-dire un autre nom, ici image2d_uint8, pour désigner la classe image2d[uint8]. Une autre façon de faire est d'écrire à la volée le paramètre de la classe ; dans la ligne ci-dessous, on définit une instance d'image, ima, dont le type de valeurs est float :

```
ima : !image2d[float]
```

Le type de la variable ima est une classe ayant pour paramètre float : image2d[float]. Dans cette approche générique, image2d n'est *pas* une classe mais une fonction dont l'appel produit, i.e. retourne, une classe.

La définition de image2d permet ainsi d'accéder à toute une *famille de classes*. Cette dernière se caractérise par trois propriétés fortes.

- Les classes partagent exactement le même code, écrit une fois pour toute. En conséquence, la cohérence de comportement de ces classes est assurée naturellement, les risques de boggue sont fortement réduits et, du point de vue, du mainteneur du logiciel, son travail est grandement facilité.
- Seul le type des valeurs de pixels change d'une classe à une autre et l'utilisateur dispose d'un choix ouvert, non limitatif, pour ce type. Ainsi, il peut imaginer manipuler des images contenant des données de toute nature et *a fortiori* d'une toute nouvelle nature.
- Pour le langage, et donc pour le compilateur, chaque classe de cette famille se distingue des autres par son type. Cette différence permet au compilateur de vérifier que le programme de l'utilisateur est correct, qu'il n'a pas confondu par mégarde des types distincts de données.

Enfin, une quatrième propriété est forcée par la présence du symbole ! (par exemple dans "ima : ! image2d[float]") : l'utilisateur demande explicitement au compilateur de définir complètement les types d'images à la compilation. Dit autrement, aucun travail ne sera laissé de côté par le compilateur, ne sera différé à la phase d'exécution. Cela se traduit par la propriété suivante :

- Chaque classe est compilée séparément et, en conséquence, l'utilisation de chaque classe est optimisé au mieux de ses propres caractéristiques. Les exécutables générés peuvent donc être très performants [1].

En corollaire de cette dernière propriété, il faut comprendre que le compilateur génère lui-même la classe image2d_uint8, qui en début de section, code 2.2 page 24, avait été écrite à la main par le programmeur. En introduisant le paramètre V, du travail de programmation est donc transféré de l'humain vers le compilateur. Rappelons que ce dernier est connu pour travailler de façon beaucoup plus performante et plus fiable que nous.

L'emploi de paramètres pour la définition de classes est connu depuis longtemps et disponible dans de très langages dont Ada, Eiffel, C++, Java, .NET, et D. Leur utilisation principale, maintenant très classique, est la définition de structures de données. Les exemples emblématiques sont donc des types paramétrés comme list[E] pour des liste chaînées ou array[E] pour des tableaux, où E est le type

1. Les aspects de compilation, d'optimisation et de performance sont décrits de façon détaillée en section 3.1.

REMARQUE 1: Choix de présentation.

Ici, image2d est une fonction qui s'applique sur un type. Nous considérons donc implicitement que les types sont des valeurs. Cela se traduit dans le mini-langage par la syntaxe :

```
fun_ord_sup : [T1 : type...] -> type
```

proche de celle de fonctions classiques :

```
fun : (v1 : float...) -> float
```

Dans certains langages à classes, une approche différente existe, aboutissant approximativement au même résultat. L'entité image2d est considérée comme un type (au lieu d'une fonction) dont les instances sont des classes ; un tel type s'appelle une *méta-classe*. Avec cette approche alternative, l'écriture en mini-langage serait :

```
image2d_uint8 : image2d := uint8
```

pour initialiser l'objet (la classe image2d_uint8) à partir de son type (la méta-classe image2d). Le lien entre ces deux approches (méta-classe et fonction d'ordre supérieure) peut être fait en réalisant simplement que la construction d'un objet à partir de son type est en soi une fonction.

des éléments stockés dans ces structures. En calcul numérique, l'exemple classique est la classe paramétrée vector[n,T] pour représenter des vecteurs algébriques de dimension n et de coordonnées de type T.

Les paramètres sont aussi utilisées pour définir des fonctions génériques comme, par exemple, la fonction "élévation au carré" :

```
sqr : auto[T : type] (x : T) -> T is return x * x
```

La présence du mot-clef auto s'explique par le fait que la valeur du type T peut être déduite par le compilateur à chaque appel à cette fonction. Le client de cette fonction n'est alors pas obligé de renseigner la valeur du paramètre. En effet, comme T est le type de l'argument x de la fonction, T est le type de la valeur passée en argument lors d'un appel. Ainsi, on a :

```
i : int := 3
j : int := sqr(i) // appelle sqr[int]

f : float := 1.1
g : float := sqr(f) // appelle sqr[float]
```

Du point de vue du client de cette fonctionnalité, de l'appelant, elle est vue comme pouvant accepter des données de types différents (ici, aussi bien int que float). Une telle fonctionnalité est dite *polymorphe*. Plus précisément, il s'agit ici de *polymorphisme paramétrique* car la propriété 'polymorphe' de sqr est obtenue grâce à la présence d'un paramètre.

Revenons au domaine du traitement d'images. Comme nous nous sommes doter d'une famille de classes image2d[V], nous pouvons songer à écrire des algorithmes génériques pour cette famille. Pour reprendre l'exemple de la routine fill non-générique donné en section 2.3.1 (code 2.1 page 21), elle peut maintenant s'écrire :

```
fill : auto[V : type] (ima : ref image2d[V], v : V) -> void
is
  row, col : uint
  for row := 0; row < ima.nrows; row := row + 1
    for col := 0; col < ima.ncols; col := col + 1
      ima.set(row, col) := v
  end
```

Code source 2.4: Version un peu plus générique de fill.

Nous venons d'abstraire le type des valeurs des pixels ; c'est un premier pas vers la généricité. Néanmoins, de façon pragmatique, si la première limitation de la liste donnée page 21 est maintenant levée, il reste encore 7 limitations...

2.3.4 *Généricité classique*

En fait, la mise en œuvre de la généricité se traduit par un "niveau de généricité" plus ou moins élevé. Plus le niveau de généricité augmente, plus il se fait rare dans les logiciels. Le premier niveau de généricité est celui de la section précédente, accessible facilement au programmeur débutant : paramétrer une structure de données par le type de ses données. À l'opposé, les niveaux de généricité plus avancés, loin d'être triviaux pour des informaticiens même expérimentés, seront traités dans le chapitre suivant (chapitre 3).

Cette section a pour sujet le deuxième niveau de généricité, déjà présent dans quelques logiciels et discuté dans une poignée de publications scientifiques. Nous l'appellerons "généricité classique". Il correspond à un objectif assez naturel pour qui traite des structures de données de nature différente : réussir à gérer cette diversité. Concrètement, cela signifie qu'un algorithme connu pour être applicable sur différentes structures devrait être implémenté tout en préservant son aspect versatile.

La seconde limitation forte de l'exemple fil rouge de la routine fill est en effet liée à la dimension de la structure de données : pour l'instant, seules les images bidimensionnelles peuvent être traitées. En traitement d'images, d'autres structures de données existent pour lesquelles la fonction de remplissage devrait être applicable, par exemple les images tridimensionnelles. Cette fonction ne devrait pas se limiter au cas 2D, ni d'ailleurs à un "petit" ensemble de structures, mais rester ouverte à tout nouveau type de structure.

La généricité classique s'explique ainsi :

> **Dissocier les algorithmes, les structures de données et leurs valeurs rend possible l'utilisation de tout triplet (algorithme, structure, valeur).**

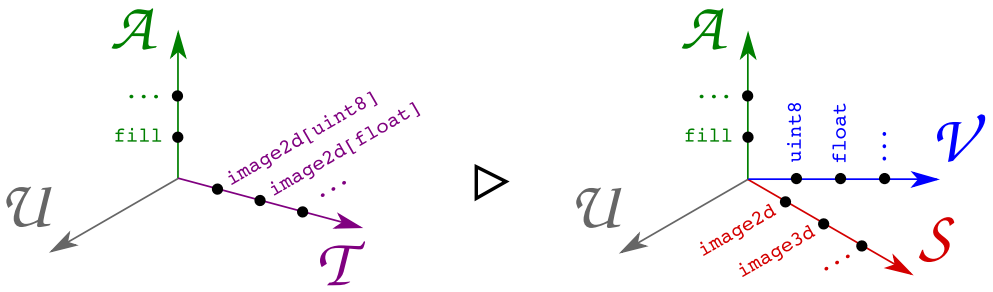


FIGURE 6.: Décomposition de l'axe des *𝒯ypes* en deux axes orthogonaux.

Cette façon de présenter la généricité est la plus courante du côté utilisateur, c'est-à-dire de la part de praticiens d'un domaine. Citons par exemple pour le traitement d'images [24, 6, 9]. Cette présentation de la généricité est illustrée dans la partie droite de la figure 6 ; on y retrouve les trois axes correspondant aux triplets (algorithme, structure, valeur) possibles. Cette figure met également en évidence l'évolution de la généricité de premier niveau, décrite dans la section précédente, à la généricité classique ; de plus, l'axe des *𝒰*tilitaires est conservé, même si, *a priori*, il semble extérieur à la notion de triplets de la généricité classique [2].

On peut considérer que, en traitement d'images, disposer d'un outil logiciel proposant la généricité classique est un objectif minimal. En effet de nombreux types de structures sont utilisées : image 1D (signal), image 2D, image volumique, graphe, complexe cellulaire, et ce, en conjonction avec de nombreux types de valeurs : booléen (pour les images binaires), différents encodages de niveaux de gris et de couleurs, mais aussi des vecteurs, matrices, tenseurs, etc. Les traitements fournis par une bibliothèque devraient alors être compatibles avec tous ces types.

Le but de la généricité classique est de concilier algorithmes et structures de données pour que ceux-ci puissent fonctionner sur la diversité de structures et de valeurs contenues dans ces structures. Il s'agit donc *in fine* d'une propriété de l'implémentation d'algorithmes qui est recherchée. Une citation de l'auteur d'une des rares bibliothèques (vraiment / très) génériques en traitement d'images en témoigne :

> *After all, the point of* VIGRA *is algorithms, not the core ; the core is just there to give the algorithms something to work with.*
>
> —Ullrich Köthe, auteur et mainteneur de VIGRA, correspondance privée, décembre 2006.

Cette focalisation sur les algorithmes s'explique assez facilement. Le but d'un traiteur d'images est avant tout de traiter donc d'exécuter des algorithmes. Avoir un logiciel non limitatif se traduit par considérer que les cas d'utilisation des algorithmes ne sont pas limités.

2. Nous montrons en section 3.1 que l'*a priori* consistant à exclure l'axe des *𝒰*tilitaires de l'explication de la généricité classique est maladroit.

REMARQUE 2: Polymorphisme paramétrique v. polymorphisme ad hoc.

Le code de sqr est unique, écrit une fois pour toute. Pourtant, c'est comme si plusieurs versions de fonctions existaient, chaque version correspondant à un type différent en entrée. Un autre bout de code, à rapprocher de celui de l'exemple, est le suivant :

```
sqr : (x : int) −> int is return x * x
sqr : (x : float) −> float is return x * x
```

Ici, différentes versions de sqr sont écrites ; elles coexistent au sein d'un même programme bien qu'elles portent le même nom (sqr). Lors d'un appel à la fonctionnalité sqr, la version *ad hoc* est appelée, choisie comme par avant, grâce au type de la valeur passée en argument lors de l'appel. On a alors :

```
i : int := 3
j : int := sqr(i) // appelle la première version

f : float := 1.1
g : float := sqr(f) // appelle la seconde version
```

Il s'agit ici de *polymorphisme ad hoc*, appelé aussi *surcharge*. Contrairement au polymorphisme paramétrique, le polymorphisme *ad hoc* n'est **pas générique**. En effet, on ne pourra pas appeler sqr avec une valeur de type double car il n'y a pas de version prévue pour ce "nouveau" type. On se retrouve dans une situation de limitation des cas d'utilisation. En revanche, derrière l'écriture de la version générique :

```
sqr : auto[T : type] (x : T) −> T is return x * x
```

se cache en fait une famille de fonctions automatiquement élargie avec la diversité d'appels. Cette écriture unique est alors rigoureusement équivalente à une surcharge de fonctions que l'on n'a pas besoin de gérer à la main :

```
sqr[int] : (x : int) −> int is return x * x
sqr[float] : (x : float) −> float is return x * x
...
// et dans le cas d'un appel à un 'double' :
sqr[double] : (x : double) −> double is return x * x
```

La différence fondamentale entre ces deux polymorphismes et que l'un permet de définir un ensemble fini de cas possibles (polymorphisme *ad hoc*) tandis que le second laisse ouvert l'ensemble des possibles (polymorphisme paramétrique). Le premier pose donc des limites à son utilisation, ce qui est un avantage lorsque l'on souhaite bien déterminer et contrôler les différents cas d'utilisation, mais un inconvénient lorsque l'on cherche à s'affranchir de telles limites.

Deuxième partie

LE CŒUR

3 PARADIGMES

Petit à petit, j'aurais mon building
Petit à petit, je serais en France
C'est un gratte-ciel, que j'aimerais construire
C'est un gratte-ciel, plus haut que celui de Niamey
Petit à petit, les poissons du fleuve
M'aideront à la construire, c'est le commerce plus facile
Par ce moyen je l'aurais, cet énorme gratte-ciel

Paroles de la chanson du film Petit à Petit*, Jean Rouch, 1971.*

3.1 QUATRE APPROCHES À LA GÉNÉRICITÉ

L'objectif de généricité se traduit par vouloir offrir à l'utilisateur la possibilité d'accéder à une fonctionnalité quelque soit le type de structure et le type de valeurs qu'il manipule. Dit autrement, une bibliothèque doit mettre en œuvre l'espace formé par les axes $\mathcal{A}$lgorithmes, $\mathcal{S}$tructures et $\mathcal{V}$aleurs de la figure 2.3.4. Nous avons identifié quatre approches différentes de la part des concepteurs de logiciel pour atteindre cet objectif ou, tout du moins s'en rapprocher. Ces approches sont décrites dans les sections suivantes.

3.1.1 *Exhaustivité*

Une première approche, assez brutale, consiste tout simplement à gérer tous les cas. C'est une approche "manuelle" où la duplication de code permet de prendre en compte la multiplicité des structures et des valeurs.

```
image2d : type is
  class
  {
   ...
   + set self : (row, col : uint, v : any) -> void
     is
        assign(data@row@col, v) // voir code 3.3
        end
  }

image3d : type is
  class
  {
   ...
   + set self : (sli, row, col : uint, v : any) -> void
     is
        assign(data@sli@row@col, v)
        end
  }
```

Code source 3.1: Version des classes images pour l'approche exhaustive.

```
fill : (ima : ref image2d, v : any) -> void // version 2D
is
   row, col : uint
   for row := 0; row < ima.nrows; row := row + 1
     for col := 0; col < ima.ncols; col := col + 1
       ima.set(row, col) := v
   end

fill : (ima : ref image3d, v : any) -> void // version 3D
is
   sli, row, col : uint
   for sli := 0; sli < ima.nslis; sli := sli + 1
     for row := 0; row < ima.nrows; row := row + 1
       for col := 0; col < ima.ncols; col := col + 1
         ima.set(sli, row, col) := v
   end

... // autres versions
```

Code source 3.2: Version de fill par exhaustivité.

```
assign : (vl : ref any, vr : any) −> void // mime l'affectation :   vl := vr
is
    !precondition typeof(vl) = typeof(vr)
    if typeof(vr) = uint8
      then vl := cast[uint8](vr)
    else if typeof(vr) = float
      then vl := cast[float](vr)
    ... // autres cas...
    end
```

Code source 3.3: Gestion des types de valeurs par exhaustivité.

Ici, nous souhaitons prendre en compte deux structures d'images différentes : les images bidimensionnelles et tridimensionnelles, définies par les classes du code 3.1.

Avec l'approche exhaustive, chaque type différent de structure d'images donne lieu à une version dédiée de la fonction fill ; voir code 3.2. De façon similaire, toujours par duplication de code, la multiplicité des types de valeurs peut être gérée par une liste de cas comme le montre le code 3.3.

Ce couple d'exemples montre les deux mécanismes utilisés en conjonction avec la duplication de code, respectivement la surcharge et une structure de contrôle disjonctive. Avec la surcharge, plusieurs versions de fill cohabitent, se distinguant par leurs entrées de types différents. Le choix de la version de code appropriée est alors automatique ; ce choix est opéré par le compilateur en fonction de la nature de l'entrée. Avec une structure de contrôle, c'est le programmeur qui doit écrire l'appariement entre la nature des données à traiter et la bonne version de code à exécuter. À noter : l'usage de if...then...else est complètement équivalent à d'autres constructions, que ce soit d'autres structures de contrôle comme switch...case, ou des mécanismes à base de tableaux de pointeurs et d'étiquettes-indices.

Lorsque le langage de programmation ne supporte pas la surcharge, ce qui est le cas du langage C par exemple, le développeur est contraint de se rabattre sur le second outil :

```
fill2d : (ima : ref image2d, v : any) −> void ...
fill3d : (ima : ref image3d, v : any) −> void ...
... // autres versions

fill : (ima : ref any, v : any) −> void is
  if typeof(ima) = image2d
    then fill2d(cast[image2d](ima), v)
  else if typeof(ima) = image3d
    then fill3d(cast[image3d](ima), v)
  ... // autres cas...
  end
```

Ici, la routine fill est une *façade* permettant à l'utilisateur de ne pas préciser quelle version doit être appelée. La présence de cette facade est saine ; en son absence, l'utilisateur écrirait des appels explicites sujets à des erreurs d'appels, comme :

```
fill3d(ima2d, v)
```

ou écrirait des appels avec redondance d'information, comme :

REMARQUE 3: Pire cas de l'exhaustivité.

Nous ne présentons pas ici le pire cas de l'approche exhaustive, pourtant utilisée par quelques jusqu'au-boutistes. Ce pire cas consiste à dupliquer le code des algorithmes non seulement pour les structures différentes mais aussi pour les différents types de valeurs. Cela donne :

```
// pour les images 2D :
fill : (ima : ref image2d_uint8, v : uint8) −> void is ...
fill : (ima : ref image2d_float, v : float) −> void is ...
... // autres types de valeur

// pour les images 2D :
fill : (ima : ref image3d_uint8, v : uint8) −> void is ...
fill : (ima : ref image3d_float, v : float) −> void is ...
... // autres types de valeur

... // autres types d'images
```

La combinatoire pour l'écriture d'un algorithme est donc d'une version par type de données différent : avec S types de structures et V types de valeurs, cette combinatoire est donc de $S \times V$ versions ! Beaucoup d'auteurs de bibliothèque ont compris que c'était beaucoup et propose maintenant une approche exhaustive "simplifiée", moins lourde, présentée dans cette section. En éliminant le facteur "valeur" de la combinatoire, cette dernière tombe à S versions d'un même algorithme. Cette réduction est drastique car il y a beaucoup plus de types de valeurs que de types de structures. Enfin, notons qu'il n'est pas toujours simple de faire abstraction du type de valeurs. Cette constatation transparaît à travers les codes 3.1 et 3.3 : pour réussir à écrire une méthode set simple, il a fallu déléguer l'affectation à une routine assign complexe. Finalement, on a caché la combinatoire sous le tapis mais elle est toujours bien présente. Lors de l'écriture d'un algorithme de traitement d'images, les valeurs des pixels interviennent inévitablement. Pour réussir à cacher la multiplicité des types de valeurs, pour qu'elle ne pollue pas l'écriture d'algorithmes, il faut donc se doter de nombreux outils annexes, comme assign. Ces outils font partie de l'axe $\mathcal{U}$.

```
fill2d(ima2d, v)
```

Grâce au mécanisme de surcharge ou, à défaut, à celui de façade, le client de l'algorithme fill n'est finalement pas obligé de voir la duplication de code. Il appelle une fonctionnalité et le programme embranche sur la version *ad hoc*. Il n'est donc pas *a priori* gêné par la multiplicité des versions existantes ; la complexité du logiciel lui est donc cachée.

3.1.2 *Généralisation*

Une deuxième approche consiste à généraliser les types. Au lieu de multiplier les types d'images et de valeurs, il s'agit de les réifier en un type général. On aura donc un unique type d'images et un unique type de valeurs, chaque type devant embrasser la diversité des données qu'il représente.

```
image : type is
  class
  {
    - data : array of * double,
    + get : (sli, row, col : int) -> double ...
    + set self : (sli, row, col : int) -> ref double ...
    ...
  }
```

Code source 3.4: Version de l'unique classe d'images par généralisation.

Les choix les plus représentés dans l'existant sont de considérer que *le* type d'images correspond à une image de dimension 3 et que *le* type de valeurs est un flottant à double précision. Ces choix se traduisent par un code similaire à celui donné en 3.4.

```
fill : (ima : ref image, v : double) -> void
is
  sli, row, col : uint
  for sli := 0; sli < ima.nslis; sli := sli + 1
    for row := 0; row < ima.nrows; row := row + 1
      for col := 0; col < ima.ncols; col := col + 1
        ima.set(sli, row, col) := v
  end
```

Code source 3.5: Version de fill par généralisation.

Si l'utilisateur veut manipuler une image bidimensionnelle, il peut construire une image dont le nombre de coupes (ima.nslis) est réduit à 1 ; s'il veut une gamme discrète de niveaux de gris, comme avec des entiers non signés sur 8 bit, il peut car le type double est plus général. En conséquence, la fonction fill possède une unique implémentation, donnée par le code source 3.5, ce qui contraste fortement avec l'approche exhaustive de la section précédente (code 3.2 page 34). Finalement, l'exhaustivité est réduite à l'unique cas général.

3.1.3 *Inclusion*

Une troisième approche est liée au modèle à classes avec héritage, appelé modèle *orienté-objet* par anglicisme. Dans ce modèle, la relation de généralisation existe explicitement entre classes. Cette notion de généralisation est souvent effacée dans la littérature au profit du concept qu'elle met en œuvre, l'héritage, entre une classe particulière et une classe plus générale.

```
image : type is // classe abstraite
  class
  {
    + get : (p : point) -> value is abstract
    + set self : (p : point) -> ref value is abstract
    + create_iterator : (void) -> iterator is abstract
  }

image2d : type is // classe concrète, particulière
  class inherits image
  {
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    + create_iterator : (void) -> iterator2d is ...
    ...
  }
```

Code source 3.6: Classe abstraite d'images et une classe particulière.

La relation de généralisation traduit en programme la relation "est-un" ; par exemple, une image 2D *est une* image. La classe image2d est donc un type particulier du type plus général, la classe abstraite image ; le code 3.6 illustre cette relation.

Dans ces définitions, deux autres classes abstraites apparaissent, point et value, qui généralisent respectivement les différents types de points et de valeurs rencontrés dans le domaine. Par exemple, pour les classes de points :

```
point : type is class { ...
point2d : type is class inherits point { ...
point3d : type is class inherits point { ...
```

Dans les langages à objets et héritage de classes, la relation de généralisation conduit à une classification des types du domaine. Cette classification est véritablement ensembliste : une classe décrit l'ensemble de ses instances possibles, ses éléments donc, et la généralisation correspond à l'union ensembliste. On a :

$$image = image_{2d} \cup image_{3d} \cup ...$$

Le partitionnement de classes (ensembles) en sous-classes (sous-ensembles) donne une hiérarchie de classes et une classe particulière, par exemple image2d, est véritablement *incluse* dans sa sur-classe (sur-ensemble), ici image.

REMARQUE 4: À propos d'inclusion, d'abstraction et de typage.

L'inclusion ensembliste lors d'une classification, c'est-à-dire d'une mise en classes des entités / notions / concepts d'un domaine, s'interprète également en terme de typage. Une classe étant un type, la relation d'inclusion correspond également à une relation de sous-typage (notée $<:$). Mathématiquement, les types forment un treillis, construit sur cette relation. Il y a équivalence entre la vision ensembliste :

$$image2d \subset image$$

et le typage :

$$\mathsf{image2d} <: \mathsf{image}.$$

De l'inclusion se déduit une propriété forte : lorsqu'une instance de classe est attendue, toute instance d'une sous-classe convient. Ce n'est pas surprenant car :

$$ima \in image2d \Rightarrow ima \in image$$

Utiliser une classe générale, comme image, permet de ne pas parler d'une classe en particulier (une classe particulière, une sous-classe) et, plus généralement, de ne parler d'aucune de ses sous-classes. La relation de généralisation permet donc d'*abstraire*.

Néanmoins, pour pouvoir manipuler des abstractions sans que des problèmes ne surviennent, il faut garantir qu'il y ait sous-typage strict dès qu'il y a inclusion. Malheureusement, sous-classer n'implique pas formellement sous-typer...

Les sous-classes concrètes de la classe abstraite image doivent fournir une implémentation pour les méthodes déclarées abstraites. Le code 3.7 montre une partie de la définition d'une classe concrète représentant les images bidimensionnelles classiques. La méthode set est définie pour toutes les sous-classes concrètes (instanciables) de la classe abstraite image. La fonctionnalité set est disponible pour tous les types différents d'images ; elle est donc polymorphe. Le support de cette fonctionnalité est l'inclusion inhérente à une hiérarchie de classes ; il s'agit de *polymorphisme d'inclusion*.

```
image2d : type is
  class inherits image
  {
    - data : array of * value, // valeurs des pixels
    - nrows, ncols : uint // taille de l'image

    + set self : (p : point) -> ref value
      is
          precondition data not = 0 // les données sont allouées
          p_ : point2d := cast[point2d](p) // p_ prend son type précis
          precondition p_.row < nrows and p_.col < ncols // p_ est OK
          return data@p_.row@p_.col
          end

    + create_iterator : (void) -> iterator2d
      is
          return iterator2d.make(nrows, ncols)
    ...
  }
```

Code source 3.7: Détail d'une classe concrète d'images par inclusion.

La classe abstraite image convient pour désigner le type de n'importe quelle image, aussi bien bidimensionnelle que tridimensionnelle ; elle est même utilisable pour des images dont le type est encore inconnu parce qu'il ne sera défini qu'ultérieurement. À l'aide de ce type abstrait, il est donc possible de donner un code unique, bien concret, à la fonction fill. Ce code ressemble à :

```
fill : (ima : ref image, v : value) -> void
is
  p : point ref := ...
  ... // boucle "pour tous les points p du domaine de ima faire"
      ima.set(p) := v
  end
```

Pour parcourir tous les points du domaine de définition d'une image, quelque soit le type de cette dernière, un objet maintenant classique est utilisé : un *itérateur*. Un itérateur, sous sa forme abstraite, reproduit une simple boucle avec un jeu d'opérations très réduit : start pour se positionner sur le premier point, next pour passer à l'élément suivant, is_valid pour tester s'il est encore valide d'itérer (si l'itérateur n'a pas dépassé le dernier élément), et get_point pour récupérer le point courant. L'interface de la classe abstraite iterator est donc très simple :

```
iterator : type is
```

```
class
{
  + start ref : () −> void is abstract
  + is_valid : () −> bool is abstract
  + next ref : () −> void is abstract
  + get_point : () −> point is abstract
}
```

Un itérateur concret, dédié à un type d'images en particulier, est facile à mettre en œuvre ; il reproduit le parcours des points de ce type d'images, ici une double-boucle :

```
iterator2d : type is
  class inherits iterator
  {
    − p : point2d
    − nrows, ncols : int
    + make : (nr, nc : int) −> self is nrows := nr; ncols := nc; end
    + start ref : () −> void is p.row := 0; p.col := 0; end
    + is_valid : () −> bool is return p.row < nrows
    + next ref : () −> void is
        p.col := p.col + 1
        if p.col = ncols then p.col := 0; p.row := p.row + 1; end
        end
    + get_point : () −> point2d ref is return p
    ...
  }
```

Les classes abstraites image, point et iterator permettent enfin d'écrire un traitement sans préciser les types concrets, effectifs, sur lesquels il se réalisera. Ce code est donné en 3.8.

```
fill : (ima : ref image, v : value) −> void
is
  i : iterator := ima.create_iterator()
  p : point ref is i.get_point() // p est une référence au point désigné par i
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

Code source 3.8: Version de fill par inclusion.

Cette fonction peut être appelée sur une image bidimensionnelle, ou tridimensionnelle, ou autre ; dans ce cas, ce sont les méthodes des classes idoines qui seront appelées à l'exécution. Ainsi, si ima est de type image2d, alors i sera un iterator2d et p un point2d. La boucle utilisera les méthodes définies par l'itérateur bidimensionnel et l'instruction ima.set(p) := v modifiera la valeur du pixel de l'image 2D au point courant.

Le polymorphisme d'inclusion supporté par des hiérarchies de classes (ici, liées aux notions d'images, de points et d'itérateurs) permet d'abstraire le comportement des fonctions.

3.1.4 *Paramétrisation*

Une dernière approche consistant à atteindre la généricité classique est la *programmation générique* via l'utilisation intensive de paramètres. Dans ce contexte, l'héritage de classe n'apparaît plus car il devient inutile. Jusqu'à une certaine limite, on peut considérer que le polymorphisme d'inclusion et le polymorphisme paramétrique sont deux formes duales.

Nous avons vu qu'à une classe d'images étaient associé plusieurs types : son type de valeurs, son type de points et son type d'itérateurs. Pour la classe image2d de l'"avant-propos" (code 2.3, page 25), paramétrée par son type V de valeurs, le type de valeurs est connu grâce à ce paramètre. Pour la classe image2d telle que définie dans la section précédente, les types de points et d'itérateurs sont connus explicitement ; ce sont respectivement les classes point2d et iterator2d.

```
image2d : [V : type] -> type is
  class
  {
    + value : type is V
    + point : type is point2d
    + iterator : type is iterator2d

    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

Code source 3.9: Version d'une image 2D avec paramétrisation.

Au sein de la définition d'une classe d'images bidimensionnelles, nous pouvons lister ses types associés. Ils sont alors utilisables en interne, pour typer les méthodes de cette classe, comme en externe, grâce à la syntaxe :

<classe> : :<type_associé>

Ainsi, nous pouvons reprendre la définition générique de la classe d'images 2D (paramétrée par son type de valeurs) mais avec des méthodes similaires à celles de la définition de la section précédente. La nouvelle définition est donnée par le code 3.9.
Ici, value, point et iterator ne sont pas des abstractions. Ils sont parfaitement définis. Ce ne sont que des alias, des noms généraux, qui désignent des types connus. Néanmoins, d'un point de vue extérieur à cette classe, ils peuvent *ressembler* à des abstractions. En effet, si I est un type d'images, mais n'importe lequel (on ne sait pas lequel), alors il est possible de désigner le type de valeur qui lui est associé :

I : :value

On peut ainsi manipuler des types que l'on ne connaît pas explicitement.

Cela conduit à une version générique de la fonctionnalité fill. Son image en entrée (premier argument) peut avoir n'importe quel type, ce qui se traduit par un paramètre formel I pour noter

```
fill : auto[I : type] (ima : ref I, v : I::value) −> void
is
  i : I::iterator := ima // initialisation de i.nrows et i.ncols avec les valeurs d'ima
  p : I::point ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

Code source 3.10: Version de fill par paramétrisation.

le type de cette entrée. La valeur en entrée (second argument), v, a pour type le type des valeurs de l'image en entrée ; il s'agit donc de I : :value. Le code de fill défini par inclusion, code 3.8 page pagereflst.paradigmes.inclusion.fill, est légèrement modifié dans sa version paramétrée. Il est donné par le code source 3.10. La déduction de types associés, nommément I : :value, I : :iterator et I : :point, remplace l'utilisation des classes abstraites (respectivement value, iterator et point) de la version orientée-objet classique. Dans ces deux approches, il y a *abstraction* mais via des moyens différents : la classe abstraite image devient ici un paramètre formel, abstrait, I.

Un appel à fill tel :

```
ima : image2d[uint8]
fill(ima, 51)
```

se traduit par l'exécution de la version de la fonction compilée avec pour paramètre le type image2d[uint8]. Cette version particulière, s'il fallait l'écrire, serait exactement :

```
fill[ image2d[uint8] ] : (ima : ref image2d[uint8], v : uint8) −> void
is
  i : iterator2d := ima
  p : point2d ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

Dans cette version, comme tous les types sont explicitement connus, le compilateur peut remplacer tous les appels de méthodes par leur définition. C'est donc le programme suivant qui est traduit en code machine (et optimisé) par le compilateur :

```
fill[ image2d[uint8] ] : (ima : ref image2d[uint8], v : uint8) −> void
is
  row, col : int := 0 // start
  while row < ima.nrows // is_valid
    ima.data@i.row@i.col := v // set
    col := col + 1; if col = ima.ncols then col := 0; row := row + 1; end // next
    end
  end
```

Finalement, le compilateur génère une version de l'algorithme fill dédiée au type particulier d'images bidimensionnelles à valeurs en uint8. Cette version est rigoureusement équivalente à la version non-générique critiquée en section 2.3.1 (code 2.1 page 21). La différence est que ce n'est plus le programmeur qui écrit cette version mais le compilateur qui s'en charge.

REMARQUE 5: Itérateur staticisé et autres “design patterns”.

Avec le passage de l’orienté-objet classique (par inclusion) à la programmation générique (par paramétrisation), nous passons d’un paradigme dynamique, avec des résolutions de types à l’exécution, à un paradigme statique, où les résolutions sont réalisées à la compilation. La dualité entre l’orienté-objet classique et la généricité a été analysée dans [29] sous le jour du typage statique et donc de l’obtention de performances.
Un “itérateur” est un exemple de modèle de conception (*design pattern*), c’est-à-dire un élément de conception réutilisable, un bout de modélisation très général, que l’on retrouve dans de nombreux programmes. Les modèles de conception, formalisés et popularisés par [12], sont traditionnellement décrits dans le contexte de l’orienté-objet classique.
Partant de l’idée qu’un itérateur pouvant être staticisé, nous avons proposé d’étendre cette traduction au monde statique à plusieurs autres modèles de conception. Cela s’est traduit par deux publications : [16] et [10]. Cette dernière est portée en annexe B.1 de ce rapport.

3.2 DISCUSSION DES APPROCHES

TABLE 2.: Résumé des 4 approches vers la généricité.

approche	*description courte*	code
exhaustivité	copié-collé-modifié à la main	3.1 et 3.2
généralisation	un type d’image pour les gérer tous	3.4 et 3.5
inclusion	hiérarchies de classes	3.6 et 3.8
paramétrisation	paramètres et types associés	3.9 et 3.10

Dans cette section, nous adoptons un point de vue critique envers les quatre approches vers la généricité présentées dans la section précédente. Ces approches sont résumées en une description courte dans la table 2.

3.2.1 *Comparaison entre exhaustivité et généralisation*

L’approche exhaustive est très laborieuse puisque le programmeur multiplie les opérations de “copié-collé-modifié”. Ces opérations sont sources d’erreur et le risque d’incohérence entre les versions est élevé. Toute modification doit être reportée sur tous les duplicata ; cela, ajouté à la plétore de code, aboutit a une lourdeur de maintenance. Cette approche est à l’opposée des règles de génie logiciel. Tout d’abord, la factorisation de code est quasi nulle. Ensuite, l’ajout d’une nouvelle

entité, un type de structure ou de valeurs, conduit à être intrusif dans le code existant (une bonne pratique est que toute addition de fonctionnalité n'affecte pas l'existant pour ne pas risquer le casser). Pire, non seulement cette intrusion nécessite d'écrire beaucoup de code mais elle n'est pas localisée. En effet, elle touche de nombreuses parties de code déjà existant : à chaque nouvelle structure, par exemple, il faut ajouter une version par algorithme. L'approche exhaustive ne réussit donc pas à traduire convenablement en logiciel la dissociation des axes $\mathcal{A}$lgorithmes / $\mathcal{S}$tructures / $\mathcal{V}$aleurs (figure 6). La pénibilité de devoir gérer tous les cas à la main décourage généralement les programmeurs, ce qui les entraîne à ne prendre en compte que les cas les plus importants ou les plus utiles statistiquement. Pour cette raison, dans les logiciels qui adoptent cette approche, l'exhaustivité n'y est que très partielle. Aussi, en pratique, ces logiciels sont peu génériques.

L'approche par généralisation est dérangeante car l'unique type d'images fixe l'ensemble des cas d'utilisation possibles de l'outil. Les limites de ce dernier sont ainsi pré-déterminées. L'utilisateur n'a donc pas le choix ; si ces limites ne lui conviennent pas, soit il doit se résigner, soit il doit repousser ces limites. Cette dernière option implique de retoucher la classe d'images puis de modifier tous les algorithmes pour prendre en compte ses modifications. En terme de génie logiciel, cette tâche, extrêmement coûteuse, présente les mêmes défauts que dans le cas de l'approche exhaustive. Finalement, la dissociation $\mathcal{A}$lgorithmes / $\mathcal{S}$tructures s'avère encore être un échec.

L'approche par généralisation induit également un nouveau défaut : le fait de vouloir gérer le cas général entraîne souvent une pollution du code. Notons que ce défaut est déjà présent puisqu'une troisième boucle est présente (pour toutes les coupes) même si l'image à traiter n'est que bidimensionnelle. Néanmoins, ce défaut présente des inconvénients plus importants. Prenons pour exemple le concepteur qui a pensé à intégrer un masque binaire dans son type d'image, afin de pouvoir éventuellement appliquer les traitements de façon partielle, c'est-à-dire seulement à une région de l'image. Un algorithme ressemblera alors à ça :

```
fill : (ima : ref image, v : double) -> void
is
  sli, row, col : uint
  for sli := 0; sli < ima.nslis; sli := sli + 1
    for row := 0; row < ima.nrows; row := row + 1
      for col := 0; col < ima.ncols; col := col + 1
        if ima.msk not = 0 and ima.msk.get(sli, row, col) = true
        then
          ima.set(sli, row, col) := v
  end
```

Non seulement, le code est pollué par le test mais encore le traitement se trouve pénalisé en temps d'exécution, même si l'image n'a pas de masque défini. Pallier ce problème induit revient à écrire :

```
fill : (ima : ref image, v : double) -> void
is
  sli, row, col : uint
  if ima.msk = 0
  then
    // l'image n'a pas de masque défini
    for sli := 0; sli < ima.nslis; sli := sli + 1
      for row := 0; row < ima.nrows; row := row + 1
```

```
        for col := 0; col < ima.ncols; col := col + 1
          ima.set(sli, row, col) := v
else
  // prise en compte du masque
  for sli := 0; sli < ima.nslis; sli := sli + 1
    for row := 0; row < ima.nrows; row := row + 1
      for col := 0; col < ima.ncols; col := col + 1
        if ima.msk.get(sli, row, col) = true
        then
          ima.set(sli, row, col) := v
end
```

et, finalement, l'approche par généralisation se met à ressembler à l'approche exhaustive.

La généralisation conduit donc à une double limitation : la limite du cas considéré comme général, tout cas particulier (en dehors du cas général) n'étant pas géré, et celle du paradigme même, puisque on est conduit à casser la généralité en spécialisant (par le test if ima.msk = 0 dans l'exemple ci-dessus). Néanmoins, on peut s'attendre globalement à moins de code produit avec l'approche par généralisation qu'avec l'approche exhaustive.

Conceptuellement, généraliser revient à nier tout particularisme et à tout confondre. Cela aboutit naturellement à de nombreuses erreurs perverses, difficilement identifiables par l'utilisateur. Par exemple, si toutes les valeurs sont encodées par le type double, l'outil permet l'addition d'une image binaire avec une image à niveaux de gris. En effet, on peut réaliser l'opération "true + gris_moyen" sans que cette monstruosité théorique soit signalée comme erreur à l'utilisateur. Toute écriture aberrante devient possible. En généralisant, les informations fondamentales qui font la différence entre des types distincts, des entités mathématiques différentes, sont effacées. Avec la perte de typage fort, le compilateur ne peut pas assister le programmeur en contrôlant la justesse de son code.

À l'inverse, particulariser à l'extrême, comme c'est le cas avec l'approche exhaustive, représente un gaspillage d'énergie dû à une trop grande dispersion que nécessaire. Notons d'ailleurs que l'unique cas de l'approche générale, pour l'approche exhaustive, est un cas en particulier, un cas particulier donc.

3.2.2 *Apport d'abstraction*

Avec les deux dernières approches, par inclusion et par paramétrisation, la notion d'abstraction est introduite. Cette notion est absente des deux premières approches : pour l'exhaustivité, tous les cas particuliers sont gérés et, pour la généralisation, le seul cas général est traité. Ce sont à chaque fois des cas concrets, en ce sens qu'on est capable, très précisément, de décrire les types d'images concernés.

Les abstractions revêtent deux formes différentes : avec l'inclusion, elles se traduisent en classes abstraites, tandis qu'avec la paramétrisation, elles prennent la forme de paramètres et de types associés. A cette différence près, l'écriture de la fonction fill est quasi-identique avec ces deux approches, comme le montre les codes (a) et (b) de la figure 7. Les classes abstraites utilisées y sont indiquées en italique gris (pour l'inclusion, à gauche) ; le paramètre formel, I, est en rouge et les types associés en orange (pour la paramétrisation, à droite). Pour les classes concrètes d'images

```
fill : (ima : ref image, v : value) −> void
is
  i : iterator := ima.create_iterator()
  p : point ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

(a) fill par inclusion

```
fill : auto[I : type] (ima : ref I, v : I::value) −> void
is
  i : I::iterator := ima
  p : I::point ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

(b) fill par paramétrisation

```
image : type is
  class
  {
    + get : (p : point) −> value is abstract
    + set self : (p : point) −> ref value is abstract
    ...
  }
image2d : type is
  class inherits image
  {
    + get : (p : point) −> value is ...
    + set self : (p : point) −> ref value is ...
    ...
  }
```

(c) image2d par inclusion

```
image2d : [V : type] −> type is
  class
  {
    + value : type is V
    + point : type is point2d
    ...
    + get : (p : point) −> value is ...
    + set self : (p : point) −> ref value is ...
    ...
  }
```

(d) image2d par paramétrisation

FIGURE 7.: Abstractions via inclusion (à gauche) et paramétrisation (à droite).

bidimensionnelles, il en va de même. Les deux approches, montrées codes (c) et (d) de la figure 7, donnent une interface identique :

```
    + get : (p : point) −> value is ...
    + set self : (p : point) −> ref value is ...
```

la seule différence provenant, là-encore, de la nature des abstractions point et value.

Une remarque importante mérite d'être notée. Si dans ces deux approches l'écriture de la fonction est abstraite, c'est grâce à l'introduction d'un *itérateur*. Un tel objet réussit à capturer le parcours des éléments d'un ensemble sous une forme unifiée : quelle que soit la dimension de l'image, ce parcours revient à une unique boucle, dont l'écriture est toujours for i.start() ; i.is_valid() ; i.next(). Cela contraste par rapport aux deux premières approches où les boucles étaient explicitement dédiées à la dimension de l'image en entrée ; précisément, une double ou triple boucle suivant le cas traité dans l'approche exhaustive, une triple boucle dans l'approche par généralisation. De façon pragmatique, nous pouvons affirmer que l'abstraction algorithmique est due dans cet exemple à l'outil utilitaire 'itérateur'. Le découplage entre les types données et les algorithmes peut être effectif, bien réel, puisque c'est un tiers, un utilitaire, qui réalise *a posteriori* un pont entre types données et algorithmes. Au final, les axes $\mathcal{T}$ et $\mathcal{A}$ peuvent être indépendants (Cf. figure 6) car l'axe $\mathcal{U}$ peut fournir le *liant* et garantir la cohérence de l'ensemble.

Vus sous ce jour, les utilitaires sont des entités essentielles pour la capacité à abstraire les algorithmes et, par conséquence, pour l'obtention de généricité. Ils doivent donc être disponibles au sein la bibliothèque, fournis avec elle (au sens de la catégorie "fournisseurs", expliquée en section 2.2.1).

3.2.3 *Comparaison entre inclusion et paramétrisation*

Si l'écriture de la fonction fill est très similaire entre la version par inclusion et celle par paramétrisation, il y a en revanche deux différences fondamentales.

Avec l'approche orientée-objet, les abstractions sont des classes donc des *types nommés*. Ainsi, les deux arguments de la fonction fill, ima et v, peuvent être explicitement typés ; en l'occurrence, leurs types respectifs sont ***image*** et ***value***. Le typage des arguments de fill est donc *fort*, précis, contraint : cette routine n'accepte que des images en premier argument et que des valeurs en second argument. Avec l'approche par paramétrisation, aucune contrainte de la sorte n'est imposée puisque le type du premier argument, ima, est *libre* ; précisément, il est désigné par un paramètre formel, I, qui ne contraint pas ce type à être une image. La seule contrainte qui transparaît est que l'expression "I : :value" doit être un type valide, donc que I doit contenir un type associé nommé value. Cela n'en fait toujours pas une image pour autant. Du point de vue du typage, il est *explicite* avec l'approche par inclusion. Cela conduit dans ce cas à de nombreux avantages dont des messages clairs de la part du compilateur en cas d'erreur de l'utilisateur, une documentation plus lisible, et la possibilité de surcharger cette fonction (ce dernier point est détaillé dans la section 3.3). L'absence de ces avantages pour l'approche par paramétrisation sont des inconvénients notables de l'approche par paramétrisation.

La seconde différence fondamentale entre ces deux approches réside dans leurs performances respectives. Avec l'approche par inclusion, une même opération se traduit par plusieurs implémentations différentes au sein d'une hiérarchie de classes ; c'est une méthode polymorphe. Par exemple, l'opération next sur la hiérarchie d'itérateurs (déclarée dans la classe abstraite iterator) est implémentée dans la classe iterator2d pour la gestion des images bidimensionnelles ; elle possède une implémentation différente dans la iterator3d, pour les images tridimensionnelles cette fois. L'appel à une méthode polymorphe a un coût unitaire extrêmement faible ; c'est la somme du coût d'un appel de fonction (avec formation d'une pile pour le passage d'arguments et le retour d'une valeur) et d'une indirection. Cette dernière correspond à la résolution de l'appel, qui dépend du type concret de l'objet. Dans le cas de la méthode next, cette résolution est illustrée à droite ci-dessous :

```
fill : (ima : ref image, v : value) −> void
is
  i : iterator := ima.create_iterator()
  p : point ref is i.get_point()
  i.start()
  while i.is_valid()
    ima.set(p) := v
    i.next() − − − − − − − − − − − − >
    end
  end
```

```
// pseudo-code d'un appel polymorphe

i.iterator::next()
is
  if typeof(i) = iterator2d
    then i.iterator2d::next()
  else if typeof(i) = iterator3d
    then i.iterator3d::next()
  else ...
end
```

Pour l'algorithme fill, les appels à une méthode polymorphe sont indiqués en cyan ci-dessus à gauche. Malheureusement, ils grèvent les performances de l'algorithme. En effet, à chaque itération trois appels sont réalisés et, comme une image comporte de nombreux pixels, le nombre total de ces appels est important et le coût total lié à ces appels devient conséquent. Pour cet exemple précis, la version avec inclusion est quatre fois moins rapide que la version par paramétrisation.

Au final, les abstractions classiques liés à l'orienté-objet, avec la résolution des méthodes polymorphes à l'exécution, induisent une dégradation notable des performances. Ce paradigme n'est pas adapté pour le calcul scientifique intensif performant.

3.2.4 *Hybridation*

Dans la section 3.1, nous avons présenté quatre approches de la généricité en les dissociant complètement. Dans la pratique, on observe assez souvent un mélange de ces approches, ce qui aboutit à des solutions multi-paradigmes, hybrides. L'illustration la plus emblématique car la plus courante en pratique est que l'axe des $\mathcal{V}$aleurs est maintenant systématiquement géré par la paramétrisation. Il s'agit donc de généricité de premier niveau, classique en programmation, décrite dans la section 2.3.3 : une structure de données est paramétrée par le type de ces éléments. Dans les bibliothèques récentes, les différents types d'images, comme image2d, sont paramétrés par le type des valeurs de leurs pixels. L'hybridation consiste alors à choisir une approche pour la gestion de la multiplicité des types de structures.

Une hybridation avec l'approche exhaustive se trouve dans PANDORE [1] :

```
fill2d : auto[V : type] (ima : ref image2d[V], v : V) -> void ...
fill3d : auto[V : type] (ima : ref image3d[V], v : V) -> void ...
... // autres versions
```

Elle a bien sûr tous les défauts de l'approche exhaustive mais le paramétrage a permis de réduire l'exhaustivité vis-à-vis des types de valeurs.

Une hybridation avec l'approche par généralisation est présente dans ITK [2] :

```
image : [V : type] -> type is
  class
  {
    - data : array of * V,
    + get : (sli, row, col : int) -> V ...
    + set self : (sli, row, col : int) -> ref V ...
    ...
  }
```

L'axe des types de valeurs est bien géré mais les défauts de la généralisation ne sont pas gommés pour autant : on ne sait pas faire la différence entre (on confond) les images 2D et 3D.

Une hybridation avec l'approche par inclusion pourrait s'écrire comme ci-dessous (à notre connaissance, il n'y a pas d'exemple de cette hybridation, tiré d'une bibliothèque) :

```
image : [V : type] -> type is
  class {
    + get : (p : point) -> V is abstract
    + set self : (p : point) -> ref V is abstract
    ...
  }
```

1. Site de PANDORE : http://www.greyc.ensicaen.fr/~regis/Pandore/.
2. Site d'ITK : http://www.itk.org/.

```
image2d : [V : type] -> type is
  class inherits image[V] {
    ...
  }
```

Du point de vue des propriétés de cette hybridation, elle a peu d'avantage : le type de valeurs est connu de l'abstraction Image mais... il n'existe plus d'*unique* abstraction puisque Image est devenue une classe paramétrique. De plus, les méthodes restent polymorphes au sens de l'inclusion donc elles grèvent toujours les performances à l'exécution. En revanche, cette écriture est intéressante car elle soulève une question : pourquoi seul le type de valeurs est remonté dans la classe de base ? En effet, du point de vue des images concrètes, d'autres types varient et pourraient être remontés de cette même façon. Par exemple, le type de points :

```
image : [V, P : type] -> type is
  class {
    + get : (p : P) -> V is abstract
    + set self : (p : P) -> ref V is abstract
    ...
  }
```

et, en poursuivant ce raisonnement, on pourrait remonter au niveau de l'abstraction quasiment toutes les parties variables des classes concrètes. C'est le point de départ de ce que nous allons voir dans les sections suivantes...

3.3 PARADIGMES STATIQUES

Nous avons expérimentés plusieurs paradigmes statiques. Par statique, il faut entendre que les types sont toujours parfaitement connus à la compilation ce qui permet au compilateur de générer des exécutables performants. Le premier, discuté en section 3.3.1, est connue aujourd'hui comme étant la *programmation générique* ; sous sa forme la plus classique, il s'agit de l'approche par paramétrisation, décrite en section 3.1.4. Pour pallier ses limites, nous avons proposé un paradigme, donné en section 3.3.2, qui allie les bénéfices de l'orienté-objet classique et de la programmation générique. Nous en proposons une simplification en section 3.3.3.

3.3.1 *Limites de la programmation générique "classique"*

L'approche par paramétrisation, décrite en section 3.1.4, est la plus représentée dans les bibliothèques les plus récentes. Le premier exemple concret et à large échelle de cette approche est la STL, pour *Standard Template Library*. Cette bibliothèque, conçue par Stepanov, d'abord en Ada puis finalisée en C++, propose des conteneurs classiques (tableau dynamique, liste chaînée, ensemble, file, etc.) et une collection d'algorithmes classiques applicables à ces structures (tri, recherche, remplissage, etc.) Chaque algorithme devait être à la fois efficace et générique : avoir un code performant, unique et applicable sur les différentes structures. Ce logiciel, normalisé comme bibliothèque du langage C++, a ouvert à la voie à de nombreux autres projets logiciels ; citons comme exemples :

- *Computational Geometric Algorithms Library* (CGAL)
  http://www.cgal.org/
- *Matrix Template Library* (MTL)
  http://osl.iu.edu/research/mtl/
- *Boost Graph Library* (BGL)
  http://www.boost.org/libs/graph/
- ...

et, pour le domaine du traitement d'images :

- *VIGRA Computer Vision Library*
  http://hci.iwr.uni-heidelberg.de/vigra/
- *Insight Segmentation and Registration Toolkit* (ITK)
  http://www.itk.org/
- *ImLib3D*
  http://imlib3d.sourceforge.net/
- *ExactImage*
  http://www.exactcode.de/site/open_source/exactimage/
- *Simple Library for Image Processing* (SLIP)
  http://www.sic.sp2mi.univ-poitiers.fr/slip
- ...

L'attrait de l'approche par paramétrisation, la programmation générique, s'explique par ses trois avantages principaux qui la distingue des autres approches. **Unicité des algorithmes :** L'implémentation d'un algorithme est un unique bout de code. Cette propriété, partagée avec les approches par généralisation et par inclusion, supprime les défauts de l'approche exhaustive. Cette unicité facilite le travail du programmeur, limite le risque d'erreurs (boggues) et allège la maintenance. **Non limitative :** Les algorithmes sont indépendants à la fois des types de structures de données et des types de valeurs. Les axes $\mathcal{A}$, $\mathcal{S}$ et $\mathcal{V}$ étant découplés (Cf. figure 6), l'utilisation de tout triplet (algorithme, structure, valeurs) est rendue possible. Les limitations de l'approche par généralisation sont levées. **Performante :** Le code généré par le compilateur est efficace car dédié aux types de données passées en entrée aux algorithmes. La pénalisation en performances de l'approche par inclusion est évitée.

L'approche par paramétrisation apparaît comme la plus adaptée des quatre approches pour tout domaine relevant du calcul scientifique intensif. Sa popularité actuelle semble donc pleinement justifiée.

Néanmoins, l'approche par programmation générique "classique" n'est pas exempte de défauts. S'il y en a un qui est beaucoup plus pénalisant que les autres, il s'agit de l'*absence de réification des types abstraits*. Précisément, aucun type nommé explicite ne traduit la notion abstraite d'"image". Cela conduit à de nombreuses difficultés.

Comme nous l'indiquions en section 3.2.3, l'approche par paramétrisation conduit à des fonctions dont la signature n'est pas fortement typée. Prenons par exemple une routine foo qui attend une image en entrée :

```
foo : auto[I : type] (ima : I) -> void is ...
```

nul part est précisé le fait que l'entrée doit être une image. Formellement le type de cette fonction est "[I : type] (I) −> void" ; dit littéralement :

pour tout type I, prend un I et ne retourne rien.

Ce typage faible des fonctions induit les problèmes suivants.

La surcharge des fonctions est impossible. Il arrive souvent que l'on souhaite différencier le code d'une fonction suivant la nature de son entrée ou de ses entrées. Par exemple :

```
foo : (ima : image) −> void is ... // version pour les images
foo : (ps : point_set) −> void is ... // version pour les ensembles de points
foo : (f : function) −> void is ... // version pour les fonctions
...
```

avec la programmation générique classique, ce n'est pas possible car les paramètres formels ne sont que des noms de variables et ces noms sont interchangeables. Aussi, les trois lignes :

```
foo : auto[Image : type] (ima : Image) −> void is ...
foo : auto[Point_Set : type] (ps : Point_Set) −> void is ...
foo : auto[Function : type] (f : Function) −> void is ...
```

ne sont pas trois définitions distinctes du point de vue du compilateur. Il y a ambiguïté et ce bout de programme ne compile pas.

Le typage structurel autorise l'écriture de programmes erronés. Par exemple, avec les définitions suivantes :

```
cowboy : type = class { shoot : () −> void is ... }
soccer_player : type = class { shoot : () −> void is ... }

score_a_goal : auto[Soccer_Player : type] (p : Soccer_Player) −> void
is
  p.shoot() // GOAL ! \o/
  end
```

les lignes ci-dessous sont parfaitement correctes :

```
lucky_luke : cowboy
zidane : soccer_player
score_a_goal(lucky_luke) // hum...
```

et, pourtant, le programmeur a vraisemblablement commis une erreur. Contrairement au typage nommé, la seule vérification effectuée par le compilateur lors de l'appel à la fonction score_a_goal est la présence de la méthode shoot pour le type du seul argument passé. Les deux classes définies dans cet exemple ont une sémantique distincte mais la définition de la fonction est malheureusement oublieuse.

Enfin, un troisième problème résulte du précédent : cette approche de programmation apporte son lot de messages cryptiques d'erreurs à la compilation. Comme les types ne sont pas précisés dans la signature des arguments de fonctions, les appels aux fonctions sont *a priori* correctes ; les erreurs ne sont alors détectées que plus tard, c'est-à-dire plus loin dans le code.

En guise de conclusion partielle, l'approche par programmation générique classique est séduisante de par ses propriétés d'unicité d'écriture d'algorithmes, de généricité et de performances

obtenues. En revanche, cette approche ne matérialise pas dans le code les notions abstraites. Pour le domaine du traitement d'images, s'il semble naturel de définir la notion d'image bidimensionnelle par une classe, un type bien concret, on peut se demander pourquoi faire l'impasse, dans les programmes, sur la réification des images et autres entités abstraites de ce domaine.

3.3.2 *Orienté-objet statique*

Notre objectif est de matérialiser explicitement des notions abstraites en classes, tout en préservant les performances des programmes résultants. Reprenons les définitions de classes, déjà données en figure 7, pour les approches par inclusion et par paramétrisation :

```
image : type is
  class
  {
    + get : (p : point) -> value is abstract
    + set self : (p : point) -> ref value is abstract
    ...
  }
image2d : type is
  class inherits image
  {
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

```
image2d : [V : type] -> type is
  class
  {
    + value : type is V
    + point : type is point2d
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

À gauche, des définitions qui aboutissent à des exécutables lents, mais avec une notion abstraite explicite d'image ; à droite, une définition concrète performante mais sans abstraction. Nous avons proposé un mélange de ces deux approches qui consiste à traduire statiquement le paradigme orienté-objet. Pour cela, nous conservons l'inclusion et la classe qui abstrait n'importe quel type d'images : *!*Image !. Son interface reste inchangée :

```
    + get : (p : point) -> value
    + set self : (p : point) -> ref value
```

En revanche, afin de préserver de bonnes performances, nous ne pouvons pas utiliser des classes abstraites pour désigner le type de valeurs (value) et le type de points (point). Nous devons donc nous éloigner de la solution orientée-objet classique (ci-dessus à gauche).

Notre proposition s'appuie sur la notion de *types virtuels*. Cette notion n'existe qu'à l'état de recherche ; elle n'est pas présente dans les langages de programmation répandus. Il s'agit d'une transposition aux types de la notion de méthodes polymorphes définies sur une hiérarchie de classes : une classe abstraite déclare un type "abstrait" et chaque sous-classe concrète donne plus tard une définition concrète de ce type. Dans la classe abstraite Image, nous déclarons donc deux types virtuels, value et point. Ces deux types sont alors utilisés dans la déclaration des méthodes get et set. La différence avec l'approche orientée-objet classique réside dans le fait que ces types ne sont pas des classes mais des variables qui permettent de déclarer des alias (des synonymes) pour les notions de type de valeurs et de type de points. Avec ce mécanisme, value et point ne sont plus des classes abstraites de la bibliothèque ; ils représentent des types, dont la définition est différée, mais qui sont utilisables dès à présent. La sous-classe paramétrée image2d[V] définit alors ces alias ; par exemple,

value est V dans le cas de cette classe. Nous retrouvons alors pour les classes concrètes l'utilisation de types associés propre à l'approche par paramétrisation. Pour l'instant, les seuls changements par rapport à cette approche résident dans la matérialisation en classe de l'abstraction "image" et dans la déclaration des types associés sous la forme de types virtuels. L'écriture du couple "classe abstraite / classe concrète" est donnée ci-dessous :

```
Image : type is
  class
  {
    + value : type is abstract // déclaration du type virtuel value
    + point : type is abstract
    invariant point inherits Point
    ...
    + get : (p : point) -> value is abstract // utilisation des types virtuels
    + set self : (p : point) -> ref value is abstract
    ...
  }

image2d : [V : type] -> type is
  class inherits Image
  {
    + value : type is V // définition du type virtuel value
    + point : type is point2d
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

Code source 3.11: Classes d'images avec héritage et types virtuels.

L'interface de la classe Image reste inévitablement abstraite puisque cette classe représente une abstraction. Cela se traduit par les déclarations marquées "*abstract*" des types et des méthodes de son interface. Les définitions de ces types et méthodes sont données dans chaque sous-classe concrète et peuvent alors varier d'une classe à l'autre pour tenir compte de leur spécificité. L'utilisation de types virtuels à la place de classes abstraites témoigne d'une évolution de paradigme puisque les abstractions revêtent une forme différente dans la signature des méthodes. Cela nous permet de ne pas mêler des types abstraits à l'interface de la classe Image et, ainsi, d'envisager que le corps des méthodes soit performant à l'exécution. En effet, dans les classes concrètes, nous retrouvons à l'identique le même code efficace du paradigme par paramétrisation. Nous avons donc maintenant des méthodes dont l'exécution n'est pas pénalisée par la présence d'abstractions. Cette amélioration est facilement visible. La version par inclusion de la méthode set de la classe image2d (extrait du code 3.7 page 40) s'écrivait :

```
    + set self : (p : point) -> ref value
      is
        p_ : point2d := cast[point2d](p) // p_ prend son type précis
        return data@p_.row@p_.col // le retour perd son type précis
```

```
        end
```

Maintenant, grâce à l'utilisation des types virtuels, on a exactement :

$$\text{point} = \text{point2d} \quad \text{et} \quad \text{value} = \text{V}$$

et le corps de la méthode set peut s'appuyer sur des types connus précisément :

```
  + set self : (p : point) -> ref value
    is
      invariant point = point2d and value = V
      // p est un point2d
      return data@p.row@p.col // retour de type V
      end
```

Si le code des méthodes est maintenant efficace, ce n'est pas le cas de la résolution des appels à ces méthodes polymorphes. L'utilisation de l'abstraction Image, comme pour le paradigme orienté-objet classique, engendre encore la perte de performance discutée en section 3.2.3. Il nous reste donc à rendre statique les définitions abstraites qui, classiquement, sont dynamiques.

Une façon de parvenir à ce résultat est de paramétrer l'abstraction par son type exact :

> Image[I] est le type abstrait d'une image dont le type concret est I.

Par exemple :

- une image de type concret image2d[uint8] (dont le type exact est image2d[uint8]) aura pour type abstrait Image[I] avec I valant image2d[uint8] ;
- le type d'images image3d[float] pourra être abstrait en Image[I] avec I valant image3d[float].

Dit autrement, si I représente un type concret d'images, alors ce type dérive de Image[I]. Le type qui nous permet d'abstraire la notion d'images passe donc du statut de classe "simple" à celui de classe paramétrée : Image[I]. Notons que ce type reste abstrait malgré la présence de I, qui représente un type concret. En effet, tout paramètre formel permet d'abstraire ce qu'il représente ; dans le cas présent, le type exact d'images, tel qu'il apparaît au niveau de l'abstraction Image[I], est donc abstrait. La définition de l'abstraction de la notion d'images s'écrit :

```
Image : [I : type] -> type is
  class
  {
    + value : type is ...
    + point : type is ...
    invariant point inherits Point[?]
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

Au final, tout type concret d'images dérive de la classe paramétrique Image. Ces classes paramétriques traduisent des abstractions ; elles sont donc utilisables pour vérifier qu'un type correspond bien à la

notion attendue. Par exemple, si l'on souhaite vérifier que le type virtuel point, tel que défini par la suite, sera effectivement un type de points, il suffit de vérifier qu'il dérive bien de l'abstraction Point. Cela se traduit par l'invariant point **inherits** Point[?] dans le code ci-dessus.

Les portions de code omises (les "...") peuvent correspondre maintenant à des définitions puisque l'on peut parler du type exact d'images : même si nous ne connaissons pas quel est véritablement (exactement) ce type, il est désigné par I. Ces portions de code correspondent à ce qui est réalisé classiquement à l'exécution, c'est-à-dire au remplacement des déclarations par les définitions : des types pour les types virtuels abstraits et des implémentations pour les méthodes abstraites. Par exemple, le type des valeurs d'une image de type exact I est I : :value donc le type associé value dans Image[I] est I : :value. De même, la méthode set pour une image de type exact I est I : :set ; le code de Image[I] : :set revient donc à appeler celui de I : :set.

```
// Version statique du code 3.11 page 54

Image : [I : type] -> type is
  class
  {
    + value : type is I::value
    + point : type is I::point
    invariant point inherits Point[?]
    ...
    + get : (p : point) -> value is self.I::get (p)
    + set self : (p : point) -> ref value is self.I::set (p)
    ...
  }

image2d : [V : type] -> type is
  class inherits Image[ image2d[V] ] // le paramètre I vaut donc image2d[V]
  {
    + value : type is V
    + point : type is point2d
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

Code source 3.12: Version des classes d'images pour l'approche par inclusion statique.

Finalement, la définition complète de la classe paramétrique abstraite est donnée par le code 3.12. Les résolutions des types virtuels et des appels aux méthodes y sont écrites en dur. L'interface des classes abstraites est donc *pseudo-abstraite* : ses éléments (types virtuels et méthodes) restent intrinsèquement abstraits puisqu'il est impossible de donner leur définition finale mais ils sont définis, bien concrètement, par le code permettant leur résolution à la compilation. Par conséquence, les programmes résultants sont aussi efficaces que s'il n'y avait aucune abstraction. La classe image2d[V] dérive de l'abstraction en valuant le paramètre formel I avec son propre type : image2d[V].

```
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void
is
  i : I::iterator := ima
  p : I::point ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

Code source 3.13: Version de fill par inclusion statique.

La routine fill, quant à elle, est paramétrée par le type exact d'images qu'elle reçoit en entrée : I ; le type de son argument ima est donc "une image de type I", soit Image[I]. La version de fill par inclusion statique est donnée par le code 3.13. Elle représente un parfait mélange des versions respectives de fill par inclusion (code 3.8) et par paramétrisation (code 3.10) :

```
fill : (ima : ref image, v : value) -> void // par inclusion
fill : auto[I : type] (ima : ref I, v : I::value) -> void // par paramétrisation
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void // par inclusion statique
```

Lors d'un appel à cette routine, le code générique est remplacé par le compilateur par un code dédié, propre au type exact de l'image à traiter. Ainsi, si l'image en entrée est de type précis image2d[uint8], une version spécialisée de fill est générée et optimisée. On retrouve à l'identique le comportement de la version par paramétrisation donnée page 43. La seule différence réside dans le typage fort de cette nouvelle version, avec la présence explicite du type "abstrait" Image.

3.3.3 *Orienté-objet statique simplifié*

L'héritage présenté dans la section précédente (code 3.12) réalise :

```
class image2d[V] inherits Image[ I := image2d[V] ] // pseudo-code
```

ce qui se lit : "une image2d[V] est une *Image* de type image2d[V]". Nous voyons apparaître ici que la définition des classes est récursive. Le motif "I dérive de Image[I]" est connu comme le *curiously recurring template pattern* [4], mis en évidence pour factoriser du code au niveau de la sur-classe sans être pénalisé en terme d'efficacité à l'exécution. Ici nous avons généralisé cette construction récursive puisque toute l'interface des classes abstraites, les types virtuels y compris, suivent ce motif.

Le problème que pose cette approche est la multiplication du travail demandé au compilateur. En effet, la nature récursive des définitions de classes n'est pas implémentable directement. Cela entraîne des contorsions, décrites dans la publication portée en annexe B.2. En particulier, les types virtuels doivent être définis en dehors des classes, afin de casser toute récursion lors du typage. L'écriture effective du paradigme par inclusion statique, en vrai code C++ qui compile, s'avère donc plus lourde que son expression en mini-langage donné dans la section précédente. En pratique, le nombre de classes doit déjà être doublé :

```
// Version effective du code 3.12 page 56
```

```
vtypes : [ I : type ] -> type is later // base de données des types virtuels

Image : [I : type] -> type is
  class
  {
    + value : type is vtypes[I]::value
    + point : type is vtypes[I]::point
    invariant point inherits Point[?]
    ...
    + get : (p : point) -> value is self.I::get (p)
    + set self : (p : point) -> ref value is self.I::set (p)
    ...
  }

image2d : [V : type] -> type is later

vtypes : [ auto V : type, image2d[V] ] -> type is
  class // définitions des types virtuels de la classe image2d[V]
  {
    + value : type is V
    + point : type is point2d
  }

image2d : [V : type] -> type is
  class inherits Image[ image2d[V] ]
  {
    + value : type is super::value // récupération des types virtuels
    + point : type is super::point // définis dans la sur-classe Image
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

De plus, la prise en compte des types d'images spéciaux que sont les *morpheurs*, décrits en section 4.3.2, ajoute encore un niveau d'indirection supplémentaire dans la définition des types virtuels ; Cf. la publication en annexe B.3. Il faut alors encore dé-doubler les définitions de classes. Au final, le nombre de classes nécessaires à l'implémentation de ce paradigme est un multiple du nombre de classes que l'on veut effectivement définir. Les temps de compilation sont alors de 10 à 40 fois supérieurs à ceux que l'on est en mesure d'attendre avec de tels programmes.

Pour revenir à des temps de compilation classiques donc raisonnables, nous proposons de simplifier l'implémentation de ce paradigme. Le paradigme en est allègement transformé mais il conserve toutes les propriétés décrites précédemment.

Au lieu de déclarer des types virtuels dans les classes abstraites et d'implémenter en dur leur résolution, nous nous contentons de vérifier la présence de ces types dans les classes concrètes. Une classe abstraite ne fait plus apparaître ses types virtuels comme composante classique de son interface mais ils restent présents dans son interface sous la forme de requis. De la même façon, les méthodes pseudo-abstraites n'apparaissent plus comme méthodes dans l'interface mais

comme requis. Le code 3.14 montre le résultat de cette transformation. L'interface devient une liste d'invariants vérifiés à la compilation.

La routine fill, donnée par le code 3.15, est légèrement modifiée. Une ligne y est ajoutée réalisant le transtypage descendant de l'image d'entrée : son type passe de son type abstrait Image[I] vers son type concret exact I. Ce transtypage est nécessaire car le type Image[I] est *creux*, dénué d'interface ; une image possédant ce type abstrait (la variable ima_ dans l'exemple de code) n'est pas manipulable. En revenant au type exact, l'image (tenue par la variable ima) retrouve son interface et est pleinement manipulable.

```
Image : [I : type] -> type is
  class
  {
    !invariant
      I::value = type and
      I::point inherits Point[?] and
      typeof(get) = (point) -> value and
      typeof(set self) = (point) -> ref value and
      end
    ...
  }

// la classe image2d[V] retrouve une définition simple :

image2d : [V : type] -> type is
  class inherits Image[ image2d[V] ] // le paramètre I vaut donc image2d[V]
  {
    + value : type is V
    + point : type is point2d
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...
  }
```

Code source 3.14: Version de la classe abstraite d'images pour l'approche par inclusion statique simplifiée.

Au final, le paradigme par inclusion statique devient véritablement pratiquable. Les seules modifications apportées par sa simplification sont assez légères, presque cosmétiques :

- l'interface des classes d'abstractions devient une vérification d'interface ;
- l'utilisation de cette interface dans les algorithmes, au lieu d'être directe, nécessite une ligne de transtypage.

```
fill : auto[I : type] (ima_ : ref Image[I], v : I::value) -> void
is
  ima : ref I = exact(ima_) // transtypage descendant vers le type exact
  i : I::iterator := ima
  p : I::point ref is i.get_point()
  for i.start(); i.is_valid(); i.next()
    ima.set(p) := v
  end
```

Code source 3.15: Version de fill par inclusion statique simplifiée.

4

TAXONOMIE DES IMAGES

> Ces catégories ambiguës, superfétatoires, déficientes rappellent celles que le docteur Franz Kuhn attribue à certaine encyclopédie chinoise intitulée *Le Marché céleste des connaissances bénévoles*. Dans les pages lointaines de ce livre, il est écrit que les animaux se divisent en a) appartenant à l'empereur, b) embaumés, c) apprivoisés, d) cochons de lait, e) sirènes, f) fabuleux, g) chiens en liberté, h) inclus dans la présente classification, i) qui s'agitent comme des fous, j) innombrables, k) dessinés avec un très fin pinceau de poils de chameau, l) et caetera, m) qui viennent de casser la cruche, n) qui de loin semblent des mouches.
>
> *Jorge Luis Borges, [« La langue analytique de John Wilkins », dans] Enquêtes.*

4.1 QU'EST-CE QU'UNE IMAGE

La mise en informatique de la notion abstraite d'*image* ainsi que des types concrets d'images nécessite de définir dans le détail ce qu'est une image et, en corrolaire, ce qui caractérise chaque type d'images. Cela signifie répondre à un double objectif :

- définir le dénominateur commun de "toutes" les différentes sortes d'images, c'est-à-dire, ce que l'on est en mesure d'attendre de toute image ;
- définir pour chaque type d'images particulier ce que ce type apporte de plus par rapport à la notion abstraite d'images.

Pour cela, l'approche que nous proposons consiste à décrire un ensemble de caractéristiques qui forme la personnalité et l'identité des images.

4.1.1 *Une approche pour répondre*

Une façon de répondre à la question “qu’est-ce qu’une image ?”, sachant qu’on souhaite apporter une réponse informatique à cette question, est de regarder ce que signifient, ou représentent, les structures d’images dans les bibliothèques existantes.

De façon pragmatique, lorsqu’un type d’image est défini dans une bibliothèque, il représente une occurrence de “réponse” à la mise en informatique d’une certaine sorte d’images. Dans les langages informatiques, un type décrit le comportement des instances que l’on déduit de ce type ; le couple type-instance joue le même rôle que le couple ensemble-élément en mathématiques : le “comportement” des éléments est dicté par les “propriétés” de leur ensemble. La définition d’un type est donc extrêmement descriptive. Regarder comment les types d’images sont définis dans les bibliothèques permet d’extraire le “sens” que l’on a prêté aux images, leur sémantique.

Une autre façon d’expliquer cette approche consiste à réaliser qu’il s’agit de rétro-ingénierie (ingénierie inverse). L’implémentation d’un type d’images revient à effectuer une traduction d’une notion appartenant au domaine du traitement d’images en une construction en langage informatique. Sous l’hypothèse que cette traduction est fidèle, le résultat informatique, le type d’images, correspond effectivement à l’intention du traiteur d’images, une sorte d’images. Partir du résultat pour revenir à l’intention revient à parcourir le chemin inverse du développeur, à passer du monde informatique vers celui du traitement d’images. L’ingénierie inverse permet de mieux comprendre les mécanismes mis en jeu dans le passage entre ces deux mondes.

En particulier, cette approche a deux avantages. Tout d’abord, en ayant une meilleure connaissance de l’arrivée, on a une meilleure perception du chemin à parcourir et on peut espérer être mieux guidé pour réaliser la traduction image vers informatique. Un second avantage est qu’en prenant des distances avec l’intention nous laissons de côté la subjectivité de l’auteur, du programmeur. En nous écartant de la documentation (scientifique, technique et marketing), nous échappons aux *a priori*, aux approximations (voire aux erreurs) et aux oublis ou non-dits. Un exemple de ce phénomène a déjà été donné en section 2.3.1 page 21 : l’utilisation d’un type d’images dévoile des hypothèses implicites dont l’auteur n’a peut-être jamais mesuré l’aspect limitatif.

Regarder un type d’images dans les yeux signifie observer son comportement et, par là-même, lui faire avouer ce qu’il est sensé représenter. En d’autres termes, l’interface, liste des types et méthodes publiques du type, témoigne du comportement de ce type et donc de sa signification. Si le comportement peut être décrit sous forme textuelle, une traduction “fonctionnelle” Ici, on part de l’hypothèse non innocente que le comportement d’un type est complètement dicté par la sémantique du type, à-la “ dis-moi ce que tu fais, je te dirais qui tu es”. Cette approche est valide car en adéquation avec la réalité de l’implémentation : une instance de type *est* ce que l’on peut en faire.

Il est difficile de ne pas constater que nous laissons volontairement de côté l’approche classique, que l’on trouve dans de nombreux ouvrages : répondre à la question “qu’est-ce qu’une image ?” en donnant la définition d’une image dans chaque contexte théorique. Une image est, suivant le contexte, soit une matrice, soit des valeurs plaquées sur une structure topologique, soit une fonction discrète, etc. L’hétérogénéité de ces différentes visions, ou interprétations d’images, appartient clairement au domaine du traitement d’images. Nous nous éloignons alors bien trop du

monde informatique dans lequel nous souhaitons nous immerger. Les interprétations d'images sont connues du traiteur d'images ; elles n'apportent pas *directement* d'informations sur leur traduction informatique. Notons que l'on pourrait avoir l'impression qu'une traduction informatique unique de toutes ces interprétations d'images nous donnerait une généralisation / une unification des différentes approches. C'est faux : nous recherchons ici une intersection, c'est-à-dire, ce qu'il y a de commun à toutes ces visions ; nous ne nous intéressons pas à l'union / à la généralisation de ces visions.

Nous allons chercher à dresser une taxonomie des images. Une façon de procéder est de lister tous les types d'images désirés, d'en extraire un ensemble de propriétés et d'utiliser ces propriétés pour exhiber des classes d'images, pour classifier les types d'images. Nous avons en fait suivi cette démarche ; dans les pages qui suivent n'apparaît que le résultat ce notre travail, précisément, un catalogue de propriétés et de types d'images avec leurs propriétés.

4.1.2 *État de l'art des types*

Dans la section 2.3.4, nous avons vu que l'axe des types ($\mathcal{T}$) pouvait être scindés en deux axes distincts : l'axe des structures ($\mathcal{S}$) et l'axe des valeurs ($\mathcal{V}$). Cela correspond à l'utilisation très classique de la généricité pour définir des types de structures de données. Dans le cas du traitement d'images, cette même dissociation prend tout son sens : les types de valeurs des pixels et les types de structures d'images sont indépendants. Pour preuve, nous pouvons imaginer toute combinaison (type de structure, type de valeurs). Aussi, au lieu de définir un type monolithique d'images, comme l'est image2d_uint8, qui associe de façon irrémédiable la structure d'image bidimensionnelle au type de valeur 'entier non signé sur 8 bit', la plupart des bibliothèques modernes utilisent un type paramétrique pour définir une structure et de reléguer le type de valeurs en paramètre. Ainsi, la dissociation se traduit par :

image2d_uint8 $\rightarrow$ image2d[**uint8**]

où les deux types, image2d et **uint8**, sont définis dans le programme de façon indépendante. Le dédoublement de l'axe des types d'images, $\mathcal{T}$, en deux axes, $\mathcal{S}$ et $\mathcal{V}$, est véritablement effectif dans le programme. Cela est illustré par la première ligne de la figure 8. Relevons néanmoins l'aspect trompeur de la représentation en axes ; les axes ne jouent pas véritablement un rôle symétrique puisque, traduits en code, ils ne revêtent pas la même nature : une structure est un type paramétrique tandis que le type des valeurs est un paramètre.

Un type paramétrique, qui sera toujours représenté sur l'axe $\mathcal{S}$ par la suite, est en fait *le* type d'image par opposition à ses paramètres, portés par les autres axes. Un type paramétrique a en fait un double rôle : il représente une notion (matérialise cette notion par un type) et sert de glu pour tirer partie d'informations provenant de paramètres. Les paramètres sont en fait les parties variables du type complet ; la partie fixe est donnée par le type paramétrique. Ainsi image2d est (une famille de types) qui représente la notion d'image bidimensionnelle et choisir un type de valeurs permet de compléter la définition de ce type d'image bidimensionnelle. Dit autrement un type d'image bidimensionnelle a besoin d'un type de valeurs et l'inverse n'est pas vrai ; aussi, le type de valeurs est donc glué au type de structure. Cette dissymétrie traduit une dépendance

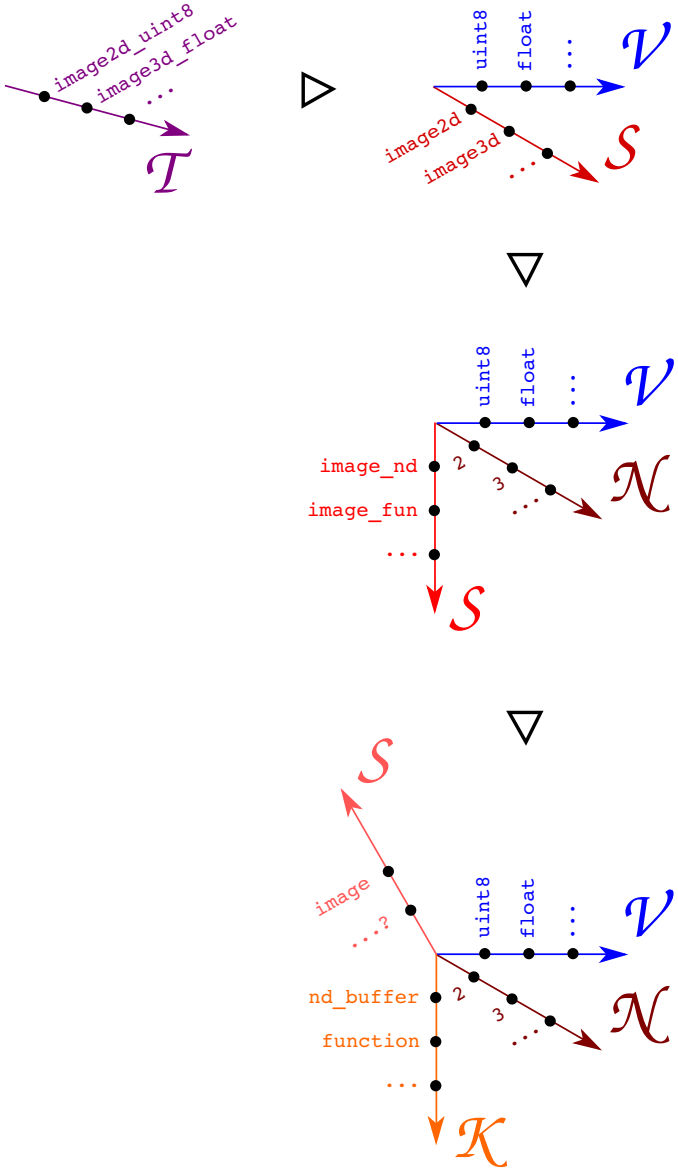


FIGURE 8.: Décomposition progressive de l'espace des types d'images.

dans l'écriture de programme : on a besoin de paramètres pour définir un type à partir d'un type paramétré. Néanmoins, cette dépendance n'est pas si nette d'un point de vue sémantique et la représentation par axes orthogonaux est alors justifiée : une structure d'image bidimensionnelle *plus* le type de valeurs **uint8** forme effectivement un type d'images. Schématiquement, nous pouvons considérer que :

$$\text{image2d[} \textbf{uint8} \text{]} \quad \approx \quad \text{image2d} \; + \; \textbf{uint8}.$$

Cette façon de voir permet de changer de perspective. Si, du point de vue du type final image2d[**uint8**], **uint8** est un paramètre et dénote la partie variable du type paramétré image2d, d'un autre côté, du point de vue de l'utilisateur, les deux composantes du types peuvent être choisies indépendamment. Finalement, en terme de choix pour la définition d'un type d'images, le type de structure et le type de valeurs jouent le même rôle. Nous avons deux caractéristiques orthogonales de types d'images et l'utilisateur a effectivement deux choix (structure *et* valeur) pour former un type d'images.

L'introduction d'un paramètre lors de la transformation de image2d_uint8 en image2d[**uint8**] permet de réduire de façon drastique la quantité de code nécessaire à la définition de types d'images. La redondance de code entre les types tels que image2d_uint8 et image2d_float est éliminé grâce au code unique de la classe paramétrée image2d.

> La paramétrisation est *à la fois* un procédé d'orthogonalisation de caractéristiques *et* un procédé de factorisation de code. De plus, la paramétrisation permet de créer des types réutilisables (donc non-limitatifs) car le choix de la valuation d'un paramètre reste ouvert.

En poursuivant cette démarche, quelques rares bibliothèques proposent des types d'images à plusieurs paramètres. C'est le cas d'ITK où la classe paramétrique d'image est paramétrée par le type de valeurs et par la dimension de l'image. La décomposition des types d'images à l'aide de paramètres continue donc ; elle peut se schématiser suivant :

$$\text{image2d\_uint8} \quad \rightarrow \quad \text{image2d[} \textbf{uint8} \text{]} \quad \rightarrow \quad \text{image\_nd[2D, } \textbf{uint8} \text{]}.$$

Le type obtenu est un type d'images n-dimensionnelles avec n valant 2 et des valeurs de pixels en **uint8**. Nous retrouvons ici une factorisation de code : au lieu d'avoir deux codes distincts, respectivement pour image_2d et image_3d, nous disposons d'un code unique, celui de image_nd. Ce type permet automatiquement de ne pas être limité aux dimensions 2 et 3 : il est valide pour d'autres dimensions d'images donc nous pouvons imaginer de traiter des signaux (dimension 1) ou des images de dimension supérieure à 3.

L'orthogonalisation de caractéristiques aboutit à un type d'images composés de trois parties :

$$\text{image\_nd} \; + \; \text{2D} \; + \; \textbf{uint8}.$$

Pour former un type d'images, l'utilisateur a la possibilité de choisir la dimension, le type de valeurs mais aussi la composante paramétrique. Ici image_nd peut être un choix parmi d'autres types proposés. Une nouvelle question est de savoir ce que représente cette composante, ce troisième

axe $\mathcal{S}$. La question suivante est de savoir quelles sont les différentes options offertes à l'utilisateur, c'est-à-dire quels sont les types de cet axe. (Rappel : lors de l'étape précédente, l'axe $\mathcal{S}$ représentait la structure de l'image et les types offerts étaient image_2d, image_3d, etc.) Puisque la dimension de l'image est maintenant "sortie" de l'axe $\mathcal{S}$ pour former un axe à part entière, l'axe $\mathcal{N}$, les questions peuvent être reformulées suivant :

- qu'est-ce qu'une image_nd ?
- qu'est-ce qu'une image lorsque l'on a déjà dit quel est le type des valeurs des pixels et quelle est la dimension de son espace de définition ?
- quelle autre partie variable que le type de valeurs et la dimension intervient dans la définition d'un type d'images ?
- quels sont les différents types d'images ayant le même type de valeurs et la même dimension ?

Une réponse est donnée par des bibliothèques autres que celles de traitement d'images : pour un conteneur, la façon de fournir les valeurs est une caractéristique que l'on a envie de faire varier. L'axe $\mathcal{S}$ se caractérise donc par la façon dont est défini l'ensemble de valeurs des pixels. Le cas le plus courant dans les bibliothèques est de stocker ces valeurs en mémoire vive (RAM) sous la forme d'une matrice n-dimensionnelle. Ce cas correspond bien à l'idée que l'on peut se faire du type d'images désigné par le type image_nd. D'autres façons de stocker les valeurs des pixels donneront alors d'autres types paramétriques de l'axe $\mathcal{S}$: si le stockage n'est pas en mémoire vive, par exemple sur le disque dur (*memory mapping*) ou non-local car sur le réseau, si le stockage n'utilise pas une matrice n-D, par exemple avec un tableau associatif. Les autres pistes concernant la fourniture des valeurs sont nombreuses : valeurs calculées à la volée par une fonction, avec gestion d'un cache, avec mémoïsation, etc. Enfin, citons deux types d'images très classiques qui se caractérisent par une gestion des valeurs particulière : une image constante, où tous les pixels partagent la même valeur, nécessite d'une cellule mémoire pour stocker cette valeur, et une image avec une table de correspondance (*look-up-table*) où les valeurs stockées sont les indices de la table et non les valeurs effectives des pixels.

La figure 8 montre l'évolution des axes qui définissent l'espace des types d'images. L'état de l'art des bibliothèques de traitement d'images existantes s'arrête à la dissociation structure + dimension + type de valeurs. C'est le cas d'ITK. En ne fournissant qu'un type (paramétrique) d'images unique :

image[n,V]

avec n pour la dimension et V comme type de valeurs, l'utilisateur n'a pas la possibilité de choisir la façon dont les valeurs des pixels sont fournis. Pire, en nommant image le type paramétrique, les auteurs de cette bibliothèque déclarent implicitement qu'il s'agit bien du type d'images, comprendre "la notion d'image", absolue, générale et unique. En conséquence, cette position occulte le fait que des hypothèses implicites sont liées à ce type, qui, justement, n'en font pas le type d'images réifié. Afin de rapprocher cet exemple des approches décrites en section 3.1, il s'agit ici d'une approche de "généralisation par paramétrisation". En effet, un unique type général est écrit : le type paramétrique image, qui est sensé représenter "toutes" les images possibles. En addition, les paramètres permettent de le rendre plus souple que le type général décrit en section 3.1.2.

Finalement, décomposer un type d'images général en plusieurs axes, un par caractéristique, revient à étendre la généricité "classique" (type de structure + type de valeurs) avec pour objectif d'être encore plus générique.

4.1.3 *Multiplicité des types*

Poursuivre la dissociation des axes en caractéristiques orthogonales devrait aboutir *in fine* à un type paramétrique qui soit réduit au rôle de glu. Toute partie variable de la nature des images serait représentée par un axe dédié et mise en œuvre par un paramètre. Au final, nous aurions effectivement le type paramétrique unique :

image[axe_1, axe_2, .. , axe_n].

Cette approche par décomposition est assez populaire dans les bibliothèques en programmation générique. Elle a pourtant de sévères inconvénients.

Tout d'abord, l'écriture d'un type concret, en valuant tous les paramètres est très lourde ; imaginez ce que doit écrire un utilisateur qui a seulement besoin du type d'images le plus utilisé (nommément image2d[**uint8**] avec une version plus limitée de la généricité). Un autre défaut est l'intrication des paramètres. Une autre défaut réside dans l'écriture du code du type paramétrique : comme les paramètres apportent des informations orthogonales mais complémentaires, ce code doit fusionner ces informations disparates et une intrication terrible en résulte. En corollaire, on peut signaler que ces difficultés se manifestent déjà lors de l'écriture des paramètres. En effet, tout indépendant qu'ils sont du point de vue sémantique, il n'en va pas de même du point de vue du typage. Par exemple, le type de stockage des valeurs est généralement paramétré par le type des valeurs. Cacher ces dépendances à l'utilisateur, pour qu'il ne se perde pas dans leur écheveau, complexifie beaucoup le code et, ne pas les cacher, fait apparaître comme paramètres des méta-types tout aussi difficiles à manipuler. Un dernier défaut réside dans l'extension de cette écriture en liste de paramètres. Si l'on imagine ajouter une nouvelle caractéristique aux images, parce que l'on a tout simplement pas pensé à tout (!), il faut donc augmenter le type d'images par un nouveau paramètre. Cette conclusion est dramatique du point de vue du génie logiciel puisqu'elle revient à modifier le type le plus utilisé partout ; la propagation de ce changement touche donc toutes les parties de code, y compris celles des clients de la bibliothèque. Ces bouts de code, qui dorment sûrement dans un coin en attendant d'être de nouveau appelés, ne sont pas sensés devoir autant évoluer pour se plier aux changements de la bibliothèque.

L'orthogonalisation des caractéristiques des images en liste de paramètres présente trop de défauts majeurs pour être une solution à la généricité des types d'images. L'idée d'un type paramétrique général n'est pas viable si l'on souhaite avoir une bibliothèque extensible, c'est-à-dire, propre à multiplier les types d'images.

Nous proposons ici une solution à l'opposée de la quête de généralisation. Trois règles dictent la définition de types d'images :

- un type d'images s'écrit avec un nombre minimal de paramètres ;
- chaque type d'images a un trait principal ;
- toute caractéristique qui revêt un caractère optionnel ou singulier (spécifique, original) est traitée dans un type à part.

Afin d'expliciter ces règles, procédons tout d'abord à l'examen de contre-exemples. Pour la première règle, le type paramétrique général image[axe_1, axe_2, .. , axe_n] est le parangon du type pénible à lire et à manipuler. L'utilisateur doit spécifier la nature de son choix sur tous les

axes en valuant tous les paramètres formels ; il doit donc préciser de façon détaillée toutes les caractéristiques du type d'images, y compris les moins importantes et les "non-caractérisques". Un exemple de non-caractéristique est qu'aucun masque n'est associé à l'image (son domaine de définition n'est pas restreint à l'aide d'une image binaire) ; devoir dire ce qui n'est pas est ici maladroit. *In fine*, non seulement l'écriture est lourde mais le code devient difficilement lisible car l'important est noyé dans la verbosité. À l'inverse, disposer de types au nommage concis, comme image2d[V], améliore la lisibilité et la compréhensibilité du code ; en réduisant le nombre de paramètres, on évite de désigner des types par des périphrases. Pour la deuxième règle, un contre-exemple est un type d'images ayant au moins deux traits principaux. S'il existait, le type image2d[V,M], qui représente une image bidimensionnelle classique dotée d'un masque (de type M), est donc la réunion de ces deux notions : le type classique d'images *et* la présence d'un masque. Ce type va coexister au sein de la bibliothèque avec le type classique seul, sans masque, image2d[V]. Au final, il y a aura redondance de code entre ces deux versions (nous sommes dans la situation contraire du type général où le code unique gère tous les cas dont la présence éventuelle d'un masque). Du point de vue du génie logiciel, cette redondance est à proscrire puisqu'elle est source de problèmes : multiplicité du code à maintenir, risque d'incohérence entre les duplicata, etc. Le cas du type image2d[V,M] est également un contre-exemple de la troisième règle. Ici la notion de masque est couplée au type d'images bidimensionnelles ; il faudra donc sûrement créer de même un type image3d[V,M] afin de bénéficier des masques sur des images tridimensionnelles. Comme la caractéristique "masquage" n'est pas traitée indépendamment des types d'images à masquer, cette absence d'orthogonalisation entraîne soit de la duplication de code (pour traiter plusieurs cas), soit du manque de code (car finalement tous les cas possibles ne seront pas traités).

Ces trois règles sont facilement justifiables. Un type d'images doit pouvoir s'écrire simplement ; ne préserver qu'un nombre minimal de paramètres peut d'éviter la complexité. Lorsque chaque type d'images a un trait principal, on garantit une bonne compréhension de ce type et on limite la redondance entre types. Traiter à part toute caractéristique optionnelle ou singulière permet de conserver la nature *atypique* de cette caractéristique : elle peut alors s'appliquer à de nombreux types d'images.

Finalement, nous aboutissons naturellement à la conclusion que les types d'images concrets doivent être nombreux afin de satisfaire à ces règles. Au lieu de ne proposer qu'un type ou qu'une poignée de types, la diversité de la nature des images doit se traduire par une multitudes de types offerts à l'utilisateur. Ce dernier n'a alors qu'à choisir celui ou ceux qui sont les plus appropriés à son application.

4.2 INTERFACES ET PROPRIÉTÉS

En la présence de multiples types d'images, il est nécessaire que l'utilisateur sache, donc retienne facilement, ce qu'il peut attendre de tel ou tel type d'images. Pour manipuler un type, ce dernier offre une interface, ensemble de méthodes et de types associés. Cet ensemble précis, augmenté de sa documentation, traduit l'identité du type en question. Répondre à la question "qu'est ce qu'une image" consiste alors à comprendre pourquoi telle interface est offerte par telle image.

4.2.1 *Abstraction, interface commune et interfaces spécifiques*

Tous les types concrets d'images dérivent de l'abstraction ***image***[**I**]. Ils partagent un socle commun qui se traduit par une interface réduite : un ensemble de types associés et de méthodes. Définir précisément cette interface permet de répondre partiellement à la question "qu'est-ce qu'une image". En effet, cela signifie que l'on a décrit la façon d'utiliser toutes les images, c'est-à-dire le comportement commun de toutes les images et, par extension, l'identité des "images".

La représentation théorique la plus simple d'une image numérique f est une fonction :

$$f : \left\{ \begin{array}{ccc} \mathcal{D} & \longrightarrow & \mathcal{V} \\ p & \longmapsto & f(p) \end{array} \right.$$

où $\mathcal{D}$ et $\mathcal{V}$ sont respectivement le domaine et le co-domaine de f. Ainsi, une image peut être vue comme un ensemble d'associations $f \equiv \{(p, f(p))\}$. Suivant les différentes interprétations théoriques de la nature des images, les éléments p du domaine de définition d'une image sont de nature très différente. Ils peuvent être des points, des diracs, des sommets d'un graphe, des faces de complexes, etc. Nous les appellerons *sites* pour rester neutres vis-à-vis de toutes ces interprétations et ne retenir uniquement le fait qu'ils logent les valeurs des images. Le type de domaine reste abstrait dans la définition abstraite d'une image. Aucune hypothèse sur la nature de ce type n'est réalisée ; ce peut être le type d'un sous-ensemble de $\mathbb{Z}^2$, ou le type d'un graphe, ou n'importe quoi d'autre. La réification naturelle de la multiplicité de domaines d'images est la notion d'ensemble de sites. Tout comme nous avons le concept d'***Image***, nous avons celui de ***Site_Set*** pour abstraire les types de domaines. Pour accéder aux éléments d'un domaine, nous utilisons la notion d'itérateurs (introduite en section 3.1.3). Pour des raisons pratiques, nous offrirons à partir d'une image, la possibilité d'itérer sur les sites de son domaine ; tout type d'images expose alors un type d'itérateurs (quasiment toujours, celui des itérateurs de son domaine). Lors d'une itération en cours (non terminée), l'objet itérateur est valide et *désigne* donc un site du domaine de définition de l'image. Bien qu'un itérateur ne soit pas exactement un site, il est pratique de confondre volontairement ces deux notions ce qui, concrètement, se traduit par le fait que l'utilisation d'un itérateur à la place du site qu'il désigne doit non seulement être possible mais aussi doit toujours rester transparente [1]. Une conséquence directe de cette transparence est que nous prendrons invariablement les mêmes noms de variables pour un site et un itérateur sur les sites d'un ensemble ; par exemple, `p` désignera bien aussi bien un site ou l'itérateur qui désigne ce site. À l'aide d'un itérateur, la valeur de l'image en un site peut être *lue* (insistons : cette valeur est toujours lisible mais, *a priori*, elle n'est pas modifiable). À cet effet, toute image propose un opérateur parenthèses, "()". Avec une image `f` et un itérateur `p`, la valeur de l'image au site désigné par l'itérateur est obtenue en écrivant `f(p)` (ce qui est un appel à la méthode "opérateur parenthèses").

Une traduction informatique simple de la représentation des images comme des fonctions d'un ensemble itérable de sites vers un ensemble de valeurs est donnée par la classe abstraite ***image***[**I**] du code 4.1.

L'interface de l'abstraction image est extrêmement simple et courte Cela ne veut pas dire qu'elle est simpliste, ni trop pauvre. Elle suffit à exprimer de nombreux algorithmes non naïfs et, dans la

1. Dans les faits, cette facilité est difficile à mettre en œuvre informatiquement...

```
Image_interface : interface is
  {
    + domain_t : type is abstract // type du domaine
    + domain : () -> domain_t is abstract // retourne le domaine 𝒟

    + site : type is abstract // type des sites (éléments du domaine)
    + value : type is abstract // type des valeurs

    + "()" : (p : site) -> value is abstract // opérateur “parenthèses”

    + piter : type is abstract // type d’un itérateur
  }
```

Code source 4.1: Interface du concept abstrait d’image (version simplifiée).

pratique, elle s’est avérée ne pas être bloquante pour l’implémentation des nombreux algorithmes existants actuellement dans la bibliothèque MILENA. L’exemple le plus élémentaire permettant d’apprécier l’utilisation de cette interface est un algorithme qui affiche l’image sous la forme de couples “site, valeur de l’image en ce site”. Cet algorithme s’écrit ainsi :

```
print : auto[I : type] (f : Image[I]) -> void
is
  p : I::piter := f.domain() // p itérateur sur 𝒟({})
  for_all(p)
    print(p, f(p)) // affiche le couple (p, f(p))
  end
```

for_all est une macro (raccourcis d’écriture) telle que l’expansion de *for_all*(p) donne la boucle d’itération classique : “**for** p.start(); p.is_valid(); p.next()” ce qui se lit : “p débute l’itération ; tant que p est valide (n’a pas fini d’itérer), on fait quelque chose puis p passe à l’élément suivant”.

Le type abstrait *Image* représente l’“intersection” de tous les types d’images possibles. Dit autrement, ce type expose ce que toutes les images ont en commun, i.e., l’intersection de leur interface, leur plus grand dénominateur commun. (Il ne s’agit donc pas d’un mécanisme de généralisation par union comme présenté en section 3.1.2 qui, lui, signifierait que l’interface exposée est le plus petit commun multiple des types d’images.)

Chaque type d’image particulier va avoir une interface constituée de l’interface commune à toutes les images *plus* une part additionnelle d’interface. Cette dernière reflète les particularités de ce type d’images, ce qui est propre à ce type mais que ne possèdent pas toutes les images. Plus précisément :

- cette interface additionnelle est l’union d’interfaces spécifiques qui traduisent des propriétés particulières de ce type d’images ;
- en règle général, chaque propriété prise indépendamment se traduit par une interface propre à cette propriété ;
- une propriété particulière est généralement partagée par plusieurs types.

Illustrons cela avec le type classique image2d_uint8, tel qu'on le trouve implémenter dans la plupart des bibliothèques. Les propriétés qui décrivent ce type d'images sont (de façon quasi exhaustive) les suivantes :

- c'est un type primaire d'images, i.e., instantiable par l'utilisateur sans qu'aucune autre image ne préexiste ;
- le nombre de pixels est connu sans qu'un parcours soit nécessaire (dit autrement l'accès à ce nombre est en temps constant) ;
- la boîte englobante du domaine de définition est accessible en temps constant ;
- la structure topologique sous-jacente est une grille régulière ;
- ces valeurs sont à accès aléatoire (pas d'itération nécessaire pour accéder à la i-ème valeur, comme pour les données d'un tableau ; par opposition, une structure de liste chaîne n'est pas à accès aléatoire) ;
- en addition des deux propriétés précédentes, comme le domaine est bidimensionnel, un accès par un couple d'indices (ligne, colonne) est possible ;
- chaque valeur d'un pixel peut être modifiée indépendamment d'une autre ;
- toutes les valeurs des pixels sont stockées (elles ne sont pas calculées à la volée) ;
- l'ensemble des valeurs est stocké de façon linéaire en mémoire ;
- la taille de l'image ne peut pas être gigantesque (typiquement, on s'attend à ce que cette image "tienne en RAM") ;
- la liste des valeurs effectivement prises n'est pas connue *a priori* ;
- il n'y a pas d'accès prévu pour accéder à tous les pixels ayant une valeur donnée ;
- cette image contient des valeurs entières (donc scalaires) ;
- les valeurs de cette image sont faiblement quantifiées (8-bit) ;
- l'image possède un bord extérieur tournant (afin de pouvoir maîtriser des "effets de bord" et d'en tirer partie lors de parcours du voisinage des pixels du bord intérieur de l'image) ;
- ce bord extérieur est re-dimensionnable ;
- les valeurs des pixels de ce bord sont modifiables, et ce, indépendamment les uns des autres.

C'est cet ensemble de propriétés qui caractérise ce type d'images. À la lecture de certaines propriétés, on devine facilement l'interface qu'elles apportent ; par exemple :

- comme le "nombre de pixels est connu", une méthode nsites() peut le donner ;
- comme le "stockage linéaire des valeurs en mémoire", plusieurs méthodes permettent de manipuler ce tableau :
  - buffer() pour connaître l'adresse du premier élément de ce tableau ;
  - nelements() pour connaître sa taille ;
  - element(i) pour accéder au i-ème élément ;
- etc.

L'interface du type d'image image2d_uint8 sera donc l'union des interfaces spécifiques apportées par l'ensemble des propriétés de ce type. À noter : on devine également les interfaces que ce type d'images n'a pas (mais que d'autres images auront) à la lecture de certaines propriétés ; par exemple :

- comme il n'y a "pas d'accès prévu pour accéder à tous les pixels ayant une valeur donnée", on n'aura pas de méthode sites(v) pour obtenir l'ensemble de points de valeur v ;

– comme la liste des “valeurs prises n’est pas connue”, on n’aura pas de méthode values() pour l’obtenir ;
– etc.

Enfin, il est très facile d’imaginer à travers la liste des propriétés qui décrivent le type d’images image2d_uint8 d’autres types d’images, véritablement différents, et qui partagerons certaines des propriétés de ce type.

Dans un soucis d’exactitude, nous devons mentionner que la règle générale "chaque propriété prise indépendamment se traduit par une interface propre à cette propriété" n’est pas toujours vérifiée. En effet, deux catégories de cas particuliers existent. Quelquefois, une propriété ne se traduit pas par la présence d’une interface (cela signifie que cette propriété n’apporte donc aucun outil supplémentaire de manipulation (type associé ou méthode). Cela peut survenir lorsqu’une propriété est d’ordre purement “sémantique”, par opposition à “opérationnelle” ; en reprenant l’exemple ci-dessus, le fait que les valeurs soient entières n’apporte en effet aucun élément d’interface particulier. La seconde exception à la règle générale est que, quelquefois, c’est un petit ensemble de propriété qui se traduit par la présence d’une interface. Cette dernière n’est donc pas due à une propriété en particulier mais à une conjonction de propriétés. Toujours avec le même exemple, l’“accès aux valeurs par un couple d’indices” se traduit par une méthode at(row, col) : value présente grâce à la conjonction de trois propriétés (grille régulière, valeurs en accès aléatoire et domaine bidimensionnel).

4.2.2 *Programmation déclarative*

Une propriété est de façon classique en informatique un couple : (nature, valeur). Comme les propriétés s’appliquent sur des types, et comme ces types sont connus à la compilation, les valeurs de propriétés peuvent également être statiques. La modélisation que nous avons choisie est la suivante : chaque nature de propriété est un espace de nom et ses valeurs sont des classes de cet espace. Prenons un exemple. Afin d’exprimer que l’accès aux valeurs des sites d’une image peut être soit en lecture seule soit en lecture-écriture, nous avons défini :

– l’espace de nom pw_io (pour *point-wise input-output*) qui traduit la nature de la propriété ;
– la classe pw_io::read pour la valeur “lecture seule” de la propriété ;
– et la classe pw_io::read_write pour la valeur “lecture-écriture”.

Afin d’exprimer la notion de valeur inconnue, nous avons opté pour de l’héritage de classes. La valeur inconnue se traduit en une classe dont dérive les classes de valeurs :

– la classe de base pw_io::**any** représente ainsi soit pw_io::read soit pw_io::read_write.

Au final on obtient :

```
pw_io : namespace is
{
  any : type is class {}
  read : type is class inherits any {}
  read_write : type is class inherits any {}
}
```

Le fait de s’appuyer sur de l’héritage de classes pour définir les propriétés permet en outre d’avoir des modélisations à plusieurs niveaux, certaines valeurs de propriétés en raffinant d’autres. Ainsi,

une hiérarchie de propriétés est facilement mise en œuvre. Voici un exemple aisé à comprendre et qui concerne la caractérisation des types d'images par la nature des données (les valeurs des pixels) :

```
kind : namespace is
{
  any : type is class {}
  color : type is class inherits any {}
  gray : type is class inherits any {}

  logic : type is class inherits any {}
  binary : type is class inherits logic {}
  ternary : type is class inherits logic {}
  ...
}
```

Dans la section 3.3.3 page 57, nous avons vu un paradigme orienté-objet simplifié, sur lequel nous allons nous appuyer pour prendre en compte explicitement dans le code la notion de propriétés. Dans ce paradigme, la classe abstraite *Image* est vide d'interface : elle ne comporte ni méthodes, ni types associés (elle se contente de vérifier statiquement des invariants). Les classes concrètes d'images dérivent de cette abstraction, définissent l'interface commune attendue de toute image et ajoutent leur part d'interfaces spécifiques :

```
Image : [I : type] -> type is
  class
  {
    !invariant ...
  }

image2d : [V : type] -> type is
  class inherits Image[ image2d[V] ] // le paramètre I vaut donc image2d[V]
  {
    // interface commune à toute image :

    + value : type is V
    + point : type is point2d
    ...
    + get : (p : point) -> value is ...
    + set self : (p : point) -> ref value is ...
    ...

    // interfaces spécifiques :

    + nsites : () -> unsigned is ...
    + buffer : () -> ref V is ...
    ...
  }
```

La définition d'une classe d'image doit maintenant être augmentée de la prise en compte des propriétés. Cela est réalisé sur un mode *déclaratif* (en programmation déclarative, on décrit le "quoi", par opposition à la programmation impérative où l'on décrit le "comment"). Les propriétés

d'un type d'images sont listées dans la spécialisation *ad hoc* d'une structure, property, et leurs valeurs sont définies par des types associés :

```
property : [ auto V : type, image2d[V] ] -> type is
  class
  {
    + pw_io_value : type is pw_io::read_write
    + kind_value : type is property [V]::kind_value
    ...
  }
```

L'accès à une propriété particulière revient alors à accéder au type associé correspondant. Ici, d'ailleurs, la définition de la propriété "nature (kind) de l'image" (image en niveaux de gris ou en couleur, etc.) se fait en lisant la propriété équivalente "nature" du type de valeurs contenues dans l'image. On a donc l'accès à la propriété property[V] : :kind_value ce qui permet la définition de la propriété property[Image[V]] : :kind_value. Donnons un second exemple. Pour savoir si une image de type I accepte la modification des valeurs de ses pixels, il suffit de réaliser un test :

```
!if (property[I]::pw_io_value = pw_io::read_write)
```

Cela se lit : "si la valeur de propriété d'accès aux valeurs pour le type I est égal à la valeur lecture-écriture." Ce test de l'égalité de deux types est résolu à la compilation, il s'agit donc d'un test *statique* (d'où le caractère ' !'). Le fait de concrétiser les propriétés de types explicitement dans le code a de très nombreux avantages. Les propriétés deviennent ainsi *manipulables* et il est alors possible d'écrire du code, des bouts de programmes, qui seront gérés intégralement à la compilation. Ces méta-programmes peuvent donc raisonner sur les propriétés des types ; dit autrement, on peut écrire des programmes qui tirent réellement partie de ces propriétés. Cette caractéristique n'est généralement offerte que par des langages dynamiques. Ici, nous en disposons dans un cadre statique ce qui présente un double avantage. D'une part, cela va nous permettre de contrôler, à la compilation donc, la validité du code ; ces vérifications portent aussi bien sur le code de la bibliothèque que sur celui d'un utilisateur de cette bibliothèque. D'autre part, il faut comprendre qu'en fait, nous allons pouvoir programmer le comportement du compilateur grâce à la manipulation des propriétés des types. Il s'agit bien qu'une caractéristique puissante (!) et rare, l'écrasante majorité des langages et environnements de programmation ne proposant pas de telles fonctionnalités.

Nous venons de voir comment déclarer les propriétés des types et nous disposons d'outils nous permettant de manipuler ces propriétés. Des exemples concrets de ces manipulations sont donnés par la suite en sections 4.3.2 et 4.3.3.

4.3 À PROPOS DES ALGORITHMES

Jusqu'à présent, dans ce manuscrit, nous avons plutôt orienté notre discours en restant guidé par les types d'images. L'écriture d'algorithmes n'a été alors considérée qu'à la lumière des différents paradigmes permettant d'implanter les types d'images. Dit autrement, telle écriture d'algorithme a été expliquée comme étant la conséquence de tel paradigme. Dans cette section, nous allons

renverser ce point de vue et commencer à discuter de l'impact de l'écriture d'algorithmes sur les types de données...

4.3.1 *Un cas d'école*

Reprenons notre exemple "fil rouge" d'algorithme, le remplissage d'une image par une valeur. Nous en étions resté à cette version générique, on ne peut plus compacte :

```
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void
is
  p : I::piter := ima.domain()
  for_all(p)
    ima(p) := v
  end
```

Jusqu'à présent, nous avons qualifié cette mise en œuvre de générique puisqu'elle accepte comme entrée tout type d'images. Nous pouvons considérer que c'est encore vrai... mais, d'un autre côté, de nombreuses bibliothèques offrent des *variations* de cet algorithmes. La plus commune, parce que la utile en traitement d'images, est de restreindre le domaine du remplissage à une *région* de l'image d'entrée. Les deux solutions les plus courantes sont la généralisation et la surcharge. La première, discutée en section 3.1.2, n'est pas raisonnable d'un point de vue du génie logiciel. Nous ne la détaillerons pas ici ; rappelons seulement qu'elle signifie obscurcir le code des types et des algorithmes sans être jamais complètement satisfaisante. La seconde solution revient à s'appuyer sur la surcharge pour ajouter à la routine existante une autre routine qui, elle, nécessite un masque. Cela donne le code additionnel ci-dessous :

```
fill : auto[I : type] (
  ima : ref Image[I],
  msk : I::site -> bool,
  v : I::value) -> void
is
  p : I::piter := ima.domain()
  for_all(p)
    if (msk(p) = true) // on teste ici l'appartenance de p au masque
      ima(p) := v
  end
```

Dans cette version du remplissage, le masque est donnée par un prédicat, fonction qui associe un booléen à chaque site. (Remarquons que cette formulation est plus large que la définition la plus employée, la définition d'un masque par une image binaire ; en effet, il est facile de voir une image comme une fonction et il y a plus de façons de définir des fonctions que des images.) Il y a toutefois une autre façon de réaliser le remplissage restreint à une région : en passant en argument un domaine caractérisant cette région. Une écriture additionnelle est alors la suivante :

```
fill : auto[I : type, S : type] (
  ima : ref Image[I],
  msk : Site_Set[S],
  v : I::value) -> void
is
```

```
  p : S::piter := msk
  // p itère maintenant sur le masque donc la boucle n'a pas besoin de réaliser de test
  for_all(p)
    ima(p) := v
  end
```

Une deuxième variation désirable survient lorsque les valeurs des pixels ne sont plus scalaires mais sont des structures ou des vecteurs. On trouve alors, dans les quelques bibliothèques les mieux dotées, la possibilité de n'effectuer le remplissage que pour une des composantes des valeurs de l'image. Cela s'écrit à l'aide d'une fonction de projection, a, qui prend une valeur par référence et renvoie une composante donnée par référence :

```
fill : auto[I : type, V : type] (
  ima : ref Image[I],
  v : V,
  a : ref I::value -> ref V) -> void
is
  p : I::piter := ima.domain()
  for_all(p)
    a(ima(p)) := v
  end
```

Ainsi, il est possible de réaliser l'appel fill(lena, 0, red) qui met à zéro la composante rouge de tous les pixels de l'image couleur Lena. Pour finir, nous allons présenter une troisième variation autour de l'algorithme de remplissage. Cette dernière étape a pour but de compléter un tour de cette notion de "variation". La première variation portait sur le domaine à traiter, la deuxième sur les valeurs, la troisième ne concerne ni le domaine ni les valeurs. Il s'agit de vouloir visualiser la progression de l'algorithme à l'aide d'une interface graphique. À chaque modification dans l'image, il faut alors rafraîchir la vue de l'image au sein de l'interface :

```
fill : auto[I : type] (
  ima : ref Image[I],
  v : I::value,
  gui : ima_gui) -> void
is
  p : I::piter := ima.domain()
  for_all(p)
    ima(p) := v
    gui.refresh() // rafraîchit la visualisation de l'image
  end
```

Si ce dernier exemple peut sembler ici ridicule, étant donné la simplicité du traitement, il n'est pas anodin. Déjà, pour des algorithmes plus trapus, pouvoir suivre pas à pas leur exécution est véritablement un plus. Mais au delà de l'exemple de la visualisation se cache une idée forte :

> nous devrions être capable d'instrumenter le code des algorithmes de façon non intrusive.

En effet, les trois variations présentées ne sont autres que des instrumentations du code originel de la routine fill donné initialement. Et le fait que ces instrumentations soient de natures différentes (puisqu'elles touchent respectivement le domaine, les valeurs et un artefact visuel de l'exécution) n'y change rien.

Considérons maintenant du point de vue du génie logiciel ce à quoi nous avons abouti. Nous avons ajouté trois variations à une routine originelle et, pour cela, nous nous sommes appuyés sur le mécanisme de surcharge. Plusieurs morceaux de code portent un même nom de routine mais, grâce à des signatures différentes (la liste typée de leurs arguments), peuvent coexister sans qu'il n'y ait d'ambiguïtés. Pour écrire ces routines, nous avons copié-collé-modifié la version originelle. Cela ressemble très fortement à la démarche donnée en section 3.1.1 et qui correspond à l'approche *exhaustive* de la généricité : pour gérer différents cas, écrivons-les tous. Nous aboutissons exactement aux mêmes défauts de cette approche : écrire tous les cas est non seulement fastidieux et sujet à de nombreuses erreurs, mais tout simplement impossible. En effet, la combinatoire de toutes les variations possibles est énorme, et *quid* des variations auxquelles on n'a pas encore pensé. En reprenant l'exemple ci-dessus et en y repensant, nous pourrions déjà nous demander pourquoi nous n'avons pas encore écrit de variantes qui mixent "restriction du domaine" et "accès à une composante des valeurs", et pourquoi pas aussi "restriction du domaine" et "visualisation de l'exécution", etc. Une bibliothèque qui n'offrirait que des variantes élémentaires mais pas leur combinaison serait frustrante pour l'utilisateur.

Finalement, retrouver ici les défauts de l'approche exhaustive de la généricité nous mène à nous interroger sur l'ensemble des cinq routines que nous venons voir. Avant tout, avons-nous bien un unique algorithme ? La réponse est positive : il s'agit d'*un* remplissage et le fait que ce remplissage puisse revêtir plusieurs formes n'est que la manifestation du caractère intrinsèquement abstrait des algorithmes. En poursuivant ce raisonnement, on aboutit à l'idée suivante. Vouloir rendre concret un algorithme n'est peut-être finalement que l'aspect visible d'une instrumentation. L'exemple manifeste de cela est le dernier, celui de l'ajout de la visualisation de l'exécution à l'aide d'une interface graphique. Mais si les premiers exemples n'étaient également que des instrumentations, alors cela signifierait que l'algorithme ci-dessous, tel qu'écrit, est véritablement *l'algorithme générique* du remplissage :

```
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void
is
  p : I::piter := ima.domain()
  for_all(p)
    ima(p) := v
  end
```

Si cela était vrai, nous n'aurions pas besoin d'écrire de variations de cet algorithme et, en corollaire, toute variation de cet algorithme serait possible sans lui faire perdre son intégrité.

Nous allons voir dans la section suivante qu'effectivement c'est le cas.

4.3.2 *Les morpheurs*

Ne pas toucher à l'écriture d'un algorithme mais pouvoir modifier son comportement n'est possible que d'une façon : changer les entrées de cet algorithme, ses arguments. Ainsi, nous pouvons imaginer d'affecter l'algorithme, de changer son comportement, de manière non intrusive.

Pour comprendre comment cela peut être effectif, nous allons reprendre les exemples de la section précédente. La seule routine présente dans la bibliothèque est celle donnée juste avant,

en fin de section. Ses entrées sont une image, ima, et une valeur, v. Afin de réaliser les variations suivantes :

- ne remplir que si un prédicat est vérifié ;
- ne remplir d'un sous-domaine ;
- ne remplir qu'une composante des valeurs ;
- rafraîchir une visualisation à chaque remplissage élémentaire,

il faut faire porter les informations nécessaires à un des deux arguments de la routine. Comme cela ne peut être la valeur, ce doit être obligatoirement l'image. Cette image doit avant tout rester une "vraie" image, au sens de toutes les autres fournies par la bibliothèque. Nous devons donc ré-exprimer les variations non plus du point de vue de l'algorithme mais du point de vue des images passées à l'algorithme ! Cela nous donne quatre nouvelles saveurs d'images :

- une image dont le domaine est restreint par un prédicat ;
- une image dont le domaine est réduit à un sous-domaine ;
- une image dont les valeurs "visibles" ne sont en fait qu'une composante des valeurs "effectives" ;
- une image qui rafraîchit une visualisation à chaque modification qu'elle subit.

Pour chacun de ces cas, il faut bien comprendre que le résultat des modifications donne une image. Par exemple, pour le premier cas, il faut donc lire : "une image qui correspond à une première image dont le domaine a été restreint par un prédicat".

Du point de vue de l'utilisateur, il s'agit donc de créer ces nouvelles images avant d'appeler l'algorithme ou au moment de l'appel de l'algorithme. Le code ci-dessous crée une image représentant un échiquier de 8×8 cases dont les couleurs sont alternativement rouge et verte :

```
chess : (p : point2d) −> bool is
  // chess renvoie un booléen suivant la parité des coordonnées
  return (p.row() + p.col()) mod 2 = 0

ima : image2d[rgb_3x8] := (8, 8) // dimensions de l'image
fill(ima | chess , green) // remplissage restreint aux cases où 'chess' renvoie 'vrai'
fill(ima | not chess , red) // idem mais pour les autres cases
```

Nous partons donc d'une première image, ima, dont nous restreignons le domaine à deux reprises. L'écriture "ima | chess" crée une *nouvelle* image, un nouvel objet. Cet objet représente "ima dont le domaine est restreint par le prédicat chess". C'est un objet dit *léger*, i.e., un objet qui ne prend pas beaucoup de mémoire. Schématiquement, les seules données (attributs) de cet objet sont deux pointeurs : un pointeur vers l'image de départ, ima, et un vers la fonction, chess. La création de cet objet ne coûte à peu près rien, ni en mémoire, ni en temps de création. La nouvelle image "ima | chess" ainsi créée n'est en fait qu'une *vue* de l'image de départ ima. Mais le nouvel "objet-image" est effectivement une image du point de vue de l'algorithme et, bien sûr, de tout ce que l'on est en droit d'attendre d'une "vraie" image. De façon similaire, voici comment l'utilisateur peut traiter les autres exemples :

```
fill(ima | box2d(0,0, 1,7), black) // remplissage restreint aux deux 1ères lignes
fill(blue(ima), 0) // toutes les composantes bleues des couleurs sont mises à 0
fill(display(ima), white) // visualisation du blanchiment
```

Ces images sont issues de la modification d'images existantes ; nous les appellerons *morpheurs* par la suite, car elles transforment des images en d'autres images. Il existe plusieurs catégories de morpheurs, comme le laisse supposer les exemples de la section précédente :

– les morpheurs qui affectent le domaine d'une image ;
– ceux qui affectent les valeurs d'une image ;
– ceux qui ne modifient pas vraiment une image mais lui ajoutent du comportement ou des données, donc des fonctionnalités ;

et enfin, une dernière catégorie :

– les morpheurs qui changent la façon dont la relation domaine/valeurs est gérée.

Un requis non trivial pour les morpheurs est que, dans un objectif omniprésent de généricité, ils doivent pouvoir s'appliquer sur un maximum de types d'images. En effet, pourquoi ne voudrait-on transformer que certains types d'images ? Par exemple, dans le cas de la visualisation, on veut pouvoir visualiser n'importe quelle image ; dans le cas de la restriction du domaine par un prédicat, cette restriction est applicable sur toute image. Aussi, les morpheurs doivent également être génériques, au même titre que le sont les types d'images de base, comme image2d[V], vis-à-vis de leur(s) paramètre(s), ici V.

Un morpheur est donc une fonction qui prend un type d'images et vraisemblablement des informations supplémentaires, et qui retourne un nouveau type d'images. Plus précisément, un morpheur, disons morpher, transforme un type d'images I en un type d'images morpher[I,...], où les points de suspension symbolisent ici les informations additionnelles statiques dont le morpheur a besoin. Un morpheur se traduit donc sous la forme d'un type paramétrique d'images, soit schématiquement :

```
morpher : [ I : type, ... ] −> type is
  class inherits Image[ morpher[I,...] ] // un morpher est un type d'images...
  {
    !invariant I inherits Image[?] // ...et son paramètre I est également un type d'images
    ...
  }
```

Et finalement, un type de morpheur ressemble en fait à n'importe quel autre type d'images.

Voyons maintenant comment les morpheurs peuvent se concrétiser dans l'environnement décrit jusqu'à présent, où le paradigme est un mélange simplifié d'orientation-objet et de staticité (Cf. la section 3.3.3) sur lequel s'ajoute une couche déclarative (Cf. section 4.2.2) pour exprimer la notion de propriétés. Pour cela, prenons pour illustration le morpheur "image restreinte à un sous-domaine".

Nous allons tout d'abord mettre en œuvre le type de ce morpheur ; nommons ce type sub_image, puisqu'il s'agit en fait du type d'une image résultante d'une restriction à un sous domaine, donc une sorte de sous-image. Ce type doit bien entendu fournir une implémentation de l'interface commune attendue de toute image :

```
sub_image : [ I : type, S : type ] −> type is
  class inherits Image[ sub_image[I,S] ]
  {
    // I est un type d'images et S est un type de domaines :
```

```
  !invariant I inherits Image[?] and S inherits Site_Set[?]


  // attributs :

  - ima : ref I // l'image transformée
  - s : S // le sous-domaine (la restriction du domaine de ima)


  // implémentation de l'interface commune :

  + domain_t : type is S // type du domaine
  + domain : () -> S is return s // retourne le domaine

  + site : type is S::site // type des sites (éléments du domaine)
  + value : type is I::value // type des valeurs

  + "()" : (p : site) -> value is // opérateur d'accès aux valeurs
      precondition s.has(p) and ima.domain().has(p)
      return ima(p)
    end

  + piter : type is S::piter // type d'un itérateur
}
```

Remarquons tout d'abord que cette implémentation est très courte et extrêmement simple. Déjà le morpheur conserve le même type de valeurs que l'image transformée, ima, référencée en attribut. Maintenant, le domaine du morpheur est celui donné par le sous-domaine s, stocké en attribut. L'accès à une valeur, via l'opérateur "()" doit tester que le site p passé en argument est bien un élément du sous-domaine s. Enfin, le type du sous-domaine, le paramètre S du morpheur, n'est pas obligatoirement celui de l'image initial (I : :domain_t) ; de ce (potentiellement nouveau) type se déduisent les types de sites et d'itérateurs. Passer une instance de sub_image[I,S] à la routine générique fill va changer le comportement de cette dernière. Pour lever toute ambiguïté avec les noms et types d'images, commençons par renommer l'image d'entrée et son type en les suffixant par un tiret bas :

```
fill : auto[I_ : type] (ima_ : ref Image[I_], v : I_::value) -> void
is
  p : I_::piter := ima_.domain()
  for_all(p)
    ima_(p) := v
  end
```

Si l'image d'entrée est de type sub_image[I,S], nous avons I_ = sub_image[I,S] et ima_ est l'instance de morpheur dont les attributs sont ima, l'image dont on va restreindre le domaine, et s, le sous-domaine (la restriction). On peut maintenant *substituer* à I_ et aux appels de méthodes sur ima_ leur définition. Il vient :

```
fill : auto[I, S : type] (ima_ : ref Image[ sub_image[I,S] ], v : I::value) -> void
is
  p : S::piter := ima_.s // on va itérer sur le sous-domaine
```

```
  for_all(p)
    ima_.ima(p) := v // on modifie la valeur d'un pixel de ima
  end
```

De plus, en *éliminant* l'objet léger morpheur ima_, pour ne garder que les deux objets à l'origine de sa création, l'image originale ima et le sous-domaine s, ce code se transforme finalement en :

```
fill : auto[I, S : type] ( ( ima : ref Image[I], s : Site_Set[S] ) , v : I::value) -> void
is
  p : S::piter := s // on va itérer sur le sous-domaine
  for_all(p)
    ima(p) := v // on modifie la valeur d'un pixel de ima
  end
```

Les deux étapes que nous venons de voir comprennent, à l'appel de la routine fill, la substitution des types, la mise en ligne des méthodes et l'élimination de l'objet léger morpheur. Ces trois mécanismes sont en fait véritablement réalisés *automatiquement* par le compilateur. La routine fill générique qui sera exécutée lors de l'appel donné en exemple plus haut et repris ici :

```
fill(ima | box2d(0,0, 1,7), black) // remplissage restreint aux deux 1ères lignes
```

sera donc exactement équivalent au code que nous venons de transformer à la main en une routine à trois arguments (ima, s et v) et à deux paramètres (I et S). Pour mémoire, rappelons la variation algorithmique de fill que nous avions écrite explicitement avec un masque défini par un sous-domaine :

```
fill : auto[I : type, S : type] (
  ima : ref Image[I],
  msk : Site_Set[S],
  v : I::value) -> void
is
  p : S::piter := msk
  for_all(p)
    ima(p) := v
  end
```

Le code générique de fill, appelé avec un morpheur de restriction à un sous-domaine, est donc en fait strictement équivalent au code de la variation algorithmique. Aussi, grâce à ce type de morpheur, nous n'avons plus besoin d'écrire ce type de variation. Mieux, ce qui est valable ici pour fill l'est aussi pour tous les autres algorithmes. Nous venons de réaliser l'équation suivante :

Des algorithmes + un morpheur = une variation algorithmique pour chacun de ces algorithmiques.

En termes de combinatoire, avec A algorithmes et M morpheurs, cela donne $A \times M$ variations algorithmiques élémentaires (c'est-à-dire à l'aide d'un unique morpheur, sans combiner les morpheurs en eux) qui s'ajoutent aux A algorithmes originaux...

Nous venons de découpler les variations algorithmes des algorithmes originaux et, au final, une variation se traduit par un morpheur. Ce dernier étant générique, la variation l'est aussi : elle est applicable à tout algorithme original. Enfin, notons que la généricité du morpheur vis-à-vis de

l'image sur lequel il s'applique (qu'il transforme), permet aussi de multiplier les structures de données / les types d'images. En effet, comme un morpheur peut s'appliquer *a priori* sur n'importe quel type d'image, il peut être utilisé sur un autre morpheur (par exemple, on peut restreindre le domaine d'un image dont on ne voit qu'une composante de ses valeurs). Sous les hypothèses simplificatrices (certaines favorables, d'autres défavorables) suivantes :

- si l'on applique à une image plusieurs morpheurs, on supposera que l'ordre des morpheurs n'a pas d'importance (ce qui n'est pas toujours vrai) ;
- on ignore le fait qu'un même morpheur peut être appliqué plusieurs fois à une image ;
- on suppose que toutes les combinaisons font sens et sont possibles (ce qui n'est généralement pas vrai) ;
- enfin, on ne prend pas en compte la possibilité d'avoir des informations additionnelles de différents types (les paramètres autres que I) pour un même morpheur ;

avec M morpheurs et $D = S \times V$ types primaires d'images, on peut potentiellement construire :

$$D \times \sum_{k=0}^{M} C_M^k \quad = \quad D \times 2^M \text{ types d'images différents.}$$

En corollaire, le fait de combiner des morpheurs offrent aussi 2^M variations possibles pour chaque routine. Ce chiffre est n'est pas précisément juste à cause des hypothèses simplificatrices ; néanmoins, il donne un *ordre de grandeur correcte*.

Dans la pratique, il n'est pas raisonnable de considérer qu'un type d'images soit l'"empilement" (la conjugaison) des M morpheurs différents offerts par la bibliothèque ; dit autrement, k ne va pas jusqu'à M. Néanmoins, à l'usage, il est très courant d'utiliser jusqu'à 2 morpheurs sur un image initiale, courant d'en empiler 3, et il nous est arrivé d'en cumuler 5.

Finalement, on peut retenir que la combinatoire de types, donc de variations algorithmiques, est énorme. La notion de morpheur décuple donc incroyablement l'ensemble des possibilités offertes par une bibliothèque.

Afin de compléter la description et l'aspect implémentatoire des morpheurs, il faut parler de nouveau des propriétés. En effet, elles jouent un rôle primordial dans l'obtention effective des morpheurs. Contrairement à ce que le cours du discours pourrait le laisser supposer, ce n'est *pas* parce qu'un morpheur est une image qu'on doit définir ses propriétés mais c'est parce qu'en fait on doit déclarer les propriétés des types d'images *pour* pouvoir avoir des morpheurs ! L'explication de cela est relativement simple. Nous avons mis l'accent en section 4.2.1 sur la présence et la nécessité des interfaces spécifiques or un morpheur est générique donc *a priori* ne sait pas gérer les spécificités.

Si nous reprenons le code du morpheur sub_image, nous pouvons réaliser que seule l'interface commune à toutes les images a été implémentée :

```
sub_image : [ I : type, S : type ] -> type is
  class inherits Image[ sub_image[I,S] ]
  {
    ...
    // implémentation de l'interface commune :
    ...
  }
```

ce qui est normal car il s'agit de la seule interface que l'on est sure de retrouver dans tous les types d'images. De par sa propre généricité, le type sub_image[I,S] va avoir des propriétés qui dépendent en fait de ses paramètres, donc de celles du type d'images I et du type d'ensemble de sites S. Prenons un exemple. Si I est une image dont les valeurs sont accessibles respectivement en lecture seule ou en lecture-écriture, il en sera de même pour ce morpheur ; cette propriété se transmet du type I vers le type du morpheur. Dans le cas où l'image initiale est en lecture-écriture, le morpheur doit donc avoir une interface spécifique additionnelle pour cela. Cette interface ne peut être mise en œuvre dans la définition de la classe du morpheur car ce serait du code invalide dans le cas contraire où I est en lecture seule... Un raisonnement similaire s'applique pour toute autre propriété.

Ainsi nous n'avons pas d'autres choix que d'implémenter les interfaces spécifiques en dehors des classes de morpheurs et, au final, les classes de morpheurs doivent alors être augmentées automatiquement [2] des implémentations des interfaces spécifiques. Cette récupération automatique de code est rendue possible grâce à la déclaration des propriétés. Commençons par cette déclaration.

```
property : [ auto I, S : type, sub_image[I,S] ] -> type is
  class
  {
    + pw_io_value : type is property [I]::pw_io_value
    + ext_domain_value : type is ext_domain::none
    ...
  }
```

Dans cet exemple, nous déclarons que la propriété d'accès aux valeurs (lecture seule ou lecture-écriture) pour le type de morpheur provient de (égale à) la propriété du type I de l'image initiale. De la même façon, d'autres propriétés du type de morpheur proviendront de celles du type d'images I, mais d'autres proviendront du type de sous-domaines S.

La seconde déclaration de propriété en revanche est une particularité de ce type de morpheur, les types I et S n'ayant ici aucune influence. Cette seconde propriété concerne la nature de l'extension du domaine de définition des images. Une *extension* est la généralisation de la notion de "bord extérieur" du domaine de l'image. Cette notion est très pratique lorsque l'on parcourt le voisinage d'un site et, ce faisant, que l'on sort du domaine de définition ; en présence d'une extension, on peut alors effectivement "sortir" du domaine sans que le comportement soit erroné. En fait, ce comportement, ou effet de bord, est très utile (voire indispensable dans certains cas). Le fait de restreindre le domaine de définition d'une image, ce que fait le morpheur sub_image, efface toute extension éventuellement présente dans l'image initiale. Ce qui explique la déclaration "ext_domain_value is none".

En conclusion, la déclaration des propriétés d'un morpheur s'écrit librement, et nous avons même la possibilité de définir des règles plus ou moins évoluées (des petits bouts de méta-programmes) afin de fixer les valeurs de propriétés. Dans la pratique, nous n'avons pas observé de difficultés, ni d'ambiguïtés, lors de la déclaration des propriétés des morpheurs.

La partie magique provient de la récupération automatique des implémentations spécifiques. Afin d'insister sur cette caractéristique souhaitée, rappelons d'une part que ce sont les interfaces

2. Bienvenu dans le monde merveilleux de la méta-programmation dans lequel nous pouvons, en quelque sorte, "programmer" le compilateur pour qu'il nous obéisse : ici, en fonction de certains types (les valeurs de propriétés), nous feront récupérer du code automatiquement aux classes de morpheurs !

spécifiques qui forgent l'identité de chaque type d'image. D'autre part, laisser de côté les interfaces spécifiques aurait une saveur amère pour l'utilisateur. En effet, avec le type classique d'images bidimensionnelles, l'utilisateur peut accéder aux valeurs des pixels à l'aide d'un couple d'indices :

```
print( ima.at(0,0) ) // OK, rien de surprenant
```

alors comment lui expliquer que, si l'on restreint l'image à ses deux premières lignes (ci-dessous en créant l'image légère ima'), ce même accès lui serait interdit :

```
ima' : ? := ima | box2d(0,0, 1,7) // restriction aux deux 1ères lignes
print( ima'.at(0,0) ) // KO !!!
```

Abandonner les interfaces spécifiques, c'est se priver des fonctionnalités naturelles qu'un utilisateur est en droit d'attendre. Il est donc nécessaire de récupérer les implémentations *ad hoc*, ce qui est réalisable avant tout grâce à de l'héritage d'implémentation. Une classe de morpheur va hériter d'une classe particulière, nommément fetch_image_impl, qui se chargera d'aller récupérer les implémentations attendues :

```
sub_image : [ I : type, S : type ] -> type is
  class inherits Image[ sub_image[I,S] ], fetch_image_impl[ sub_image[I,S] ]
  {
    ...
  }
```

Ce qui se cache derrière la classe paramétrique sub_image est un méta-programme dont la description précise, assez touffue et technique, sort du cadre de ce manuscrit ; Cf. les détails dans [17] porté en annexe B.3. En revanche, son fonctionnement peut être résumé. La classe fetch_image_impl[M] hérite elle-même de plusieurs classes, chacune ayant dédiée à une propriété en particulier et ayant pour but de fournir, si besoin, l'implémentation de l'interface spécifique idoine. Pour cela, les propriétés du type M du morpheur sont introspectées. Ce mécanisme assure qu'à la compilation toute classe de morpheur est bien complète.

Pour terminer, notons que, du point de vue du génie logiciel, la nature varie d'un morpheur à l'autre. Certains sont des décorateurs, d'autres des ponts, d'autres des vues, encore d'autres des enveloppes, etc. Dans [16] et [10], nous avons présenté comment ces différentes notions pouvaient se traduire dans le paradigme orienté-objet muni de polymorphisme paramétrique. *In fine*, nous avons abouti à une unique solution pour la mise en œuvre des morpheurs dans la bibliothèque MILENA.

4.3.3 *Les algorithmes rencontrent les types*

Jusqu'à présent, nous avons parlé d'"algorithmes" et, quelquefois, nous avons employé le terme "routine" (ou son synonyme "procédure"). Il faut maintenant préciser un certain nombre de notions liées aux algorithmes.

Nous appellerons un *opérateur* une entité abstraite qui réalise une opération dont la sémantique est connue théoriquement. Cette entité correspond exactement à ce que l'on entend dans l'expression "opérateur de traitement d'images". Par exemple, la *reconstruction géodésique par dilatation d'une fonction* est un opérateur appartenant à la famille de la morphologie mathématique. Par opposition,

FIGURE 9.: Quatre algorithmes pour un même opérateur.

```
Rd_parallel(f : Image, g : Image)
  → o : Image
begin
  var
    o' : Image
    stability : 𝔹

  o := f
  repeat
    o' := o
    for_all p ∈ E
      o(p) := max{ o'(q) |
                   q ∈ 𝒩(p) ∪ {p} }
    for_all p ∈ E
      o(p) := min{ o(p), g(p) }
    stability := (o = o')
  until stability
  return o

end
```

```
Rd_sequential(f : Image, g : Image)
  → o : Image
begin
  var
    o' : Image
    stability : 𝔹

  o := f
  repeat
    o' := o
    for_all p ∈ E ▷
      o(p) := min{ max{ o(q) |
                   q ∈ 𝒩⁻(p) ∪ {p} }, g(p) }
    for_all p ∈ E ◁
      o(p) := min{ max{ o(q) |
                   q ∈ 𝒩⁺(p) ∪ {p} }, g(p) }
    stability := (o = o')
  until stability
  return o

end
```

```
Rd_queue_based(f : Image, g : Image)
  → o : Image
begin
  var
    q : Queue of Point
    M : Image

  M := regional_maxima(f)
  for_all p ∈ M
    for_all n ∈ 𝒩(p)
      if n ∉ M
        q.push(p)
  o := f
  while not q.empty()
    p := q.first()
    for_all n ∈ 𝒩(p)
      if o(n) < o(p) and o(n) ≠ g(n)
        o(n) := min{o(p), g(n)}
        q.push(n)
  return o
end
```

```
Rd_hybrid(f : Image, g : Image)
  → o : Image
begin
  var
    q : Queue of Point

  o := f
  for_all p ∈ E ▷
    o(p) := min{ max{ o(q) |
                 q ∈ 𝒩⁻(p) ∪ {p} }, g(p) }
  for_all p ∈ E ◁
    o(p) := min{ max{ o(q) |
                 q ∈ 𝒩⁺(p) ∪ {p} }, g(p) }
    // avec mise en queue
    for_all n ∈ 𝒩⁺(p)
      if o(n) < o(p) and o(n) < g(n)
        q.push(p)
  while not q.empty()
    p := q.first()
    for_all n ∈ 𝒩(p)
      if o(n) < o(p) and o(n) ≠ g(n)
        o(n) := min{o(p), g(n)}
        q.push(n)
  return o
end
```

nous appellerons *algorithme*, une séquence d'instructions permettant la mise en œvre d'un opérateur. Généralement, *un* même opérateur se traduit par *plusieurs* algorithmes. Voyons tout de suite un exemple concret. La définition mathématique de reconstruction géodésique par dilatation est la suivante :

$$\mathcal{R}_g^{\delta}(f) = \lim_{n\to\infty} \delta_g^n(f) = \delta_g^{\infty}(f)$$
$$\text{où } \delta_g^1(f) = \delta(f) \bigwedge g \text{ et } \delta_g^{n+1}(f) = \delta(\delta_g^n(f)) \bigwedge g.$$

et quatre algorithmes vraiment différents, réalisant cet opérateur, sont donnés par la figure 9. Une remarque importante : ces algorithmes sont véritablement interchangeables, i.e., ils traitent rigoureusement les mêmes types d'images (nous verrons par la suite que certains algorithmes ne sont *pas* interchangeables). Une bibliothèque doit être capable de fournir différents algorithmes et de permettre à l'utilisateur d'appeler un algorithme en particulier. Si l'utilisateur appelle "juste" l'opérateur, sans préciser une implémentation particulière, alors cet appel est re-dirigé vers l'algorithme le plus approprié (le plus efficace généralement). La routine qui sert à l'appel de l'opérateur sera appelée "façade" et, en fait, il s'agit bien du point d'entrée utilisé le plus couramment pour accéder aux algorithmes. L'utilisateur ne veut pas toujours savoir dans le détail ce qu'il se passe lorsqu'il a besoin d'un opérateur : il se contente le plus souvent de l'appeler et fait confiance à la bibliothèque pour que l'opération correspondante soit exécutée au mieux. Au final, l'organisation "opérateur / algorithmes" se traduit par ce code schématique :

```
namespace impl is
{
  // les différentes implémentations d'algorithmes
  Rd_parallel : ...
  Rd_sequential : ...
  Rd_queue_based : ...
  Rd_hybrid : ...
}


Rd : (f, g) ... // façade de l'opérateur
is
  return impl::Rd_hybrid(f, g) // re-direction vers l'algorithme le plus efficace
end
```

Nous avons vu que les propriétés sont utiles pour vérifier qu'un type implémente correctement toute ses interfaces et pour réussir à mettre en œuvre la notion de types morpheurs. Où les propriétés sont également intéressantes, c'est pour *instrumenter* les algorithmes. Nous allons voir deux bénéfices apportés par les propriétés aux algorithmes.

Contrairement à l'exemple précédent, certains algorithmes ne fonctionnent que sur certains types d'images. Par exemple, pour indiquer qu'une routine comme fill, qui écrit des valeurs dans une image, n'est valide qu'avec des images dont les valeurs des pixels sont modifiables, il suffit alors seulement d'ajouter un test statique de la valeur de la propriété correspondante :

```
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void
is
  !invariant property[I]::pw_io_value = pw_io::read_write
```

```
p : I::piter := ima.domain()
for_all(p)
  ima(p) := v
end
```

L'ajout de tests statiques basés sur les propriétés permet au compilateur de signaler des erreurs à l'utilisateur. Qu'une routine soit une façade d'opérateur ou une implémentation d'algorithmes, de telles vérifications empêchent l'utilisateur de faire n'importe quoi.

Mais il y a mieux. Il arrive qu'un opérateur n'ait pas la même implémentation suivant que les types en entrée vérifient telles ou telles propriétés. Reprenons la routine fill ci-dessus, et observons l'affectation qui est au cœur de la boucle :

```
p : I::piter := ima.domain()
for_all(p)
  ima(p) := v
end
```

Est-elle possible sur tout type I d'images ? Non puisque les types d'images dont la propriété d'accès ponctuel vaut **pw_io**::**read_only** n'acceptent pas une telle affectation. Maintenant, allons plus loin : pouvons-nous imaginer des types d'images dont la modification de la valeur d'un pixel n'est pas raisonnablement possible mais pour lesquels le remplissage puisse être opérable ? Mais oui, et il y a de nombreux types qui vérifient ce cas. Nous en citons trois.

– Un type d'images dont tous les pixels partagent la même valeur peut être encodé à l'aide d'*une* valeur (*la* valeur) et de son domaine de définition. Pour une telle image, on ne peut pas changer la valeur d'un pixel indépendamment des autres pixels, ce qui est la sémantique de l'affectation "ima(p) := v". Pourtant il est très facile de remplir cette image à l'aide de la valeur v puisqu'il s'agit d'écrire :

  ```
  ima.the_value() := v // tout simplement !
  ```

– Un deuxième type d'images intéressant pour le cas présent correspond à un encodage des valeurs par plage (LRE, pour *run-length encoding*). Au lieu d'avoir en mémoire une valeur par pixel, les données sont stockées sous la forme d'une suite de couples "valeur et longueur de plage" (notons que de nombreuses variantes de ce stockage peuvent être imaginées). Ces images n'autorisent pas l'accès en modification de valeur pour chaque pixel pris indépendamment car, non seulement ce serait trop coûteux en mise à jour de la structure de plages, mais aussi, on perdrait alors totalement l'intérêt de l'utilisation de cette structure. Néanmoins, il est très facile de remplir de telles images avec une unique valeur : pour chaque plage, on met à jour la valeur.

– Un dernier type d'images qui tombe dans les cas attendus est relativement classique (mais quasiment jamais implémenté dans les bibliothèques de traitement d'images...) : les images dont les valeurs sont définies avec une table de correspondances, en anglais une *look-up table* ou LUT. Les valeurs "en mémoire" sont stockées sous la forme d'un tableau d'indices et la LUT donne pour chaque indice sa "vraie" valeur correspondante. Accepter une affectation élémentaire (au sens d'un pixel indépendant) telle "ima(p) := v" signifierait tout d'abord parcourir la LUT pour savoir si la valeur v y été représentée. Dans la positive, il faut changer la valeur "en mémoire" à p avec l'indice correspondant. Dans la négative, il faut changer la LUT

(en imaginant que ça soit possible). Un accès élémentaire n'est donc pas réalisable en temps constant ; ce type d'images n'a pas la propriété pw_io : :read_write. En revanche, affecter une nouvelle valeur à *tous* les pixels, ce qui est le sens de fill, est tout à fait possible.

On remarque facilement que les trois exemples donnés ci-dessus partagent une même propriété : le stockage des valeurs des pixels de l'images n'est pas géré par un ensemble d'associations "point - valeur" mais, à l'opposé, par des associations "valeur - ensemble de points". Cette propriété est traduite dans la bibliothèque par vw_io, qui signifie qu'il existe des accès *value-wise*, c'est-à-dire, via les valeurs (par opposition aux accès *point-wise*, via les points). En particulier, l'interface spécifique liée à cette propriété propose les outils pour itérer sur l'ensemble des valeurs que liste ces images (à bien distinguer de l'ensemble des valeurs des pixels). Pour reprendre les trois exemples précédents, une itération sur ces valeurs cible respectivement :

- la seule valeur de l'image constante ;
- les valeurs des plages de l'image ;
- et les valeurs de la table de correspondance.

Comme dans le cas *point-wise*, la propriété de l'accès *value-wise* peut valoir soit lecture seule soit lecture-écriture. En effet, certains types d'images pourront interdire de modifier leur valeurs.

Finalement, nous aurons donc une implémentation particulières pour les types d'images en accès valeur, fill_vw , qui s'ajoutera à celle qui procède en accès ponctuel, fill_pw :

```
namespace impl is
{
  fill_pw : auto[I : type] (ima : ref Image[I], v : I::value) -> void
  is
    !invariant property[I]::pw_io_value = pw_io::read_write
    p : I::piter := ima.domain()
    for_all(p)
      ima(p) := v
  end

  fill_vw : auto[I : type] (ima : ref Image[I], v : I::value) -> void
  is
    !invariant property[I]::vw_io_value = vw_io::read_write
    val : I::viter := ima.values()
    for_all(val)
      val := v
  end
}
```

La façade correspondante à l'opérateur de remplissage doit donc choisir d'appeler l'implémentation *ad hoc* en fonction des propriétés de l'image d'entrée. Ce choix pourra être réalisé à la compilation puisque les propriétés sont statiques. Cette disjonction s'écrit :

```
fill : auto[I : type] (ima : ref Image[I], v : I::value) -> void
is
  !if property[I]::vw_io_value = vw_io::read_write
     impl::fill_vw(ima, v)
   else
   if property[I]::pw_io_value = pw_io::read_write
     impl::fill_pw(ima, v)
```

```
  else
    error
end
```

Le fait que le choix d'une implémentation soit statique est en fait fort heureux. En effet, chaque implémentation tire partie de l'interface spécifique correspondant à une propriété donc toute implémentation est ici très spécifique : une image dont le type est compatible avec fill_pw ne peut pas être passée à fill_vw, et vice-versa. Si l'embranchement vers les implémentations avait été réalisé avec un test dynamique (un simple IF, par opposition à un !IF statique), un unique appel à fill aurait entraîné la compilation de toutes les branches, et cette compilation se serait terminée sur une erreur dans la branche où le type d'image ne convient pas.

L'utilisation statique des propriétés permet donc d'avoir des implémentations incompatibles (du point de vue de leurs entrées) derrière une façade unique.

Se permettre de proposer plusieurs implémentations peut également servir des objectifs de performances. En effet, certaines images offrent des interfaces spécifiques liées aux optimisations possibles. C'est le cas des types classiques, comme image2d, où les valeurs des pixels sont en RAM dans une zone mémoire contiguë. La bibliothèque fournit alors tous les outils pour potentiellement exploiter tout gain en performances. On y lira par exemple les deux implémentations ci-dessous :

```
namespace impl is
{
  fill_pw_fast : auto[I : type] (ima : ref Image[I], v : I::value) -> void
  is
    !invariant ...
       and property[I]::speed_value = speed::fast
    pxl : I::pixter := ima
    for_all(pxl)
      pxl.val() := v // affectation au pixel (un pointeur se cache dans l'objet pxl)
  end

  fill_pw_fastest : auto[I : type] (ima : ref Image[I], v : I::value) -> void
  is
    !invariant ...
       and property[I]::storage_value = storage::one_block
       and sizeof(I::value) = 8
    memset(ima.buffer(), ima.nelements(), v) // affectation bas-niveau d'une zone mémoire
  end
}
```

La première effectue une itération, non plus sur les points (les sites), mais sur les pixels. L'itérateur pxl est lié à une image en particulier et contient un pointeur vers la cellule mémoire de la valeur du pixel qu'il représente. La boucle déroule donc un déplacement de pointeur et la méthode val() permet de dé-référencer le pointeur pour changer la valeur du pixel correspondant. La seconde implémentation est encore plus particulière que la précédente, puisqu'elle s'appuie sur la routine de bas-niveau memset.

Contrairement à tout ce qui a été vu jusqu'à présent, nous utilisons ici les propriétés afin de *spécialiser* les algorithmes (et ce, accessoirement, afin de garantir les meilleures performances

à l'exécution). Il ne s'agit donc plus d'avoir plusieurs algorithme pour un même opérateur mais d'avoir plusieurs écritures d'un même algorithme.

Enfin, remarquons que dans tous les exemples montrés dans cette section, les algorithmes présentés, y compris les algorithmes spécialisés, restent toujours les "plus génériques possibles."

Troisième partie

LA FIN (?)

5 CONCLUSION

Et on peut touſiours reduire ainſi toutes les quantités inconnuës à vne ſeule, lorſque le Probleſme ſe peut conſtruire par des cercles & des lignes droites, ou auſſy par des ſections coniques, ou meſme par quelque autre ligne qui ne ſoit que d'vn ou deux degrés plus compoſée. Mais ie ne m'areſte point a expliquer cecy plus en detail, a cauſe que ie vous oſterois le plaiſir de l'apprendre de vous meſme, & l'vtilité de cultiuer voſtre eſprit en vous y exerceant, qui eſt a mon auis la principale, qu'on puiſſe tirer

La Géometrie (Livre Premier), René Descartes, 1637.

Dans ce rapport, nous avons présenté un travail mené en filigrane de nos autres activités (Cf. le *curriculum vitæ* à part) : comment réaliser une bibliothèque de traitements d'images qui soit véritablement *générique*. Pour mesurer la difficulté de ce sujet, très loin d'une "simple" problématique d'ingénierie, cette citation peut nous aider :

> *Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.*
>
> —Alexander Stepanov,
> conversation avec Bjarne Stroustrup [34].

En effet, l'obtention d'une généricité satisfaisante est une quête à la fois ardue et sans fin car *la* généricité n'existe pas. Nous devrions parler de *degré de généricité*. De ce point de vue-là, les bibliothèques existantes restent plutôt pauvres : très peu d'entre peuvent gérer de multiples types d'images de façon vraiment transparente et aucunes ne proposent l'équivalent de ce nous offrons dans la bibliothèque MILENA.

Une explication de cet état de fait est donnée très tôt dans ce rapport, par le préambule en section 1.1. La situation est paradoxale : ce sujet n'est, ni tout à fait du traitement d'images, ni seulement de l'informatique. Il s'en suit que les outils "bibliothèques" tels que nous les connaissons ne sont absolument pas satisfaisants. Nous avons donc mené une réflexion sur ce à quoi devrait ressembler une bibliothèque idéale (section 2.1) et nous en avons déduit des objectifs (section 2.2). L'un des plus importants d'entre eux, la généricité, est détaillé en section 2.3. Il s'agit d'essayer de repousser les frontières du possible, d'avoir un outil qui ne nous limite pas. Le but de notre travail est de démontrer qu'il n'est pas "normal", comme le font nombre d'utilisateurs de bibliothèques, de considérer qu'un tel outil soit difficile à manier et obligatoirement limité. L'utilisateur n'a pas à accepter de souffrir à cause de l'informatique, ni à accepter les limitations de ses outils logiciels. Il doit apprendre que l'état de l'art des connaissances en informatique lui permet maintenant de sortir de cette situation.

Afin d'apporter des solutions, nous avons étudié différents moyens d'accéder à de la généricité. Le chapitre 3 est consacré à cette étude. Quatre paradigmes ont été examinés (section 3.1) et discutés (section 3.2).

Nous avons vu que l'approche exhaustive, la plus utilisée dans les plus "vieilles" bibliothèques de traitement d'images, ne permet pas d'écrire de façon unique les opérateurs de traitement. Ce défaut, bien trop pénalisant, est rédhibitoire pour une bibliothèque "moderne". Pour l'approche par généralisation et ses variantes, la notion d'image se traduit par une *unique* classe concrète (ou une *unique* famille de classes concrètes lorsqu'un ou plusieurs paramètres sont introduits). L'aspect univoque de la généralisation impose un moule figé bien trop rigide pour tolérer toute extension de la notion d'image. L'approche par paramétrisation, quant à elle, est séduisante mais fait l'impasse sur la matérialisation de la notion abstraite d'image. Cette absence de réification est handicapante car elle prive de typage fort les signatures des opérateurs et, par voie de conséquence, interdit l'utilisation "naturelle" de surcharge pour les fonctions et méthodes. Enfin, l'approche orientée-objet classique, avec classe abstraite et inclusion, est rejetée car pénalisante en terme de performances. En revanche, nous avons montré (en section 3.3) que le mélange entre inclusion et paramétrisation conduit à un paradigme qui propose le meilleur des deux mondes : matérialisation des abstractions *sans* perte de performances à l'exécution. Afin que ce paradigme statique soit exploitable en pratique, nous en avons proposé une simplification.

Cette étude est présentée sous un angle qui est plutôt original pour deux raisons. Tout d'abord, nous avons voulu nous adresser en priorité à la communauté du traitement d'images : le discours est illustré pour ce domaine en particulier et les aspects informatiques—paradigmes, typage et génie logiciel—sont expliqués de façon à rester abordables pour le traiteur d'images [1]. Le sujet que constitue "l'informatique au service du traitement d'images" est plutôt original car il n'est pas très "recherché". On dénombre en effet moins de 10 références sur le sujet depuis les 30 dernières années (précisément : [35, 25, 32, 3, 33, 8, 21, 22, 24]). La seconde originalité est d'avoir tenté de mener toutes nos explications du point de vue des *types* et des mises en œuvre qui en découlent. Par exemple, pour la seule notion d'abstraction de ce qu'est une image, nous avons étudié différentes

1. L'informatique *hard-core* a été circonscrite au loin, dans l'annexe B.

TABLE 3.: Matérialisations de la notion d'image.

approche	**code**	**matérialisation**
exhaustivité		*néant*
généralisation	image3d	classe unique
généralisation / paramétrisation	image3d[V]	classe unique paramétrée
généralisation par paramétrisation	image[V,D...]	classe unique multi-paramétrée
paramétrisation	I	paramètre
inclusion	*image*	classe abstraite
inclusion / paramétrisation	*image*[V]	classe abstraite paramétrée
approche proposée	*image*[I]	classe abstraite statiquement

matérialisations (Cf. le tableau récapitulatif 3) et nous avons étudié l'impact de ces différentes représentations sur l'écriture des types concrets d'images et des algorithmes. Une des particularité de cette approche "orientée types" a été de considérer l'ensemble des types d'images dans un espace dont les *axes* portent des sémantiques différentes (Cf. par exemple la figure 8 page 64).

Notre travail avons abouti à plusieurs conclusions parmi lesquelles trois sont particulièrement importantes : abstraire est important ; il faut mélanger le meilleur des différents paradigmes ; les types d'images doivent rester simples (avec peu de paramètres).

La suite de notre travail, chapitre 4, s'est inscrit dans la continuité naturelle de la décomposition des types sur différents axes. Nous nous sommes alors demandé ce qu'est vraiment une image en essayant de capturer les notions antagonistes que sont leur trait commun et leur diversité (section 4.1). Cette sorte de psychanalyse introspective de traiteur d'images donc. Nous avons alors montré la nécessité d'avoir une interface commune et de multiples interfaces spécifiques. Pour concrétiser ces interfaces, nous sommes arrivés à la conclusion que nous avions besoin d'une taxonomie des images par leurs propriétés (section 4.2). Les propriétés d'un type expriment la spécificité de ce type et lui confèrent une identité, ce qui contraste singulièrement avec les deux notions vues précédemment : l'abstraction et la généralisation. Nous avons alors proposé une solution à la mise en œuvre des propriétés mêlant héritage et programmation *déclarative*.

Le point d'orge de ce rapport a été de renverser notre perspective. Au lieu de considérer les types et de voir leurs influences sur les algorithmes, nous avons dressé des *desiderata* pour les opérateurs de traitement d'images et nous avons regardé ce que cela impliquait sur les types d'images. La conclusion à laquelle nous sommes parvenu est heureuse : l'existence du système de propriétés nous permet en effet d'obtenir toutes les caractéristiques attendues (section 4.3. Nous avons montré :

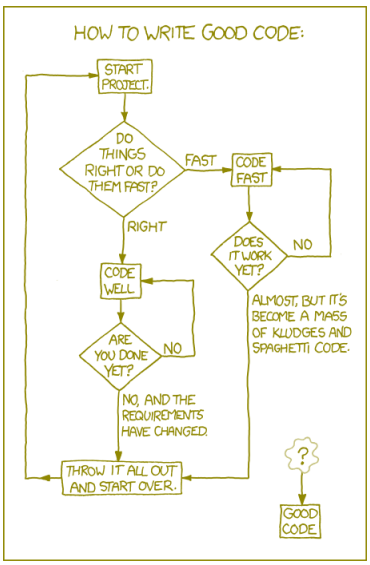
- que nous pouvions bénéficier de typage fort pour les procédures de traitement ;
- que nous pouvions implémenter, pour un même opérateur, plusieurs algorithmes, même si ces derniers sont "incompatibles" du point de vue du typage ;
- que, lors de l'appel à un traitement, un mécanisme de sélection automatique de l'implémentation *ad hoc* (en fonction du type des données en entrée) pouvait être mis en place et résolu à la compilation ;

– et enfin, qu'il existe une solution pour ne pas compromettre les performances en temps d'exécution.

Ainsi, nous avons vérifié la pertinence du multi-paradigme "orienté-objet, statique et déclaratif", proposé pour la définition des *types d'images*, vis-à-vis des *opérateurs et algorithmes* de la bibliothèque. Dit autrement, la pertinence de ce multi-paradigme a donc été validée également *a posteriori*.

6

PERSPECTIVES



xkcd *#844, Randall Munroe, 2011* (© CC BY-NC).

Le travail présenté dans ce rapport est un corps en mouvement ; comme pour tout sujet de recherche et tout logiciel, il est sain qu'il mature, mue et évolue. Les perspectives de ce travail s'inscrivent sur quatre axes relativement distincts :

- la formalisation ;
- la transposition ;
- l'utilisation ;
- la refonte.

LA FORMALISATION. La suite la plus évidente et la plus naturelle du travail présenté ici est la définition précise des concepts du domaine du traitement d'images et des interfaces associées (globales et spécifiques). Cette formalisation pourra être validée par une mise en œuvre dans la bibliothèque d'un catalogue très exhaustif de types concrets correspondants. On peut noter que

plusieurs autres chercheurs, également auteurs d'outils logiciels, sont intéressés par "faire converger" les définitions des propriétés et des interfaces pour le domaine métier du traitement d'images.

LA TRANSPOSITION. Les solutions proposées ici pour construire une bibliothèque générique en s'appuyant sur un multi-paradigme (orienté-objet, générique et déclaratif) ne sont pas limitées au traitement d'images. Une perspective ouverte est la transposition de nos propositions de construction logicielle à un autre domaine métier que le traitement d'images. En effet, elles sont générales ("paradigmiques" ?) et peuvent s'appliquer à tout domaine scientifique où les algorithmes comportent du calcul intensif et où les données en entrée peuvent être de différentes natures.

L'UTILISATION. L'aspect ultra-générique de notre bibliothèque de traitement d'images facilite grandement l'exploration de nouvelles voies de recherche. Effectivement, il est possible de réutiliser les algorithmes et méthodes déjà disponible en remplaçant la nature "classique" des entrées à traiter par des structures, topologies et/ou types de valeurs moins classiques. On peut ainsi tirer des ponts entre des entités, méthodes et types de données, qui ne sont pas généralement utilisés ensembles.

LA REFONTE. Les résultats d'une taxonomie des entités du domaine du traitement d'images ont une portée générale ; ils ne dépendent pas de choix d'implémentation liés à un langage en particulier. Précisément, le fait d'avoir choisi le langage C++ pour la réalisation de notre bibliothèque n'a été guidé il y a dix ans que par deux raisons : ce langage est répandu (et proche du C donc accessible aux "anciens" programmeurs) et, surtout, il offre un large spectre de possibilités dû à son aspect "multi-paradigme". Cette dernière caractéristique se retrouve maintenant dans des occurrences modernes de langages. Mener des expérimentations, par exemple avec D [1] ou Scala [30], qui fournissent l'éventail suffisant de caractéristiques pour nous permettre de reproduire nos résultats, peut mener à des propositions de paradigmes *ad hoc* pour le calcul scientifique.

Ces quatre axes de perspectives de notre travail reflètent la pluri-disciplinarité du travail déjà mené. Ils proposent également un autre chemin que celui qui lie "*are you done yet ?*" à "*throw it all out and start over*".

Quatrième partie

ANNEXES

A

MINI-LANGAGE

Le mini-langage utilisé dans ce rapport est décrit dans les pages qui suivent ; on y trouve une description organisée ainsi :

- les mots-clefs (tableau 4, page 102) ;
- les bases de sa syntaxe (tableau 5, page 103) ;
- les fonctions et classes (tableau 6, page 103) ;
- et ses aspects statiques (tableau 7, page 104).

TABLE 4.: Mots-clefs du mini-langage.

généraux :

end	fin d'un bloc d'instructions
class	introduit la définition d'une classe
later	pré-déclaration (la définition est retardée, repoussée plus loin)
FIXME	FIXME : à continuer...

types particuliers :

type	type d'un type (!)
any	n'importe quel type
?	un type mais on ne sait pas lequel ou on ne veut pas l'écrire
typeof(<expr>)	type de l'expression <expr>

modifieurs de types :

array of <size> <type>	tableau de
ref <type>	référence à

types prédéfinis :

bool	booléen
int	entier signé
uint	entier non signé
int$n$ et uint$n$	entiers sur n bit
float	flottant
double	flottant double précision

échappements :

return	retour d'une fonction
break	sortie d'une boucle
continue	passage à l'itération suivante

opérateurs :

=	égalité
not =	inégalité
<	infériorité stricte
< or =	infériorité large

TABLE 5.: Syntaxe du mini-langage : les bases.

instruction	syntaxe
ajout d'un commentaire <txt> en fin de ligne	// <txt>
omission de code	...
bloc d'instructions (CR : retour chariot)	<instr1> CR <instr2> ... CR end
plusieurs instructions sur la même ligne	<instr1> ; <instr2> ...
déclaration d'une variable <var> de type <type>	<var> : <type>
définition par <def> d'une variable <var> de type <type>	<var> : <type> is <def>
affectation de la valeur <val> à la variable <var>	<var> := <val>
déclaration abstraite (pas de définition) de l'entité <ent>	<ent> is abstract
liste	<item1>, <item2> ...
accès au <i> ème élement d'un tableau <a>	<a>@<i>

TABLE 6.: Syntaxe du mini-langage : fonctions et classes.

instruction	syntaxe
déclaration d'une fonction <fun> prenant la liste d'arguments <arglist> et ayant <ret> comme type de retour	<fun> : (<arglist>) -> <ret>
absence d'argument ou de retour	void
appel d'une fonction <fun> avec les arguments <args>	<fun>(<args>)
retour de l'expression <expr> par une fonction	return <expr>
définition d'une classe par sa liste de champs <flds>	{ <flds> }
déclaration d'une méthode <m> impérative	<m> ref : <type>
constructeur	make : (<arglist>) -> self is ...
l'objet lui-même	self
la classe immédiatement supérieure	super
déclaration d'un opérateur <o>	"<o>" : <type>
déclaration de l'accessibilité <a> d'un champ <fld> (respectivement +, +- et - pour publique, protégée et privée)	<a> <fld>
déclaration d'héritage d'une classe mère <super>	class inherits <super>
accès à un champ <fld> à partir d'une instance <obj> d'une classe	<obj>.<fld>
accès à un champ-type <typ> d'une classe <cls>	<cls>::<typ>

TABLE 7.: Syntaxe du mini-langage : staticité.

instruction	syntaxe
déclaration d'une liste de paramètres formels <parmlist> pour une entité <ent>	<ent> [<parmlist>]
déclaration de valuation automatique d'un paramètre formel <parm>	auto <parm>
déclaration de valuation automatique de toute une liste de paramètres <parmlist>	auto [<parmlist>
appel d'une entité paramétrée <ent> avec les paramètres <parms>	<ent>[<parms>]
version statique d'une entité <ent>	!<ent>

B

ARTICLES SUR L'INFORMATIQUE

Cette annexe présente trois articles dont la saveur est purement informatique. Ils illustrent le fait que les résultats obtenus au cours de notre travail ont fait l'objet de publications dans la communauté informatique (hors traitement d'images donc).

- L'annexe B.1 (page 106) recopie un article de 2001 qui présente des motifs de conception (*design patterns*) correspondants à la traduction statique de motifs connus.
- Le mélange entre le paradigme classique orienté-objet et le paradigme de programmation générique, nommé SCOOP, est expliqué dans un article long de 2003, donné en annexe B.2 (page 121).
- Enfin, un article long de 2008 introduit la notion de propriétés mais dans le cadre du paradigme SCOOP ; cet article est repris en annexe B.3 (page 158).

B.1 DESIGN PATTERNS FOR GENERIC PROGRAMMING IN C++

Design Patterns for Generic Programming in C++. *USENIX Conference on Object-Oriented Technologies and Systems* (**COOTS**), Jan-Feb 2001. pp. 189-202, San Antonio, Texas, USA. With Alexandre Duret-Lutz and Akim Demaille.

Design Patterns for Generic Programming in C++

Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
{Alexandre.Duret-Lutz,Thierry.Geraud,Akim.Demaille}@lrde.epita.fr
http://www.lrde.epita.fr/

Abstract

Generic programming is a paradigm whose wide adoption by the C++ community is quite recent. In this scheme most classes and procedures are parameterized, leading to the construction of general and efficient software components. In this paper, we show how some design patterns from Gamma *et al.* can be adapted to this paradigm. Although these patterns rely highly on dynamic binding, we show that, by intensive use of parametric polymorphism, the method calls in these patterns can be resolved at compile-time. In intensive computations, the generic patterns bring a significant speed-up compared to their classical peers.

1 Introduction

This work has its origin in the development of Olena [11], our image processing library. When designing a library, one wants to implement algorithms that work on a wide variety of types without having to write a procedure for each concrete type. In short, one algorithm should be *generic* enough to map to a single procedure. In object-oriented programming this is achieved using *abstract types*. *Design Patterns*, which are design structures that have often proved to be useful in scientific computing, rely even more on abstract types and inclusion polymorphism[1].

However, when it comes to numerical computing, object-oriented designs can lead to a *huge performance loss*, especially as there may be a high number of virtual functions calls [7] required to perform operations over an abstraction. Yet, rejecting design patterns for the sake of efficiency seems radical.

In this paper, we show that some design patterns from Gamma *et al.* [10] can be adapted to generic programming. To this aim, virtual functions calls are avoided by replacing inclusion polymorphism by parametric polymorphism.

This paper presents patterns in C++, but, although they won't map directly to other languages because "genericity" differs from language to language, our work does not apply only to C++: our main focus is to devise flexible designs in contexts where efficiency is critical. In addition, C++ being a multi-paradigm programming language [28], the techniques described here can be limited to critical parts of the code dedicated to intensive computation.

In section 2 we introduce generic programming and present its advantages over classical object-oriented programming. Then, section 3 presents and discusses the design of the following patterns: GENERIC BRIDGE, GENERIC ITERATOR, GENERIC ABSTRACT FACTORY, GENERIC TEMPLATE METHOD, GENERIC DECORATOR, and GENERIC VISITOR. We conclude and consider the perspectives of our work in section 4.

2 Generic programming

By "generic programming" we refer to a use of parameterization which goes beyond simple genericity on data types. Generic programming is an abstract and efficient way of designing and assembling components [15] and interfacing them with algorithms.

[1] *Inclusion polymorphism* corresponds to virtual member functions in C++, deferred functions in Eiffel, and primitive functions in Ada.

Generic programming is an attractive paradigm for scientific numerical components [12] and numerous libraries are available on the Internet [22] for various domains: containers, graphs, linear algebra, computational geometry, differential equations, neural networks, visualization, image processing, etc.

The most famous generic library is probably the *Standard Template Library* [26]. In fact, generic programming appeared with the adoption of STL by the C++ standardization committee and was made possible with the addition of new generic capabilities to this language [27, 21].

Several generic programming idioms have already been discovered and many are listed in [30]. Most generic libraries use the GENERIC ITERATOR that we describe in 3.2. In POOMA [12] — a scientific framework for multi-dimensional arrays, fields, particles, and transforms — the GENERIC ENVELOPE-LETTER pattern appears. In the REQUESTED INTERFACE pattern [16], a GENERIC BRIDGE is introduced to handle efficiently an adaptation layer which mediates between the interfaces of the servers and of the clients.

2.1 Efficiency

The way abstractions are handled in the object-oriented programming paradigm ruins the performances, especially when the overhead implied by the abstract interface used to access the data is significant in comparison with the time needed to process the data.

For example, in an image processing library in which algorithms can work on many kinds of aggregates (two or three dimensional images, graphs, etc.), a procedure that adds a constant to an aggregate may be written using the object-oriented programming paradigm as follows.

```
template< class T >
void add (aggregate<T>& input, T value)
{
  iterator<T>& iter = input.create_iterator ();
  for (iter.first(); !iter.is_done(); iter.next())
    iter.current_item () += value;
}
```

Here, **`aggregate<T>`** and **`iterator<T>`** are abstract classes to support the numerous aggregates available: parameterization is used to achieve genericity on pixel types, and object-oriented abstractions are used to get genericity on the image structure.

	dedicated C	classical C++	generic C++
`add`	10.7s	37.7s	12.4s
`mean`	47.3s	225.8s	57.5s

Table 1: Timing of algorithms written in different paradigms. (The code was compiled with **`gcc 2.95.2`** and timed on an AMD K6-2 380MHz machine running GNU/Linux.)

As a consequence, for each iteration the direct call to **`T::operator+=()`** is drowned in the virtual calls to **`current_item()`**, **`next()`** and **`is_done()`**, leading to poor performances.

Table 1 compares classical object-oriented programming and generic programming and shows a speed-up factor of 3 to 4. The **`add`** test consists in the addition of a constant value to each element of an aggregate. The **`mean`** test replaces each element of an aggregate by the mean of its four neighbors. The durations correspond to 200 calls to these tests on a two dimensional image of 1024×1024 integers. "Dedicated C" corresponds to handwritten C specifically tuned for 2D images of integers, so the difference with classical C++ is what people call *the abstraction penalty*. While this is not a textbook case —we do have such algorithms in Olena— it is true that usually the impact of object-oriented abstraction is insignificant. High speed-ups are obtained from generic programming compared to object-oriented programming when data processing is cheap relatively to data access. For example for simple list iteration or matrix multiplication.

The generic programming writing of this algorithm, using a GENERIC ITERATOR, will be given in section 3.2.

2.2 Generic programming from the language point of view

Generic programming relies on the use of several programming language features, some of which being described below.

Genericity is the main way of generalizing object-oriented code. Not all languages support both generic classes and generic procedures (e.g., Eiffel features only generic classes).

Nested type names refers to the ability to look up a type as member of a class and allow to link related types (such as **`image2d`** and **`iterator2d`**)

together.

Constrained genericity is a way to restrict the possible values of formal parameters using signatures (e.g., when using ML functors [19]) or constraining a type to be a subclass of another (as in Eiffel or Ada 95). C++ does not provide specific language features to support constrained genericity, but subclass constraints [29, 23] or feature requirements [18, 25] can be expressed using other available language facilities [27].

Generic specialization allows the specialization of an algorithm (e.g., dedicated to a particular data type) overriding the generic implementation.

Not all languages support these features, this explains why the patterns we present in C++ won't apply directly to other languages.

2.3 Generic programming guidelines

From our experience in building Olena, which is entirely carried out by generic programming, we derived the following guidelines. These rules may seem drastic, but their appliance can be limited to critical parts of the code dedicated to intensive computation.

Guidelines for generic classes:

- Avoid inclusion polymorphism.
  In other words, the type of a variable (static type, known at compile-time) is exactly that of the instance it holds (dynamic type, known at run-time). The main requirement of generic programming is that *the concrete type of every object is known at compile-time*.
- Avoid operation polymorphism.
  Abstract methods are forbidden: dynamic binding is too expensive. Simulate operation polymorphism with either: (i) parametric classes thanks to the *Curiously Recurring Template* idiom (see section 3.4), or (ii) parametric methods, which lead to a form of *ad-hoc* polymorphism (overloading).
- Use inheritance only to factor methods and to declare attributes shared by several subclasses.

Guidelines for procedures which use generic patterns:

- Parameterize the procedures by the types of their inputs, even if the input itself is parameterized.
- Parameterize the procedures by the types of the components used (unless they can be obtained by a nested type lookup in another parameter-type).

3 Generic Design Patterns

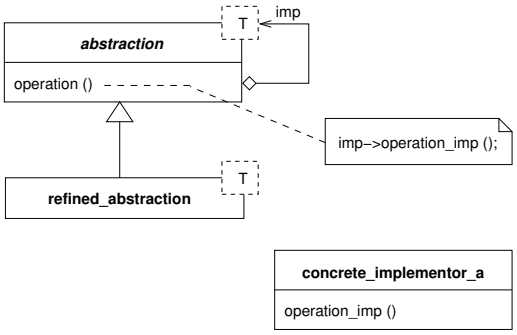
Our generic design patterns exposition is Gamma *et al.*'s description of the original, abstract version of the patterns [10]. We do not repeat the elements that can be found in this book.

3.1 Generic Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Structure



Participants

An ***abstraction*** class is parameterized by the ***Implementation*** used. Any (low-level) operation on the abstraction is delegated to the implementation instance.

Consequences

Because the implementation is statically bound to the abstraction, you can't switch implementation at runtime. This kind of restriction is common to generic programming: configurations must be known at compile-time.

Known Uses

This pattern is really straightforward and broadly used in generic libraries. For example the **`allocator`** parameter in STL containers is an instance of GENERIC BRIDGE.

The POOMA team [5] use the term *engine* to name implementation classes that defines the way matrices are stored in memory. This is also a GENERIC BRIDGE.

The Ada 95 rational [14, section 12.6] gives an example of GENERIC BRIDGE: a generic empty package (also called signature) is used to allow multiple implementation of an abstraction (here, a *mapping*).

As in the case of the original patterns, the structure of this pattern is the same as the GENERIC STRATEGY pattern. These patterns share the same implementation.

3.2 Generic Iterator

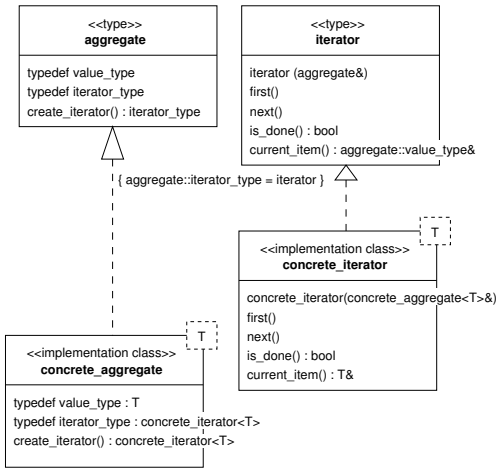
Intent

To provide an *efficient* way to access the elements of an aggregate without exposing its underlying representation.

Motivation

In numeric computing, data are often aggregates and algorithms usually need to work on several types of aggregate. Since there should be only one implementation of each algorithm, procedures must accept aggregates of various types as input and be able to browse their elements in some unified way; *iterators* are thus a very common tool. As an extra requirement compared to the original pattern, iterations must be efficient.

Structure



We use **`typedef`** as a non-standard extension of UML [24] to represent type aliases in classes.

Participants

The term *concept* was coined by M. H. Austern [1], to name a set of requirements on a type in STL. A type which satisfies these requirements is a *model* of this concept. The notion of concept replaces the classical object-oriented notion of abstract class.

For this pattern, two concepts are defined: *aggregate* and *iterator*, and two concrete classes model these concepts.

Consequences

Since no operation is polymorphic, iterating over an aggregate is more efficient while still being generic. Moreover, the compiler can now perform additional optimizations such as inlining, loop unrolling and instruction scheduling, that virtual function calls hindered.

Efficiency is a serious advantage. However we lose the dynamic behavior of the original pattern. For example we cannot iterate over a tree whose cells do not have the same type[2].

[2] A link between an abstract aggregate and the corresponding generic procedures can be achieved using lazy compilation and dynamic loading of generic code [8].

Implementation

Although a concept is denoted in UML by the stereotype <<type>>, in C++ it does not lead to a type: a concept only exists in the documentation. Indeed the fact that concepts have no mapping in the C++ syntax makes early detection of programming errors difficult. Several tricks have been proposed to address this issue by explicitly checking that the arguments of an algorithm are models of the expected concepts [18, 25]. In Ada 95, concept requirements (types, functions, procedures) can be captured by the formal parameters of an empty generic package (the *signature* idiom) [9].

For the user, a type-parameter (such as ***Aggregate_Model*** in the sample code) represents a model of *aggregate* and the corresponding model of *iterator* can then be deduced statically.

Sample Code

```
template< class T >
class buffer
{
  public:
    typedef T data_type;
    typedef buffer_iterator<T> iterator_type;
  // ...
};
```

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename Aggregate_Model::iterator_type&
  iter = input.create_iterator ();

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

Known Uses

Most generic libraries, such as STL, use the GENERIC ITERATOR.

Variations

We translated the Gamma *et al.* version, with methods ***first()***, ***is_done()***, and ***next()*** in the iterator class. STL uses another approach where pointers should also be *models* of iterators: as a consequence, iterators cannot have methods and most of their operators will rely on methods of the container's class. This makes implementation of multiple schemes of iteration difficult: for example compare a forward and a backward iteration in STL:

```
container::iterator i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

container::reverse_iterator i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

First, the syntax differs. From the STL point of view this is not a serious issue, because iterators are meant to be passed to algorithms as instances. For a wider use, however, this prevents parametric selection of the iterator (i.e., passing the iterator as a type). Second, you have to implement as many ***xbegin()*** and ***xend()*** methods as there are schemes of iteration, leading to a higher coupling [17] between iterators and containers.

Another idea consists in the removal of all the iterator related definitions, such as ***create_iterator()*** or ***iterator_type***, from ***concrete_aggregate<T>*** in order to allow the addition of new iterators without modifying the existing aggregate classes [32]. This can be achieved using *traits classes* [20] to associate iteration schemes with aggregates: the iterated aggregate instance is given as an argument to the iterator constructor. For example we would rewrite the ***add()*** function as follows.

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename forward_iterator< Aggregate_Model >::type
  iter (input);

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

This eliminates the need to declare iterators into the aggregate class, and allows further additions of iteration schemes by the simple means of creating a new traits class (for example ***backward_iterator<T>***).

3.3 Generic Abstract Factory

Intent

To create families of related or dependent objects.

Motivation

Let us go back over the different iteration schemes problem discussed previously. We want to define several kind of iterators for an aggregate, and as so we are candidates for the ABSTRACT FACTORY pattern. The STL example can be rewritten as follows to make this pattern explicit: iterators are *products*, built by an aggregate which can be seen as a *factory*.

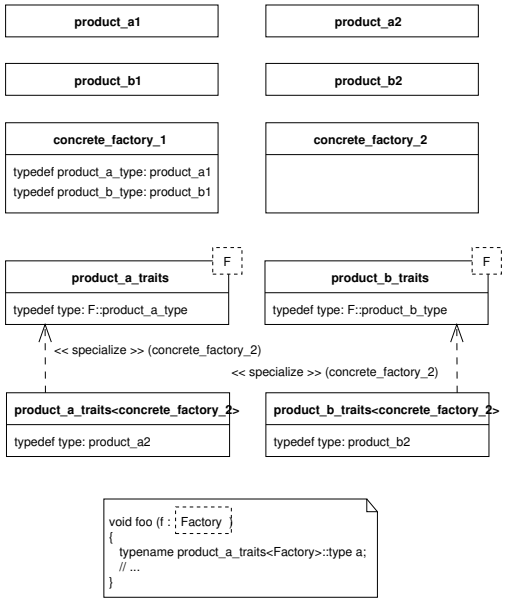
```
factory_a::product_1 i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

factory_a::product_2 i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

Implementing a GENERIC ABSTRACT FACTORY is therefore just a matter of defining the product types in the classes that should be used as a factory. This is really simpler than the original pattern. Yet there is one significant difference in usage: an ABSTRACT FACTORY returns an *object* whereas a GENERIC ABSTRACT FACTORY returns a *type*, giving more flexibility (e.g. constructors can be overloaded).

We have shown that if we want to implement multiple iteration schemes, it is better to use traits classes, to define the schemes out of the container. A trait class if a GENERIC ABSTRACT FACTORY too (think of `trait::type` as `factory::product`). But one issue is that these two techniques are not homogeneous. Say we want to add a new iterator to the STL containers: we cannot change the container classes, therefore we define our new iterator in a traits, but now we must use a different syntax whether we use one iterator or the other.

The structure we present here takes care of this: both internal and external definitions of products can be made, but the user will always use the same syntax.

Structure



Here, we represent a parametric method by boxing its parameter. For instance, ***F***actory is a type-parameter of the method ***A***ccept. This does cannot conform to UML since UML lacks support for parametric methods.

Participants

We have two factories, named `concrete_factory_1` and `concrete_factory_2` which each defines two products: `product_a_type` and `product_b_type`. The first factory define the products intrusively (in it's own class), while the second do it externally (in the product's traits).

To unify the utilization, the traits default is to use the type that might be defined in the "factory" class. For example the type `a` defined in `foo<Factory>`, defined as `product_a_trait<Factory>::type` will equal to `concrete_factory_1::product_a_type` in the case `Factory` is `concrete_factory_1`.

Consequences

Contrary to the pattern of Gamma, inheritance is no longer needed, neither for factories, nor for products. Introducing a new product merely requires adding a new

parametrized structure to handle the types aliases (e.g., ***product_c_traits***), and to specialize this structure when the alias ***product_c_type*** is not provided by the factory.

Known Uses

Many uses of this pattern can be found in STL. For example all the containers whose contents can be browsed forwards or backwards[3] define two products: forward and backward iterators.

The actual type of a list iterator never explicitly appears in client code, as for any class name of concrete products. Rather, the user refers to ***A::iterator***, and ***A*** is an STL container used as a concrete factory.

3.4 Generic Template Method

Intent

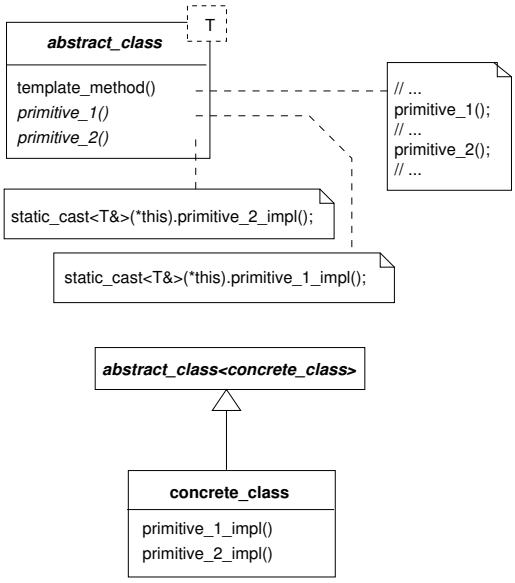
To define the canvas of an *efficient* algorithm in a superior class, deferring some steps to subclasses.

Motivation

In generic programming, we limit inheritance to factor methods [section 2.3]; here, we want a superior class to define an operation some parts of which (primitive operations) are defined only in inferior classes. As usual we want calls to the primitive operations, as well as calls to the template method, to be resolved at compile-time.

[3] vectors, doubly linked lists and dequeues are models of this concept, named *reversible containers*

Structure



Participants

In the object-oriented paradigm, the selection of the target function in a polymorphic operation can be seen as a search for the function, browsing the inheritance tree upwards from the dynamic type of the object. In practice, this is done at run-time by looking up the target in a table of function pointers.

In generic programming, we want that selection to be solved at compile-time. In other words, each caller should statically know the dynamic type of the object from which it calls methods. In the case of a superior class calling a method defined in a child class, the knowledge of the dynamic type can be given as a template parameter to the superior class. Therefore, any class needing to know its dynamic type will be parameterized by its leaf type.

The parametric class ***abstract_class*** defines two operations: ***primitive_1()*** and ***primitive_2()***. Calling one of these operations leads to casting the target object into its dynamic type. The methods executed are the implementations of these operations, ***primitive_1_impl()*** and ***primitive_2_impl()***. Because the object was cast into its leaf type, these functions are searched for in the object hierarchy from the leaf type up as desired.

When the programmer later defines the class ***concrete_class*** with the primitive operation implementations, the method ***template_method()*** is inherited and a call to this method leads to the execution of the proper implementations.

Consequences

In generic programming, operation polymorphism can be simulated by "parametric polymorphism through inheritance" and then be solved statically. The cost of dynamic binding is avoided; moreover, the compiler is able to inline all the code, including the template method itself. Hence, this design is more efficient.

Implementation

The methods ***primitive_1()*** and ***primitive_2()*** do not contain their implementation but a call to an implementation; they can be considered as *abstract methods*. Please note that they can also be called by the client without knowing that some dispatch is performed.

This design is made possible by the typing model used for C++ template parameters. A C++ compiler has to delay its semantic analysis of a template function until the function is instantiated. The compiler will therefore accept the call to ***T::primitive_1_impl()*** without knowing anything about ***T*** and will check the presence of this method later when the call to the ***A<T>::primitive_1()*** is actually performed, if it ever is. In Ada [13], on the contrary, such postponed type checking does not exist, for a function shall type check even if it is not instantiated. This pattern is therefore not applicable *as is* in this language.

One disadvantage of this pattern over Gamma's implementation is directly related to this: the compiler won't check the actual presence of the implementations in the subclasses. While a C++ compiler will warn you if you do not supply an implementation for an abstract function, even if it is not used, that same compiler will be quiet if pseudo-virtual operations like ***primitive_1_impl()*** are not defined and not used. Special care must thus be taken when building libraries not to forget such functions since the error won't come to light until the function is actually used.

We purposely added the suffixes ***_impl*** to the name of primitives to distinguish the implementation functions. One could image that the implementation would use the same name as the primitive, but this require some additional care as the abstract primitive can call itself recursively when the implementation is absent.[4]

Sample Code

The following code shows how to define a ***get_next()*** operation in each iterator of a library of containers. Obviously, ***get_next()*** is a template method made by issuing successive calls to the ***current_item()*** and ***next()*** methods of the actual iterator.

We define this method in a superclass ***iterator_common*** parametrized by its subtype, and have all iterators derive from this class.

```
template< class Child, class Value_Type >
class iterator_common
{
 public:
  Value_Type& get_next () {
   // template method
   Value_Type& v = current_item ();
   next ();
   return v;
  }
  Value_Type& current_item () {
   // call the actual implementation
   static_cast<Child&>(*this).current_item_impl();
  }
  void next () {
   // call the actual implementation
   static_cast<Child&>(*this).next_impl();
  }
};


// sample iterator definition
template< class Value_Type >
class buffer_iterator: public
  iterator_common< buffer_iterator< Value_Type >,
                   Value_Type >
{
  public:
    Value_Type current_item_impl () { ... };
    void next_impl () { ... };
    void first () { ... };
    void is_done () { ... };
    // ...
};
```

Known Uses

This pattern relies on an idiom called *Curiously Recurring Template* [4] derived from the *Barton and Nackman*

[4] You can ensure at compile-time that two functions (the primitive and its implementation) are different by passing their addresses to a helper template specialized in the case its two arguments are equal.

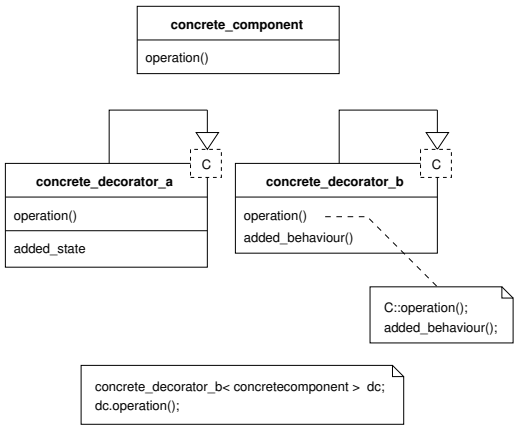
Trick [2]. In [2] this idiom is used to define a binary operator (for instance `+`) in a superior class from the corresponding unary operator (here `+=`) defined in an inferior class. Further examples are given in [30].

3.5 Generic Decorator

Intent

To *efficiently* define additional responsibilities to a set of objects or to replace functionalities of a set of objects, by means of subclassing.

Structure



We use a special idiom: having a parametric class that derives from one of its parameters. This is also known as *mixin inheritance*[5] [3].

Participants

A class ***concrete_component*** which can be decorated, offers an operation ***operation()***. Two parametric decorators, ***concrete_decorator_a*** and ***concrete_decorator_b***, whose parameter is the decorated type, override this operation.

[5] Mixins are often used in Ada to simulate multiple inheritance [14].

Consequences

This pattern has two advantages over Gamma's. First, any method that is not modified by the decorator is automatically inherited. While Gamma's version uses composition and must therefore delegate each unmodified operation. Second, decoration can be applied to a set of classes that are not related via inheritance. Therefore, a decorator becomes truly generic.

On the other hand we lose the capability of dynamically adding a decoration to an object.

Sample Code

Decorating an iterator of STL is useful when a container holds structured data, and one wants to perform operations only on a field of these data. In order to access this field, the decorator redefines the data access operator ***operator*()*** of the iterator.

```
// A basic red-green-blue struct
template< class T >
struct rgb
{
  typedef T red_type;
  red_type red;

  typedef T green_type;
  green_type green;

  typedef T blue_type;
  blue_type blue;
};


// An accessor class for the red field.
template< class T >
class get_red
{
  public:
    typedef T input_type;
    typedef typename T::red_type output_type;

    static output_type&
    get (input_type& v) {
      return v.red;
    }

    static const output_type&
    get (const input_type& v) {
      return v.red;
    }
};
```

Note how the ***rgb<T>*** structure exposes the type of each attribute. This makes cooperation between objects easier: here the ***get_red*** accessor will look up the ***red_type*** type member and doesn't have to know that fields of ***rgb<T>*** are of type ***T***. ***get_red*** can therefore

apply to any type that features ***red*** and ***red_type***, it is not limited to ***rgb<T>***.

```
// A decorator for any iterator
template< class Decorated,
          template< class > class Access >
class field_access: public Decorated
{
 public:
  typedef typename Decorated::value_type value_type;
  typedef Access< value_type >           accessor;
  typedef typename accessor::output_type output_type;

  field_access () : Decorated () {}
  field_access (const Decorated& d) : Decorated (d) {}

  // Overload operator*, use the given accessor
  // to get the proper field.
  output_type& operator* () {
    return accessor::get (Decorated::operator* ());
  }

  const output_type& operator* () const {
    return accessor::get (Decorated::operator* ());
  }
};
```

field_access is a decorator whose parameters are the types of the decorated iterator, and of a helper class which specifies the field to be accessed. Actually, this second parameter is an example of the GENERIC STRATEGY pattern [6, 30].

```
int main ()
{
  typedef std::list< rgb< int > > A;
  A input;
  // ... initialize the input list ...

  // Build decorated iterators.
  field_access< A::iterator, get_red >
    begin = input.begin (),
    end   = input.end ();
  // Assign 10 to each red field.
  std::fill (begin, end, 10);
}
```

The ***std::fill()*** procedure is a standard STL algorithm which assigns a value to each element of a range (specified by two iterators). Since ***std::fill()*** is here given decorated iterators it will only assign red fields to 10.

Note that the decorator is independent of the decorated iterator: it can apply to any STL iterator, not only ***list<T>::iterator***. The ***std::fill()*** algorithm will use methods of ***field_access*** inherited from the decorated iterator, such as the assignment, comparison, and pre-increment operators.

Known Uses

Parameterized inheritance is also called *mixin inheritance* and is one way to simulate multiple inheritance in Ada 95 [14]. This can also be used as an alternate way for providing *template methods* [6].

3.6 Generic Visitor

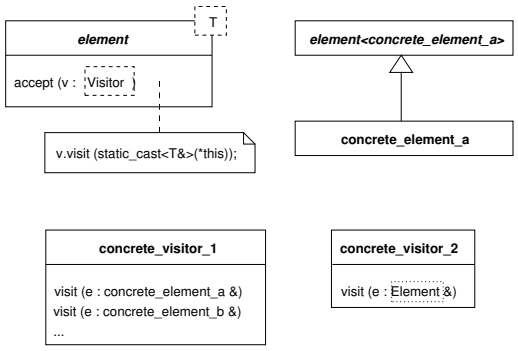
Intent

To define a new operation for the concrete classes of a hierarchy without modifying the hierarchy.

Motivation

In the case of the VISITOR pattern, the operation varies with the type of the operation target. Since we assume to know the exact type as compile-time, a trivial design is thus to define this operation as a procedure overloaded for each target. Such a design, however, does not have the advantages of the translation of the VISITOR pattern proposed in the next section.

Structure



Participants

In the original Gamma's pattern the method ***accept*** has to be defined in each ***element***. The code of each of these ***accept*** method can be the same[6], only type of the

[6]This is not actually the case in Gamma's book, because the name of the visiting method to call is dependent on the element type; how-

this pointer changes. Here we use the same trick as the GENERIC TEMPLATE METHOD to factor **accept** in the superclass.

Each visitor defines a method **visit**, for each element type that it must handle. **visit** can be either an overloaded function (as in **concrete_visitor_1**) or a function template (as in **concrete_visitor_2**). In both case, the overload resolution or function instantiation is made possible by the exact knowledge of the element type.

One advantage of using a member template (as in **concrete_visitor_2**), over an overloaded function (as in **concrete_visitor_1**) is that the **concrete_visitor_2** class does need to be changed when new type are added: the visitor can be specialized externally should the default be inadequate.

Consequences

The code is much closer to the one of Gamma than the trivial design presented before, because the visitor is here an object with all its advantages (state, life duration).

While **accept** and **visit** does not return anything in the original pattern, they can be taught to. It the GENERIC ITERATOR they can even return a type dependent on the visitor's type. As the following example shows.

Sample Code

Let's consider an image2d class the pixels of which should be addressable using different kind of positions (Cartesian or polar coordinates, etc.). For better modularity, we don't want the image2d to known all position types. Therefore we see positions as visitors, which the image **accept**s. **accept** returns the pixel value corresponding to the supplied position. The image will provide only one access method, and it is up to the visitor to perform necessary conversion (e.g. polar to Cartesian) to use this interface.

A position may also refer to a particular channel in a color image. The **accept** return type is thus dependent on the visitor. We will use a traits to handle this.

ever, using the same name (**visit**) for all these methods make no problem in any language as C++ which support function overloading.

```
template< class Visitor, class Visited >
struct visit_return_trait;
```

For each pair (**Visitor**, **Visited**) **visit_return_trait<Visitor,Visited>::type** is the return type of access and visit.

```
// factor the definition of accept for all images
template < class Child >
class image {
public:
  template < typename Visitor >
  typename visit_return_trait< Visitor, Child >::
  type accept (Visitor& v) {
    return v.visit (static_cast< Child& > (*this));
  }
  // ... likewise for const accept
};
```

```
template< typename T >
class image_2d : public image< image_2d< T > > {
public:
  typedef T pixel_type;
  // ...
  T& get_value (int row, int col){...}
  const T& get_value const (int row, int col){...}
};
```

Here is one possible visitor, with it's corresponding **visit_return_trait** specialization.

```
class point_2d {
public:
  point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Visited::pixel_type&
  visit (Visited& v) {
    return v.get_value (row, col);
  }
  // ...
  int row, col;
};

template< class Visited >
struct visit_return_trait< point_2d, Visited > {
  typedef typename Visited::pixel_type type;
};
```

channel_point_2d is another visitor, which must be parametered to access a particular layer (as in the decorator example).

```
template< template< class > class Access >
class channel_point_2d {
public:
  channel_point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Access< typename Visited::pixel_type >::
  output_type& visit (Visited& v) {
    return Access< typename Visited::pixel_type >::
      get (v.get_value (row, col));
  }
  // ...
};

template< template< class > class Access,
          class Visited >
struct visit_return_trait
  < channel_point_2d< Access >, Visited > {
  typedef typename
    Access< typename Visited::pixel_type >::
      output_type type;
};
```

Finally, the following hypothetical main shows how the return value of **`accept`** differ according to the visitor used.

```
int main () {
  image_2d< rgb< int > > img;
  point_2d p(1, 2);
  channel_point_2d<get_red> q(3, 4);

  int v      = img.accept (p);
  rgb<int> w = img.accept (q);
}
```

In our library, **`accept`** and **`visit`** are both named **`operator[]`** so we can write **`img[p]`** or **`p[img]`** at will.

4 Conclusion and Perspectives

Based on object programming, generic programming allows to build and assemble reusable components [15] and proved to be useful where efficiency is required.

Since generic programming (or more generally *Generative programming* [31, 6]) is becoming more popular and because much experience and knowledge have been accumulated and assimilated in structuring the object-oriented programming, we believe that it is time to explore the benefits that the former can derive from well-proven designs in the latter.

We showed how design patterns can be adapted to the generic programming context by presenting the generic versions of three fundamental patterns from Gamma *et al.* [10]: the GENERIC BRIDGE, GENERIC ITERATOR, the GENERIC ABSTRACT FACTORY, the GENERIC TEMPLATE METHOD, the GENERIC DECORATOR, and the GENERIC VISITOR. We hope that such work can provide some valuable insight, and aid design larger systems using generic programming.

Acknowledgments

The authors are grateful to Philippe Laroque for his fruitful remarks about the erstwhile version of this work, to Colin Shaw for his corrections on an early version of this paper, and to Bjarne Stroustrup for his valuable comments on the final version.

Availability

The source of the patterns presented in this paper, as well as other generic patterns, can be downloaded from **`http://www.lrde.epita.fr/download/`**.

References

[1] Matthew H. Austern. *Generic programming and the STL – Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley, 1999.

[2] John Barton and Lee Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *procdings of ACM Conference on Object-Oriented Programming: System, Languages, and APPLICATION (OOPSLA) 1990*. ACM, 1989. URL **`http://java.sun.com/people/gbracha/oopsla90.ps`**.

[4] James Coplien. Curiously recurring template pattern. In Stanly B. Lippman, editor, *C++ Gems*. Cambridge University Press & Sigs Books, 1996. URL **`http://people.we.mediaone.net/stanlipp/gems.html`**.

[5] James A. Crotinger, Julian Cummings, Scott Haney, William Humprey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Dagstuhl seminar on Generic Programming*, April 1998. URL

http://www.acl.lanl.gov/pooma/papers/GenericProgrammingPaper/dagstuhl.pdf.

[6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming. Methods, Tools, and Applications*. Addison Wesley, 2000. URL http://www.generative-programming.org/.

[7] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, 1996. URL http://www.cs.McGill.CA/ACL/papers/oopsla96.html.

[8] Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and oo programming. In *symposium on Generative and Component-Based Software Engineering, Young Researchers Workshop*, 10 2000. URL http://www.lrde.epita.fr/publications/.

[9] Ulfar Erlingsson and Alexander V. Konstantinou. Implementing the C++ Standard Template Library in Ada 95. Technical Report TR96-3, CS Dept., Rensselaer Polytechnic Institute, Troy, NY, January 1996. URL http://www.adahome.com/Resources/Papers/General/stl2ada.ps.Z.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[11] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *15th International Conference on Pattern Recognition (ICPR'2000)*, September 2000. URL http://www.lrde.epita.fr/publications/.

[12] Scott Haney and James Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4), 1999. URL http://www.acl.lanl.gov/pooma/papers.html.

[13] *Ada95 Reference Manual*. Intermetrics, Inc., December 1994. URL http://adaic.org/standards/95lrm/. Version 6.00 (last draft of ISO/IEC 8652:1995).

[14] *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, January 1995. URL ftp://ftp.lip6.fr/pub/gnat/rationale-ada95/.

[15] Mehdi Jazayeri. Component programming – a fresh look at software components. In *Procedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 457–478, September 1995.

[16] Ullrich Köthe. Requested interface. In *In Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*, Munich, Germany, 1997. URL http://www.riehle.org/events/europlop-1997/p16final.pdf.

[17] John Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.

[18] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[20] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5):32–35, June 1995. URL http://www.cantrip.org/traits.html.

[21] Nathan C. Myers. Gnarly new C++ language features, 1997. URL http://www.cantrip.org/gnarly.html.

[22] OON. The object-oriented numerics page. URL http://oonumerics.org/oon.

[23] Esa Pulkkinen. Compile-time determination of base-class relationship in C++, June 1999. URL http://lazy.ton.tut.fi/~esap/instructive/base-class-determination.html.

[24] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference manual*. Object Technology Series. Addison-Wesley, 1999.

[25] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[26] Alex Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1995. URL **http://www.cs.rpi.edu/~musser/doc.ps**.

[27] Bjarne Stroustrup. *The Design and Evolution of C++*. Addision Wesley, June 1994. URL **http://www.research.att.com/~bs/dne.html**.

[28] Bjarne Stroustrup. Why C++ isn't just an Object-Oriented Programming Language. In *OOPSLA'95*, October 1995. URL **http://www.research.att.com/~bs/papers.html**.

[29] Petter Urkedal. Tools for template metaprogramming. web page, March 1999. URL **http://matfys.lth.se/~petter/src/more/metad/index.html**.

[30] Todd L. Veldhuizen. Techniques for scientific C++, August 1999. URL **http://extreme.indiana.edu/~tveldhui/papers/techniques/**.

[31] Todd L. Veldhuizen and Dennis Gannon. Active libraries – Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, October 1998. URL **http://extreme.indiana.edu/~tveldhui/papers/oo98.html**.

[32] Olivier Zendra and Dominique Colnet. Adding external iterators to an existing Eiffel class library. In *32nd conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)*, Melbourne, Australia, November 1999. IEEE Computer Society. URL **http://SmallEiffel.loria.fr/papers/tools-pacific-1999.pdf.gz**.

B.2 A STATIC C++ OBJECT-ORIENTED PROGRAMMING PARADIGM (SCOOP)

A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming. *Workshop on Multiple Paradigm with Object-Oriented Languages* (**MPOOL**), October 2003. At *ACM Conference on OO Programming, Systems, Languages, and Applications* (**OOPSLA**), Anaheim, CA, USA. With Nicolas Burrus, Alexandre Duret-Lutz, David Lesage, and Raphaël Poss. 

A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss

EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France
firstname.lastname@lrde.epita.fr

Abstract. Object-oriented and generic programming are both supported in C++. The former provides high expressiveness whereas the latter leads to more efficient programs by avoiding dynamic typing. This paper presents SCOOP, a new paradigm which enables both classical object-oriented design and high performance in C++ by mixing object-oriented programming and generic programming. We show how classical and advanced object-oriented features such as virtual methods, multiple inheritance, argument covariance, virtual types and multimethods can be implemented in a fully statically typed model, hence without run-time overhead.

1 Introduction

In the context of writing libraries dedicated to scientific numerical computing, expressiveness, reusability and efficiency are highly valuable features. Algorithms are turned into software components that handle mathematical abstractions while these abstractions are mapped into types within programs.

The object-oriented programming (OOP) paradigm offers a solution to express reusable algorithms and abstractions through abstract data types and inheritance. However, as studied by Driesen and Hölzle [18], manipulating abstractions usually results in a run-time overhead. We cannot afford this loss of performance since efficiency is a crucial issue in scientific computing.

To both reach a high level of expressiveness and reusability in the design of object-oriented scientific libraries and keep an effective run-time efficiency for their routines, we have to overcome the "abstractions being inefficient" problem. To cope with that, one can imagine different strategies.

A first idea is to find an existing language that meets our requirements, i.e., a language able to handle abstractions within programs without any penalty at execution time. This language has to be either well-known or simple enough to ensure that a scientist will not be reluctant to use our library. Unfortunately we do not feel satisfied with existing languages; for instance LOOM and PolyTOIL

by Bruce et al. [11, 9] have the precise flavor that we expect but, as prototypes, they do not offer all the features one can expect from a mature language.

A second approach, chosen by Baumgartner and Russo [6] and Bracha et al. [8] respectively for C++ and Java, is to extend an existing language by adding ad hoc features making programs more efficient at run-time. Yet, this approach requires a too important amount of work without any guarantee that extensions will be adopted by the language community and by compiler vendors. To overcome this problem, an alternate approach is to propose a front-end to translate an extended and more expressive language into its corresponding primary efficient language, such as Stroustrup [49] did with his erstwhile version of the C++ language. This approach has been made easier than in the past thanks to recently designed tools dedicated to program translation, for instance Xt [57]. However, we have not chosen this way since we are not experimented enough with this field.

Another strategy is to provide a compiler that produces efficient source codes or binaries from programs written in an expressive language. For that, several solutions have been developed that belong to the fields of static analysis and partial evaluation, as described by Chambers et al. [14], Schultz [42], Veldhuizen and Lumsdaine [56]. In particular, how to avoid the overhead of polymorphic method calls is studied by Aigner and Hölzle [2], Bacon and Sweeney [4] for C++ and by Zendra et al. [58] for Eiffel. However, most of these solutions remain prototypes and are not implemented in well-spread compilers.

Last, we can take an existing object-oriented language and try to bend it to make some constructs more efficient. That was for instance the case of the expression templates construct defined by Veldhuizen [54] in C++, later brought to Ada by Duret-Lutz [19], and of mixin-based programming by Smaragdakis and Batory [44] in C++. These solutions belong to the field of the *generic programming* (GP) paradigm, as described by Jazayeri et al. [26]. This programming style aims at implementing algorithms as general hence reusable as possible without sacrificing efficiency obtained by parameterization—related to the `template` keyword in C++ and to the `generic` keyword in Ada and Eiffel. However, from our experience in developing a scientific library, we notice several major drawbacks of GP that seriously reduce expressiveness and affect user-friendliness, whereas these drawbacks do not exist with "classical" OOP. A key point of this paper is that we do *not* subscribe to "traditional" GP because of these drawbacks. Said shortly, they have their origin in the unbounded structural typing of parameterization in C++ which prevents from having strongly typed signatures for functions or methods. Consequently, type checking at compile-time is awkward and overloading is extremely restricted. Justifications of our position and details about GP limitations are given later on in this paper.

Actually, we want to keep the best of both OOP and GP paradigms—inheritance, overloading, overriding, and efficiency—without resorting to a new language or new tools—translators, compilers, or optimizers. The advent of the C++ Standard Template Library, mostly inspired by the work of Stepanov et al. [47], is one the first serious well-known artifact of GP. Following that example

a lot of scientific computing C++ libraries arose during the past few years(they are referenced by oonumerics [39]), one of the most predominant being Boost [7]. Meanwhile, due to the numerous features of C++, many related GP techniques appeared and are described in the books by Czarnecki and Eisenecker [17], Alexandrescu [3], Vandevoorde and Josuttis [53]. Moreover, Striegnitz and Smith [48], Järvi and Powell [25], Smaragdakis and McNamara [45] have shown that some features offered by a non-object-oriented paradigm, namely the functional one, can be supported by the native C++ language. Knowing these C++ programming techniques, we then thought that this language was able to support an OOP-like paradigm without compromising efficiency. The present paper describes this paradigm, namely a proposal for "Static C++ Object-Oriented Programming": SCOOP.

This paper is composed of three parts. Section 2 discusses the OOP and GP paradigms, their limitations, existing solutions to overcome some of these limitations, and finally what we expect from SCOOP. Section 3 shows how SCOOP is implemented. Finally some technical details and extra features have been moved into appendices.

2 OOP, GP, and SCOOP

A scientific library offers data structures *and* algorithms. This procedural point of view is now consensual [35] although it seems to go against OOP. Actually, an algorithm is intrinsically a general entity since it deals with abstractions. To get the highest possible decoupling between data and algorithms, a solution adopted by the C++ Standard Library and many others is to map algorithms into functions. At the same time, data structures are mapped into classes where most of the methods are nothing but the means to access data. Last, providing reusable algorithms is an important objective of libraries so we have to focus on algorithms. It is then easier to consider that algorithms and all other entities are functions (such as in functional languages) to discuss typing issues. For all these reasons, we therefore adopt in this section a function-oriented approach of algorithms.

2.1 About Polymorphisms

A function is polymorphic when its operands can have more than one type, either because there are several definitions of the function, or because its definition allows some freedom in the input types. The proper function to call has to be chosen depending on the context. Cardelli and Wegner [13] outline four different kinds of polymorphism.

In **inclusion polymorphism**, a function can work on any type in a *type class*. Type classes are named sets of types that follow a uniform interface. Functional languages like Haskell allow programmers to define type classes explicitly, but this polymorphism is also fundamental for OO languages. In C++, inclusion polymorphism is achieved via two mechanisms: subclassing and overriding of virtual functions.

Subclassing is used to define sets of types. The `class` (or `struct`) keyword is used to define types that can be partially ordered through a hierarchy: i.e., an inclusion relation[1]. A function which expects a pointer or reference to a class `A` will accept an instance of `A` or any subclass of `A`. It can be noted that C++'s typing rules make no difference between a pointer to an object whose type is exactly `A` and a pointer to an object whose type belongs to the type class of `A`[2].

Overriding of virtual functions allows types whose operations have different implementations to share the same interface. This way, an operation can be implemented differently in a subclass of `A` than it is in `A`. Inclusion polymorphism is sometime called *operation polymorphism* for this reason.

These two aspects of inclusion polymorphism are hardly dissociable: it would make no sense to support overriding of virtual functions without subclassing, and subclassing would be nearly useless if all subclasses had to share the same implementation.

In **parametric polymorphism**, the type of the function is represented using at least one generic type variable. Parametric polymorphism really corresponds to ML generic functions, which are compiled only once, even if they are used with different types. Cardelli and Wegner states that Ada's generic functions are not to be considered as parametric polymorphism because they have to be *instantiated explicitly* each time they are used with a different type. They see Ada's generic functions as a way to produce several monomorphic functions by macro expansion. It would therefore be legitimate to wonder whether C++'s function templates achieve parametric polymorphism. We claim it does, because unlike Ada's generics, C++'s templates are instantiated *implicitly*. In effect, it does not matter that C++ instantiates a function for each type while ML compiles only one function, because this is transparent to the user and can be regarded as an implementation detail[3].

Inclusion and parametric polymorphisms are called *universal*. A nice property is that they are open-ended: it is always possible to introduce new types and to use them with existing functions. Two other kinds of polymorphism do not share this property. Cardelli and Wegner call them *ad-hoc*.

Overloading corresponds to the case where several functions with different types have the same name.

Coercion polymorphism comes from implicit conversions of arguments. These conversions allow a monomorphic function to appear to be polymorphic.

All these polymorphisms coexist in C++, although we will discuss some notable incompatibilities in section 2.3. Furthermore, apart from virtual functions,

[1] Inclusion polymorphism is usually based on a subtyping relation, but we do not enter the debate about "subclassing v. subtyping" [15].

[2] In Ada, one can write `access A` or `access A'Class` to distinguish a pointer to an instance of `A` from a pointer to an instance of any subclass of `A`.

[3] This implementation detail has an advantage, though: it allows specialized instantiations (i.e., template specializations). To establish a rough parallel with *inclusion polymorphism*, template specializations are to templates what method overriding is to subclassing. They allow to change the implementation for some types.

the resolution of a polymorphic function call (i.e., choosing the right definition to use) is performed at compile-time.

2.2 About the Duality of OOP and GP

Duality of OOP and GP has been widely discussed since Meyer [33]. So we just recall here the aspects of this duality that are related to our problem.

Let us consider a simple function `foo` that has to run on different image types. In traditional OOP, the image abstraction is represented by an abstract class, `Image`, while a concrete image type (for instance `Image2D`) for a particular kind of 2D images, is a concrete subclass of the former. The same goes for the notion of "point" that gives rise to a similar family of classes: `Point`, which is abstract, and `Point2D`, a concrete subclass of `Point`. That leads to the following code[4]:

```
struct Image {
  virtual void set(const Point& p, int val) = 0;
};

struct Image2D : public Image {
  virtual void set(const Point& p, int val) { /* impl */ }
};

void foo(Image& input, const Point& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is a polymorphic function thanks to *inclusion through class inheritance*. The call `input.set(p, 51)` results in a run-time dispatch mechanism which binds this call to the proper implementation, namely `Image2D::set`. In the equivalent GP code, there is no need for inheritance.

```
struct Image2D {
  void set(const Point2D& p, int val) { /* impl */ }
};

template <class IMAGE, class POINT>
void foo(IMAGE& input, const POINT& p) {
  // does something like:
  input.set(p, 51);
}
```

[4] Please note that, for simplification purpose, we use `struct` instead of `class` and that we do not show the source code corresponding to the `Point` hierarchy.

```
int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is now polymorphic through *parameterization*. At compile-time, a particular version of `foo` is instantiated, `foo<Image2D, Point2d>`, dedicated to the particular call to `foo` in `main`. The basic idea of GP is that all exact types are known at compile-time. Consequently, functions are specialized by the compiler; moreover, every function call can be inlined. This kind of programming thus leads to efficient executable codes.

The table below briefly compares different aspects of OOP and GP.

notion	OOP	GP
typing	named typing through class names so explicit in class definitions	structural so only described in documentation
abstraction (e.g., image)	abstract class (e.g., `Image`)	formal parameter (e.g., `IMAGE`)
inheritance	is the way to handle abstractions	is only a way to factorize code
method (set)	no-variant (`Image::set(Point, int)` `Image2D::set(Point, int)`)	— — —
algorithm (foo)	a single code at program-time (`foo`) and a unique version at compile-time (`foo`)	a single meta-code at program-time (`template<..> foo`) and several versions at compile-time (`foo<Image2D,Point2D>`, etc.)
efficiency	poor	high

From the C++ compiler typing point of view, our OOP code can be translated into:

$$\text{type Image} = \{\ \text{set} : \text{Point} \to \text{Int} \to \text{Void}\ \}$$
$$\text{foo} : \text{Image} \to \text{Point} \to \text{Void}$$

`foo` is restricted to objects whose types are respectively subclasses of `Image` and `Point`. For our GP code, things are very different. First, the image abstraction is not explicitly defined in code; it is thus unknown by the compiler. Second, both formal parameters of `foo` are anonymous. We then rename them respectively "I" and "P" in the lines below and we get:

$$\forall\ \text{I},\quad \forall\ \text{P},\quad \text{foo} : \text{I} \to \text{P} \to \text{Void}$$

Finally, if these two pieces of code seem at a first sight equivalent, they do not correspond to the same typing behavior of the C++ language. Thus, they are treated differently by the compiler and have different advantages and drawbacks. The programmer then faces the duality of OOP and GP and has to determinate which paradigm is best suited to her requirements.

During the last few years, the duality between OOP and GP has given rise to several studies. Different authors have worked on the translation of some design patterns [22] into GP; let us mention Géraud and Duret-Lutz [23], Langer [27], Duret-Lutz et al. [20], Alexandrescu [3], Régis-Gianas and Poss [40].

Another example concerns the *virtual types* construct, which belongs to the OOP world even if very few OO languages support it. This construct has been proposed as an extension of the Java language by Thorup [51] and a debate

about the translation and equivalence of this construct in the GP world has followed [10, 52, 41].

Since the notion of virtual type is of high importance in the remainder of this paper, let us give a more elaborate version of our previous example. In an augmented C++ language, we would like to express that both families of image and point classes are related. To that aim, we would like to write:

```
struct Image {
  virtual typedef Point point_type = 0;
  virtual void set(const point_type& p, int val) = 0;
};

struct Image2D : public Image {
  virtual typedef Point2D point_type;
  virtual void set(const point_type& p, int val) { /* impl */ }
};
```

`point_type` is declared in the `Image` class to be an "abstract type alias" (`virtual typedef .. point_type = 0;`) with a constraint: in subclasses of `Image`, this type should be a subclass of `Point`. In the concrete class `Image2D`, the alias `point_type` is defined to be `Point2D`. Actually, the behavior of such a construct is similar to the one of virtual member functions: using `point_type` on an image object depends on the exact type of the object. An hypothetical use is depicted hereafter:

```
Image* ima = new Image2D();
// ...
Point* p = new (ima->point_type)();
```

At run-time, the particular exact type of `p` is `Point2D` since the exact type of `ima` is `Image2D`.

An about equivalent GP code in also an augmented C++ is as follows:

```
struct Image2D {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I>
where I {
  typedef point_type;
  void set(const point_type&, int);
}
void foo(I& input, const typename I::point_type& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
```

```
  foo(ima, p);
}
```

Such as in the original GP code, inheritance is not used and typing is fully structural. On the other hand, a *where clause* has been inserted in `foo`'s signature to precise the nature of acceptable type values for `I`. This construct, which has its origin in CLU [30], can be found in Theta [29], and has also been proposed as an extension of the Java language [36]. From the compiler point of view, `foo`'s type is much more precise than in the traditional GP code. Finally, in both C++ OOP augmented with virtual types and C++ GP augmented with where clauses, we get stronger expressiveness.

2.3 OOP and GP Limitations in C++

Object-Oriented Programming relies principally on the inclusion polymorphism. Its main drawback lies in the indirections necessary to run-time resolution of virtual methods. This run-time penalty is undesirable in highly computational code; we measured that getting rid of virtual methods could speed up an algorithm by a factor of 3 [24].

This paradigm implies a loss of typing: as soon as an object is seen as one of its base classes, the compiler looses some information. This limits optimization opportunities for the compiler, but also type expressiveness for the developer. For instance, once the exact type of the object has been lost, type deduction (`T::deducted_type`) is not possible. This last point can be alleviated by the use of virtual types [52], which are not supported by C++.

The example of the previous section also expresses the need for covariance: `foo` calls the method `set` whose expected behavior is covariant. `foo` precisely calls `Image2D::set(Point2D,int)` in the GP version, whereas the call in the OOP version corresponds to `Image::set(Point,int)`.

Generic Programing on the other hand relies on parametric polymorphism and proscribes virtual functions, hence inclusion polymorphism. The key rule is that the exact type of each object has to be known at compile-time. This allows the compiler to perform many optimizations. We can distinguish three kinds of issues in this paradigm:

- the rejection of operations that cannot be typed statically,
- the closed world assumption,
- the lack of template constraints.

The first issue stems from the will to remain statically typed. Virtual functions are banished, and this is akin to rejecting inclusion polymorphism. Furthermore there is no way to declare an heterogeneous list and to update it at run-time, or, more precisely to dynamically replace an attribute by an object of a compatible subtype. These operations cannot be statically typed, there can be no way around this.

The closed world assumption refers to the fact that C++'s templates do not support separate compilation. Indeed, in a project that uses parametric polymorphism exclusively it prevents separate compilation, because the compiler must

always know all type definitions. Such monolithic compilation leads to longer build times but gives the compiler more optimization opportunities. The C++ standard [1] supports separate compilation of templates via the `export` keyword, but this feature has not been implemented in mainstream C++ compilers yet.

```
void foo(A1& arg)
{
  arg.m1()
}

void foo(A2& arg)
{
  arg.m2()
}
```

```
template<class A1>
void foo(A1& arg)
{
  arg.m1()
}

template<class A2>
void foo(A2& arg) // illegal
{
  arg.m2()
}
```

```
template<class A1>
void foo(A1& arg)
{
  arg.m1()
}

template<>
void foo<A2>(A2& arg)
{
  arg.m2()
}
```

Fig. 1. Overloading can be mixed with inclusion polymorphism (left), but will not work with unconstrained parametric polymorphism (middle and right).

The remaining issue comes from bad interactions between parametric polymorphism and other polymorphisms in C++. For instance, because template arguments are unconstrained, one cannot easily overload function templates. Figure 1 illustrates this problem. When using inclusion polymorphism (left), the compiler knows how to resolve the overloading: if `arg` is an instance of a subclass of `A1` (resp. `A2`), it should be used with the first (resp. second) definition of `foo()`. We therefore have two implementations of `foo()` handling two different sets of types. These two sets are not closed (it is always possible to add new subclasses), but they are constrained. Arbitrary types cannot be added unless they are subtypes of `A1` or `A2`. This constraint, which distinguishes the two sets of types, allows the compiler to resolve the overloading.

In generic programming, such an overloading could not be achieved, because of the lack of constraints on template parameters. The middle column on Figure 1 shows a straightforward translation of the previous example into parametric polymorphism. Because template parameters cannot be constrained, the function's arguments have to be generalized *for any type* `A`, and *for any type* `B`. Of course, the resulting piece of code is not legal in C++ because both functions have the same type. A valid possibility (on the right of Figure 1), is to write a definition of `foo` for any type `A1`, and then *specialize* this definition for type `A2`. However, this specialization will only work for one type (`A2`), and would have to be repeated for each other type that must be handled this way.

Solving overloading is not the only reason to constrain template arguments, it can also help catching errors. Libraries like STL, which rely on generic programming, document the requirements that type arguments must satisfy. These constraints are gathered into *concepts* such as *forward iterator* or *associative con-*

tainer [47]. However, these concepts appear only in the documentation, not in typing. Although some techniques have been devised and implemented in SGI's STL to check concepts at compile-time, the typing of the library still allows a function expecting a *forward iterator* to be instantiated with an *associative container*. Even if the compilation will fail, this technique will not prevent the compiler from instantiating the function, leading to cryptic error messages, because some function part of the *forward iterator* requirements will not be found in the passed associative container. Had the *forward iterator* been expressed as a constraint on the argument type, the error would have been caught at the right time i.e. during the attempt to instantiate the function template, not after the instantiation.

2.4 Existing Clues

As just mentioned, some people have already devised ways to check constraints. Siek and Lumsdaine [43] and McNamara and Smaragdakis [32] present a technique to check template arguments. This technique relies on a short checking code inserted at the top of a function template. This code fails to compile if an argument does not satisfy its requirements and is turned into a no-op otherwise. This technique is an effective means of performing structural checks on template arguments to catch errors earlier. However, constraints are just *checked*, they are not *expressed* as part of function types. In particular, overloading issues discussed in the previous section are not solved. Overloading has to be solved by the compiler *before* template instantiation, so any technique that works after template instantiation does not help.

Ways to *express* constraints by subtyping exist in Eiffel [34] and has been proposed as a Java extension by Bracha et al. [8]. Figure 2 shows how a similar C++ extension could be applied to the example from Section 2.2.

We have introduced an explicit construct through the keyword `concept` to express the definition of `image`, the structural type of images. This construct is also similar to the notion of signatures proposed by Baumgartner and Russo [6] as a C++ extension. Having explicitly a definition of `image` constraints the formal parameter `I` in `foo`'s type.

Some interesting constructions used to constrain parametric polymorphism or to emulate dynamic dispatch statically rely on a idiom known as the *Barton and Nackman trick* [5] also known as the *Curiously Recurring Template Pattern* [16]. The idea is that a super class is parameterized by its immediate subclass (Figure 3), so that it can define methods for this subclass.

For instance the Barton and Nackman trick has been used by Furnish [21] to constrain parametric polymorphism and simplify the Expression Template technique of Veldhuizen [54].

```
concept image {
  typedef point_type;
  void set(const point_type& p, int val);
};

struct Image2D models image {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I models image>
void foo(I& input, const typename I::point_type& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

Fig. 2. Extending C++ to support concept constraints

```
template <class T>
struct super
{
  void foo(const T& arg)
  {
    // ...
  }
};
```

```
struct infer : public super<infer>
{
  // ...
};
```

Fig. 3. The Barton and Nackman trick

2.5 Objectives of SCOOP

Our objective in this paper is to show how inclusion polymorphism can be almost completely emulated using parametric polymorphism in C++ while preserving most OOP features. Let us define our requirements.

Class Hierarchies. Developers should express (static) class hierarchies just like in the traditional (dynamic) C++ OOP paradigm. They can draw UML static diagrams to depict inheritance relationships between classes of their programs. When they have an OOP class, say `Bar`, its SCOOP translation is a single class template: `Bar`[5].

Named Typing. When a scientific practitioner designs a software library, it is convenient to reproduce in programs the names of the different abstractions of the application domain. Following this idea, there is an effective benefit to make the relationships between concrete classes and their corresponding abstractions explicit to get a more readable class taxonomy. We thus prefer named typing over structural typing for SCOOP.

Multiple Inheritance. In the object model of C++, a class can inherit from several classes at the same time. There is no reason to give up this feature in SCOOP.

Overriding. C++ inheritance comes with the notions of pure virtual functions, virtual functions, and overriding functions in subclasses. We want to reproduce their behavior in SCOOP but without their associated overhead.

Virtual Types. This convenient tool (see sections 2.2 and 2.3) allows to express that a class encloses polymorphic `typedef`s. Furthermore, it allows to get covariance for member functions. Even if virtual types does not exist in primary C++, we want to express them in SCOOP.

Method Covariance. It seems reasonable to support method covariance in SCOOP, and particularly binary methods. Since our context is static typing with parametric polymorphism, the C++ compiler may ensure that we do not get typing problems eventually.

[5] We are aware of a solution to encode static class hierarchies that is different to the one presented later on in this paper. However, one drawback of this alternate solution is to duplicate every class: having a class `Bar` in OOP gives rise to a couple of classes in the static hierarchy. In our opinion, this is both counter-intuitive and tedious.

Overloading. In the context of scientific computing, having overloading is crucial. For instance, we expect from the operator "`+`" to be an over-overloaded function in an algebraic library. Moreover, overloading helps to handle a situation that often arises in scientific libraries: some algorithms have a general implementation but also have different more efficient implementation for particular families of objects. We want to ensure in SCOOP that overloading is as easy to manage as in OOP.

Multimethods. Algorithms are often functions with several inputs or arguments. Since the source code of an algorithm can also vary with the nature and number of its input, we need multimethods.

Parameter Bounds. Routines of scientific libraries have to be mapped into strongly typed functions. First, this requirement results in more comfort for users since it prevents them from writing erroneous programs. Second, this requirement is helpful to disambiguate both overloading and multimethod dispatch.

3 Description of SCOOP

3.1 Static Hierarchies

Static hierarchies are meta-hierarchies that result in real hierarchies after various static computations like parameter valuations. With them, we are able to know all types statically hence avoiding the overhead of virtual method resolution. Basically, the core of our static hierarchy system is a generalization of the Barton & Nackman trick [5]. Veldhuizen [55] had already discussed some extensions of this technique and assumed the possibility to apply it to hierarchies with several levels. We effectively managed to generalize these techniques to entire, multiple-level hierarchies.

Our hierarchy system is illustrated in Figure 4. This figure gives an example of a meta-hierarchy, as designed by the developer, and describes the different final hierarchies obtained, according to the instantiated class. The corresponding C++ code is given in Figure 5. This kind of hierarchy gives us the possibility to define abstract classes (class `A`), concrete extensible classes (class `B`), and final classes (class `C`). Non final classes[6] are parameterized by `EXACT` that basically represents the type of the object effectively instantiated. Additionally, any class hierarchy must inherit from a special base class called `Any`. This class factorizes some general mechanisms whose role are detailed later.

Instantiation of abstract classes is prevented by protecting their constructors. The interfaces and the dispatch mechanisms they provide are detailed in Section 3.2.

Extensible concrete classes can be instantiated and extended by subclassing. Since the type of the object effectively instantiated must be propagated through

[6] Non final classes are abstract classes or concrete classes that can be extended. Non parameterized classes are necessarily final in our paradigm.

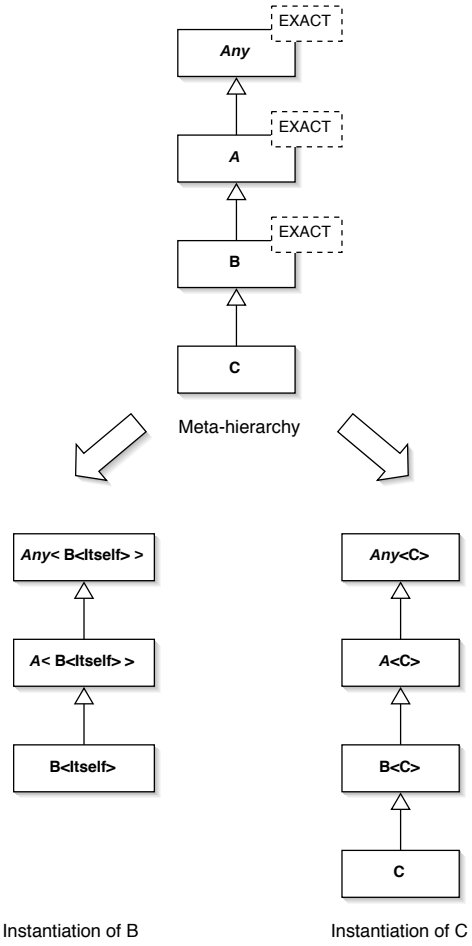


Fig. 4. Static hierarchy unfolding sample
A single meta-hierarchy generates one class hierarchy per instantiable class. Our model can instantiate both leaf classes and intermediate ones. In this example, only `B` and `C` are instantiable, so only the above two hierarchies can be instantiated. Non final classes are parameterized by `EXACT` which represents the type of the object effectively instantiated. The type `Itself` is used as a terminator when instantiating extensible concrete classes.

```
// Hierarchy apparel

struct Itself
{ };

// find_exact utility macro
#define find_exact(Type) // ...

template <class EXACT>
class Any
{
  // ...
};
```

```
// Hierarchy

// purely abstract class
template <class EXACT>
class A: public Any<EXACT>
{
  // ...
};

// extensible concrete class
template <class EXACT = Itself>
class B: public A<find_exact(B)>
{
  // ...
};

// final class
class C: public B<C>
{
  // ...
};
```

Fig. 5. Static hierarchy sample: C++ code
`find_exact(Type)` mechanism is detailed in Appendix A.1.

the hierarchy, this kind of class has a double behavior. When such a class `B` is extended and is not the instantiated class, it must propagate its `EXACT` type parameter to its base classes. When it is effectively instantiated, further subclassing is prevented by using the `Itself` terminator as `EXACT` parameter. Then, `B` cannot propagate its `EXACT` parameter directly and should propagate its own type, `B<Itself>`. To determine the effective `EXACT` parameter to propagate, we use a meta-program called `find_exact(Type)` whose principle and C++ implementation are detailed in Appendix A.1. One should also notice that `Itself` is the default value for the `EXACT` parameter of extensible concrete classes. Thus, `B` sample class can be instantiated using the `B<>` syntax.

Itself classes cannot be extended by subclassing. Consequently, they do not need any `EXACT` parameterization since they are inevitably the instantiated type when they are part of the effective hierarchy. Then, they only have to propagate their own types to their parents.

Within our system, any static hierarchy involving n concrete classes can be unfolded into n distinct hierarchies, with n distinct base classes. Effectively, concrete classes instantiated from the same meta-hierarchy will have different base classes, so that some dynamic mechanisms are made impossible (see Section 2.3).

3.2 Abstract Classes and Interfaces

In OOP, abstraction comes from the ability to express class interfaces without implementation. Our model keeps the idea that C++ interfaces are represented by abstract classes. Abstract classes declare all the services their subclasses should provide. The compliance to a particular interface is then naturally ensured by the inheritance from the corresponding abstract class.

Instead of declaring pure virtual member functions, abstract classes define abstract member functions as dispatches to their actual implementation. This manual dispatch is made possible by the `exact()` accessor provided by the `Any` class. Basically, `exact()` downcasts the object to its `EXACT` type made available by the static hierarchy system presented in Section 3.1. In practice, `exact()` can be implemented with a simple `static_cast` construct, but this basic mechanism forbids virtual inheritance[7]. Within our paradigm, an indirect consequence is that multiple inheritance implies inevitably virtual inheritance since `Any` is a utility base class common to all classes. Advanced techniques, making virtual and thus multiple inheritance possible, are detailed in Appendix A.2.

An example of an abstract class with a dispatched method is given in Figure 6. The corresponding C++ code can be deduced naturally from this UML diagram. In the abstract class `A`, the method `m(...)` calls its implementation `m_impl(...)`. Method's interface and implementation are explicitly distinguished by using different names. This prevents recursive calls of the interface if the implementation is not defined. Of course, overriding the implementation is permitted. Thanks to the `exact()` downcast, `m_impl(...)` is called on the type of the object effectively instantiated, which is necessarily a subclass of `A`. Thus,

[7] Virtual inheritance occurs in diamond-shape hierarchies.

overriding rules are respected. Since the `EXACT` type is known statically, this kind of dispatch is entirely performed at compile-time and does not require the use of virtual symbol tables. Method dispatches can be inlined so that they finally come with no run-time overhead.

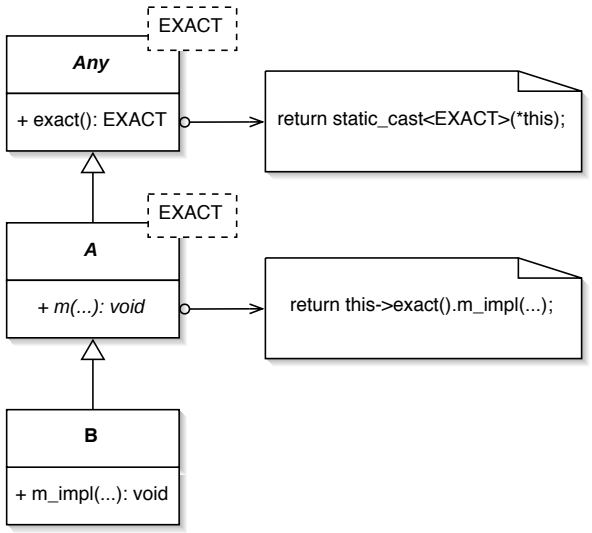


Fig. 6. Abstract class and dispatched abstract method sample

3.3 Constraints on Parameters

Using SCOOP, it becomes possible to express constraints on types. Since we have inheritance between classes, we can specify that we only want a subclass of a particular type, thereby constraining the input type. Thus, OOP's ability to handle two different sets of types has been kept in SCOOP, as demonstrated in Figure 7.

Actually, two kinds of constraints are made possible: accept a type and all its subclasses or accept only this type. Both kinds of constraints are illustrated in Figure 8. We have the choice between letting the `EXACT` parameter free to accept all its subclasses, or freezing it (generally to `Itself`) to accept only this exact type.

3.4 Associations

In SCOOP, the implementation of object composition or aggregation is very close to its equivalent in C++ OOP. Figure 9 illustrates the way an aggregation relation is implemented in our paradigm, in comparison with classical OOP. We

```
void foo(A1& arg)
{
   // ...
}

void foo(A2& arg)
{
   // ...
}
```

```
template <class EXACT>
void foo(A1<EXACT>& arg)
{
   // ...
}

template <class EXACT>
void foo(A2<EXACT>& arg)
{
   // ...
}
```

Fig. 7. Constraints on arguments and overloading
Left (classical OOP) and right (SCOOP) codes have the same behavior. Classical overloading rules are applied in both cases. Subclasses of `A1` and `A2` are accepted in SCOOP too; the limitation of GP has been overcome.

```
template <class EXACT>
void foo(A<EXACT>& a)
{
  // ...
}
```

```
void foo(A<Itself>& a)
{
  // ...
}
```

Fig. 8. Kinds of constraints
On the left, `A` and all its subclasses are accepted. On the right, only exact `A` arguments are accepted. As mentioned in section 2.1, contrary to other languages like Ada, C++ cannot make this distinction; this is therefore another restriction overcome by SCOOP.

want a class `B` to aggregate an object of type `C`, which is an abstract class. The natural way to implement this in classical OOP is to maintain a pointer on an object of type `C` as a member of class `B`. In SCOOP, the corresponding meta-class `B` is parameterized by `EXACT`, as explained in Section 3.1. Since all types have to be known statically, `B` must know the effective type of the object it aggregates. A second parameter, `EXACT_C`, is necessary to carry this type. Then, `B` only has to keep a pointer on an object of type `C<EXACT_C>`. As explained in Section 3.3, this syntax ensures that the aggregated object type is a subclass of `C`. This provides stronger typing than the generic programming idioms for aggregation proposed in [20].

As for hierarchy unfolding (Section 3.1), this aggregation pattern generates as many versions of `B` as there are distinct parameters `EXACT_C`. Each effective version of `B` is dedicated to a particular subclass of `C`. Thus, it is impossible to change dynamically the aggregated object for an object of another concrete type. This limitation is directly related to the rejection of dynamic operations, as mentioned in Section 2.3.

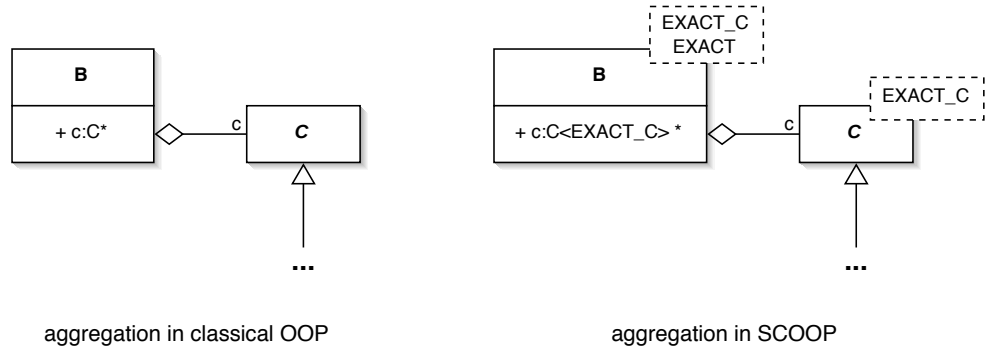


Fig. 9. Comparison of aggregation in OOP and SCOOP

3.5 Covariant Arguments

Covariant parameters may be simulated in C++ in several ways. It can be done by using a `dynamic_cast` to check and convert at run-time the type of the argument. This method leads to unsafe and slower programs. Statically checked covariance has already been studied using templates in Surazhsky and Gil [50]. Their approach was rather complex though, since their typing system was weaker.

Using SCOOP, it is almost straightforward to get statically checked covariant parameters. We consider an example with images and points in 2 and 3 dimensions to illustrates argument covariance. Figure 10 depicts a UML diagram of our example. Since an `Image2d` can be seen as an `Image`, it is possible to give

a `Point3d` (seen as a `Point`) to an `Image2d`. This is why classical OO languages either forbid it or perform dynamic type checking when argument covariance is involved.

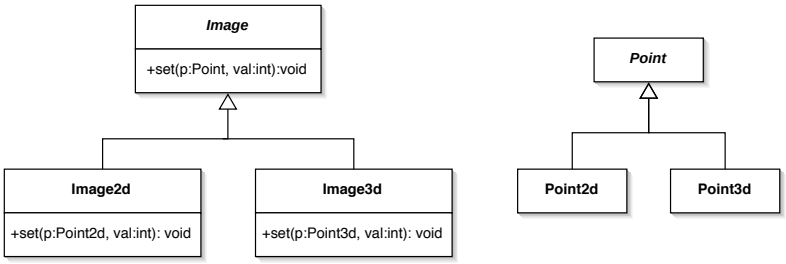


Fig. 10. Argument covariance example in UML

Figure 11 details how this design would be implemented in SCOOP. This code works in three steps:

- Take a `Point<P>` argument in `Image::set` and downcast it into its exact type `P`. Taking a `Point<P>` argument ensures that `P` is a subclass of `Point` at this particular level of the hierarchy.
- Lookup `set_impl` in the exact image type. Since the point argument has been downcasted towards `P`, methods accepting `P` (and not just `Point<P>`) are candidate.
- In SCOOP, since method dispatch is performed at compile-time, argument covariance will be checked statically. The compilation fails if no method accepting the given exact point type is available.

Finally, we have effectively expressed argument covariance. Points have to conform to `Point` at the level of `Image`, and to `Point2d` at the level of `Image2d`.

3.6 Polymorphic `typedefs`

In this section we show how we can write virtual `typedef`s (we also call them polymorphic `typedef`s) in C++. From a base class we want to access `typedef`s defined in its subclasses. Within our paradigm, although base classes hold the type of their most derived subclass, it is not possible to access fields of an incomplete type. When a base class is instantiated, its `EXACT` parameter is not completely constructed yet because base classes have to be instantiated before subclasses. A good solution to cope with this issue is to use traits [37, 55]. Traits can be defined on incomplete types, thereby avoiding infinite recursion.

The overall mechanism is described in Figure 12. To allow the base class to access `typedef`s in the exact class, traits have been defined for the exact type (`image_traits`). To ensure correct typedef inheritance, we create a hierarchy of

```
template <class EXACT>
struct Point : public Any<EXACT> {};

template <class EXACT = Itself>
struct Point2d : public Point<find_exact(Point2d)>
{
  // ...
};

template <class EXACT = Itself>
struct Point3d : public Point<find_exact(Point3d)>
{
  // ...
};

template <class EXACT>
struct Image : Any<EXACT>
{
  template <class P>
  void set(const Point<P>& p, int val) {
    // static dispatch
    // p is downcasted to its exact type
    return this->exact().set_impl(p.exact(), val);
  }
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  template <class P>
  void set_impl(const Point2d<P>& p, int val) {
    // ...
  }
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 11. Argument covariance using SCOOP
Compilation fails if the compiler cannot find an implementation of `set_impl` for the exact type of the given point in `Image2d`.

traits which reproduces the class hierarchy. Thus, `typedef`s are inherited as if they were actually defined in the class hierarchy. As for argument covariance, virtual `typedef`s are checked statically since method dispatch is performed at compile-time. The compilation fails if a wrong point type is given to an `Image2d`.

There is an important difference between classical virtual types and our virtual `typedef`s. First, the virtual `typedef`s we have described are not constrained. The `point_type` virtual `typedef` does not have to be a subclass of `Point`. It can be any type. It is possible to express a subclassing constraint though, by checking it explicitly using a meta-programming technique detailed in Appendix A.3.

One should note that in our paradigm, when using `typedef`s, the resulting type is a single type, not a class of types (with the meaning of Section 3.3). A procedure taking this type as argument does not accept its subclasses. For instance, a subclass `SpecialPoint2d` of `Point2d` is not accepted by the `set` method. This problem is due to the impossibility in C++ to make `template typedef`s, thus we have to bound the exact type of the class when making a `typedef` on it. It is actually possible to overcome this problem by encapsulating open types in boxes. This is not detailed in this paper though.

3.7 Multimethods

Several approaches have been studied to provide multimethods in C++, for instance Smith [46], which relies on preprocessing.

In SCOOP, a multimethod is written as a set of functions sharing the same name. The dispatching is then naturally performed by the overloading resolution, as depicted by Figure 13.

4 Conclusion and Perspectives

In this paper, we described a proposal for a Static C++ Object-Oriented Programming (SCOOP) paradigm. This model combines the expressiveness of traditional OOP and the performance gain of static binding resolution thanks to generic programming mechanisms. SCOOP allows developers to design OO-like hierarchies and to handle abstractions without run-time overhead. SCOOP also features constrained parametric polymorphism, argument covariance, polymorphic `typedef`s, and multimethods for free.

Yet, we have not proved that resulting programs are type safe. The type properties of SCOOP have to be studied from a more theoretical point of view. Since SCOOP is static-oriented, object types appear with great precision. We expect from the C++ compiler to diagnose most programming errors. Actually, we have the intuition that this kind of programming is closely related to the *matching* type system of Bruce et al. [11]. In addition, functions in SCOOP seem to be f-bounded [12].

The main limitations of our paradigm are common drawbacks of the intensive use of templates:

```
// Point, Point2d and Point3d

// A forward declaration is enough to define image_traits
template <class EXACT> struct Image;

template <class EXACT> struct image_traits;

template <class EXACT>
struct image_traits< Image<EXACT> >
{
  // default typedefs for Image
};

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

  void set(const point_type& p, int val) {
    this->exact().set_impl(p, val);
  }
};

// Forward declaration
template <class EXACT> struct Image2d;

// image_traits for Image2d inherits from image_traits for Image
template <class EXACT>
struct image_traits< Image2d<EXACT> >
  : public image_traits<Image <find_exact(Image2d)> >
{
  // We have to specify a concrete type, we cannot write:
  // typedef template Point2d point_type;

  typedef Point2d<Itself> point_type;
  // ... other default typedefs for Image2d
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  // ...
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 12. Mechanisms of virtual `typedef`s with SCOOP

```
template <class I1, class I2>
void algo2(Image<I1>& i1, Image<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image3d<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image2d<I2>& i2);

// ... other versions of algo2

template <class I1, class I2>
void algo1(Image<I1>& i1, Image<I2>& i2)
{
  // dispatch will be performed on the exact image types
  algo2(i1.exact(), i2.exact());
}
```

Fig. 13. Static dispatch for multi-methods
`algo1` downcasts `i1` and `i2` into their exact types when calling `algo2`. Thus, usual overloading rules will simulate multimethod dispatch.

- closed world assumption;
- heavy compilation time;
- code bloat (but we trade disk space for run-time efficiency);
- cryptic error messages;
- unusual code, unreadable by the casual reader.

The first limitation prevents the usage of separated compilation and dynamic libraries. The second one is unavoidable since SCOOP run-time efficiency relies on the C++ capability of letting the compiler perform some computations. The remaining drawbacks are related to the convenience of the *core developer*, i.e. the programmer who designs the hierarchies and should take care about core mechanisms. Cryptyc error messages can be helped by the use of structural checks mentionned in Section 2.4, which are not incompatible with SCOOP.

This paradigm has been implemented and successfully deployed in a large scale project: Olena, a free software library dedicated to image processing [38]. This library mixes different complex hierarchies (images, points, neighborhoods) with highly generic algorithms.

Although repelling at first glance, SCOOP can be assimilated relatively quickly since its principles remain very close to OOP. We believe that SCOOP and its collection of constructs are suitable for most scientific numerical computing projects.

Bibliography

[1] International standard: Programming language – C++. ISO/IEC 14882:1998(E), 1998.

[2] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *the Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–167. Springer-Verlag, 1996.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, 1996.

[5] J. Barton and L. Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[6] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[7] Boost. Boost libraries, 2003. URL `http://www.boost.org`.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[9] K. B. Bruce, A. Fiech, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (ToPLAS)*, 25(2):225–290, March 2003.

[10] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *the Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[11] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *the Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, 1997. Springer-Verlag.

[12] P. S. Canning, W. R. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *the Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 73–280, London, UK, September 1989. ACM.

[13] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[14] C. Chambers, J. Dean, and D. Grove. Wholeprogram optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering, June 1996.

[15] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, California, USA, January 1990.

[16] J. Coplien. *Curiously Recurring Template Pattern*. In [28].

[17] K. Czarnecki and U. Eisenecker. *Generative programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, SIGPLAN Notices 31(10), pages 306–323, 1996.

[19] A. Duret-Lutz. Expression templates in Ada. In *the Proceedings of the 6th International Conference on Reliable Software Technologies, Leuven, Belgium, May 2001 (Ada-Europe)*, volume 2043 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 2001.

[20] A. Duret-Lutz, T. Géraud, and A. Demaille. Design patterns for generic programming in C++. In *the Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 189–202, San Antonio, Texas, USA, January-February 2001. USENIX Association.

[21] G. Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, 1997.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[23] T. Géraud and A. Duret-Lutz. Design patterns for generic programming. In M. Devos and A. Rüping, editors, *In the Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*. UVK, Univ. Verlag, Konstanz, July 2000.

[24] T. Géraud, Y. Fabre, and A. Duret-Lutz. Applying generic programming to image processing. In M. Hamsa, editor, *In the Proceedings of the IASTED International Conference on Applied Informatics – Symposium Advances in Computer Applications*, pages 577–581, Innsbruck, Austria, February 2001. ACTA Press.

[25] J. Järvi and G. Powell. The lambda library: Lambda abstraction in C++. In *the Proceedings of the 2nd Workshop on Template Programming (in conjunction with OOPSLA)*, Tampa Bay, Florida, USA, October 2001.

[26] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, 2000. Springer-Verlag.

[27] A. Langer. Implementing design patterns using C++ templates. Tutorial at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2000.

[28] S. B. Lippman, editor. *C++ Gems*. Cambridge Press University & Sigs Books, 1998.

[29] B. Liskov, D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Technical Report 88, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge, MA, USA, February 1995.

[30] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[31] F. Maes. Program templates: Expression templates applied to program evaluation. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL; in conjunction with PLI)*, number FZJ-ZAM-IB-2003-10 in John von Neumann Institute for Computing (NIC), Uppsala, Sweden, August 2003.

[32] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[33] B. Meyer. Genericity versus inheritance. In *the Proceedings of the Conference on Object Oriented Programming Systems Languages and Aplications (OOPSLA)*, pages 391–405, Portland, OR, USA, 1986.

[34] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[35] S. Meyers. How non-member functions improve encapsulation. 18(2):44–??, Feb. 2000. ISSN 1075-2838.

[36] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *the Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[37] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7 (5):32–35, June 1995.

[38] Olena. Olena image processing library, 2003. URL `http://olena.lrde.epita.fr`.

[39] oonumerics. Scientific computing in object-oriented languages, 2003. URL `http://www.oonumerics.org`.

[40] Y. Régis-Gianas and R. Poss. On orthogonal specialization in C++: Dealing with efficiency and algebraic abstraction in Vaucanson. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Parallel/High-performance Object-Oriented Scientific Computing (POOSC; in conjunction with ECOOP)*, number FZJ-ZAM-IB-2003-09 in John von Neumann Institute for Computing (NIC), Darmstadt, Germany, July 2003.

[41] X. Rémy and J. Vouillon. On the (un)reality of virtual types. URL `http://pauillac.inria.fr/~remy/work/virtual/`. March 2000.

[42] U. P. Schultz. Partial evaluation for class-based object-oriented languages. In *Program as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 2001,*

Proceedings, volume 2053 of *Lecture Notes in Computer Science*, pages 173–198. Springer-Verlag, 2001.

[43] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *the Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[44] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. In *the Proceedings of the 2nd International Conference on Generative and Component-based Software Engineering (GCSE)*, pages 464–478. tranSIT Verlag, Germany, October 2000.

[45] Y. Smaragdakis and B. McNamara. FC++: Functional tools for object-oriented tasks. *Software - Practice and Experience*, 32(10):1015–1033, August 2002.

[46] J. Smith. C++ & multi-methods. *ACCU spring 2003 conference*, 2003.

[47] A. Stepanov, M. Lee, and D. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[48] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *the Proceedings of the 1st Workshop on Template Programming*, Erfurt, Germany, October 2000.

[49] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[50] V. Surazhsky and J. Y. Gil. Type-safe covariance in C++, 2002. URL `http://www.cs.technion.ac.il/~yogi/Courses/CS-Scientific-Writing/exampl%es/paper/main.pdf`. Unpublished.

[51] K. K. Thorup. Genericity in Java with virtual types. In *the Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

[52] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *In the Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.

[53] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[54] T. Veldhuizen. *Expression Templates*, pages 475–487. In [28].

[55] T. L. Veldhuizen. Techniques for scientific C++, August 1999. URL `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.

[56] T. L. Veldhuizen and A. Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2002.

[57] Xt. A bundle of program transformation tools. Available on the Internet, 2003. URL `http://www.program-transformation.org/xt`.

[58] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *the 12th ACM*

Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), volume 32 of *Issue 10*, pages 125–141, Athlanta, GA, USA, October 1997.

A Technical Details

A.1 Implementation of `find_exact`

The `find_exact` mechanism, introduced in Section 3.1, is used to enable classes that are both concrete and extensible within our static hierarchy system. This kind of class is parameterized by `EXACT`: the type of the object effectively instantiated. Contrary to abstract classes, concrete extensible classes cannot propagate directly their `EXACT` parameter to their parents, as explained in Section 3.1. A simple utility macro called `find_exact` is necessary to determine the `EXACT` type to propagate. This macro relies on a meta-program, `FindExact`, whose principle is described in Figure 14. and the corresponding C++ code is given in Figure 15.

FindExact($Type$, $EXACT$)
{
  if $EXACT \neq$ "$Itself$"
    return $EXACT$;
  else
    return $Type < Itself >$;
}

Fig. 14. `FindExact` mechanism: algorithmic description

```
// default version
template <class T, class EXACT>
struct FindExact
{
  typedef EXACT ret;
};

// version specialized for EXACT=Itself
template <class T>
struct FindExact<T, Itself>
{
  typedef T ret;
};

// find_exact utility macro
#define find_exact(Type) typename FindExact<Type<Exact>, Exact>::ret
```

Fig. 15. `FindExact` mechanism: C++ implementation

A.2 Static Dispatch with Virtual Inheritance

Using a `static_cast` to downcast a type does not work when virtual inheritance is involved. Let us consider an instance of EXACT. It is possible to create an `Any<EXACT>` pointer on this instance. In the following, the address pointed to by the `Any<EXACT>` pointer is called "the `Any` address" and the address of the `EXACT` instance is called the "exact address".

The problem with virtual inheritance is that the `Any` address is not necessarily the same as the exact address. Thus, even `reinterpret_cast` or `void*` casts will not help. We actually found three solutions to cope with this issue. Each solution has its own drawbacks and benefits, but only one is detailed in this paper.

The main idea is that the offset between the `Any` address and the exact address will remain the same for all the instances of a particular class (we assume that C++ compilers will not generate several memory model for one given class). The simplest way to calculate this offset is to compute the difference between an object address and the address of an `Any<EXACT>` reference to it. This has to be done only once per exact class. The principle is exposed in Figure 16.

This method has two drawbacks. First, it requires a generic way to instantiate the `EXACT` classes, for instance a default constructor. Second, one object per class (not per instance!) is kept in memory. If an object cannot be empty (for example storing directly an array), this can be problematic. However, this method allows the compiler to perform good optimizations. In addition, only a modification of `Any` is necessary, a property which is not verified with other solutions we found.

A.3 Checking Subclassing Relation

Checking if a subclass of another is possible in C++ using templates. The `is_base_and_derived<T,U>` tool from the Boost [7] `type_traits` library performs such a check. Thus, it becomes possible to prevent a class from being instantiated if the virtual types does not satisfy the required subclassing constraints.

B Conditional Inheritance

Static hierarchies presented in Section 3.1 come with simple mechanisms. These parameterized hierarchies can be considered as meta-hierarchies simply waiting for the exact object type to generate real hierarchies. It is generally sufficient for the performance level they were designed for. In order to gain modeling flexibility and genericity, one can imagine some refinements in the way of designing such hierarchies. The idea of the conditional inheritance technique is to adapt automatically the hierarchy according to statically known factors. This is made possible by the C++ two-layer evaluation model (evaluation at compile-time and evaluation at run-time) [31]. In practice, this implies that the meta-hierarchy comes with static mechanisms to discriminate on these factors and to determine the inheritance relations. Thus, the meta-hierarchy can generate different final hierarchies through these variable inheritance links.

```
template <class EXACT>
struct Any
{
  // exact_offset has been computed statically
  // A good compiler can optimize this code and avoid any run−time overhead
  EXACT& exact() {
    return *(EXACT*)((char*)this − exact_offset);
  }

private:
  static const int exact_offset ;
  static const EXACT exact_obj;
  static const Any<EXACT>& ref_exact_obj;
};

// Initialize an empty object
// Require a default constructor in EXACT
template <class EXACT>
const EXACT Any<EXACT>::exact_obj = EXACT();

// Upcast EXACT into Any<EXACT>
template <class EXACT>
const Any<EXACT>& Any<EXACT>::ref_exact_obj = Any<EXACT>::exact_obj;

// Compute the offset
template <class EXACT>
const int Any<EXACT>::exact_offset =
  (char*)(&Any<EXACT>::ref_exact_obj)
   − (char*)(&Any<EXACT>::exact_obj);
```

Fig. 16. One method to handle virtual inheritance
The offset between the `Any` address and the address of the `EXACT` class is computed once by using a static object. Since everything is static and const, the compiler can optimize and remove the cost of the subtraction.

```
// ...

template <bool b>
struct type_assert
{};

template <>
struct type_assert<true>
{
  typedef void ret;
};

#define ensure_inheritance(Type, Base)      \
typedef typename                            \
  type_assert<                              \
    is_base_and_derived<Base, Type>::value \
  >::ret  ensure_##Type

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

  // Will not compile if  point_type  is  not a Point since  ret
  // is  not  defined  if  the  assertion  fails .
  ensure_inheritance(point_type, Point<point_type>);
};

// ...
```

Fig. 17. Specifying constraints on virtual types

To illustrate the conditional inheritance mechanism, we introduced a UML-like symbol that we called an *inheritance switch*. Figure 18 gives a simple use case. This example introduces an image hierarchy with a concrete class whose inheritance is conditional: `SpecialImage`. `SpecialImage` is parameterized by an unsigned value `Dim`. We want this class to inherit from `Image2d` or `Image3d` depending on the value of `Dim`. `SpecialImage`'s inheritance is thus represented by an inheritance switch. Figure 19 presents the corresponding C++ code. The inheritance switch is implemented by the `ISwitch` trait parameterized by the dimension value. Its specialization on 2 (resp. 3) defines `Image2d` (resp. `Image3d`) as result type. Finally, `SpecialImage<Dim>` only has to inherit from `ISwitch<Dim>`'s result type.

The factors on which inheritance choices are made are necessarily static values. This includes types, provided by *typedef*s or parameterization, as constant integer values. The effective factors are not necessarily directly available data but can be deduced from static pieces of information. Trait structures can then be used to perform more or less complex information deductions. One should also note that the discriminative factors must be accessible while defining the hierarchy. This implies that these factors must be independent from the hierarchy or externally defined. In practice, class-related factors can be made available outside the hierarchy thanks to trait structures and polymorphic *typedef*s (see Section 3.6).

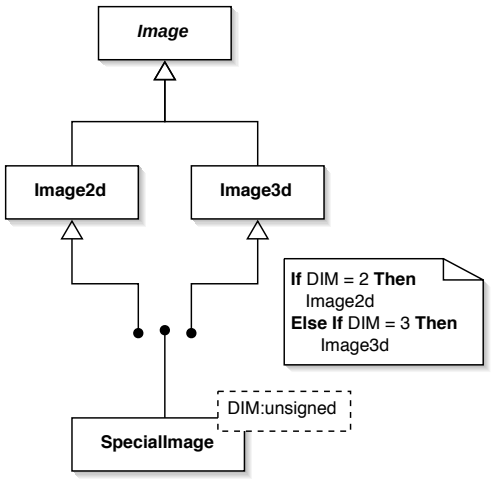


Fig. 18. Simple conditional inheritance sample: UML-like description.

Conditional inheritance mechanisms become particularly interesting when objects are defined by several orthogonal properties. A natural way to handle such a modeling problem is to design a simple sub-hierarchy per property. Unfortunately, when defining the final object, the combinatorial explosion of cases

```
class Image
{
  // ...
};

class Image2d: public Image
{
  // ...
};

class Image3d: public Image
{
  // ...
};
```

```
template <unsigned Dim>
struct ISwitch;

template <>
struct ISwitch<2>
{
  typedef Image2d ret;
};

template <>
struct ISwitch<3>
{
  typedef Image3d ret;
};

template <unsigned Dim>
class SpecialImage
  : public ISwitch<Dim>::ret
{
  // ...
};
```

Fig. 19. Simple conditional inheritance sample: C++ code

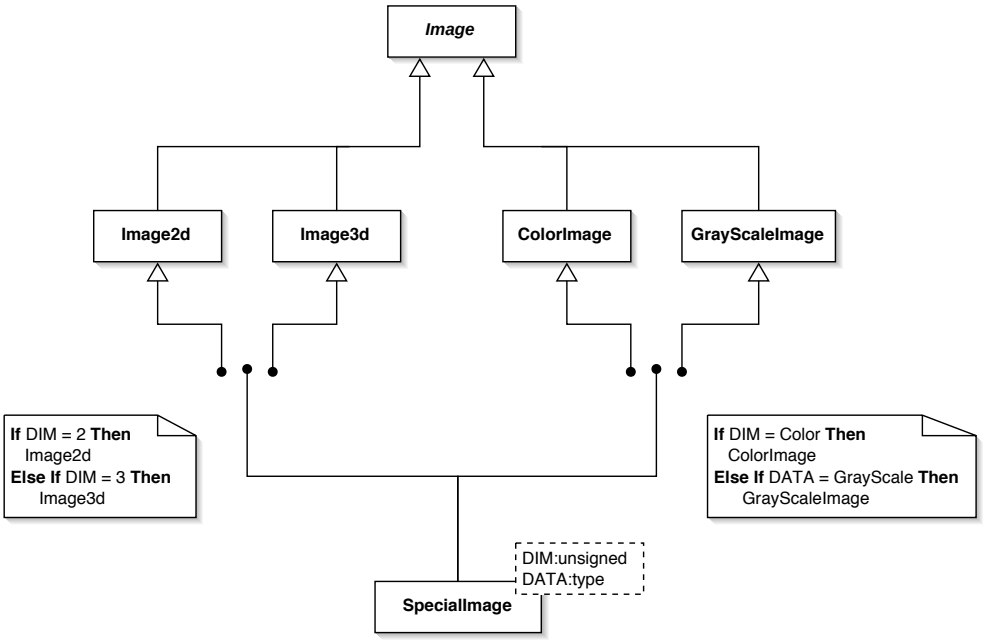


Fig. 20. Conditional inheritance: multiple orthogonal factors.

usually implies a multiplication of the number of concrete classes. Figure 20 illustrates an extension of the previous image hierarchy, with more advanced conditional inheritance mechanisms. We extended the image hierarchy with two classes gathering data-related properties, `ColorImage` and `GrayScaleImage`. The hierarchy is now split into two parallel sub-hierarchies. The first one focuses on the dimension property while the second one focuses on the image data type. The problem is then to define images that gather dimension- and data-related properties without multiplying concrete classes. The idea is just to implement a class template `SpecialImage` parameterized by the dimension value and the data type. Combining conditional and multiple inheritance, `SpecialImage` inherits automatically from the relevant classes. This example introduces the idea of a programming style based on object properties. A `SpecialImage` instance is only defined by its properties and the relevant inheritance relations are deduced automatically.

Finally, mixing conditional inheritance mechanism with other classical and static programming techniques results in powerful adaptive solutions. This work in progress has not been published yet.

B.3 SEMANTICS-DRIVEN GENERICITY : A SEQUEL TO SCOOP

Semantics-Driven Genericity : A Sequel to the Static C++ Object-Oriented Programming Paradigm. *Workshop on Multiparadigm Programming with Object-Oriented Languages* (**MPOOL**). At *European Conference on Object-Oriented Programming* (**ECOOP**), July 2008. Paphos, Cyprus. With Roland Levillain.

Semantics-Driven Genericity: A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2)

Thierry Géraud, Roland Levillain

EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire, FR-94276 Le Kremlin-Bicêtre Cedex, France
thierry.geraud@lrde.epita.fr, roland.levillain@lrde.epita.fr
Web Site: http://www.lrde.epita.fr

Abstract. Classical (unbounded) genericity in C++03 defines the interactions between generic data types and algorithms in terms of *concepts*. Concepts define the requirements over a type (or a parameter) by expressing constraints on its methods and dependent types (`typedef`s). The upcoming C++0x standard will promote concepts from abstract entities (not directly enforced by the tools) to language constructs, enabling compilers and tools to perform additional checks on generic constructs as well as enabling new features (e.g., concept-based overloading). Most modern languages support this notion of signature on generic types. However, generic types built on other types and relying on concepts—to both ensure type conformance and drive code specialization—restrain the interface and the implementation of the newly created type: specific methods and associated types not mentioned in the concept will not be part of the new type. The paradigm of concept-based genericity lacks the required semantics to transform types while retaining or adapting their intrinsic capabilities. We present a new form of semantically-enriched genericity allowing static, generic type transformations through a simple form of type introspection based on type metadata called *properties*. This approach relies on a new Static C++ Object-Oriented Programming (SCOOP) paradigm, and is adapted to the creation of generic and efficient libraries, especially in the field of scientific computing. Our proposal uses a metaprogramming facility built into a C++ library called STATIC, and doesn't require any language extension nor additional processing (preprocessor or transformation tool).

1 Introduction

In the context of software library design for numerical scientific computing, we want to meet two major objectives at the same time: *efficiency* because of the large data sets to be processed, and *genericity* since data can have multiple different types. Many libraries fulfilling such aims have been designed and developed with the C++ language [1], which has proved to be an appropriate tool [2,3,4,5]. The way *abstractions* can be handled in scientific libraries is a subject

of prime importance: practitioners mainly think about the entities of their domain in terms of abstractions, and consider implementation aspects in second place. A great benefit brought by the C++ language comes from the fact that it features multiple paradigms; as a consequence, it offers several solutions to design well-grounded scientific libraries where implementation classes are tightly related to properly defined abstractions.

The Generic Programming (GP) paradigm is classically presented as a means to achieve both efficiency and genericity in scientific libraries, leading to a strong decoupling between:

- *structures*, the different kinds of containers offered to represent data sets;
- *values*, the different types of elements that can be stored in structures;
- and *algorithms*, some non-elementary operations that are expected to run over data sets.

Schematically, providing S structure types, V value types, and A generic algorithms—i.e $S + V + A$ entities—a library features $D = S \times V$ different data sets (input types) and $D \times A$ possible processing routines. Yet, another category of entities happens to be useful:

- *data modifiers*, such as views, adapters, decorators, wrappers, or any transformation applied on a single or several data sets to provide the user with "other" data sets than the primitive ones.

The obvious requirements over such modifiers are those we already have for the other entities. They shall not penalize performance at run-time. They also have to be generic, that is, they shall apply to as many data sets as possible. Therefore with M modifiers, the number of expressible data types becomes $D^{(M^*)}$. In addition to the efficiency and genericity requirements, their main property is the following: *a modifier type is written once while being able to modify the data types it applies to, no matter the specificity of the original data types.*

Some examples of such modifiers already exist. For instance, the Boost Iterator Library [6] proposes a collection of "specialized adaptors." Though they do not directly apply on containers (data sets) but on iterators (data access), they affect the behavior of algorithms just like if the nature of the containers had changed. All iterators do not share the exact same interface, so the adapter classes have to handle this variability; thus, tag types are introduced in order to discriminate between the different abstractions of iterators. Given an iterator of type `I`, its tag can be retrieved through `iterator_traits<I>::iterator_category`, that is, an associated type enclosed in a traits class. When the result is `random_access_iterator_tag`, we know that `I` satisfies the corresponding abstractions. An adapter class targeting the type `I` shall then rely on this piece of information to offer the appropriate interface and behavior.

In a generic library featuring many different abstractions, several of them being independent from some other ones, providing dedicated *modifier* classes on a per-case basis is clumsy. We really need a very general mechanism to easily design these classes whose main characteristic is that they should be highly adaptable.

To address these problems, we have extended the Static C++ Object-Oriented Programming (SCOOP) paradigm presented in [7]. The need for this paradigm comes from the development of two numerical scientific computing libraries, namely OLENA [8,9] for image processing and VAUCANSON for finite automata manipulation [10,11]. SCOOP 1 proposed an approach mixing classical Object-Oriented Programming (OOP) and GP, but no support to implement generic modifiers. This lack finds its origins in the limitation of existing paradigms. To allow the creation of generic modifiers, we propose to reverse classical OOP, by introducing the idea of *properties* within the second version of SCOOP. While OOP is a top-down process (upper classes imposes type constraints on lower classes), properties spawns a bottom-up mechanism: concrete classes express a kind of signature from their properties, which is used to select the abstractions they conform to, and possibly, retrieve automatically some implementations based on user-written rules.

This paper is structured as follows: first, we introduce the idea of properties by studying existing GP-based paradigms in Section 2. Then, we present the SCOOP 2 paradigm (Section 3), and a component to apply it to generic libraries, STATIC (Section 4). We compare our approach to existing work in Section 5. Section 6 concludes.

2 The need for properties

This section compares various approaches to express generic programming in the C++ language, and draws a first conclusion on the current GP techniques with respect to the implementation of generic modifiers: these paradigms fail to meet our expectations. A proposition introducing the concept of properties is then presented.

2.1 Classic genericity: C++03

In contrast with OOP, generic programming does not require class inheritance but heavily relies on parametric polymorphism thanks to the C++ `template` keyword. Libraries built with GP [2,5,12,4] are efficient since the run-time cost of `virtual` methods is avoided; furthermore, methods code can be inlined which allows the compiler for extra optimizations. Abstraction interfaces are not mapped into source code but are described in documentation. The relationship between implementations and abstractions is *implicit*: an implementation class just has to provide what is required by its abstractions.

2.2 Concept-based genericity: C++0x

With the evolution of the C++ Standard, programmers will benefit from the introduction of the `concept` keyword into the upcoming C++0x proposal [13] in order to materialize into source code a set of requirements over types, a feature present in popular modern languages [14]. For instance, an abstraction

interface could be mapped into a C++0x concept and, given a generic algorithm—a parametrized function—this concept could be used to define a constraint over eligible parameters. This evolution thus preserves all the benefits from "classical" generic programming while enhancing code expressiveness, program safety, and design capabilities. With the advent of concepts comes the notion of where-clauses, a language construct to enforce a set of requirements using concepts on a type. Where-clauses allow a new form of function overloading based on concepts, which greatly enhances algorithm specialization [15].

Though we will have to wait for the next C++ standard to be able to manipulate concepts as entities of the language, there are already tools partially reifying concepts for C++03 [16]. They help to enforce early conformance of types to the concepts they model, and provide better error messages. Likewise, a form of concept-based overloading is possible with C++03 [17]. However, these techniques have less expressiveness and are less robust than their future C++0x counterparts.

2.3 Mixing Genericity and OOP: SCOOP 1 (using C++03)

In [7] we proposed a solution to translate a classical, hence dynamic, OOP class hierarchy into a *static* one. Put shortly, SCOOP 1 is a mix between OOP and GP in order to take advantage of both worlds. From OOP it borrows inheritance so that interfaces are explicitly defined as classes; SCOOP then allows for code factoring, even if some definitions (methods or associated types) are still unknown and deferred to sub-classes. From GP it keeps efficiency since all classes are parametrized and statically resolved. Technically speaking SCOOP 1 is based on a generalization of the Curiously Recurring Template Pattern [18]. The main difference between SCOOP 1 and C++0x GP is that the relationship between a concrete class and an abstraction is *explicit* with the former (as in OOP), whereas it is *implicit* with the latter.

2.4 Limitations of these approaches

The main difficulty of implementing generic modifiers comes from handling specificity. To illustrate our point, let us consider a modifier class that acts as an adapter. For instance, in the C++ standard library the class template `std::queue` is an adapter over an underlying sequence. Some eligible type `C` for the sequence can be `std::list<T>` (doubly linked list) or `std::deque<T>` (double-ended queue), `T` being the queue element type. Let us imagine that this queue type *is a modifier*. Under this assumption, its interface can provide a random access facility to queue elements if and only if `C` already features this same facility. That is the case when `C` is a deque but not when it is a list. The interface of `queue<T,C>`, and its related code, thus *depends upon the interface of its parameter* `C`. With current C++, `std::queue<T>` is not a modifier, and it masks the interface of `T`. This is an artifact of classical genericity that we call *restraining genericity*.

The first restriction imposed by modifiers is that they prevent the use of inheritance as a means to build upon an existing type. Modifiers are much more generic than sole decorators or proxies: they shall allow any transformation from a generic (and equipped) type, including *changing its interface* (and therefore its implementation). Let us consider an example from the domain of image processing: if we create a transformation flattening a 3-dimensional image to a 2-D one (i.e., a generic modifier reducing the dimension of a generic 3-D image type), we need to adjust the interface of the image type with respect to the type of points, the type of the grid, etc. (i.e, translate them to their 2-D counterparts).

This example shows that Classical GP is too limited to implement modifiers. Moreover, it seems the new features of C++0x evoked in section 2.2 will not suffice to implement modifiers directly either: using a type `T` conforming to a given concept `C` to create a new type `U` will not take into account the specific part of the interface of `T` absent from `C` if `T` models in fact a *refinement* of `C` (i.e., a sub-concept). Likewise, SCOOP 1 type transformations are only limited to a fixed type signature. In any case, there is no automatic way to retrieve default implementation while transforming types, based on a given interface (no static introspection mechanism).

The main idea of this paper is that type transformations should be available as a GP feature, and require an *extended generic programming paradigm* to be implemented.

More generally, though templates offer useful static metaprogramming tools in C++, the language itself lacks a general-purpose static meta-programming facility, like complete static introspection or a meta-object protocol [19].

Some modern languages like Java and C# offer a *dynamic* introspection mechanism, but this is a different service: we are looking for a static means to interact with the compilation process. For instance, the C++ language does not provide a actual means to inspect the `typedef`s of a class, though this information is required to create non-trivial type transformations.

2.5 Semantics-driven genericity

Modifiers should therefore perform some kind of static introspection on the transformed type, to acquire information on its interface and its implementation. This work will be performed by metaprogramming algorithms [20], so the information has to be encoded as types.

One cannot easily extract information from a C++ program without an external tool, in order to perform type transformations. Thus, the programmer has to provide the compiler with some information characterizing the very nature of the type to be transformed. With well known template-based static metaprogramming techniques, the compiler will be able to both *fetch* information form the type (rather than *inspect* it) and even rely on existing implementations through a delegation mechanism. The idea is that abstractions should be able to express the requirements of type transformations on data using *properties*, that is, parameters depending on the exact type of the implementation class. They should therefore be part of the definition of concrete classes of the hierarchy.

This separation lays down a first issue in C++, as it introduces a recursion in the definition of a class: a concrete class needs to build its super classes first; but these super classes, also used to represent the abstractions modeled by the concrete class, depend on types defined in the concrete class.

We must first recall that the solution cannot solely rely on direct inheritance from the transformed type: as the semantics of the transformation does not always allow it. For instance, given a 3-D image of type `T`, it would be tempting to define a slice (a 2-D image) of an object of type `T` as an instance of a subclass of `T`: a slice of a 3-D image is indeed a 2-D image; a 2-D image, however *is not* a 3-D image, and has a different interface. Nevertheless, inheritance is the only usual way to have a type `U` automatically retrieve (a part or all of) the implementation of a type `T` in C++. Therefore we propose a programming paradigm based on

- a split pattern, with abstractions expressing concepts using *abstract types* (depending on the exact type of the model) on the one hand; and implementations classes on the other hand;
- a set of properties attached to any concrete class of the hierarchy, used to give a "value" to the expected types in the signature of routines of abstractions. Properties add semantics from the target domain (e.g., image processing) to data types, and are used to automatically drive the inheritance relationships from concrete classes to the right abstractions;
- a delegation mechanism to automatically retrieve implementation from a *delegatee* to create generic type transformations.

These ideas are the heart of the Scoop 2 paradigm, which enriches the semantics of a library: qualifying a type with is properties and using them to retrieve both interfaces and implementations statically augment the expressiveness of the language.

3 The SCOOP 2 paradigm

Scoop (standing for *Static C++ Object-Oriented Programming*) is a programming paradigm that addresses common issues raised in the context of generic and efficient programming. Its first version has been described by Burrus *et al.* [7], and was used to build the Olena generic image processing library [8], version 0.10. This section presents the second version of the paradigm, which has been design for the forthcoming Olena library [21].

Scoop 2 has been developed primarily to fulfill the needs of designers of scientific computing libraries (especially Olena and Vaucanson), where genericity and run-time efficiency are fundamental.

Because the paradigm has an impact on the design of the software, Scoop 2 aims mainly at building new libraries. However, one can use it within an existing code base, either non intrusively (by wrapping existing data structures), or intrusively, by modifying existing code. We have successfully applied the first approach to a subset of the standard C++ library (some containers and algorithms), as a proof-of-concept of Scoop 2 [22].

One of the main advantages of SCOOP 2 is that this paradigm is expressed directly in the host language (C++ to be specific), hence it doesn't require any software tool other than a compiler complying with the ISO/IEC C++ Standard [1]. Nevertheless, to both formalize the implementation of the paradigm and factor out the development among clients, we propose a library providing a part of the facility used to apply the paradigm: types and metaprogramming algorithms. This component is called STATIC, because most of its code is used to create so-called *static class hierarchies*, where the exact type of every object is known at compile-time, replacing dynamic dispatch of method calls with static ones and allowing some form of static class introspection. STATIC is built upon METALIC, another library delivering basic C++ template metaprogramming tools (manipulation of types as values, logic on types, static `if` and `switch` statements on types expression, `typedef` lookup, etc.). METALIC shares similar goals with Loki [23] and the Boost MPL library [24,25], and could be replaced by these libraries in a future implementation of STATIC. Notably, the atomic conformance check technique (for typedef introspection and subtype checks), based on SFINAE and `sizeof`, is from Loki [23, section 2.9]. Section 4 is dedicated to STATIC. Please note that unexplained elements from examples illustrating STATIC uses (figures 4, 6 and 8) are addressed in later parts of the article to enhance readability. Their object is basically to show how STATIC client code reads, as these samples are actual code.

3.1 Organizational Considerations

The design of SCOOP determines the roles of people taking part in a SCOOP-based library project. We can classify these actors in a taxonomy of four categories.

SIMPLE USERS For this kind of actor, the components provided by the library (data types and algorithms) are sufficient to solve a problem of the applicative domain. A SIMPLE USER thus just assembles components and knows nothing about the library internals.

DESIGNERS This actor designs new algorithms, so he extends the number of algorithms (A).

PROVIDERS This actor provides the library with new data structures, hence he increases the number of data structures (S), and possibly the number of value types (V).

ARCHITECTS At the opposite of SIMPLE USERS, this category of actors is mainly concerned by the library internals.

A fifth category, MAINTAINERS, contains people in charge of the low-level components above which the library is built. As they are not directly related to the library, their role will not be covered in this section.

The programming skills expected from an actor to enter one of those categories usually go increasingly from SIMPLE USERS to ARCHITECTS; fortunately, the size of the population of those categories follows an opposite trend. However,

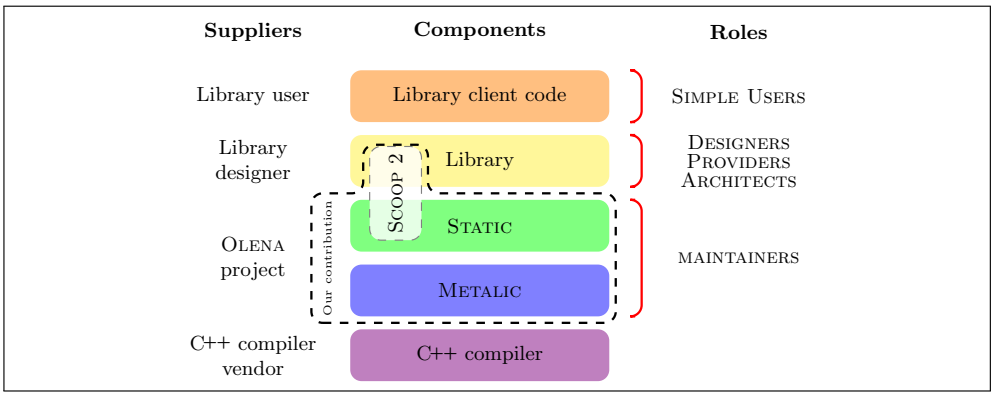


Fig. 1. Components and actors in a Scoop 2-based project. The vertical layout represents the dependencies between components: a component located on top of another depends on the latter.

as a matter of fact in the context of scientific libraries, a lot of people are involved in designing algorithms and there are a lot more Designers than Providers. A quality criterion for libraries is that the compliance between algorithms and data should be maximal. Having also in mind the notions of reusability and extensibility, this criterion has two consequences that respectively fall in the hands of Designers and Providers. Figure 1 summarizes the roles of each user.

3.2 Key concepts of the design

Scoop 2 is to be used to build libraries composed of generic algorithms (as function templates) and generic data structure (as class templates). Conceptually, Scoop 2 pays more attention to the *data* aspect than many other generic library designs. Indeed, a lot of these libraries draw their generic orientation from their algorithms, and data structures are often considered as mere models of the concepts used by these algorithms. Although these libraries provide means to extend data structures through adapters or wrappers [2,12,26], these approaches benefit little or nothing from the generic strategy (from the reusability and factoring point of view): adapters and wrappers are generally not meant to be combined.

3.3 Data structures

The developer of the library shall organize the various entities of the domain in *categories*. For instance, a generic image processing library would comprise categories such as image, point, neighborhood, etc. For each category of data structure, Scoop 2 invites the Designer to create two class hierarchies: one for the abstraction(s) (reified as "concepts" in C++0x) and one for the implementation(s). Following the previous example, corresponding abstractions for

the category image would be Image2d, Image3d, BinaryImage, ColorImage, MutableImage, etc.

Hierarchy of abstractions Concepts are expressed as abstract classes in the paradigm. Although the upcoming C++ Standard is to propose concepts as actual language constructs [13], these are not powerful enough to fully support SCOOP 2. "Concepts-as-classes" is notably a requirement of the delegation mechanism of the paradigm (see 3.3). The hierarchy is used as a means to organize the concepts according to their acquaintances (refinement, orthogonality, exclusion). The generalization relationship mimics a *refinement* in this hierarchy. Abstractions are detailed in 3.3.

Hierarchy of implementations Actual data structures are organized in class hierarchies, which serve to factor implementations. Data types are either *primary* (not relying on any other data type of the same kind) or *composed* (aggregating one or more objects of the same kind). In the latter case, the paradigm uses a powerful delegation mechanism allowing type transformations and retaining the semantics of the original type, a claim brought up in Section 2. Section 3.3 gives more insight on implementation hierarchies.

Both hierarchies can be extended independently. This is immediate for implementations, thanks to inheritance—and because properties are automatically inherited, thanks to STATIC (see section 4.3). Inheritance allows an extension from the *bottom* of the hierarchy, which fits with the location of implementations within the whole class diagram. However, things get a little trickier when it comes to abstractions, as existing implementations might already inherit from them. Extending an abstraction hierarchy *a posteriori* requires some equipment to allow a form of declarative inheritance and have implicit links between implementation and abstraction be extended non intrusively.

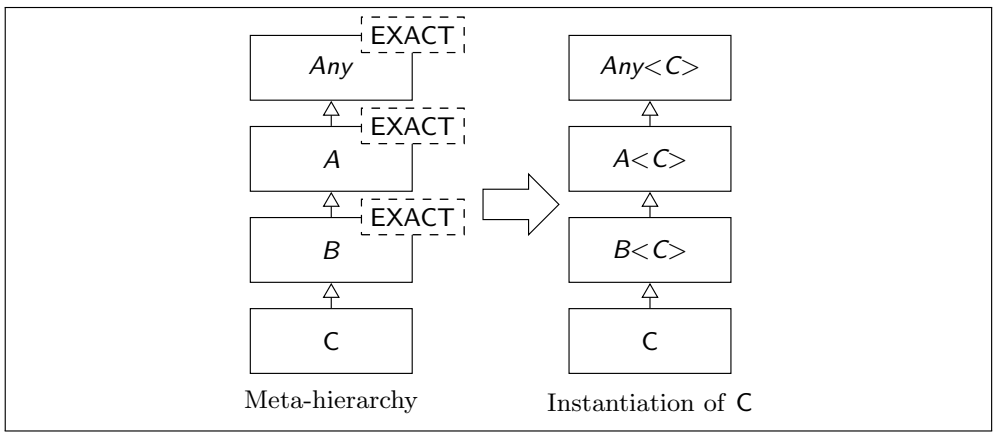


Fig. 2. An example of Static hierarchy unfolding.

Implementation hierarchies Implementation classes are templates organized in hierarchies using only single inheritance (though the paradigm can be extended to support multiple inheritance). Class hierarchies follow a generalization of the Curiously Recurring Template Pattern (CRTP) [18]. In the Generalized Curiously Recurring Template Pattern (GCRTP) [7], each derived class pass its type as parameter to its super class recursively, so that every super class knows the exact type of the object. (see Figure 2). The idea of GCRTP was previously evoked by Veldhuizen [20].

STATIC comes with a top-level class `Any` to make the integration of the GCRTP easier, and a routine `exact` returning a pointer with the exact type of an object, allowing static dispatch of method calls. GCRTP-based hierarchies are known as *static hierarchies*. Figure 3 gives an example of such a hierarchy. The implementation of `Any` is addressed in section 4.2.

```
// Abstractions.
template <typename Exact>
struct A : public Any<Exact> {
  // Static dispatch.
  void m () { exact(this)->impl_m (); }
};

template <typename Exact>
struct B : public A<Exact> {
  // Static dispatch.
  void n () { exact(this)->impl_n (); }
};

// Implementation.
struct C : public B<C> {
  // Implementations of m() and n().
  void impl_m () { /* ... */ }
  void impl_n () { /* ... */ }
};
```

Fig. 3. An example of static hierarchy.

The paradigm relies on *properties*, which determine the nature of data structures of the library and to which abstractions they conform—or, in other words, which concepts they model. The link between implementations and abstractions can be either explicit (thanks to traditional inheritance; this approach is similar to using a `concept_map` in C++0x) or implicit (computed at compile time from the properties of the structure). Section 3.3 elaborates on this subject. These properties are expressed for each class as *virtual types* (see 3.3). In the following, we call SCOOP *classes* the types that are part of a SCOOP static hierarchy. Figure 4 shows some SCOOP classes from the OLENA library.

Abstraction hierarchies SCOOP 2 uses classes to reify abstractions, the entities which express the concepts of the library. These classes are akin to Java interfaces, in the sense that

```
// image_base.
// -----------
template<typename Exact> class image_base;

template<typename Exact>
struct super_trait_< image_base<Exact> > { typedef top<Exact> ret; };

template<typename Exact>
struct vtypes< image_base<Exact> > {
private:
  typedef stc_deferred(point) point_;
public:
  // Virtual type 'category' is used by top<Image> to
  // link image_base to the right abstraction(s).
  typedef stc::final< stc::is<Image> >      category;
  typedef stc::final<typename point_::grid> grid;
  typedef stc::final<point_>                psite;
};

template <typename Exact>
struct image_base
  // Implicit (computed) link between image_base
  // and its abstraction(s).
  : top<Exact> {
  // ...
protected:
  image_base () {};
};


// image2d.
// --------
template<typename T> class image2d;

// image2d inherits properties from image_base.
template<typename T>
struct super_trait_< image2d<T> > { typedef image_base< image2d<T> > ret; };

template<typename T>
struct vtypes< image2d<T> > {
  typedef point2d point;
  typedef T       value;
};

template <typename T>
class image2d : public image_base< image2d<T> > {
public:
  typedef image2d<T> self;
  typedef image_base<self> super;
  // Import some virtual types of image2d inside
  // the scope of the class.
  stc_using(point);
  stc_using(value);
  // Implementation of Image::operator().
  value impl_read(const point& p) const { /* ... */ }
  // ...
};
```

Fig. 4. Some SCOOP 2 (implementation) classes for the category image from the OLENA library (abridged). `stc_deferred` and `top` are explained in sections 4.3 and 4.4 respectively. As for `stc::is<Image>`, a common way to "tag" classes as belonging to a given a category is to set their virtual type `category` to `stc::is<Abs>`, where `Abs` is their topmost abstraction. `Image` is an abstraction presented in Figure 6.

– their member shall be generic routines or dispatchers;
– they shall carry no data (attributes);
– they shall define no virtual type, though they can make use of them (i.e., use virtual types that will be defined in implementation classes).

Abstractions are passed the exact type of the class being instantiated, like any super class in the GCRTP. Concrete types may fulfill several abstractions (through inheritance) to model several concepts, so as to uncouple the requirements of various algorithms and add statically dispatched multimethods to the library. Static multimethods were a part of Scoop 1, and are covered in [7]. Two (or several) concepts of a library can be

related through a refinement relationship The requirements of a concept can form a superset of another one, which translates to a relation of generalization (inheritance) between their corresponding abstractions. For example, a BinaryImage is a refinement of a LabelImage.

concurrent A given concrete class can logically model one of several concurrent concepts, because they are from the same domain (implying their corresponding abstractions belong to the same hierarchy). For instance, ColorImage and GrayLevelImage are concurrent abstractions.

orthogonal They express unrelated requirements (their abstractions belong to different hierarchies). As an example, Image2D and ColorImage are orthogonal abstractions.

The abstractions of our library are thus organized as *orthogonal hierarchies* (see Figure 5). Figure 6 gives an example of abstractions corresponding to the Scoop classes of Figure 4.

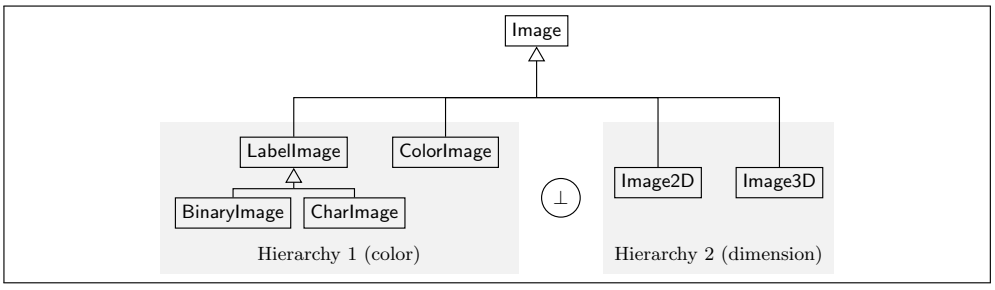


Fig. 5. An example of hierarchies of abstractions. Abstractions belonging to category image are all sub-abstractions of Image. Abstraction BinaryImage is a refinement of LabelImage. Though this is not enforced by any language construct, abstractions LabelImage (and its sub-abstractions) are semantically concurrent with ColorImage , as they are part of the same (logical) hierarchy. Likewise, abstractions of hierarchy 1 are orthogonal to abstractions of hierarchy 2.

```
// Abstractions.
// -------------
template <typename Exact>
struct Image : public virtual Any<Exact> {
  stc_typename(grid);   // Type of grid.
  stc_typename(point);  // Type of point.
  stc_typename(psite);  // Type of point site.
  stc_typename(value);  // Type of value.
  // Return the value at site P.
  value operator()(const psite& p) const {
    return exact(this)->impl_read(p);
  };
  // ...
protected:
  Image() {};
};


// Abstraction sub-hierarchy for value kind.
// -----------------------------------------
template <typename Exact>
struct LabelImage : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct BinaryImage : public LabelImage<Exact> { /* ... */ };

template <typename Exact>
struct StringImage : public LabelImage<Exact> { /* ... */ };

template <typename Exact>
struct ColorImage : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct DataImage : public virtual Image<Exact> { /* ... */ };


// Abstraction sub-hierarchy for dimension.
// ----------------------------------------
template <typename Exact>
struct Image1D : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct Image2D : public virtual Image<Exact> { /* ... */ };

template <typename Exact>
struct Image3D : public virtual Image<Exact> { /* ... */ };

// Other abstraction sub-hierarchies
// ...
```

Fig. 6. Some abstractions for the category image from the OLENA library (abridged). Note that there are no virtual type definition here, only uses of them.

Link between implementation and abstraction(s) Linking an implementation class to its abstraction(s) can either be *explicit* (manifest) or *implicit* (computed from the properties of the exact type of this class).

Explicit link In this case the link is an inheritance relationship between the implementation and the abstractions corresponding to the modeled concepts. As this type of link is explicitly written by the implementer, it is said to be "hard". It cannot be changed afterwards (other than by altering the definition of the implementation), which might break the extensibility of the library with respect to the abstractions and the type transformations that might be applied to the class.

Implicit link One of the new features of Scoop 2 is the ability to express the link between an implementation class of a given category and its abstractions as *rules on the properties* of this class, called *selectors*. For instance, a class ima from the category image whose properties contains a virtual type `grid` (type of grid) set to `grid2d`, shall inherit from the abstraction Image2D. In addition, if this class has two virtual types `psite` (type of point site) and `point` (type of point) having the same value (`point2d`), ima shall inherit from PointWiseAccessibleImage2D instead, which refines both the concepts Image2D and PointWiseAccessibleImage.

```
Stc-Is-A(source, target, abstraction)
 1  val ← Find(source, target)
 2  return Mlc-Is-A(val, abstraction)

Image-Value-Kind-Selector(source)
 1  switch
 2     case Oln-Is-Binary(Find(source, value)) : return BinaryImage
 3     case Oln-Is-String(Find(source, value)) : return StringImage
 4     case Oln-Is-Label(Find(source, value)) : return LabelImage
 5     case Oln-Is-Color(Find(source, value)) : return ColorImage
 6     case default : return DataImage

Image-Dimension-Selector(source)
 1  switch
 2     case Stc-Is-A(source, grid, Grid1d) : return Image1D
 3     case Stc-Is-A(source, grid, Grid2d) : return Image2D
 4     case Stc-Is-A(source, grid, Grid3d) : return Image3D
```

Algorithm 1: A selector used in the Olena library for the category image, written as pseudo-algorithm. The C++ template metaprogramming version is given in Figure 7. The algorithm Find is detailed in section 4.3. Mlc-Is-A is a metaprogramming algorithm provided by Metalic, while Oln-Is-Binary, Oln-Is-String, Oln-Is-Label and Oln-Is-Color are provided by Olena.

STATIC provides a means to express this semantics-driven (or property-based) inheritance. The system is based on a declarative mechanism using METALIC's meta-`switch`es on types. For each orthogonal trait of a category in the library (e.g., for a category `image`: image dimension, kind of value held by pixels), the DESIGNER writes cases as specializations of the meta-`case` statement. As METALIC's `case` statements are numbered template specializations, ARCHITECTS of the library or PROVIDERS of algorithms can extend the set of selectors non-intrusively by supplying additional `case`s, (though the order of the cases cannot be changed in the current implementation). Some rules linking the implementations of Figure 4 to the abstractions of Figure 6 are given in Algorithm 1. Figure 7 shows the C++ implementation of the corresponding selector.

Contrary to the concept-based approach which relies on where-clauses and either structural conformance or explicit modeling rules (through `concept_map`s) [13], the SCOOP 2 approach uses a property-based template metaprogramming algorithm to express the rules linking implementations to abstractions. This feature allows the library DESIGNER to write generic types based on other types of the library (modifiers, which we also call *morphers*), while considering their intrinsic specificity (see 3.3). At the present time this approach seems intractable using solely C++0x's concepts. The mechanism is explained in section 4.4.

Virtual Types The cornerstone of our proposal is the use of virtual types to express properties. Semantically, a virtual type of a SCOOP class plays the role of a polymorphic associated type of this class (or a virtual `typedef` in the C++ terminology). A virtual type is a type declaration or definition whose "value" is an actual C++ type. Like virtual member functions, virtual types are inherited and can be overridden in derived classes. Such redefinitions applies to the super classes, like polymorphic methods in the Inclusion Paradigm; that is, the value of a virtual type is always computed from the exact type.

A virtual type can be set **Abstract** in a class, i.e. declared with no definition; it might be then redefined in subclasses to be given an actual "value". Using a SCOOP class having at least one abstract virtual type is considered invalid in the SCOOP 2 paradigm. While STATIC cannot automatically enforce this check when instantiating the type, it provides some equipment macros to check the soundness of instantiated classes afterwards. Otherwise, (previously unchecked) invalid uses are caught the first time the abstract virtual type is used (see the algorithms FIND and CHECK in section 4.3). A virtual type can also be tagged **Final**, with the same meaning as its homonym in the Java programming language, i.e. to forbid redefinitions of this virtual type in subclasses.

Virtual types are used to define associated types like traits in most C++ generic libraries. Generic algorithms make use of these virtual types to abstract and generalize their definition ensuring maximum reusability [27]. *Defining* virtual types is very similar to defining traits, although SCOOP 2 cannot just rely on traits to *use* virtual types. It require metaprogramming algorithms to handle specific features like virtual type inheritance, **Abstract**, **Final** and delegation (see 3.3). The implementation of virtual types is discussed in section 4.3.

```
// Selector #1: value kind.
typedef selector<Image, 1> Image_value_kind;

template <typename Exact>
struct case_< Image_value_kind, Exact, 1 >
  : where_< value::is_binary<stc_get_type(value)> > {
  typedef BinaryImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 2 >
  : where_< value::is_string<stc_get_type(value)> > {
  typedef StringImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 3 >
  : where_< value::is_label<stc_get_type(value)> > {
  typedef LabelImage<Exact> ret;
};
template <typename Exact>
struct case_< Image_value_kind, Exact, 4 >
  : where_< value::is_color<stc_get_type(value)> > {
  typedef ColorImage<Exact> ret;
};
template <typename Exact>
struct default_case_< Image_value_kind, Exact > {
  typedef DataImage<Exact> ret;
};

// Selector #2: image dimension.
typedef selector<Image, 2> Image_dimension;

template <typename Exact>
struct case_< Image_dimension, Exact, 1 >
  : where_< stc_is_a(grid, Grid_1D) > {
  typedef Image1D<Exact> ret;
};
template <typename Exact>
struct case_< Image_dimension, Exact, 2 >
  : where_< stc_is_a(grid, Grid_2D) > {
  typedef Image2D<Exact> ret;
};
template <typename Exact>
struct case_< Image_dimension, Exact, 3 >
  : where_< stc_is_a(grid, Grid_3D) > {
  typedef Image3D<Exact> ret;
};

// Other selectors.
// ...
```

Fig. 7. A selector from the Olena library (C++ template metaprograms). `stc_is_a` relies on a Metalic algorithm to test the is-a relationship (see also Algorithm 1).

Delegation SCOOP 2 features a delegation mechanism allowing any concrete instance of a static hierarchy to declare another object as being its *delegatee* (the former object becoming then a *delegator* to the latter). For practical purposes, the delegator holds a reference to its delegatee.

The virtual type `delegatee`, when defined for a type `T` (e.g., set to type `D`), alters the behavior of STATIC. In fact, `T` may use information from zero, one or two of these branches:

- the inheritance branch, possibly defined by the `super` relationship ;
- the delegation branch, defined by the `delegatee` virtual type.

Thus, in addition to the virtual types of its (possible) super class, `T` inherits from the virtual types of `D` by default. In case of conflict (i.e., a virtual type defined in both branches), the inheritance supersedes the definition of the delegatee. Moreover, if `T` and `D` are linked to their abstractions thanks to STATIC's implicit link mechanism (see 3.3), `T`'s abstractions can be selected according to `D`'s properties, if the category they belong to defines rules based on virtual types defined by `D` and retrieved by `T`. Consequently, the paradigm can let `T` collect all or some of `D`'s abstractions, or even model concepts computed from `D`'s properties. Hence, the delegation mechanism allows a type to acquire *interfaces*, using the properties of the delegatee.

SCOOP 2 also allows a class to retrieve the *implementation* corresponding to these interfaces. The guidelines of a SCOOP 2-based library require ARCHITECTS of the library to have each abstraction `A` derive from the special type `automatic::get_impl<A, Exact>` (`Exact` being the exact concrete type of the class), provided by STATIC. By specializing another STATIC template, `automatic::set_impl`, ARCHITECTS can provide implementations for each method of the abstraction `A`, e.g. passing on the call to the delegatee. This way, SCOOP 2 allows library DESIGNERS to fully write real type transformations, not relying on direct inheritance, but on the properties of the transformed type. Instances of such implementation classes performing transformations are called *morphers* in the paradigm, while non-morpher implementation classes are called *basic types*. Morphers offer a new vision of the generic programming paradigm called *semantics-driven genericity*. These generic, composable and lightweight objects built on one or several images, are useful in a wide range of situations. For example, they can be used to implement:

mixins A morpher can add extra data (e.g. a neighborhood) or operations (e.g., an ordering on the values) to an image;

adapters E.g., a slice morpher can be used to view a slice of a 3-D image (spacemap) as a 2-D image (bitmap);

modifiers a morpher can add a mask to an image, to restrict its (iterable) domain;

lazy function applications A morpher can present an image seen through a function, either bijective or not;

More generally, morphers can be used to build three kinds of services.

extrusive equipment Changing the behavior of a generic algorithm non intrusively is a feature often wanted. Such a variation can be a logging facility tracing the execution of an algorithm, or an attached Graphical User Interface (GUI) display depicting the evolution of an image processing. An intrusive approach implies adding extra code in existing algorithms to handle the variations or maintain several versions of these algorithms, which is error-prone and tedious. Morphers provide an elegant, non intrusive solution: instead of altering the algorithms, one can decorate input data with the needed equipment, using a morpher. When the algorithms manipulate this new type, they trigger the expected additional actions (logging, updating a GUI, etc.).

lightweight replacements These are data types constructed over existing ones, saving the creation and copy of temporary variables, resulting in a gain of time and space. For instance, applying a generic algorithm only to the red channel of a color red-green-blue (RGB) image usually requires the creation and manipulation of an object duplicating the data of the red channel, and storing back the result of the algorithm in the original RGB image. Instead, one can use a morpher to create a lightweight (read-write) *view* on the RGB image presenting only the red channel information. The algorithm can then be applied seamlessly, as this new type presents an interfaces similar to the morphed image. This "adaptation" introduces very little, or even no run time cost, thanks to compiler optimizations such as inlining; and no duplication of initial data. Such types are named *replacements* as they do not appear as new data types to practitioners.

actual data (new types) As opposed to the previous item, morphers can be used to implement new types, built on existing types. An image region is such a type: it behaves as an images, and yet it requires an image object to be defined. Combined with the topological definition of a region, one can write an image region morpher from an existing image.

Figure 8 shows an example of a morpher. Beforehand we must equip the abstraction `Image` so as to retrieve implementations if needed (which is the case when the implementation is a morpher class).

```
template <typename Exact>
struct Image : public virtual Any<Exact>,
               public automatic::get_impl<Image, Exact>
```

Then we shall give the implementation in question, which is just a set of delegations to the morphed class in the case of identity-based morphers (i.e., whose default behavior is to copy the interfaces and implementations).

```
// Morpher kind tag.
namespace behavior { struct identity; }
// Automatically-retrieved implementation.
namespace automatic {
  template <typename Exact>
  struct set_impl<Image, behavior::identity, Exact>
    : public virtual Any<Exact>
```

```
  {
    stc_typename(psite);
    stc_typename(value);
    // Delegate the call.
    value impl_read(const psite& p)
    {
      return exact(this)->image()(p);
    }
    // ...
  };
}
```

Last, we can write the morpher, and use it like any other basic type.

```
image2d<bool> ima1;
casted_image< image2d<bool>, int > ima2 (ima1);
// ima2 has the interface of Image2D (among others).
bool b = ima2 (point2d (42, 51));
unsigned size = bbox_size (ima2);
```

Stopping the delegation When a virtual type, either defined in `T` or inherited from its super class, is set to **Not_Delegated**, the delegation branch is ignored for the resolution of this virtual type. This feature is useful from the software engineering point of view, when subclassing a base class for a kind of morpher, to force PROVIDERS of algorithms to define the virtual types that should be retrieved automatically from the delegator semantically.

For example, consider a basic library type class `image2d<Value>`, used to store a 2-dimensional image (with actual data in memory); and an abstract morpher template class `value_morpher<Image>`, having a delegatee of type `Image`. Types from the category `image` must define a virtual type `value`, which is perfectly defined for `image2d<Value>`, and is `Value`. However, this virtual type is set to **Not_Delegated** for `value_morpher<Image>`. Indeed, as this class serves to factor the development of morphers altering the values of an image (also called *value-wise transformations*), its subclasses shall define a valid virtual type `value`, and not use the value type of the morphed type (i.e., the delegatee) automatically. Now, let us imagine that a PROVIDER wants to implement a concrete `casted_image<Image, TargetValue>` morpher, whose purpose is to present an image of type `Image` as a (read-only) image having values of type `TargetValue`. Thus, `casted_image<image2d<int>, float>` would be a lazy transformation of an image of `int`s to an image of `float`s (no data is allocated, nor modified: an instance of this class is just a function of its delegatee `image2d<int>`). If this PROVIDER forgets to give the virtual type `value` of `casted_image<Image, NewValue>` a valid definition (presumably, `NewValue`), STATIC's equipment will trigger an error at compile-time.

3.4 Algorithms

Following the classical generic programming (GP) paradigm, SCOOP 2 expresses algorithms as function templates, not as methods. However, methods are used

```
// value_morpher.
// --------------
// Base class for morphers altering values.
template <typename Exact> class value_morpher;

template <typename Exact>
struct super_trait_< value_morpher<Exact> > { typedef image_base<Exact> ret; };

template <typename Exact>
struct vtypes< value_morpher<Exact> > {
  // This behavior makes subclasses of value_morpher
  // retrieve methods from their delegatee by default.
  typedef stc::final< behavior::identity > behavior;
  // This virtual type must be defined in subclasses.
  typedef stc::not_delegated               value;
};

template <typename Exact>
class value_morpher : public image_base<Exact> {
  typedef image_base<Exact> super;
public:
  stc_typename(delegatee);
protected:
  value_morpher() {};
};


// casted_image.
// -------------
template<typename I, typename U> class casted_image;

// casted_image gets properties from value_morpher...
template<typename I, typename U>
struct super_trait_< casted_image<I, U> >
{ typedef value_morpher< casted_image<I, U> > ret; };

template<typename I, typename U>
struct vtypes< casted_image<I, U> > {
  //...and from its delegatee.
  typedef I                 delegatee;
  // Redefine virtual types.
  typedef typename I::point point;
  typedef U                 value;
};

template<typename I, typename U>
class casted_image
  : public value_morpher< casted_image<I, U> > {
public:
  typedef casted_image<I, U> self;
  typedef value_morpher<self> super;
  stc_using(point);
  stc_using(value);
  stc_using(delegatee);
  casted_image(Image<I>& image)
    : image_ (*exact(&image)) {}
  value impl_read(const point& p) const {
    // Casted to the new value type (U).
    return image_(p);
  }
  protected:
    I& image_;
};
```

Fig. 8. An example of morpher.

for non-generic routines attached to objects and to access to internal data of objects. Generic algorithms benefit from the same traits of SCOOP 1: abstraction-based constraints on inputs, covariant arguments, polymorphic associated types (`typedefs`), statically-dispatched multimethods. We do not discuss much of the subject because this paper focuses on the data structure aspect, and most of its contents is covered in [7]. Figure 9 gives a small example of abstraction-based overloading.

```
// Compute the number of points of an image.
template <typename I>
unsigned npoints (const Image<I>& input) {
  // Slow version iterating over all the points.
  // ...
}
// Specialized version for Image2D.
template <typename I>
unsigned npoints (const Image2D<I>& input) {
  // Fast version computing the result directly.
  return input.nrows () * input.ncols ();
}
```

Fig. 9. An example of abstraction-based overloading.

4 Introducing Static

This section presents STATIC, a component to build libraries using the SCOOP 2 paradigm. For space reasons, we try to provide as much insight as possible, but the article cannot cover all the details of the implementation. For instance, some metaprogramming algorithms are given in a synthetic form, in place of the longer and complex original C++ template code[1].

4.1 Equipment

To use static within a library, one must *equip* a `namespace` with some types and functions. This prevent the mechanism from polluting the global namespace. In this namespace, the library DESIGNER can declare new virtual types to be used as properties of SCOOP 2 classes.

```
// Macros.
#include <stc/scoop.hh>
namespace my {
  // Equip with types and functions.
  #include <stc/scoop.hxx>
  // Declare the virtual types used in this context.
```

[1] Interested readers may want to consult the original code at http://trac.lrde.org/olena/wiki/Static.

```
  mlc_decl_typedef(grid);
  // ...
}
```

`mlc_decl_typedef` uses a technique similar to the typedef introspection technique from [28].

4.2 Static hierarchies

STATIC provides a class `Any` to make the construction and use of static hierarchies easier. The top-most classes of the hierarchies of the library shall inherit from `Any`, so that the exact type is available through the `exact` function.

```
template <typename Exact> struct Any {};
template <typename Exact> Exact* exact(Any<Exact>* ref)
{ return (Exact*)(void*)ref; }
```

This is admittedly the most intrusive point in using STATIC within a library, as the C++ language does not allow to add super classes a posteriori. Thus, the equipment of existing libraries might be tedious and require wrapping the existing classes.

4.3 Virtual types

Virtual types for the type `T` are defined as a specialization of the class `vtypes<T>`, defined in the `stc/scoop.hxx` equipment.

```
struct point2d;
template<> struct vtypes<point2d> {
  typedef mlc::uint_<2> dim;
  typedef int           coord;
  typedef grid2d        grid;
};
```

Semantically, a class inherits from the virtual types of its super class, but this inheritance is implicit: the user needs not mention it, since this task is taken care of by STATIC's virtual type look-up algorithm (see Algorithm 2). To make the recursive retrieval of virtual types possible, STATIC cannot simply rely on the C++ inheritance relationship. The language doesn't provide any means to actually retrieve the type of a super class. Hence, STATIC needs the author to inform a special traits to keep this information, `super_traits_`. This relationship shall not mirror the C++ inheritance exactly: it shall be limited to *implementation classes*, as they are the only classes *defining* properties. A class having no implementation super class must inherit from the special type **None** (`mlc::none`).

```
template<>
struct super_trait_<point2d> { typedef mlc::none ret; };
```

Virtual types can take any "value" (i.e. C++ type, defined by the language or the user). However, STATIC uses some special values (the C++ names are given in parenthesis).

Abstract (`stc::abstract`) Used to declare an abstract virtual type.

Final(*val*) (`stc::final<val>`) **Final** is a qualifier that does not change the value of the virtual type defined, but prevents any subclass or delegator to redefine the virtual type.

Not_Delegated (`stc::not_delegated`) Prevents the lookup algorithm from using the delegation branch to retrieve a virtual type; only the inheritance branch will be used.

Virtual types are domain-related; their meanings have no impact on STATIC, except **delegatee** (`delegatee`). This special virtual type is used to attach a *delegation branch* to a given class and its subclasses (see 3.3). The retrieving of the virtual type `T` from class `C` uses the lookup algorithm FIND. If no error occurred during the lookup (e.g., due to a erroneously-designed hierarchy) FIND(`C`, `T`) returns the "value" of the virtual type if found, **Not_Found** (`mlc::not_found`) otherwise. Algorithm 2 shows FIND as a pseudo-algorithm. The translation to the corresponding C++ metaprogram is immediate.

FIND makes use of several routines and values. The ones that are not defined in Algorithm 2 are described hereinafter.

Super(*type*) (`super_traits_<type>::ret`) This function returns the super class of *type*.

None (`mlc::none`) The value returned by a class having no super class (in the STATIC acceptation).

Find(*source*, *target*) (`find<source, target>::ret`) Recursively look for the value of the virtual type *target* for class *source*. The virtual type is searched in the inheritance branch as well as in the delegation branch (if it exists). FIND expects a virtual type to have a concrete definition. If the virtual type lookup ends up with **Abstract**, FIND triggers a compile-time error.

Find-Local(*source*, *target*)
(`find_local<source, target>::ret`) Query the class *source* for the value of the virtual type *target* directly, using `vtypes`. This algorithm is used by FIND.

Not_Found(`mlc::not_found` The value returned by FIND when a virtual type is not found.

In addition to FIND, STATIC proposes a CHECK algorithm performing additional checks not directly required by the lookup task (see Algorithm 3).

Macros As STATIC is almost only composed of template types, we wrote several macros to serve as syntactic sugar. This section explains the meaning of each macro used in the paper.

`stc_typename` and `stc_using` are just shortcuts to equip classes: they respectively inject a virtual type in the scope of the current class and retrieve a virtual type from a super class.

```
FIND(source, target)
 1  if source = delegatee
 2    then return SUPERIOR-FIND(source, target)
 3  (where, res) ← FIRST-STM(source, target)
 4  switch
 5    case res = Not_Found :
 6          return DELEGATOR-FIND(source, target)
 7    case res = Abstract :
 8          res_delegator ← DELEGATOR-FIND(source, target)
 9          if res_delegator = Not_Found
10            then error "target is abstract."
11    case res = Not_Delegated :
12          return SUPERIOR-FIND(source, target)
13    case default : return res

DELEGATOR-FIND(source, target)
 1  deleg ← SUPERIOR-FIND(source, delegatee)
 2  if SUPERIOR-FIND(source, delegatee) = Not_Found
 3    then return Not_Found
 4    else  return FIND(deleg, target)

SUPERIOR-FIND(source, target, current = source)
 1  if current = None
 2    then return Not_Found
 3  stm ← FIND-LOCAL(current, target)
 4  switch
 5    case stm = Abstract :
 6          error "target is abstract"
 7    case stm = Not_Found or stm = Not_Delegated :
 8          sup ← SUPER(current)
 9          return SUPERIOR-FIND(source, target, sup)
10    case stm = Final(val) :
11          return val
12    case default : return stm

FIRST-STM(source, target)
 1  if source = None
 2    then return (None, Not_Found)
 3  stm ← FIND-LOCAL(source, target)
 4  switch
 5    case stm = Not_Found :
 6          return FIRST-STM(SUPER(source), target)
 7    case stm = Final(val) :
 8          return (source, val)
 9    case default : return (source, stm)

FIND-LOCAL(source, target) locally queries the source class for the virtual type
target. If the set of virtual types attached to source (i.e., vtypes<source>) has a
definition for target, this "value" is returned, otherwise Not_Found is returned.
```

Algorithm 2: Virtual type lookup.

```
CHECK(source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← FIND-LOCAL(source, target)
 4  sup ← SUPER(source)
 5  switch
 6    case stm = Abstract :
 7          orig ← (source, Abstract)
 8          CHECK-NO-STM-DEFINED(orig, sup, target)
 9    case stm = Final(val) :
10          CHECK-FINAL-STM(val)
11          orig ← (source, val)
12          CHECK-NO-FINAL-DEFINED(orig, sup, target)
13    case stm = Not_Delegated :
14          orig ← (source, Not_Delegated)
15          CHECK-NO-FINAL-DEFINED(orig, sup, target)
16    case stm = Not_Found :
17            — Nothing.
18    case default :
19          orig ← (source, stm)
20          CHECK-NO-FINAL-DEFINED(orig, sup, target)
21
22  return CHECK(sup, target)

CHECK-NO-STM-DEFINED(orig, source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← LOCAL-FIND(source, target)
 4  if stm ← Not_Found
 5    then error "target re-declared abstract in orig."
 6    else  sup ← SUPER(source)
 7          CHECK-NO-STM-DEFINED(orig, sup, target)

CHECK-FINAL-STM(source, target, stm)
 1  if stm = Abstract
 2    or stm = Final(val)
 3    or stm = Not_Delegated
 4    or stm = Not_Found
 5    then error "Ill-formed final vtype."

CHECK-NO-FINAL-DEFINED(orig, source, target)
 1  if source = None
 2    then return      — Stop condition.
 3  stm ← LOCAL-FIND(source, target)
 4  if stm = Final(val)
 5    then error "Final vtype target redefined in orig."
 6    else  CHECK-NO-FINAL-DEFINED(orig, SUPER(source), target)
```

Algorithm 3: Additional type-checking rules.

```
# define stc_typename(T) typedef stc_type(Exact, T) T
# define stc_using(T)    typedef typename super::T  T
```

`stc_type (source, target)` and `stc_deferred (source, target)` are macros performing the same task: they call the template metaprogramming algorithm FIND and expand as the result of the lookup. However, the former also calls the CHECK algorithm (Algorithm 3) to ensure the rules given above are followed. Because of the recursive nature of the approach, we cannot always perform all checks on virtual types (C++ compilers consider types coming across their own definition through successive type instantiations as *empty*, which would break our algorithms). Hence we use `stc_deferred` in problematic cases (virtual type definitions, for instance).

4.4 Extensible inheritance

SCOOP 2 features an extensible inheritance mechanism used to link the top classes of implementation hierarchies to the right abstractions. The entry point of this system is the class `top<Exact>`, which recursively inherits from the results of the static `switch` statements (selectors) tagged `internal::selector<Exact, n>`, where n covers the first values of $\mathbb{N}^*$. The technique used is similar to generating scattered hierarchies (`GenScatterHierachy`) exposed in [23].

5 Related Work

Many generic programming techniques have been invented to develop C++ libraries. First of all, traits [29] have been widely used to attach properties and associated types to generic types, notably within the Standard Template Library [2].

The idea of adding and checking constraints on the parameters of C++ templates with respect to a given contract (structural conformance, name conformance) is not new, and has led the way to the current work on future C++0x concepts [13]. McNamara and Smaragdakis have proposed a solution based on static interfaces [30]. Siek and Lumsdaine have formalized the principle of concept checking [16] within the Boost Concept Check Library (BCCL). Building on similar metaprogramming techniques, a new form of concept-based polymorphism can be obtained [17]. One of the advantages of these solutions is that they require no language extension, and can therefore be applied under the form of portable libraries with C++ compilers conforming to the 2003 ISO/IEC standard. Many of them will be superseded by the use of reified concept features in the forthcoming C++ standard.

As a matter of fact, several of these techniques are used within the OLENA project: traits serve to define properties (although querying their value requires a non-trivial metaprogramming algorithm); Static interfaces and concept-checking are part of SCOOP 2 thanks to the Generalized Curiously Recurring Template Pattern.

Static introspection is a technique implemented in METALIC and used by STATIC to retrieve properties. Zólyomi and Porkoláb have described a framework providing these services (among others) as part of a C++ library [28].

Last, the extensible inheritance in STATIC is based on a technique similar to the typelists described by Alexandrescu in [23] and which has proved to be useful to design recursively-defined class hierarchies [31].

Several existing libraries have proposed solutions to express type transformations [2,32,6]. However, all these approaches create types whose interfaces are limited to a known abstraction, and do not handle the specificity of the original type if it conformed to a subabstraction.

6 Conclusion

We have presented an extended generic programming paradigm (SCOOP 2), allowing library designers to express generic type transformations called morphers. The paradigm relies on the presence of semantically-enhanced data types thanks to properties expressed as virtual types in implementation classes. These properties are used to connect concrete classes to the abstractions they model. As abstractions are expressed as actual C++ classes, they can provide implementations as well, based on the nature of the object, thanks to a delegation mechanism which is part of the paradigm.

SCOOP 2 is intended for designers of new libraries; it might not be easy to adapt to existing libraries, though the task can be achieved by wrapping existing classes. Indeed, we have successfully applied this approach on a small subset of the standard C++ library.

We provide a software component to help users design their libraries using SCOOP 2 and reify idioms of the paradigm, STATIC. This component has been conceived in the context of the OLENA project. We are working on providing syntactic sugar for the constructs of the paradigm using a language masking the verbosity of the C++ templates; however, we don't want to make this extra language mandatory, and we will keep STATIC as a pure C++ project, since library-based language extensions are generally less costly and more sustainable than domain-specific languages [33].

We ran our tests using the GNU C++ Compiler (GCC) version 4.1 on Debian GNU/Linux, and we are working on porting STATIC to other configurations, notably using the Intel C++ Compiler (ICC). The paradigm itself is expressed solely using C++03 constructs. We have conducted some experiments with the ConceptGCC compiler [34] to check whether SCOOP 2 can benefit from C++0x new features, but the compiler did not properly support mixing inheritance and where-clauses at that time.

Acknowledgments

We thank Alexandre Duret-Lutz, Olivier Gaça, Maxime van Noppen, Benoît Sigoure and Didier Verna for their proofreading and comments.

References

1. ISO/IEC: ISO/IEC 14882:2003 (e). Programming languages — C++ (2003)
2. Stepanov, A., Lee, M., Musser, D.: The C++ Standard Template Library. Prentice-Hall (2000)
3. Project, T.B.: Boost C++ libraries. http://www.boost.org/ (2008)
4. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: International Symposium on Computing in Object-Oriented Parallel Environments. Number 1505 in Lecture Notes in Computer Science (1998) 59–70
5. The CGAL Project: CGAL, Computational Geometry Algorithms Library (2008) http://www.cgal.org.
6. Abrahams, D., Siek, J.G.: Policy adaptors and the Boost Iterator Adaptor Library. In: Second Workshop on C++ Template Programming. (October 2001)
7. Burrus, N., Duret-Lutz, A., Géraud, Th., Lesage, D., Poss, R.: A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In: Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL), Anaheim, CA, USA (October 2003)
8. Duret-Lutz, A.: Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In: Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—Young Researchers Workshop; published in "Net.ObjectDays2000", Erfurt, Germany (October 2000) 653–659
9. Géraud, Th.: Advanced static object-oriented programming features: A sequel to SCOOP. http://www.lrde.epita.fr/people/theo/pub/olena/olena-06-jan.pdf (January 2006)
10. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing Vaucanson. Theoretical Computer Science **328** (November 2004) 77–96
11. Claveirole, Th., Lombardy, S., O'Connor, S., Pouchet, L.N., Sakarovitch, J.: Inside Vaucanson. In Springer-Verlag, ed.: Proceedings of Implementation and Application of Automata, 10th International Conference (CIAA). Volume 3845 of Lecture Notes in Computer Science Series., Sophia Antipolis, France (June 2005) 117–128
12. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. 1st edn. C++ In-Depth Series. Addison Wesley Professional (December 2001)
13. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic support for generic programming in C++. In: Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press (October 2006) 291–310
14. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. In: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, And Applications (OOPSLA), New York, NY, USA, ACM Press (2003) 115–134
15. Järvi, J., Gregor, D., Willcock, J., Lumsdaine, A., Siek, J.: Algorithm specialization in generic programming: challenges of constrained generics in C++. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Ottawa, Ontario, Canada, ACM Press (June 2006) 272–282

16. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)
17. Järvi, J., Willcock, J., Lumsdaine, A.: Concept-controlled polymorphism. In Pfennig, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering (GPCE). Volume 2830 of LNCS., Erfurt, Germany, Springer-Verlag (September 2003) 228–244
18. Coplien, J. In: A Curiously Recurring Template Pattern. In [35].
19. Chiba, S.: A metaobject protocol for C++. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). SIGPLAN Notices 30(10), Austin, Texas, USA (October 1995) 285–299
20. Veldhuizen, T.L.: Techniques for scientific C++. Technical Report 542, Indiana University Department of Computer Science (August 1999)
21. EPITA Research and Developpement Laboratory (LRDE): The Olena image processing library. http://olena.lrde.epita.fr (2003)
22. EPITA Research and Developpement Laboratory (LRDE): A prototype using SCOOP 2 and the C++ standard library. https://trac.lrde.org/olena/wiki/SCOOP/MiniStd (2007)
23. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
24. Gurtovoy, A., Abrahams, D.: The Boost MPL library. http://www.boost.org/libs/mpl/doc/index.html (2004)
25. David Abrahams, A.G.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. 1st edn. C++ In-Depth Series. Addison Wesley Professional (December 2004)
26. Adobe: The Adobe Generic Image Library (GIL). http://opensource.adobe.com/gil/ (2007)
27. Siek, J.G., Lumsdaine, A.: Modular generics. In: Concepts: a Linguistic Foundation of Generic Programming, Adobe Systems (April 2004)
28. Zólyomi, I., Porkoláb, Z.: Towards a general template introspection library. In Karsai, G., Visser, E., eds.: Generative Programming and Component Engineering (GPCE). Volume 3286 of LNCS., Vancouver, Canada, Springer-Verlag (October 2004) 266–282
29. Myers, N.C.: Traits: a new and useful template technique. C++ Report **7**(5) (June 1995) 32–35
30. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (October 10 2000)
31. Zólyomi, I., Porkoláb, Z., Kozsik, T.: An extension to the subtype relationship in C++ implemented with template metaprogramming. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering (GPCE). Volume 2830 of LNCS., Erfurt, Germany, Springer-Verlag (September 2003) 209–227
32. Weiser, M., Powell, G.: The View Template Library. In: First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)
33. Stroustrup, B.: A rationale for semantically enhanced library languages. In: Proceedings of the Workshop on Library-Centric Software Design (LCSD), San Diego, California, USA (October 2005)
34. Indiana University: ConceptGCC. http://www.generic-programming.org/software/ConceptGCC/ (2007)
35. Lippman, S.B., ed.: C++ Gems. Cambridge Press University & Sigs Books (1998)

C

ARTICLES “INFORMATIQUE ET IMAGE”

Cette annexe présente quatre articles dont le sujet est similaire à celui du corps de ce manuscrit ; ils complètent donc au mieux ce dernier. Ils ont été choisi pour des raisons distinctes expliquées ci-dessous.

- L’annexe C.1 (page 190) est un article court, publié en 2000, qui présente nos premiers résultats en termes de généricité. Cet article devrait permettre au lecteur de constater le chemin parcouru depuis.
- En annexe C.2 (page 195), un article publié en 2002, après un rappel de la généricité en C++, met l’accent sur les aspects naturel et lisible de l’écriture des algorithmes.
- L’article de 2009 donné en annexe C.3 (page 206) explique le besoin de généricité afin de pouvoir traiter aussi bien des images à structure topologique régulière que des graphes, des maillages et des complexes cellulaires.
- Enfin, un point peu abordé dans ce manuscrit est détaillé par l’article récent, 2011, de l’annexe C.4 (page 219) ; il s’agit de montrer comment nous préservons à l’exécution les performances des algorithmes “malgré” la généricité de ces derniers.

C.1 OBTAINING GENERICITY FOR IMAGE PROCESSING AND PATTERN RECOGNITION ALGORITHMS

Obtaining Genericity for Image Processing and Pattern Recognition Algorithms. *International Conference on Pattern Recognition* (**ICPR**), September 2000. IEEE Computer Society, vol. 4, pp. 816-819, Barcelona, Spain. With Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. 

Obtaining Genericity for Image Processing and Pattern Recognition Algorithms

Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
thierry.geraud@lrde.epita.fr

Dimitri Papadopoulos-Orfanos, Jean-François Mangin
Service Hospitalier Frédéric Joliot, CEA
4 place du Général Leclerc, F-91401 Orsay cedex, France
papadopo@shfj.cea.fr

Abstract

Algorithm libraries dedicated to image processing and pattern recognition are not reusable; to run an algorithm on particular data, one usually has either to rewrite the algorithm or to manually "copy, paste, and modify". This is due to the lack of genericity of the programming paradigm used to implement the libraries. In this paper, we present a recent paradigm that allows algorithms to be written once and for all and to accept input of various types. Moreover, this total reusability can be obtained with a very comprehensive writing and without significant cost at execution, compared to a dedicated algorithm. This new paradigm is called "generic programming" and is fully supported by the C++ language. We show how this paradigm can be applied to image processing and pattern recognition routines. The perspective of our work is the creation of a generic library.

1 Introduction

Great effort has gone into building image processing and pattern recognition libraries that can be used and augmented by different research centers. However, the main difficulty that is systematically encountered and that remains unsolved is how to manage the large number of input types used in this domain. An algorithm developed for a particular input can rarely be reused. As a consequence, no one library has succeeded in being unanimously adopted by the scientific community.

An ideal library should be generic, i.e. supply generic algorithms. A generic image processing algorithm is written once, and indistinctly accepts 2D and 3D images (isotropic or not), regions, region adjacency graphs, image and graph pyramids, sequences, collections and so forth; the types of data contained in these structures is scalar (Boolean, integer or float), complex, composed (e.g. RGB). Existing libraries are usually dedicated to a particular data structure (mostly 2D images) and their algorithms are restricted to few data types (mostly unsigned 8 bit integers). Ideal pattern recognition algorithms also have this problem: for instance, a given primitive such as a contour can have different representations.

Most algorithm data can have different forms (i.e., different types) and should be used as the input of algorithms in a transparent way. In this paper, we show that recent advances in C++ programming allow this genericity with a very comprehensive syntax and without leading to a significant extra cost of execution time as compared to dedicated algorithms. This new paradigm is called "generic programming". We have successfully applied it both to low level image processing routines and to high level pattern recognition algorithms in a library that we are currently developing.

In section 2, we present the generic programming paradigm and its benefits compared to usual paradigms. Then, in section 3, we explain how to design a generic algorithm and we show with a simple example algorithm how this paradigm can be applied to the field of image processing, and how we can obtain maximal genericity. Lastly, in section 4, we conclude and give future perspectives of our work.

2 The generic programming paradigm

To emphasize the benefits of generic programming applied to image processing and computer vision, we will first point out some drawbacks of existing libraries.

2.1 Current libraries

In current *C* libraries, an algorithm has to be written as many times as there are input types [1]; see figure 1. For instance, a simple addition of a constant to image elements leads to four routines if we want this algorithm to deal with 2D and 3D images with Boolean or floating elements. Since the combination "algorithms × structure types × data types" can be enormous, many libraries limit the number of structure types and data types handled by each algorithm. The *C*-like programming paradigm has two main drawbacks: the capabilities of such libraries are limited, and introducing a new structure type or data type is a tedious task.

Some object languages such as C^{++} offer genericity, which means that classes and procedures can be parametrized. A common use of genericity is to parameterize the definitions of data structures and routines by the data type of their elements. As a consequence, reusability is enhanced but, although some libraries use genericity in this way [6, 5], the reusability is far from being total: routines still have to be written for each structure type. In fact, existing libraries do not rely on the generic programming paradigm presented below.

The limits of reusability of image processing algorithms induced by different programming paradigms are explained with further details in [3].

2.2 The novelty

Generic programming is a new paradigm to write fully generic algorithms without leading to significant overheads at run-time as compared to dedicated code. This paradigm is very attractive for scientific numerical programming and is used in some recent libraries: CGAL [2] and Blitz++ [7], respectively dedicated to geometric and algebraic calculi.

The generic programming paradigm is based on two key ideas:

- an algorithm is parametrized by its input types (in contrast to data parameterization, see section 2.1),
- the tools, helper objects needed by the algorithm, are deduced from its input (like the iterators presented in section 3.2).

When an algorithm is used, the compiler generates the appropriate machine code for the particular input types; see figure 2. Moreover, each method call in this code can be replaced by its implementation, so the cost of method calls is avoided. Therefore, the executable code is similar to that of a routine written for the particular input types and generic procedures are roughly as fast as dedicated procedures. Generic programming makes the compiler do the work that the programmer has to do in usual programming.

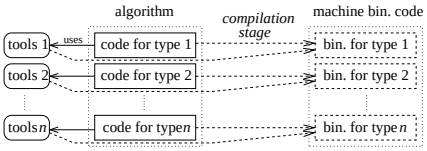


Figure 1. Usual mechanism in *C*.

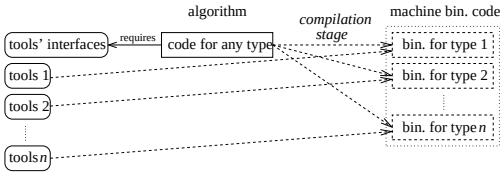


Figure 2. Generic mechanism in C^{++}.

C^{++} is a language that offers a good compromise between efficiency and generic capabilities [4], so all existing generic libraries are implemented in C^{++}. In the next section, we present the syntactic bases which are of prime importance to understand the generic programming paradigm.

2.3 Syntax

The sample code below gives a part of the definition of a generic 2D image class in C^{++}:

```
template< typename T >  // parameterization
class Image2D {
  public:
    typedef T value_type;  // a type alias
    size_t size() const;   // a method
    //...
};
```

This code contains two main syntactical elements.

Firstly, the class definition is parametrized (keyword `template`) and a single parameter is defined, `T`, whose nature is a type (keyword `typename`). This parameter represents the type of the image elements. For instance, the programmer can then use the type `Image2D<float>` to manipulate 2D images containing floating values. The parameterization thus allows the definition of only one class per structure type in the library whatever the data type.

Lastly, one can define a type alias (keyword `typedef`) within a class; for instance, the parametric class `Image2D` contains the alias `value_type`, which gives the element type, `T`. Such an alias is used as follows:

```
typename class_name::alias_name
```

.

Let us consider a procedure that is parametrized by the type `I` of an input aggregate; this aggregate is the argument

`input` of the procedure. In the procedure body, the programmer can use type aliases such as `I::value_type`, and methods of `I` such as in `input.size()`; see for instance `proc` in the code below:

```
template< typename I >  // parametrization
void proc( I& input ) {
  size_t  a_size = input.size();
  typename I::value_type  a_value;
  //...
}

int main() {
  Image2D<float>  ima;
  proc( ima );
}
```

When `proc` is called, the type `I` is known, and the dedicated procedure is compiled (if it has not already been compiled). The compiler then checks that the type effectively contains a method `size()` and an alias `value_type` (these two points represent a required interface). Since it is the case for the type `Image2D<float>`, in `main()`, the variable `value` is of type `float` in the compiled routine.

3 Generic algorithm design

The generic programming paradigm being quite recent, algorithm design does not follow a standard process. So, we first present the method that we have established.

3.1 Design method

The design method is sequential and composed of several steps.

1. Express the algorithm in mathematical language while paying attention to remaining as general as possible. This step ensures that we will not provide an algorithm that depends upon particular considerations (for instance, an algorithm linked to a given data structure).
2. Identify the objects that are involved in the expression obtained at the previous step and describe their role and their required behavior.
3. Point out each algorithm option that should be set by the user, and give each option a default value if it is pertinent. If needed, return to step 2 to take into account these options.
4. Analyze the dependencies between the types of the objects used in the algorithm, and deduce its parameters.
5. Finally, write the generic algorithm.

3.2 A simple example

To illustrate how to obtain the highest genericity for an image processing or pattern recognition algorithm, let us study a very simple example: the addition of a constant to the elements of an aggregate.

Step 1 For the sake of genericity, let us first generalize this example and consider that the algorithm has to be designed for any operator similar to `val += cst`. We can then formulate this algorithm as follows : “for each element of an input aggregate, apply the operator to the element value”. Please note that this description conceals all implementation details related to the particular types of the objects that could be involved in the algorithm.

Step 2 The objects involved in the algorithm are: an input aggregate (argument `input`), an iterator (internal variable `iter`), a constant value (argument `cst`).

To translate this description into a generic algorithm, the introduction of an object which iterates over the elements of an aggregate is required; such an object, called an *iterator*, is a common tool in software design. One iterator class is defined per structure type and all iterators provide the same interface (i.e., the same subset of methods and aliases) in order to be uniformly manipulated. In this way, iterating over graph vertices leads to the same syntax as iterating over image points.

Step 3 The options of the algorithm are:

- an operator (for instance, a saturated addition),
- a predicate to restrict the addition to certain elements of the aggregate (by default, the predicate returns always true),
- an accessor to specify, if the element type is structured, which field is concerned by the addition (by default, the accessor is the identity; its name is `get_value<>`),

Each option is translated into one extra object in the algorithm.

Step 4 Let us denote by `I` the type of the input aggregate, by `C` the type of the constant value, by `O` the type of the operator, by `P` the type of the predicate, by `get_A` the type of the accessor (when accessing field `A` of `I`), and by `T` the type of the iterator. Then, we have the following statements:

- `C` is given by `get_A::output_type`,
- `T` is given by `I::iterator_type`,
- `get_A<>::input_type` is given by `I::value_type` and `get_A<>::output_type` is given by `I::value_type::A_type`,
- `O::args_type` is given by `get_A::output_type`.

The parameters of the algorithm are: `I`, `O`, `P`, and `get_A`. The first parameter is known when the algorithm is called and the last two parameters have default values. The only parameter that the user has to set is `O`.

Step 5 The core of the algorithm is transformed into the following description. Define `iter` on `input`; then, for each iteration handled by `iter`, conditioned by `pred`, apply `oper` with `cst` on the value given by `iter` through `access`. Finally, this algorithm is ready to be translated into the generic C^{++} code[1]:

```
template< typename O,
          template< class U > class get_A = get_value,
          typename P = Pred_true >
struct op
{
   template< typename I > static
   void on( I& input,
            const get_A< I::value_type >::output_type& cst,
            P pred = P() )
   {
      O oper;    get_A< I::value_type > access;
      I::iterator_type  iter( input );
      for ( iter.first(); ! iter.isDone(); iter.next() )
         if ( pred( access( iter() ) ) )
            oper( access( iter() ), cst );
   }
};
```

This algorithm accepts various structure and data types. Moreover, the user can parameterize its behavior (for instance, the user can subtract a constant value from the red component of some elements of a 3D image). Since the user is not required to set all parameters, the simplest call of the algorithm (arithmetical plus) is:

```
op<plus>::on( input, cst );
```

4 Conclusion

In this paper, we have presented the generic programming paradigm that enables the implementation of generic algorithms. Then, we have demonstrated with an example how to apply this paradigm to the field of image processing and pattern recognition and how to obtain the highest genericity.

The main difficulty of building a generic library lies in correct designing of algorithms and tools. For that, the closer the code is to the theory, the better the genericity is. Although the example given in this paper is very simple, the design process is rigorously equivalent for high-level routines.

This paradigm has five major advantages that we set out below.

Reusability Since algorithms are generic, their reusability is maximal. Each algorithm is programmed only once and accepts data of various types. When one wants to introduce a new data type in the library, one only has to conform to few requirements in order to benefit from the existing algorithms.

Functionalities From the viewpoint of the user, such a library is no more complicated than current libraries. Algorithms can be called very simply because they define their own default settings, while it remains possible for the user to be more specific.

Development The development cost of a generic algorithm is dramatically reduced as compared to that of current libraries (see figures 1 and 2). Consequently, maintenance and reliability are significantly improved.

Efficiency Generic algorithms are roughly as fast as dedicated algorithms (the compiler expands generic code and makes it similar to dedicated code).

Federative A generic library is able to federate tools and algorithms developed by different research centers. We believe that this point is of prime importance for the community because such a library enhances the capitalization of knowledge.

We are currently developing such a library whose first version will be soon freely available. We do not aim at providing a wide range of algorithms and tools but the most usual ones to facilitate algorithm programming.

[1] The full implementation of the example is available at the URL `http://www.lrde.epita.fr/download/`

References

[1] M. Dobie and P. Lewis. Data structures for image processing in C. *Pattern Recognition Letters*, 12(8):457–466, 1991.

[2] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, 1998.

[3] T. Géraud, Y. Fabre, D. Papadopoulos-Orfanos, and J.-F. Mangin. Vers une réutilisabilité totale des algorithmes de traitement d'images. In *17th Symposium on Signal and Image Processing (GRETSI'99)*, volume 2, pages 331–334, Vannes, France, September 1999. In French; available in English as a technical report at `http://www.lrde.epita.fr/publications`

[4] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4), 1999.

[5] C. Kohl and J. Mundy. The development of the Image Understanding Environment. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, pages 443–447, 1994.

[6] G. Ritter, J. Wilson, and J. Davidson. Image Algebra: an overview. *Computer Vision, Graphics, and Image Processing*, 49(3):297–331, 1990.

[7] T. Veldhuizen. Arrays in Blitz++. In *Proc. of the 2nd Intl. Conf. in Object-Oriented Parallel Environments (ISCOPE'98)*, number 1505 in Lectures Notes in Computer Science, pages 223–230. Springer Verlag, 1998.

C.2 GENERIC IMPLEMENTATION OF MORPHOLOGICAL IMAGE OPERATORS

Generic Implementation of Morphological Image Operators. *International Symposium on Mathematical Morphology* (**ISMM**), April 2002. pp. 175-184, Sydney, Australia. With Jérome Darbon and Alexandre Duret-Lutz. 

GENERIC IMPLEMENTATION OF MORPHOLOGICAL IMAGE OPERATORS

When Harry's C++ Code Met Sally's Algorithms

J. DARBON*, T. GÉRAUD, A. DURET-LUTZ
EPITA Research and Development Laboratory
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
E-mail: {jerome.darbon, olena}@lrde.epita.fr
Phone +33 1 53 14 59 16 – Fax +33 1 53 14 59 22

Abstract Several libraries dedicated to mathematical morphology exist. But they lack genericity, that is to say, the ability for operators to accept input of different natures —2D binary images, graphs enclosing floating values, etc. We describe solutions which are integrated in OLENA, a library providing morphological operators. We demonstrate with some examples that translating mathematical formulas and algorithms into source code is made easy and safe with OLENA. Moreover, experimental results show that no extra costs at run-time are induced.

Keywords: mathematical morphology, image processing operators, genericity, programming.

1. Introduction

Most people involved in mathematical morphology are mathematicians or image processing practitioners rather than computer scientists. Therefore, they should find it easy to use program libraries in order to avoid dealing with implementation problems and rather focus only on the methodological aspects of their work. Köthe [7] notes that the lack of algorithmic comparison in the literature is due to the difficulty of implementing computer vision algorithms. Furthermore, Mallat [10] insists on the notion of reproducible computational science; that is to say, an author of article should make source code available. Along these lines, Pitas has recently published a book [12] fully illustrated with source code.

Quite a lot of image processing libraries are available on the Internet; however, they are usually restricted to very few image structures and data types,

*Also with: *École Nationale Supérieure des Télécommunications, Networks and Computer Science Department. 46, rue Barrault – F-75634 Paris Cedex 13 – France.*

 ISBN 0 643 06804 X

whereas mathematical morphology applies on a wide range of data: signals, 2D and 3D images, graphs, etc., containing integers with different precisions, floats, binaries, and so on. It is then very difficult for practitioners to find a library with morphological operators that meet their requirements.

When an image processing algorithm —for instance an erosion— is translated into a *single* routine in a given computer language, one says that this routine is *generic* if it accepts different input types. D'Ornellas and van den Boomgaard [4, 3] mention that generic algorithms for morphological image operators could be developed in C++ using the *generic programming* paradigm.

The aim of this paper is twofold: it presents a new library which provides mathematical morphology operators, and it shows how one can easily translate a mathematical formula into a C++ program in the context of our library.

This paper is organized as follows. In section 2, we present different paradigms to program image processing operators; we discuss their advantages and drawbacks with regard to their genericity level and their safety for the user. Then, in section 3, we study the particular cases of several morphological operators. Last, we conclude in section 4 and we give an evaluation of their performance.

2. Programming Paradigms, Types, and Safety

2.1 State of the Art

Most of image libraries are built on a C-style programming paradigm, and two families can be identified.

The first family considers that a general type, usually `float`, is enough to store data[1]. A 2D image structure is then defined as depicted below (left column). A major drawback of this approach is that there are no semantical distinctions between images with respect to their data type: procedures are therefore unable to check constraints about input images. For instance, the procedure `foo` (below, right column) expects that both input images have the same data type but cannot check it. It is then very easy for a programmer to call routines incorrectly.

```
struct image2d
{
  unsigned nrows, ncols;
  float** data;
};
```

```
void foo(image2d* ima1, image2d* ima2)
{
  /* ... */
}
```

Two other important drawbacks are that non-scalar data cannot be handled by this image structure and that images consume memory unnecessarily.

The second family of library programming style, described by Dobie and Lewis [2], addresses these problems by enforcing type control. To this end, the

[1]The general type `float` is the most convenient one for general purpose library; In mathematical morphology, a general type would rather be `short`, even if PDE-based approaches are now in fashion.

2D image structure is modified, see the left column below: a new field, `type`, allows the programmer to insert assertions to check at run-time the proper nature of the input images (right column below).

```
typedef enum { INTU8, FLOAT } data_t;

struct image2d
{
   unsigned nrows, ncols;
   data_t type;
   void** data;
};
```

```
void foo(image2d* ima1, image2d* ima2)
{
   assert(ima1->type == ima2->type);
   switch(ima1->type) {
      case INTU8:
         /* call sub-routine for INTU8 */
         foo_INTU8(ima1, ima2); break;
      /* ... */
   }
}
```

Unfortunately, if safety is enforced for the library user, the library programmer has to write as many sub-routines as there are data types. For instance, the sub-routine `foo_INTU8` contains the code dedicated to unsigned 8 bit integers. Since writing many similar routines per algorithm is long and tedious, libraries usually handle only very few data types.

We have shown that genericity with respect to data types can be handled either by the use of a general type (`float`) or in a tedious fashion by code replication (the different cases of `switch`). However, image structures are still not generic, since we can only handle 2D images. Some C libraries use macros (keyword `#define`) to emulate C++ templates. However, macros cannot handle all features that we enjoy with templates (e.g. stronger typing, recursivity, meta-programming).

2.2 C++ and Genericity

An interesting feature of the C++ language [13] is *genericity* using the `template` keyword. In the left column sample code below, `image2d` is a meta-structure parameterized by an unknown data type `T` and the procedure `foo` is similarly parameterized. Its input must share the same data type as made explicit by the procedure signature: this constraint is now checked *at compile-time*.

```
template<class T>
struct image2d
{
   unsigned nrows, ncols;
   T** data;
};

template<class T>
void foo(image2d<T>& ima1,
         image2d<T>& ima2)
{
   //...
}
```

```
int main()
{
   image2d<float> ima1, ima2;
   //...
   foo(ima1, ima2); // first call

   image2d<int> ima3, ima4;
   //...
   foo(ima3, ima4); // second call
}
```

In the right column above, the client instantiates different kinds of image (`ima1` contains floats whereas `ima3` contains integers) and calls `foo` twice. At each call, the compiler automatically deduces the type `T` from the input types and creates a specialized version of the meta-procedure `foo`. In our example, `T` is set to `float` at the first call and a version of `foo` dedicated to process

2D floats images is generated. That means that the compiler creates specific sub-routines, and therefore saves the programmer from performing this tedious task (see section 2.1).

2.3 Full Genericity and STL Style

Now we turn to *full genericity* as we want a procedure to accept different image types. The key idea is given by the style of the Standard Template Library (STL for short) now part of the C++ Standard Library [13]: procedures should be parameterized by their input types. This paradigm is called *generic programming* by the object-oriented scientific computing community [11].

For instance in order to browse the contents of images with different structures, obviously one cannot keep two loops when input is 2D or three when it is 3D. These image structure implementation details must be hidden. The solution that early appears in imaging software [9] is the use of *iterator* objects. Consider the code below. A new procedure, `bar`, sets every pixel to 0. It is now parameterized by `InputIter` which represents an iterator type. The object `p` iterates from the first point of the targeted image to the last, these iteration boundaries being given by the methods `begin()` and `end()` of the image class. When `bar` is called, the particular procedure instantiated by the compiler uses an iterator `i` of type `image2d_iterator<float>`; the single loop is thus able browse image with different structures.

```
template<class T>
struct image2d
{
   typedef image2d_iterator<T> iterator;
   unsigned nrows, ncols;
   T** data;
   iterator begin();
   iterator end();
   //...
};
```

```
template<class InputIter>
void bar(InputIter _first, InputIter _last)
{
   for (InputIter i = _first; i != _last; ++i)
      *i = 0;
}

int main()
{
   image2d<float> ima;
   //...
   foo(ima.begin(), ima.end());
}
```

Finally, the procedure `bar` can accept various kind of input. It is fully generic, type-safe and, moreover, as fast as dedicated C. Indeed, the use of parameterization (templates) along with type deductions (typedefs) is handled by the compiler, that is, statically. In a classical object-oriented way of programming, lot of work is performed at run-time (e.g, method dispatch through a hierarchy) and the resulting programs are not as efficient as the one presented in this section. As far as we know, the only other image processing library based on this programming paradigm is VIGRA by Köthe [8].

Please note that the contents of sections 2.1 to 2.3 is discussed in more detail in [5], [4], [3], [6] and [8].

2.4 Image Processing Style

We find the previous proposal unsatisfactory: programs should be closer to algorithm descriptions in mathematical language rather than in a computer

scientist language. Our proposal is not to design a new language dedicated to image processing such as in [1] but to provide *tools* that make easier programming for practitioners: we want something like:

```
template<class _I>
void bar(image<_I>& _ima)
{
   Exact_ref(I, ima);
   Iter(I) p(ima);
   for_all(p) ima[p] = 0;
}
```

```
int main()
{
   image2D<float> ima;
   //...
   bar(ima);
}
```

And that's indeed what we have in our library.

Please note that, for some mathematical morphology algorithms, there are few approaches to design them (parallel, in-place, based on priority queue and so on). Although we cannot provide a single version for all these flavors, we are still able to preserve the other levels of genericity.

3. Case Studies of Mathematical Morphology Operators

In this section, we study several morphological operators and we show that mathematical formulas and algorithms can easily be written using our tools[2]. In particular, the watershed operator is described as suggested by d'Ornellas and van den Boomgaard [4].

Table 1. Some Simple Morphological Operators.

Operator	Formula	Code
dilation	$\forall x,\ [\delta_B(f)](x) = \max_{b \in B} f(x+b)$	for_all(x) df[x] = max(f, x, B);
closing	$\phi_B(f) = \varepsilon_{\check{B}}[\delta_B(f)]$	erosion(dilation(f, B), -B);
black top-hat	$BTH_B(f) = \phi_B(f) - f$	minus(closing(f, B), f);
TH contrast op.	$\kappa^{TH} = Id + WTH_B - BTH_B$	plus(f, minus(white_top_hat(f, B), black_top_hat(f, B)));

3.1 Simple Operators

Table 1 presents how four morphological operators are translated in C++ code.

Dilation. In our library, the body of the `dilation` procedure (left column below) performs the following operations. Line 2 first defines the output image,

[2]In future versions, C++ operator overloading capabilities will be used in order to get a more natural way of expressing formulas. For instance, `minus(closing(f,B), f)` will be replaced by `closing(f,B) - f`.

`fd`, whose type is the procedure parameter `I`. To this aim, `fd` needs some structural information from the input image `f` (for instance, its size). At line 3, an iterator `x` is declared whose type is deduced from `I`. Then, the iteration is performed. To remain close to the mathematical formula, a particular function, `max`, is specialized according to the nature (type) of the structuring element. That is, whether `B` is flat or not, calling `max` does not run the same code.

```
1  border::adapt_copy(f, B.delta());
2  I  fd(f.info());
3  Iter(I)  x(f.info());
4  for_all(x)
5     fd[x] = max(f, x, B);
6  return fd;
```

```
7  Iter(SE)  b(B);
8  b = begin;
9  Value(f)  val = f[x + b];
10 for_all_remaining(b)
11    if (val < f[x + b])
12       val = f[x + b];
13 return val;
```

The body of procedure `max` when `B` is a flat structuring element is presented in the right column above. Note that in order to keep this procedure generic, all implementation details are hidden. An important feature of our library is that we do not have to care too much about accessing data out of image support. For instance, in the case of 2D images, `x+b` (e.g., line 11) may fall outside the image support if `x` is near the image boundary. In order to save the programmer from writing extra code to test if `x+b` is valid, some image type, such as 2D image, have an outside border and we can assign values to these particular points. In the case of dilation, we call `border::adapt_copy` (line 1) which first adapts the border size of the image to that of the structuring element and then copies values of the image inner boundary to the border[3]. Finally, dilating induces no side effect. Image processing routines relying on masks, windows, neighborhoods or structuring elements, are simpler to implement.

Closing. In our library, we do not want to annoy the user with memory management of the data structures such as images or graphs. In the case of the closing operator, a temporary image is first created resulting from a dilation process, and then, erosion is applied to obtain the final result. In classical libraries, the programmer should delete the temporary image to recover its memory. There is a risk of forgetting this deletion and to get some memory leaks at run-time. Another immediate consequence is that operators cannot be chained such as in:

```
erosion(dilation(f, B), -B)
```

because no variable holds the temporary image which thus is responsible for a memory loss.

For user convenience, we have implemented a transparent memory management: when a data structure is no longer referenced, it is automatically destroyed. Combining operators is thus made as simple as possible.

[3]In the case of the input image being a graph, since the notion of border does not exist, calling `border::adapt_copy` is still valid but does not execute any code. Another approach is to set the border to $-\infty$ ($+\infty$) in the case of dilation (erosion).

Top-Hat Contrast Operator. A lot of morphological operators rely on arithmetics, and usually, image processing libraries use *built-in* types —that is, types provided by the programming language— to express the nature of data. A resulting well-known problem is value overflow. In the sample line below positive values are encoded on 8 bit, ranging from 0 to 255:

```
unsigned char i = 255, j = 1, k = (i + j) / 2;
```

we finally have `k` set to 0. In the case of non-flat structuring elements and in the case of morphological operators involving arithmetics, e.g., the top-hat contrast operators, the programmer should not deal with from these problems. To this end, we have defined our own data types and the corresponding safe arithmetics. For instance, in the line above, one should use `int_u8` instead of `unsigned char` and the compiler raises en error at *at compile-time* because the type of expression `(i + j) / 2` is now `int_u9` and because the assignment from this type towards the "smaller" type `int_u8` is forbidden.

We also have equipped our library with conversion routines that can be used either as stand-alone functions or as first argument of other routines. In the sample code below, the user does not need to know the data type of the image returned by the contrast operator; she can guarantee that, at the very end, every pixel value falls between 0 and 255. So, she benefits simultaneously from safe arithmetics *and* convenient data type manipulations.

Last, the notion of *scoping*, whose correctness is verified by the compiler, ensures the programmer that she uses as little memory as possible; at the end of the `main` scope, only `lena` lies in memory.

```
// file inclusions:
#include "basics2d.hh"
#include "io/pnm.hh"
#include "convert/bound.hh"
#include "morpho/top_hat.hh"

// usage declarations:
using namespace oln;
using convert::bound;
using morpho::top_hat_contrast_op;
```

```
int main ()
{
  image2d<int_u8> lena;
  { // sub-scope
    read_pnm(lena, "lena.pgm");
    image2d<int_u8> output
      = top_hat_contrast_op(bound<int_u8>(),
                            lena,
                            square(5));
    save_pnm(output, "output.pgm");
  }
  // here, only lena is in the scope
}
```

3.2 Watershed

Mathematical morphology operators, such as the watershed, are often more complicated than those already discussed. In [4], d'Ornellas and van den Boomgaard argue that a *generic* implementation of the watershed is possible based on a wave-front propagation; they give the algorithm canvas that we recall in the left column below. Our library provides a *generic* implementation of this algorithm, that is, a function which works on various structures (n-D images, graphs, etc.) containing data of various types[4]. The corresponding code excerpts from our library are given in the right column below.

[4]The queue-based priority algorithm presented here is of course optimal for discrete data types. However, the user can alsa call it when data are floating values.

```
Initialization Step:
PQ q;

For (every p in domain D){
    If (M(p) != 0 and (exist (p') in N_G(p) :
            M(p') == 0))
        q.enq(p, f(p));}

Data Driven Propagation Step:
While(!(q.empty())){
    q.deq();

    For(every(p') in (N_G(p) ∩ D))
        If (M(p') == 0)
            If (exists (p'') in N_G(p') :
                    M(p'') != M(p))
                M(p') = WSHED;
            else {
                M(p') = M(p);
                q.enq(p', f(p'))
            }
}
```

```
// Initialization Step:
PQ q;
Iter(I2) p(M);

for_all(p)
    if (M[p] != 0 && exist_eq(M, p, Ng, 0))

        q.push(queue_elt(p, f[p]));

// Data Driven Propagation Step:
while (!q.empty()) {
    Point(I1) p = q.top().first; q.pop();
    Neighb(N) p_prime(Ng, p);
    for_all_neigh (p_prime) if (M.hold(p_prime))
        if (M[p_prime] == 0)
            if (exists_neq(M, p_prime, Ng, M[p]))

                M[p_prime] = WSHED
            else {
                M[p_prime] = M[p];
                q.push(queue_elt(p_prime, f[p_prime]));
            }
}
```

Note that we also succeeded in providing tools making "sophisticated" morphological algorithms implementation easy.

4. Conclusion

In this article, we have shown that computer programs can achieve both *genericity* and *user-friendliness*. Based on the conclusions of d'Ornellas and van den Boomgaard [4], we have proposed solutions —some of them being described here— and we have built an appropriate framework of object-oriented tools: OLENA. OLENA is a library dedicated to image processing practitioners, and in particular, to mathematical morphology users. The sources of OLENA are freely available on the Internet at the address:

`http://www.lrde.epita.fr/olena`

Last, getting all the benefits described in this article has not compromised efficiency. Table 2 gives processing time for some morphological operators. These algorithms were tested on the classical gray-level 256×256 image LENA with a 1 Ghz personal computer running GNU/LINUX; the code was compiled using the GNU C++ compiler with all optimizations enabled. The column "regular" refers to operators being implemented in a classical way; equivalent "fast" versions of morphological operators, based on [14], are also available in OLENA. We are aware of the optimal 11×11 dilation which uses a decomposition of two dilations by straight lines. The algorithm is $O(1)$ with line dilation approaches such as van Herck's [15]. But we do not use it because this is only a text-book study to evaluate and compare execution times. Finally, table 3 presents major functionalities in OLENA.

References

[1] R. Cecchini and A. Del Bimbo. A programming environement for imaging applications. *Pattern Recognition Letters*, 14:817–824, October 1993.

[2] M. Dobie and P. Lewis. Data structures for image processing in C. *Pattern Recognition Letters*, 12(8):457–466, 1991.

Table 2. Performance Evaluation (time in seconds).

Algorithms	Structural Elt.	Regular	Fast
Dilation	3×3 Square	0,04	0.05
Dilation	11×11 Square	0,46	0,12
Dilation	Disk of radius 6	0,44	0,13
Closing	Disk of radius 6	0,85	0,31
Top-Hat Contrast Op.	Disk of radius 6	1,74	0,62
Watershed	4-Connectivity	0,17	—

Table 3. Functionalities.

Dilation / Erosion
Closing / Opening
White Top Hat / Black Top Hat / Top Hat Contrast Operator
Hit-or-Miss
Hit-or-Miss Opening Foreground / Background
Hit-or-Miss Closing Foreground / Background
Beucher / Internal / External Gradient
Geodesic Dilation / Erosion
Geodesic Reconstruction by Dilation / Erosion (simple, sequential, hybrid)
Minima Imposition
Regional Minima
Watershed
Minima / Maxima Killer

[3] M. C. d'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. PhD thesis, University of Amsterdam, 2001.

[4] M. C. d'Ornellas and R. van den Boomgaard. Generic algorithms for morphological image operators — a case study using watersheds. In H. Heijmans and J. Roerdink, editors, *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 323–330, 1998.

[5] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, April 1998.

[6] T. Géraud, Y. Fabre, A. Duret-Lutz, D. Papadopoulos-Orfanos, and J.-F. Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *Proceedings of the 15th International Conference on Pattern Recognition*, volume 4, pages 816–819. IEEE Computer Society, September 2000.

[7] U. Köthe. Reusable implementations are necessary to characterize and compare vision algorithms. in DAGM-Workshop on Performance Characteristics and Quality of Computer Vision Algorithms, September 1997.

[8] U. Köthe. STL-style generic programming with images. *C++ Report Magazine*, 12(1):24–30, January 2000.

[9] D. Lawton and D. Mead. A modular object oriented image understanding environment. In *Proceeding of the 10th International Conference on Pattern Recognition*, volume 2, pages 611–616, Atlantic City, NJ, USA, June 1990.

[10] S. Mallat. *A Wavelet Tour of Signal Processing*, chapter 1, pages 17–18. Academic Press, 1999.

[11] Scientific computing in object-oriented languages. Web page. http://oonumerics.org/.

[12] I. Pitas. *Digital Image Processing Algorithms and Applications*. Wiley, 2000.

[13] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[14] M. Van Droogenbroeck and H. Talbot. Fast computation of morphological operations with arbitrary structuring elements. *Pattern Recognition Letters*, 17(14):1451–1460, 1996.

[15] M. Van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octogonal kernels. *Pattern Recognition Letters*, 13:517–521, 1992.

C.3 WRITE GENERIC MORPHOLOGICAL ALGORITHMS ONCE, RUN ON MANY IMAGES

Milena : Write Generic Morphological Algorithms Once, Run on Many Kinds of Images. *International Symposium on Mathematical Morphology* (**ISMM**), August 2009. In "Mathematical Morphology", pp. 295-306, Lecture Notes in Computer Science Series, Vol. 5720, Springer-Verlag. With Roland Levillain and Laurent Najman.

Milena: Write Generic Morphological Algorithms Once, Run on Many Kinds of Images

Roland Levillain[1,2], Thierry Géraud[1,2], and Laurent Najman[2]

[1] EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire, FR-94276 Le Kremlin-Bicêtre Cedex, France
[2] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Équipe A3SI,
ESIEE Paris, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand Cedex, France
{roland.levillain,thierry.geraud}@lrde.epita.fr, l.najman@esiee.fr

Abstract. We present a programming framework for discrete mathematical morphology centered on the concept of genericity. We show that formal definitions of morphological algorithms can be translated into actual code, usable on virtually any kind of compatible images, provided a general definition of the concept of image is given. This work is implemented in Milena, a generic, efficient, and user-friendly image processing library.

1 Introduction

Software for mathematical morphology targets several audiences: *end users*, *designers* and *providers*. *End users* of morphological tools want to apply and assemble algorithms to solve image processing, pattern recognition or computer vision problems. *Designers* of morphological operators build new algorithms by using constructs from their software framework (language, libraries, toolboxes, programs, etc.). Finally, *providers* of data structures are interested in extending their framework with new data types (images, values, structuring elements, etc.).

The size of the population of these categories is decreasing: there are more end users of morphological software than designers of algorithms, and the latter themselves outnumber providers of data structures. Morphological frameworks usually address the needs of their clients in this order, and even sometimes ignore the third or second categories. However, a full morphological framework should suit all groups of users so that structures of providers and algorithms of designers can be used by every actor. In this article, we present a software framework for mathematical morphology designed with two major goals in mind:

1. Be as simple as calling C routines for end users.
2. Be modular enough to be extended w.r.t. algorithms and data structures;

and four minor:

3. Be generic: if a morphological operator admits a general definition whatever the context (topology of the image, structuring element, etc.), then this algorithm should have a corresponding single implementation.

M.H.F. Wilkinson and J.B.T.M. Roerdink (Eds.): ISMM 2009, LNCS 5720, pp. 295–306, 2009.

4. Be close to theory: reading (and writing) algorithms should eventually become natural to scientists used to mathematical morphology notations.
5. Retain efficiency (with respect to run time speed and memory usage) when it is possible. Dedicated and efficient implementations of morphological algorithms for certain cases are known and should be selected whenever possible.
6. Be user-friendly: users should not have to address memory-related issues or deal with a program silently failing because of an arithmetic overflow. The tool should handle these situations, and help the user diagnose any problem.

The paradigm of Generic Programming (GP) [1] which is at the heart of many modern C++ libraries [2,3] and its application to the C++ language address many of these concerns. Developing a software library in the context of GP requires some effort. One of the key ideas is that such a library should be based on *abstractions* of the domain (mathematical morphology in this case). The above requirements will not be fully satisfied if we fail to reify intrinsic concepts of the domain as abstractions. Several image processing libraries relying on the GP paradigm exist (ITK [4], VIGRA [5], Morph-M [6]) but as far as the authors know, none of them seem to meet all of the above requirements.

From the general lattice theory on which is built mathematical morphology, many authors have proposed derived theoretical frameworks. The first ones are graphs [7,8], later extended to store information both *on* vertices and *between* vertices (on edges) [9,10]. The notion of *complex* (see Section 3.2) has also been used to express topological and geometrical attributes of images beyond the scope of graphs [11,12]. Generic programming frameworks to implement algorithms on complexes and grid data structures have been proposed [13,14,15]. Other possible frameworks include combinatorial maps [16] and orders [17].

Let us for instance consider the framework of graphs as the basis of morphological image processing in order to express definitions and properties as general as possible (and meet requirement 3). We could then use a graph-related library like the Boost Graph Library (BGL) [3]. However, such a design suffers from limitations, as mathematical morphology, despite having many intersections with graph theory, has its own definitions, idioms, notations, and issues. Therefore, adapting morphological algorithms to a graph software framework would distort their definitions, which is contrary to requirements 4 and 6. Moreover, we would probably lose efficiency (requirement 5) for restricted use cases in image processing (but at the same time, the most common ones): regular 2D or 3D images on grids, classical structuring elements, etc. Finally, setting graphs as the ultimate representation of images in mathematical morphology once and for all might prevent future extensions. For example the notion of complex mentioned previously, which extends the notion of graph, can be considered to form the basis of a morphological framework.

Therefore, instead of using a fixed system, we propose to rethink mathematical morphology under the light of generic programming [18]. The first step is to define software abstractions matching morphological entities (topology, sets, functions, lattices, structuring elements, geometry, etc.), starting with the concept of *discrete image*. Then, it will be possible to express algorithms in terms of

these concepts on the one hand, and provide actual data structure implementing these abstractions on the other hand.

In this paper, we present a generic and efficient C++ programming library, Milena, a part of the Olena image processing platform [19,20]. Milena uses and extends the idea of GP [21]. It implements the abstractions for mathematical morphology software mentioned previously.

This article mainly targets end users of the library and designers of algorithms. It is structured as follows: in Section 2, we study how morphological algorithms are commonly implemented and what are the issues of classical yet restrictive designs. Section 3 proposes a generic definition of an image and shows how this genericity is expressed through the image's traits. As an illustration, a small generic image processing chain is given in Section 4 and applied to various images.

2 Software Implementation of Mathematical Morphology

Translating mathematical morphology methods and objects into readable and usable algorithms is often biased either to satisfy constraints of actual data or meet software and hardware requirements. An example of the first circumstance is the prominent case of a 2-dimensional single-valued image, set on a rectangular (boxed) domain with integer coordinates (a discrete grid $D \subseteq \mathbb{Z}^2$). Many morphological algorithms are solely expressed with this framework in mind. The second bias is computer-dependent: for the sake of efficiency or simplicity of implementation, algorithms sometimes include language- or hardware-related constructs: buffers, loops, dimension decomposition, out-of-bounds behavior, etc.

Let us consider a simple example: the elementary morphological dilation of a gray-level image `ima` with a (flat) structuring element. A shortened definition in the framework of complete lattices [22] would be:

$$\delta_B(I)(x) = \sup_{h \in B} I(x + h)$$

where I (the image to process) is a function $D \to V$ associating a point from the domain D to a value from the set V; and B the structuring element associated to, e.g., the usual 4-connectivity neighborhood. A simple implementation in C++ could be as the one from Algorithm 1. However, this solution makes extra hypotheses that were not contained in the definition of the operation, e.g.:

1. The image is 2-dimensional, since it is accessed using a (row, col) notation.
2. Sites are points with nonnegative integers coordinates starting at 0.
3. The values of the image are compatible with the 8-bit `unsigned char` type.
4. The values of the image form a totally ordered set; hence the operator $<$ can be used to compute the supremum.
5. The structuring element is based on the 4-connectivity.

Each of the previous hypotheses is an actual limitation on the generality of Algorithm 1. It cannot be reused as-is if for instance one or several of the following conditions are expected:

```
image dilation(const image& input) {
  image output (input.nrows(), input.ncols()); // Initialize an output image.
  for (unsigned int r = 0; r < input.nrows() − 1; ++r) // Iterate on rows.
    for (unsigned int c = 0; c < input.ncols() − 1; ++c) { // Iterate on columns.
      unsigned char sup = input(r, c);
      if (r > 0                    && input(r−1, c) > sup) sup = input(r−1, c);
      if (r < input.nrows() − 1 && input(r+1, c) > sup) sup = input(r+1, c);
      if (c > 0                    && input(r, c−1) > sup) sup = input(r, c−1);
      if (c < input.ncols() − 1  && input(r, c+1) > sup) sup = input(r, c+1);
      output(r, c) = sup;
    }
  return output;
}
```

Algorithm 1. Non generic implementation of a morphological dilation of an 8-bit gray-level image on a regular 2D grid using a 4-c flat structuring element.

1. The input is a 3-dimensional image.
2. Its points are located on a box subset of a floating-point grid, that does not necessarily include the origin.
3. The values are encoded as 12-bit integers or as floating-point numbers.
4. The image is multivalued (e.g., a 3-channel color image).
5. The structuring element represents an 8-connectivity.

Even if the class of images accepted by Algorithm 1 covers day-to-day needs of numerous image processing practitioners, image with features from the previous list are also quite common in fields like biomedical imaging, astronomy, document image analysis or arts. Algorithm 1 also highlights less common restrictions. As is, it is unable to process images with the following features:

- A domain
  - which is not an hyperrectangle (or "box");
  - which is not a set of *points* located in a geometrical space, e.g., given a 3D triangle mesh, one can build an image by mapping each triangle to a set of values;
  - which is a restriction (subset) of another image's domain, still preserving essential properties, like the adjacency of the sites.
- A neighborhood where neighbors of a site are not expressed with a fixed-set structuring element, but through a function associating a set of sites to any site of the image. This is the case when the domain of the image is a graph, where values are attached to vertices [8].
- Non scalar image values, like color values.

Furthermore, the style used in Algorithm 1 does not allow for optimizations. An optimized code (taking advantage, for example, of a totally ordered domain of values, with an attainable upper bound), requires a whole new algorithm per compatible data structure.

```
template <typename I, typename W>
mln_concrete(I) dilation (const I& input, const W& win) {
  mln_concrete(I) output; initialize (output, input); // Initialize output.
  mln_piter(I) p(input.domain()); // Iterator on sites of the domain of 'input'.
  mln_qiter(W) q(win, p); // Iterator on the neighbors of 'p' w.r.t. 'win'.
  for_all(p) {
    accu::supremum sup = input(p); // Accumulator computing the supremum.
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
    return output;
  }
}
```

Algorithm 2. Generic implementation of a morphological dilation.

In the remainder of this paper, we show how the programming framework of Milena allows programmers to easily write generic and reusable [23] image processing chains using mathematical morphology tools. For instance a Milena equivalent of Algorithm 1 could be Algorithm 2. In this algorithm `I` is a generic image type, while `W` is the type of a generic structuring element (also named *window*). `p` and `q` are objects traversing respectively the domain of `ima` and the sites of the structuring element `win` centered on `p`. The predicate `input.has(q)` ensures that `q` is a valid site of `input` (this property may not be verified e.g. when `p` is on the border of the image). `sup` iteratively computes the supremum of the values under `win` for each site `p`. An example of use similar to Algorithm 1 would be:

```
image2d<unsigned char> ima_dil = dilation(ima, win_c4p());
```

where `win_c4p()` represents the set of neighboring sites in the sense of the 4-connectivity plus the center of the structuring element.

Algorithm 2 is a small yet readable routine and is no longer specific to the aforementioned 2-dimensional 8-bit gray-level image case of Algorithm 1. It is generic with respect to its inputs, and no longer restricted by the limitations we mentioned previously. For instance it can be applied to an image defined on a Region Adjacency Graph (RAG) where each site is a region of an image, associated to an n-dimensional vector expressing features from each underlying region, provided a supremum is well defined on such a value type.

3 Genericity in Mathematical Morphology

3.1 A Generic Definition of the Concept of Image

The previous considerations about the polymorphic nature of a discrete image require a clear definition of the concept of image. To embrace the whole set of aforementioned aspects, we propose the following general definition.

Definition. An image I is a function from a domain D to a set of values V. The elements of D are called the *sites* of I, while the elements of V are its *values*.

For the sake of generality, we use the term *site* instead of *point*: if the domain of I were a RAG, it would be awkward to refer to its elements (the regions) as "points". This definition forms the central paradigm of Milena's construction. However, an actual implementation of an image object cannot rely only on this definition. It is too general as is, and mathematical morphology algorithms expect some more information from their inputs, like whether V is a complete lattice, how the neighboring relation between sites is defined, etc. Therefore, we define additional notions to supplement the definition of an image. These notions are designed to address orthogonal concerns in image processing and mathematical morphology, so that actual definitions (*implementations*) of images can be changed along one axis (e.g., the topology of D) while preserving another (e.g., the existence of a supremum for each subset $X \subseteq V$).

Algorithms are then no longer defined in terms of specific image characteristics (e.g., a domain defined as two ranges of integers representing the coordinates of each of its points) but using *abstractions* (e.g., a *site iterator* object, providing successive accesses to each site of the image, that can be deduced from the image itself). This paradigm based on Generic Programming promotes "Write Once, Reuse Everywhere Applicable" design of algorithms by introducing abstract entities (akin to mathematical objects) in software defined by their *properties*.

The *genericity* of our approach resides in both the organization of the library around entities dedicated to morphological image processing (images, sites, site sets, neighborhoods, value sets, etc.) and in the possibility to extend Milena with new structures and algorithms, while preserving and reusing existing material.

The next section presents the main entities upon which we define morphological algorithms in Milena, and how they provide genericity in mathematical morphology.

3.2 Genericity Traits

We define actual images as models of the previous definition of an image, with extra *properties* on I, D or V. These traits express the generic nature of this definition, and are related to the notions of this section. Each of them is as much orthogonal (or loosely coupled) to the others as possible, so that an actual implementation of one of these concepts can be defined and used with many algorithms regardless of the other features of the input(s). In the rest of this section, we illustrate how the limitations of Algorithm 1 mentioned in Section 2 are lifted by the generic implementation of Algorithm 2.

Restriction of the Domain. It is possible to express the restriction of an image `ima` to a subset `s` of its domain using the dedicated operator `|`; the result can then be used as input of an algorithm:

```
image2d<int> ima_dil = morpho::dilation(ima | s, win);
```

The subset `s` can either be a comprehensive collection of sites (array, set, etc.) or a predicate. A classical example is the use of a "mask" to restrict the domain of an image. This mask can for instance be a watershed line previously computed on `ima`; the dilation above would act as a reconstruction of the pixels of `ima` belonging to this watershed line.

Structuring Elements, Neighborhoods and Windows. Structuring elements of mathematical morphology can be generalized with the notion of *windows*: functions from D to $\mathcal{P}(D)$. A special case of window is a *neighborhood*: a non-reflexive symmetric binary relation on D. In the case of images set on n-dimensional regular grids (as in the previous example of dilation of a 2D image), D is a subset of $\mathbb{Z}^n$ and is expressed as an n-dimensional bounding box. Windows' members can be expressed regardless of the considered site, using a (fixed or variable) set of vectors, called *delta-sites*, as they encode a difference between two sites. For instance a 4-connectivity window is the set of 2D vectors $\{(-1,0),(0,-1),(0,0),(0,1),(1,0)\}$.

In more general cases, windows are implemented as domain-dependent functions. For instance, the natural neighbors of a site p (called the center of the window) of a graph-based image, where D is restricted to the set of vertices, are its adjacent vertices, according to the underlying graph. Such a window is implemented by an object of type `adjacent_vertices_window_p` in Milena (see below). This window does not contain delta-sites; instead, it encodes the definition of its member sites as a function of p. Using an iterator q to iterate over this window (as in Algorithm 2) successively returns each of its members.

Topological Structure. The structure of D defines relations between its elements. Classical images types are set on the structure of a regular graph, where each vertex is a site of I. More general images can be defined on general graphs, where sites can be either the vertices of the graph, its edges or even both.

An example of dilation on regular 2D image was given in Section 2. In the case of an image associating 8-bit integer values to the elements (vertices and edges) of graph, computing an elementary dilation with respect to the adjacent vertices would be written as this:

```
graph_image<int_u8> ima_dil =
  morpho::dilation(ima | vertices , adjacent_vertices_window_p());
```

`ima | vertices` creates an image based on the subset of vertices of on-the-fly, while `adjacent_vertices_window_p()` returns a window mapping each vertex to the set of its neighbors plus the vertex itself.

We can generalize this idea by using *simplicial complexes*. An informal definition of a simplicial complex (or simplicial d-complex) is "a set of simplices" (plural of simplex), where a simplex or n-simplex is the simplest manifold that can be created using n points (with $0 \leq n \leq d$). A 0-simplex is a point, a 1-simplex a line segment, a 2-simplex a triangle, a 3-simplex a tetrahedron. Simplicial complexes extends the notion of graphs; a graph is indeed a 1-complex. They can be used to define topological spaces, and therefore serve as supports for images. Figure 1 shows an example of simplicial 3-complex.

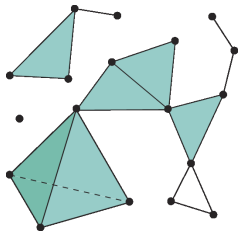


Fig. 1. A simplicial 3-complex

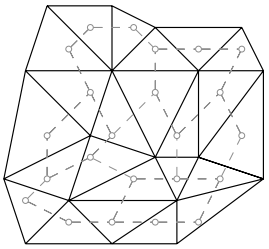


Fig. 2. A mesh seen as a simplicial 2-complex

Let us consider an image `ima` based on a simplicial 2-complex (Figure 2) where each element is located in space according to a geometry G (the notion of site location and geometry is addressed later) with 8-bit integer values. The domain D of this image is composed of points, segments and triangles. We consider a neighboring relation among triangles (also known as 2-faces) where two triangles are neighbors iff they share a common edge (1-face). The code to compute the dilation of the values associated to the triangles of D with respect to this relation is as follows:

```
complex_image<2, G, int_u8> ima_dil =
  dilation (ima | faces (2), complex_lower_dim_connected_n_face_window_p<2, G>());
```

As in the example of the graph-based image, `ima | faces(2)` is a restriction of the domain of `ima` to the set of 2-faces (triangles). The expression `complex_lower_dim_connected_n_face_window_p<2, G>()` creates the neighboring relation given earlier (for a site p of dimension n, this window is the set of n-faces sharing an $(n-1)$-face, plus the p itself).

Site Location and Geometry. In many context, the location of the sites of an image can be independent from the structure of D. For instance if the domain of I is built on the vertices of a graph, these sites can be located in $\mathbb{Z}^n$ or $\mathbb{R}^n$ with $n \in \mathbb{N}^*$. In some cases, the location of sites is polymorphic. E.g., if D is a 3-dimensional simplicial complex located in a 3D space (as in Figure 1), the location of site p can be a 3D point (if p is a vertex), a pair or points (if p is an edge), a triplet of points (if it is a triangle) or a quadruplet (if it is a tetrahedron). We encode such information as a set of locations called a *geometry*. For instance, the term `G` from the previous code is a shortcut for `complex_geometry<2, point2d>`.

Value Set. Almost all framework support several (fixed) value sets representing mathematical entities such as $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ or subsets of them. Some of them also support Cartesian products of these sets. Not so many support user-defined value types. To be able to process any kind of values, properties should be attached to these sets: quantification, existence of an order relation, existence of a supremum or infimum, etc. Then it is possible to implement algorithms with

expected constraints on V. For instance, one can perform a dilation of a color image with 8-bit R, G, B channels by defining a supremum on the `rgb8` type:

```
rgb8 sup (const rgb8& x, const rgb8& y) {
  return rgb8(max(x.r(),y.r()), max(x.g(),y.g()),  max(x.b(),y.b()));
}
image2d<rgb8> ima_dil = morpho::dilation(ima, win_c4p());
```

3.3 Design and Implementation

Milena aims at genericity (broad applicability to various inputs, reusability) and efficiency (fast execution times, minimum memory footprint). The design of the library focuses on the following features, that we can only sketch here.

Ease of Use. The interface of Milena is akin to classical C code to users, minus the idiosyncratic difficulties of the language (pointers, manual memory allocation and reclaim, weak typing, etc.). Users do not need to be C++ experts to use the library. Images and other data are allocated and released automatically and transparently with no actual performance penalty.

Efficiency. Milena handles non-trivial objects (images, graphs, etc.) through shared memory, managed automatically. The mechanism is efficient since it avoids copying data. As for algorithms, programmers can provide several versions of a routine in addition to the generic one. The selection mechanism is static (resolved at compile-time), and more powerful than function overloading: instead of dispatching with respect to *types*, it dispatches with respect to one or several *properties* attached to one or several types [21].

Usability. Milena targets both prototyping and effective image processing. In the case of very large images (1 GB), we cannot afford multiples copies of values or sometimes even loading a whole image (of e.g. several gigabytes). Therefore, the library provides alternative memory management policies to handle such inputs: in this case, memory-mapped image types which, by design, have no impact whatsoever on the way algorithms are written or called.

4 Illustrations

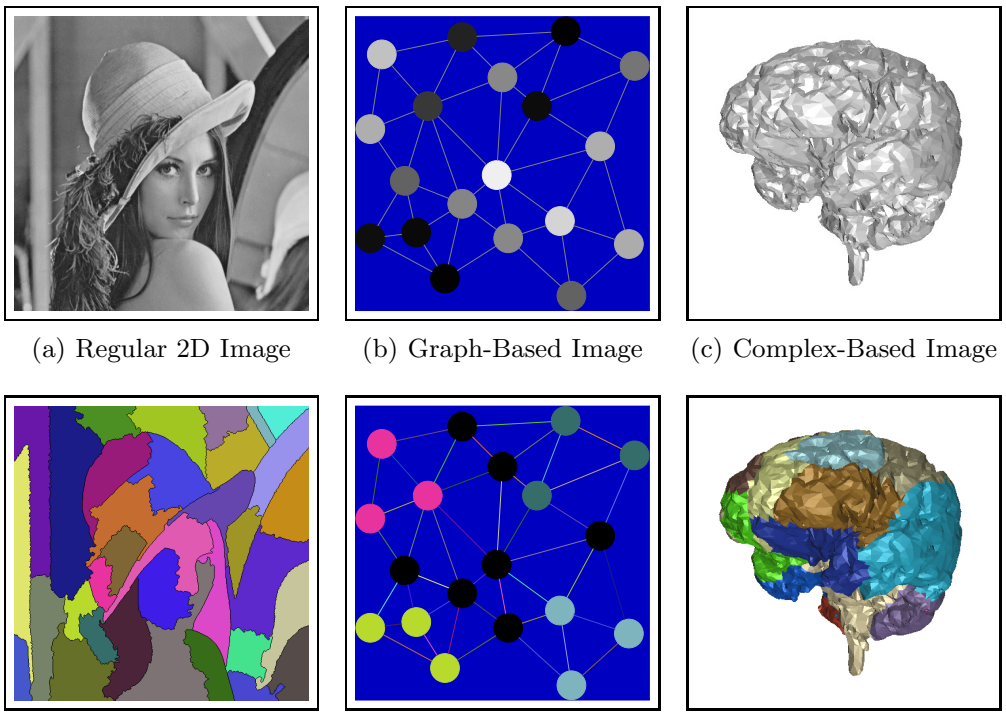
In this part, we consider a simple, classical image processing chain: from an image `ima`, compute an area closing `c` using criterion value `lambda`; then, perform a watershed transform by flooding on `c` to obtain a segmentation `s`. We apply this chain on different images `ima`. All of the following illustrations use the exact same Milena code corresponding to the processing chain above. Given an image `ima` (of type `I`), a neighborhood relation `nbh`, and a criterion value (threshold) `lambda`, this code can be written as this (`nb` is a placeholder receiving the number of catchment basins present in the watershed output image) :

```
template <typename L, typename I, typename N>
mln_ch_value(I, L) chain(const I& ima, const N& nbh, int lambda, L& nb) {
  return morpho::watershed::flooding(morpho::closing::area(ima, nbh, lambda),
                                     nbh, nb);
}
```

Regular 2-Dimensional Image. In the example of Figure 3(a), we first compute a morphological gradient used as an input for the processing chain. A 4-c window is used to compute both this gradient image and the output (Figure 3(d)), where basins have been labeled with random colors.

Graph-Based Image. Figure 3(b) shows an example of planar graph-based [7] gray-level image, from which a gradient is computed using the vertex adjacency as neighboring relation. The result shows four basins separated by a watershed line on pixels.

Simplicial Complex-Based Image. In this last example [24], a triangular mesh is viewed as a 2-simplicial complex, composed of triangles, edges and vertices (Figure 3(c)). From this image, we can compute maximum curvature values on each triangle of the complex, and compute an average curvature on edges.



(a) Regular 2D Image (b) Graph-Based Image (c) Complex-Based Image

(d) Result on a gradient of (a) (e) Result on a gradient of (b) (f) Result on the curvature of (c)

Fig. 3. Results of the image processing chain of Section 4 on various inputs

Finally, a watershed cut [25] on edges is computed, and basins are propagated to adjacent triangles and vertices for visualization purpose (Figure 3(f)).

All examples use Meyer's watershed algorithm [26], which has been proved to be equivalent to watershed cuts when used on the edges of a graph [27].

5 Conclusion

We have presented the fundamental concepts at the heart of Milena, a generic programming library for image processing and mathematical morphology, released as Free Software under the GNU General Public License. Milena allows users to write algorithms once and use them on various image types. The programming style of the library promotes simple, close-to-theory expressions.

As far as implementation is concerned, Milena extends the C++ language "from within", as a library extension dedicated to image processing. Though we designed the library to make it look familiar to image processing practitioners, it does not require a new programming language nor special tools: a standard C++ environment suffices. Moreover, as Generic Programming allows many optimizations from the compiler, the use of abstractions does not introduce actual run-time penalties.

We encourage practitioners of mathematical morphology interested in Milena to download the library at `http://olena.lrde.epita.fr/Download` and see if it can be useful to their research experiments.

Acknowledgments. The authors thank Guillaume Lazzara for his work on Milena as part of the SCRIBO project, and Alexandre Duret-Lutz for proofreading and commenting on the paper. This work has been conducted in the context of the SCRIBO project (`http://www.scribo.ws/`) of the Free Software Thematic Group, part of the "System@tic Paris-Région" Cluster (France). This project is partially funded by the French Government, its economic development agencies, and by the Paris-Région institutions.

References

1. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: Proc. of OOPSLA, pp. 115–134 (2003)
2. CGAL: Computational Geometry Algorithms Library (2008), `www.cgal.org`
3. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual, 1st edn. Addison Wesley Professional, Reading (2001)
4. Yoo, T.S. (ed.): Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis. AK Peters Ltd. (2004)
5. Köthe, U.: STL-style generic programming with images. C++ Report Magazine 12(1), 24–30 (2000)
6. Enficiaud, R.: Algorithmes multidimensionnels et multispectraux en Morphologie Mathématique: approche par méta-programmation. PhD thesis, CMM, ENSMP, Paris, France (February 2007)

7. Vincent, L.: Graphs and mathematical morphology. Signal Processing 16(4), 365–388 (1989)
8. Heijmans, H., Vincent, L.: Graph morphology in image analysis. In: Dougherty, E. (ed.) Mathematical Morphology in Image Processing, pp. 171–203. M. Dekker, New York (1992)
9. Meyer, F., Angulo, J.: Micro-viscous morphological operators. In: Proc. of ISMM, pp. 165–176 (2007)
10. Cousty, J., Najman, L., Serra, J.: Some morphological operators in graph spaces. In: Proceedings of ISMM 2009 (2009); These proceeedings
11. Bertrand, G., Couprie, M., Cousty, J., Najman, L.: Chapter title: Ligne de partage des eaux dans les espaces discrets. In: Najman, L., Talbot, H. (eds.) Morphologie mathématique: approches déterministes, pp. 123–149. Hermes Sciences (2008)
12. Loménie, N., Stamon, G.: Morphological mesh filtering and α-objects. Pattern Recognition Letters 29(10), 1571–1579 (2008)
13. Köthe, U.: Generic programming techniques that make planar cell complexes easy to use. In: Bertrand, G., Imiya, A., Klette, R. (eds.) Digital and Image Geometry. LNCS, vol. 2243, pp. 17–37. Springer, Heidelberg (2002)
14. Kettner, L.: Designing a data structure for polyhedral surfaces. In: Proc. of SCG, pp. 146–154. ACM, New York (1998)
15. Berti, G.: GrAL: the grid algorithms library. FGCS 22(1), 110–122 (2006)
16. Edmonds, J.: A combinatorial representation for polyhedral surfaces. Notices of the American Mathematical Society 7 (1960)
17. Bertrand, G., Couprie, M.: A model for digital topology. In: Bertrand, G., Couprie, M., Perroton, L. (eds.) DGCI 1999. LNCS, vol. 1568, pp. 229–241. Springer, Heidelberg (1999)
18. d'Ornellas, M.C., van den Boomgaard, R.: The state of art and future development of morphological software towards generic algorithms. International Journal of Pattern Recognition and Artificial Intelligence 17(2), 231–255 (2003)
19. LRDE: The Olena image processing library (2009), `http://olena.lrde.epita.fr`
20. Darbon, J., Géraud, T., Duret-Lutz, A.: Generic implementation of morphological image operators. In: Proc. of ISMM, Sydney, Australia, CSIRO, pp. 175–184 (2002)
21. Géraud, T., Levillain, R.: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In: Proc. of MPOOL, Paphos, Cyprus (July 2008)
22. Goutsias, J., Heijmans, H.J.A.M.: Fundamenta morphologicae mathematicae. Fundamenta Informaticae 41(1-2), 1–31 (2000)
23. Köthe, U.: Reusable software in computer vision. In: Jähne, B., Haussecker, H., Geißler, P. (eds.) Handbook of Computer Vision and Applications. Systems and Applications, vol. 3, pp. 103–132. Academic Press, San Diego (1999)
24. Alcoverro, M., Philipp-Foliguet, S., Jordan, M., Najman, L., Cousty, J.: Region-based 3D artwork indexing and classification. In: 3DTV, pp. 393–396 (2008)
25. Cousty, J., Bertrand, G., Najman, L., Couprie, M.: Watershed cuts: minimum spanning forests and the drop of water principle. IEEE PAMI (to appear, 2009)
26. Meyer, F.: Un algorithme optimal de ligne de partage des eaux. In: Actes du 8e Congrès AFCET, Lyon-Villeurbanne, France, AFCET, pp. 847–857 (1991)
27. Cousty, J., Bertrand, G., Najman, L., Couprie, M.: On watershed cuts and thinnings. In: Coeurjolly, D., Sivignon, I., Tougne, L., Dupont, F. (eds.) DGCI 2008. LNCS, vol. 4992, pp. 434–445. Springer, Heidelberg (2008)

C.4 UNE APPROCHE GÉNÉRIQUE DU LOGICIEL POUR LE TDI PRÉSERVANT LES PERFORMANCES

Une approche générique du logiciel pour le traitement d'images préservant les performances. *Symposium on Signal and Image Processing (GRETSI)*, Septembre 2011. Avec Roland Levillain et Laurent Najman. 

Une approche générique du logiciel pour le traitement d'images préservant les performances

Roland LEVILLAIN[1,2], Thierry GÉRAUD[1,2]*, Laurent NAJMAN[2]

[1]Laboratoire de Recherche et Développement de l'EPITA (LRDE)
14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre Cedex, France

[2]Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge
Équipe A3SI, ESIEE Paris, Cité Descartes, BP 99, 93162 Noisy-le-Grand Cedex, France
roland.levillain@lrde.epita.fr, thierry.geraud@lrde.epita.fr, l.najman@esiee.fr

Résumé – L'approche générique du logiciel en traitement d'image permet d'écrire des algorithmes réutilisables, compatibles avec de nombreux types d'entrées. Cependant, ce choix de conception se fait souvent au détriment des performances du code exécuté. Du fait de la grande variété des types d'images existants et de la nécessité d'avoir des implémentations rapides, généricité et performance apparaissent comme des qualités essentielles du logiciel en traitement d'images. Cet article présente une approche permettant l'écriture de variantes d'algorithmes basées sur les caractéristiques des types de données utilisés, offrant un compromis entre généricité et performance. Il est possible d'obtenir des temps d'exécutions comparables à ceux d'un code dédié, et dans certains cas de surpasser des routines optimisées manuellement.

Abstract – The generic approach to image processing software allows users to write reusable algorithms compatible with many input types. However, this design choice is often made at the expense of some performance loss. The great variety of image types and the need for fast implementations make both genericity and efficiency desirable features for image processing software. This paper presents an approach enabling the definition of algorithm variants based on data types features, offering an adjustable trade-off between genericity and efficiency. Such variants can match dedicated code in terms of execution times, and may sometimes perform better than routines optimized by hand.

1 Introduction

À l'instar de nombreux domaines de calcul scientifique, le traitement d'images (TDI) doit faire face à deux types de problèmes difficiles à résoudre simultanément. D'une part le spectre des types de données à traiter peut être très large : les méthodes de TDI peuvent être utilisées avec des images à deux dimensions « classiques » composées des pixels carrés organisés selon une grille régulière, contenant des valeurs binaires, en niveaux de gris, en couleurs, vectorielles ou même tensorielles ; la dimension peut également changer : 1D (signaux), 3D (volumes), 2D+t (séquences d'images), 3D+t (séquences de volumes) ; le domaine de l'image peut de plus être non régulier : il est possible de définir des images sur des structures mathématiques telles que des graphes, des complexes simpliciaux ou des cartes topologiques. La capacité à prendre en compte autant de structures de données différentes dépend de la *généricité* du cadre théorique choisi, et s'agissant de l'implémentation, des outils logiciels associés. On désigne par le terme *algorithme générique* un algorithme qui peut être appliqué à des entrées diverses, par opposition à un *algorithme spécifique*, qui ne peut être employé qu'avec une structure de données précise. La programmation générique (PG) est un paradigme de programmation qui s'intéresse aux question de généricité dans le logiciel [1]. L'une des motivations derrière la PG est la *réutilisabilité* du logiciel, c'est-à-dire la minimisation du coût lié à l'utilisation d'algorithmes existants avec de nouvelles structures de données (types d'entrées) et vice versa. Il existe plusieurs projets logiciels dédié au TDI qui reposent sur la PG : Vigra [4], ITK [3], Morph-M [2], GIL [1].

D'autre part, de nombreux problèmes et applications du TDI font intervenir des jeux de données important (par leur nombre ou leur taille) ou font appel à des techniques complexes nécessitant des calculs intensifs. Dans les deux cas, toute solution logicielle doit en pratique satisfaire des contraintes de *performance*, en particulier vis-à-vis de la vitesse d'exécution. Malheureusement, généricité et performance sont souvent opposés : un programme performant de TDI est le plus souvent dédié à certains types d'images, certaines méthodes ou certains domaines d'application spécifiques, et n'est de fait pas générique. À l'inverse, la plupart des *frameworks* génériques ne rivalisent pas avec des solutions dédiées en termes de performances.

Dans cet article, nous examinons les problèmes qui opposent

*Ce travail a été effectué dans le cadre du projet SCRIBO (http://www.scribo.ws/) du Groupe Thématique Logiciel Libre du pôle de compétitivité "Systematic Paris-Région". Ce projet est partiellement financé par le Gouvernement Français, ses agences de développement économique ainsi que les institutions de la région Île-de-France.

1. Dans cet article, par « générique » nous entendons « générique vis-à-vis de la programmation », et non « générique par rapport à l'approche utilisée pour résoudre un problème de traitement d'images ».

```
image dilation(const image& input) {
  image output(input.nrows(), input.ncols());
  for (unsigned r = 0; r < input.nrows(); ++r)
    for (unsigned c = 0; c < input.ncols(); ++c) {
      unsigned char sup = input(r,c);
      if (r != 0 && input(r-1,c) > sup)
        sup = input(r-1,c);
      if (r != input.nrows()-1 && input(r+1,c) > sup)
        sup = input(r+1,c);
      if (c != 0 && input(r,c-1) > sup)
        sup = input(r,c-1);
      if (c != input.ncols()-1 && input(r,c+1) > sup)
        sup = input(r,c+1);
      output(r, c) = sup;
    }
  return output;
}
```

ALGORITHME 1 – Implémentation de dilatation non générique

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Itérateur sur les sites du domaine de 'input'.
  mln_piter(I) p(input.domain());
  // Itérateur sur les voisins de 'p' selon 'win'.
  mln_qiter(W) q(win, p);
  for_all(p) {
    // Accumulateur calculant le supremum de 'win'.
    accu::supremum<mln_value(I)> sup;
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
  }
  return output;
}
```

ALGORITHME 2 – Implémentation de dilatation générique [5]

généricité et performance dans le TDI d'un point de vue algorithmique (les optimisations bas niveau ou matérielles ne sont pas abordées). Nous présentons tout d'abord comment la PG peut être appliquée au TDI et quels sont les bénéfices de cette approche (section 2). La section 3 aborde les questions de performance dans un contexte générique. Nous y étudions les raisons de l'opposition entre généricité et performance et nous proposons dans la section 4 une réponse à ce problème sous la forme d'un compromis. Cette idée développée plus encore dans la section 5, où non seulement les algorithmes, mais aussi les types de données, sont mis à contribution pour augmenter sensiblement les performances au détriment d'un peu de généricité. Les résultats chiffrés de nos expériences sont présentés et examinés dans la section 6.

2 Genericité en traitement d'images

Un framework logiciel générique fournit des algorithmes et des structures de données orthogonaux disposant chacun d'une *unique* implémentation, pouvant être utilisés ensembles pour réaliser toute combinaison valide. Cette approche permet d'éviter la redondance de code et favorise une réelle réutilisabilité des algorithmes sur une multitude de structures de données compatibles (par exemple, des types d'images), tout en évitant l'explosion combinatoire.

La généricité est basée sur le paradigme de programmation générique (PG). L'idée maîtresse de cette approche est de construire le logiciel à l'aide de *concepts*, qui représentent des entités abstraites du domaine (ici, le TDI) [6]. Un concept définit les relations entre l'entité qui lui est associée (par exemple un type d'images) et d'autre éléments (par exemple, un type de points ou un type de valeurs). Il comporte également l'ensemble minimal des services à fournir (par exemple, l'obtention d'une valeur associée à un point dans une image). On peut par la suite écrire des algorithmes génériques en utilisant les concepts à la place des types de données spécifiques. Comme ces algorithmes ne font pas état des détails liés aux types de données manipulés, ils constituent des implémentations génériques, ne dépendant d'aucun type d'entrées en particulier. Nous avons utilisé cette approche avec succès en morphologie mathématique [5] et dans d'autre domaines du TDI.

Nous pouvons illustrer cette idée à l'aide d'un algorithme élémentaire : une dilatation morphologique utilisant un élément structurant plat. L'algorithme 1 montre une implémentation simple de ce filtre. Cependant, celle-ci comporte des détails d'implémentation qui lient la routine à un type d'entrées spécifique (une image 2D sur une grille régulière contenant des valeurs compatibles avec le type `unsigned char`). De plus, l'élément structurant (4-connexe) ne peut être changé. Par conséquent, il nous est impossible d'utiliser cet algorithme pour traiter, par exemple, une image 3D en couleurs (RVB) avec un élément structurant 6-connexe.

Cependant, si un algorithme fait usage d'entités abstraites basées sur des concepts, il peut être employé avec des types d'entrées variés. C'est le cas de l'algorithme 2, qui propose une version générique de l'algorithme précédent, implémenté à l'aide d'une bibliothèque C++ générique, Milena, pièce centrale d'une plate-forme libre de TDI, Olena [7]. Les types d'entrées (resp. une image, et un élément structurant ou *fenêtre*) sont désormais des paramètres de l'algorithme (resp. `I` et `W`) ; les boucles sur les intervalles de lignes et colonnes sont remplacées par un unique objet `p` parcourant le domaine de l'image, appelé *itérateur sur sites* (ou *piter*) ; de même, les points de l'élément structurant (auparavant figé dans le code) centré en `p` ne sont plus mentionnés explicitement et sont remplacés par un second itérateur `q` (nommé *qiter*) parcourant la fenêtre `win` ; enfin, en lieu et place du calcul manuel d'une valeur maximale, un objet *accumulateur* est utilisé pour calculer le supremum des valeurs dans la fenêtre glissante.

3 Généricité et performance

Les chiffres de la table 1 font apparaître un surcoût à l'exécution pour l'implémentation générique (algorithme 2), environ dix fois plus lente que la version non générique (algorithme 1). Il ne s'agit pas d'une conséquence du paradigme de PG en soit. Ce surcoût est en fait dû au style hautement abstrait de l'algorithme 2, qui en retour rend la routine très polyvalente vis-à-vis du contexte d'utilisation. La version non générique est plus rapide que son homologue générique car elle tire parti de caracté-

ristiques connues de ses entrées. Par exemple, l'élément structurant est « intégré » dans la fonction (alors qu'il s'agit d'un objet représentant une entrée générique dans l'algorithme 2) : sa taille est constante et connue à la compilation. De tels détails d'implémentation sont des informations *statiques* que le compilateur peut utiliser pour optimiser le code produit. La raison qui empêche un code d'être générique apparaît donc comme une condition nécessaire à la génération de code efficace : les détails d'implémentation.

4 Optimisations génériques

En choisissant soigneusement la quantité d'information spécifique apparaissant dans un algorithme, il est possible de créer des variantes intermédiaires affichant de bonnes performances à l'exécution et préservant un grand nombre de traits génériques. Une technique permettant d'accélérer l'algorithme 2 consiste par exemple à ne pas employer d'itérateurs sur sites pour parcourir le domaine des images d'entrée et de sortie. Dans Milena, un itérateur sur site sait se convertir automatiquement en un site (point), c'est-à-dire une position dans le domaine d'une (ou plusieurs) image(s). Une telle information de position n'est pas liée à une image en particulier : dans le cas des images 2D régulières, un site `point2d(42, 51)` est compatible avec tout domaine 2D à coordonnées entières de la bibliothèque (ce qui inclut des espaces toriques, des sous-espaces non rectangulaires de $\mathbb{Z}^2$, etc.). C'est pour cette raison que l'itérateur `p` est utilisé pour désigner le même emplacement à la fois dans `input` et `output` dans l'algorithme 2.

L'utilisation d'une expression aussi générale se traduit habituellement par un coût à l'exécution : chaque utilisation de l'itérateur avec une image implique des calculs, car cet itérateur ne pointe pas directement sur les données. Cette souplesse n'est cependant pas toujours nécessaire, si les données à traiter présentent des propriétés particulières. Par exemple, une image dont les valeurs sont stockées dans un espace mémoire contigu et linéaire peut être parcourue en utilisant un pointeur. Ce dernier permet un accès direct aux valeurs de façon séquentielle, en utilisant leurs adresses mémoire au lieu de calculer ces emplacements à chaque accès. Dans Milena, nous encapsulons ces pointeurs dans des objets légers appelé *itérateurs sur pixels* ou *pixters* (un pixel désignant un couple (site, valeur) dans une image). Un pixter est lié à une seule image et ne peut être utilisé pour itérer sur une autre image. Les pixters peuvent également être employés pour parcourir des éléments structurants (fenêtres) spatialement invariants, à condition que le domaine sous-jacent de l'image soit régulier.

L'algorithme 3 est une réimplémentation de l'algorithme 2 où les itérateurs sur sites sont remplacés par des itérateurs sur pixels. Le code est similaire, à ceci près que les images `input` et `output` sont maintenant parcourues à l'aide de deux itérateurs différents (possédant chacun un pointeur sur les données de l'image correspondante). Cette implémentation de dilatation morphologique est moins générique que algorithme 2. Malgré

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Itérateur sur les pixels de 'input'.
  mln_pixter(const I) pi(input);
  // Itérateur sur les pixels de 'output'.
  mln_pixter(I) po(output);
  // Itérateur sur les pixels voisins de 'pi'.
  mln_qixter(const I, W) q(pi, win);
  for_all_2(pi, po) { // Itération synchrone de 'pi' & 'po'.
    accu::supremum<mln_value(I)> sup;
    for_all(q)
      sup.take(q.val());
    po.val() = sup.to_result();
  }
  return output;
}
```

ALGORITHME 3 – Dilatation optimisée, en partie générique

cela, elle est toujours utilisable avec une grande variété de types d'images, du moment que leurs données présentent une organisation régulière, ce qui inclut les images régulières de dimension quelconque dont les valeurs sont stockées sous forme d'un unique bloc de mémoire linéaire. De plus, cette implémentation est compatible avec n'importe quel élément structurant spatialement invariant (c'est-à-dire toute fenêtre constante). Cette variante reste par conséquent plus générique que l'algorithme 1. Du point de vue des performances, l'algorithme 3 est comparable à l'algorithme 1 (voir la table 1). Il s'agit donc d'une bonne alternative à la dilatation générique, où l'équilibre entre la performance et la généricité est déplacé vers la première.

L'approche présentée ici s'applique aussi à d'autres algorithmes de la littérature du TDI pour lesquels des implémentations optimisées ont été proposées. De telles optimisations sont en pratique le plus souvent compatibles avec une variété de types d'entrées ; leurs implémentations peuvent donc être considérées comme des *optimisations génériques* puisque qu'elles ne sont pas liées à un type spécifique.

5 Optimisations supplémentaires

Il est possible de développer l'approche proposée dans cet article afin d'améliorer les performances d'une optimisation générique. L'idée est de faire participer les structures de données à cet effort d'optimisation : au lieu d'intervenir uniquement sur les algorithmes, nous pouvons produire de nouvelles variantes optimisées en agissant également sur leurs entrées.

Par exemple, au lieu d'une fenêtre contenant un tableau dynamique de vecteurs (à savoir $\{(-1,0),(0,-1),(0,0),(0,+1),(+1,0)\}$ dans le cas d'un élément structurant 4-connexe spatialement invariant) – dont la taille est connue à l'exécution – nous pouvons implémenter et utiliser une fenêtre *statique*, composée d'un tableau comportant les mêmes informations, mais dont le contenu et la taille sont connus à la compilation. Les compilateurs modernes savent tirer parti de telles informations pour effectuer des optimisations efficaces (par exemple remplacer une boucle sur les éléments de la fenêtre par un code déroulé équivalent). Dans cet exemple en particulier, cette optimisation nécessite juste la création de deux types de données

TABLE 1 – Temps d'exécutions des algorithmes de dilatation

Implémentation	Temps (secondes)		
Image (pixels) :	512^2	1024^2	2048^2
Non générique (Alg. 1)	0.10	0.39	1.53
Non générique, avec pointeurs[2]	0.07	0.33	1.27
Générique (Alg. 2)	0.99	4.07	16.23
Optimisée en partie gén. (Alg. 3)	0.13	0.54	1.95
Alg. 3 avec une fenêtre statique	0.06	0.28	1.03

relativement simples (fenêtre et itérateur sur pixels statiques). Aucune nouvelle implémentation de l'algorithme de dilatation n'est nécessaire : il suffit d'utiliser l'algorithme 3 avec le nouveau type de fenêtre. Le code résultant est non seulement plus rapide que la version non générique de l'algorithme 1, mais peut aussi être plus rapide qu'une version optimisée manuellement (et par conséquent non générique) utilisant des pointeurs (voir la section suivante).

6 Résultats

La table 1 montre les temps d'exécutions de plusieurs implémentation de dilatation morphologique avec un élément structurant (fenêtre) 4-connexe, appliquée à des images de taille croissante (512×512, 1024×1024 et 2048×2048 pixels). Les temps correspondent à 10 invocations itératives. Les tests ont été effectués sur un PC sous Debian GNU/Linux comportant un processeur Intel Pentium 4 à 3,4 GHz et 2 Go de mémoire vive à 400 MHz, en utilisant la version 4.4.5 du compilateur `g++` (GCC), appelé avec le niveau d'optimisation '`-O3`'. En sus des implémentation présentées dans cet article, une version supplémentaire non générique utilisant des optimisations à base de pointeurs a été ajoutée à la suite de tests, afin de pousser plus encore la comparaison entre du code non générique – optimisé essentiellement « à la main » – et du code générique – optimisé essentiellement par le compilateur.

Le coût lié à l'abstraction dans l'implémentation la plus générique est important : il est environ dix fois plus long que l'algorithme 1. Le code particulièrement flexible de l'algorithme 2 est dépourvu de détail d'implémentation que le compilateur pourrait utiliser pour produire du code rapide. L'algorithme 3 propose un compromis entre généricité et performance : il est environ 30% plus lent que l'algorithme 1, mais il est suffisamment générique pour fonctionner avec de nombreux types d'images régulières (qui sont dans les faits les plus courants). Le cas de la dilatation avec une fenêtre statique est intéressant : la réutilisation d'un même code (algorithme 3) avec une entrée moins générique (une fenêtre statique représentant un élément structurant fixe spatialement invariant) génère un code deux fois plus rapide, au point de surpasser une implémentation optimisée manuellement à base de pointeurs. Ainsi, il est utile de disposer de plusieurs implémentations (ici les algorithmes 2 et 3) pour satisfaire des besoins de flexibilité et d'efficacité.

2. Cette implémentation n'est pas montrée ici pour des raisons de place.

7 Conclusion

Cet article propose une approche pour concilier généricité et performance dans le logiciel pour le TDI. Cette stratégie repose sur la notion d'optimisation générique, qui exprime une variante efficace d'un algorithme pour un sous-ensemble des types d'entrées valides.

L'ajout de versions moins génériques mais plus performantes d'algorithmes ne devrait pas affecter la motivation à concevoir un framework de TDI aussi générique que possible. Nous pensons que la version la plus générique d'un algorithme devrait toujours être écrite en premier, puis complétée par des implémentations plus rapides. En effet, disposer d'une version générique d'un algorithme signifie posséder au moins une implémentation fonctionnant avec tous les types d'entrées valides. De plus, les implémentations génériques sont généralement plus simples, plus courtes et plus rapides à écrire, si le framework utilisé fournit les entités nécessaires (itérateurs, fenêtres, etc.). Enfin, elles constituent une base saine pour l'écriture de variantes optimisées, de par la similarité de leurs structures.

Il est également possible d'automatiser la sélection de la version d'un algorithme en fonction du type des entrées utilisées. Ce mécanisme nécessite d'équiper les types de données de propriétés (*traits*) décrivant leurs capacités. Les techniques de métaprogrammation C++ permettent alors l'écriture d'un code de sélection statique basé sur ces propriétés.

La bibliothèque Milena, utilisée dans les exemples de cet article, est distribuée avec la plate-forme Olena. Ce projet logiciel libre, diffusé sous licence GNU GPL, est accessible à l'adresse `http://olena.lrde.epita.fr/Download`.

Références

[1] Adobe. Generic Image Library (GIL). `http://opensource.adobe.com/gil`.

[2] Centre de Morphologie Mathématique. Morph-M. `http://cmm.ensmp.fr/Morph-M/`.

[3] L. Ibáñez, W. Schroeder, L. Ng et J. Cates. *The ITK Software Guide*. Kitware, Inc., 2005.

[4] U. Köthe. Reusable software in computer vision. *In* B. Jähne, H. Haussecker et P. Geißler, éditeurs. *Handbook of Computer Vision and Applications*, volume 3. Academic Press, 1999.

[5] R. Levillain, Th. Géraud et L. Najman. Milena : Write generic morphological algorithms once, run on many kinds of images. *In* M. H. F. Wilkinson et J. B. T. M. Roerdink, éditeurs. *Proc. of the 9th International Symposium on Mathematical Morphology*, Groningen, The Netherlands, 2009.

[6] R. Levillain, Th. Géraud et L. Najman. Why and how to design a generic and efficient image processing framework : The case of the Milena library. *In Proc. of the International Conference on Image Processing*, Hong Kong, 2010.

[7] LRDE. The Olena image processing platform. `http://olena.lrde.epita.fr`.

D

PROJET OLENA, BIBLIOTHÈQUE MILENA

C'est peut-être un triangle, mais on ne voit pas le troisième côté que forment dans l'espace ces curieux oiseaux de passage.

Les chants de Maldoror (Livre 1), Lautréamont, 1968-70.

D.1 À PROPOS DU PROJET OLENA

Le projet de la plate-forme OLENA est décrit à l'URL `http://olena.lrde.epita.fr`. À partir de cette adresse, du matériel est disponible et, en particulier, la bibliothèque MILENA. Cette présente section n'a pas pour but de remplacer ce matériel ni, en particulier, de documenter la plate-forme et la bibliothèque. En revanche, vous trouverez ici quelques clefs permettant de mieux comprendre ce qu'a été et ce qu'est ce projet logiciel. Cette section devrait donc compléter un peu le corps de ce rapport.

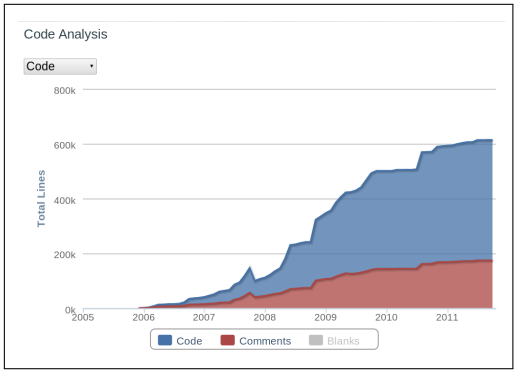
D.1.1 *Historique*

Quelques dates :

- 2000 : début du projet.
- Novembre 2001 à avril 2004 : évolution de la version 0.1 à 0.10
- Février 2007 : mise à jour pour se conformer aux compilateurs récents ; version 0.11.
- Décembre 2008 : version 1.0 béta.
- Juillet 2009 : sortie de la version 1.0.
- Septembre 2011 : version 2.0.

Jusqu'en 2004, les différentes versions ont reflété différents cycles de développement caractérisés par des paradigmes différents ; précisément, il s'agit de la programmation générique "classique", du paradigme "SCOOP [2] (voir aussi l'annexe B.2) et enfin de "SCOOP 2" [17] (voir l'annexe B.2). À partir de 2004, nous avons travaillé sur la simplification de ce dernier paradigme, présentée dans le corps de ce rapport. 2007 fut l'année de la ré-écriture de la bibliothèque avec ce nouveau paradigme ; la version sortie en 2008 ainsi que les versions postérieures l'utilisent.

Sur le graphique ci-dessous, on peut lire l'évolution du dernier cycle de développement et qui a abouti à la version 1.0 du projet :



Fourni par **ohloh** ; Cf. http://www.ohloh.net/p/olena.

Depuis 2010, l'effort a porté sur le module SCRIBO de la bibliothèque, dédié à la dématérialisation de documents scannés.

D.1.2 *Quelques informations en vrac*

Quelques chiffres

code complet d'OLENA †	440 000 lignes
code de la bibliothèque MILENA †	109 000 lignes
le cœur de MILENA †	25 000 lignes
code de test de MILENA †	20 000 lignes
commentaires dans le code d'OLENA	170 000 lignes
documentation technique	1300 pages

† Hors commentaires et lignes blanches.

Le cœur de la bibliothèque comprend les types d'images et les outils qui leur sont associés ; il ne comprend donc pas les algorithmes, ni les entités annexes comme les types de valeurs, les fonctions et accumulateurs, les fenêtres, etc. De plus, tous les méta-programmes utilitaires sont également regroupés à part. Finalement, le cœur de la bibliothèque est véritablement constitué de très peu de lignes de code, pour un énorme potentiel de fonctionnalités.

Logo

Le logo du projet représente Otto, la loutre mascotte du LRDE, déguisée en Lenna ; il a été dessiné par Alexis Angelidis (ancien étudiant au LRDE, maintenant directeur technique chez Pixar) et colorisé par Fabien Freling :

D.2 Á PROPOS DE LA BIBLIOTHÈQUE MILENA

Contenu de MILENA

La bibliothèque est décomposée en modules fonctionnels distincts. Chaque module apparaît comme un sous-répertoire, auquel correspond un sous-espace de nommage. On retrouve ici des catégories classiques du domaine du traitement d'images :

```
accu/           data/       graph/      morpho/         trace/
algebra/        debug/      histo/      norm/           trait/
                display/    io/         opt/            transform/
arith/          draw/       labeling/   pw/             upscaling/
binarization/   essential/   linear/     registration/   util/
border/         estim/      literal/    set/            value/
canvas/         extension/  logical/    subsampling/
clustering/     extract/    make/       tag/            win/
convert/        fun/        math/       test/           world/
core/           geom/       metal/      topo/
```

Les autres ?

Pour réaliser MILENA, nous avons regardé (à un moment ou à un autre, et plus ou moins en profondeur) les bibliothèques et/ou environnements suivants :

A. ADISL — AGG — AMILab
C. CamlImages — CCV — CImg — CVIPtools — CxImage
D. DGtal — DIPlib
E. ExactImage
F. Filters — Fulguro —
G. Gandalf — GENIAL — GIL — GIMP — gluas — GpuCV — GraphicsMagick
H. HIPS — Horus
I. IAC++ — IDL — IM — ImageJ — ImageMagick — ImaGene — ImaLab — ImLab — ImLib3D — Imview — Intel IPP (was IPL) — iNVT — IPL98 — IPLab — IPT — ITK — IUE
J. JIU
K. Khoros / VisiQuest
L. Leptonica — Libpipi LTI-Lib
M. Magick++ Mamba — Mathematica (with Digital Image Processing) — MATLAB (with Image Processing Toolbox) — MegaWave — MicroMorph — MITK (and 3DMED) — MorphoLib — Morph-M — mmach — mmorph (Matlab) — Multivac
N. Netpbm — NGI NVL++
O. OCRopus — Octave — OpenCV — OTB / Orfeo
P. Pandore — PIL — Pink PIPADE
Q. Qgar
S. Sage — ScilImage (from the Team behind the Horus project : ISIS, Amsterdam) — Scilab (Image Toolboxes : SIP, ...) — SLIP
T. TINA — Tivoli — topomap3d
V. Vigra — VIPS — Vista — VSIPL — VSIPL++ — VTK — VXL (includes vil)
X. XIP Xlim3D — XMV / XMegaWave
Y. Yayi

D.2.1 *Une comparaison*

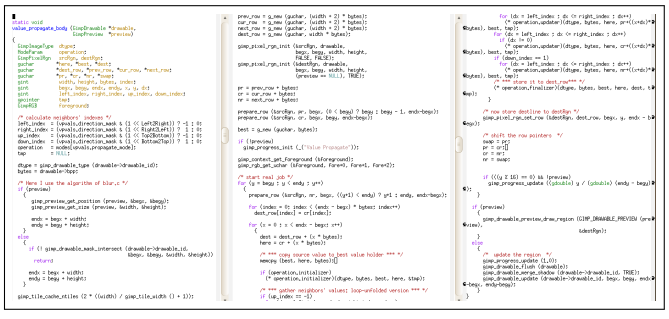
Afin de se faire une idée de la différence entre MILENA et plusieurs autres bibliothèques, nous avons extrait un même algorithme de celles-ci. Cet exemple est représentatif de l'expressivité de ces différents outils, à la fois en terme de concision d'écriture, mais aussi en terme de lisibilité du code. Les captures d'écran sont reproduites ci-dessous.

```
template <typename I, typename W>
mln_concrete(I) dilation(const I& ima, const W& win)
{
  mln_concrete(I) out;
  initialize(out, ima);

  mln_piter(I) p(ima.domain());
  mln_qiter(W) q(win, p);
  accu::sup<mln_value(I)> sup;

  for_all(p)
  {
    sup.init();
    for_all(q) if (ima.has(q))
      sup.take(ima(q));
    out(p) = sup;
  }

  return out;
}
```

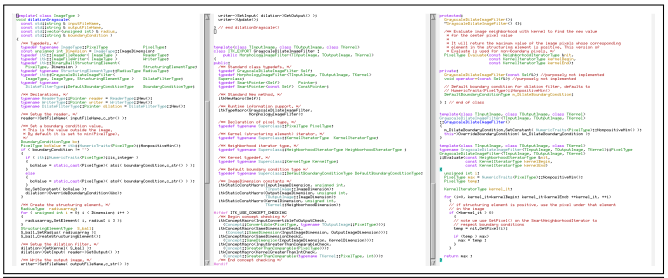
MILENA

```
CImg get_dilate(const unsigned int n=1)
{
  CImgStats stats(*this);
  const T tmax = stats.max!=0?(T)stats.max:(T)1;
  CImg_3x3x3(I,T);
  if (n==1)
  {
    CImg dest(*this);
    cimg_mapV(*this,k) cimg_map3x3x3(*this,x,y,z,k,I)
      if (!Iccc && (Incc || Ipcc || Icnc || Icpc || Iccn || Iccp)) dest(x,y,z,k) = tmax;
    return dest;
  }
  CImg img1(*this),img2(*this,false);
  CImg *src = &img1, *dest = &img2, *tmp = NULL;
  for (unsigned int iter=0; iter<n; iter++)
  {
    *dest = *src;
    cimg_mapV(*src,k) cimg_map3x3x3(*src,x,y,z,k,I)
      if (!Iccc && (Incc || Ipcc || Icnc || Icpc || Iccn || Iccp)) (*dest)(x,y,z,k) = tmax;
    tmp = src;
    src = dest;
    dest = tmp;
  }
  return *src;
}
```

CIMG



TheGimp (extrait)



```
template<class TInputImage, class TOutputImage, class TKernel>
typename GrayscaleDilateImageFilter<TInputImage, TOutputImage, TKernel>::PixelType
GrayscaleDilateImageFilter<TInputImage, TOutputImage, TKernel>
::Evaluate(const NeighborhoodIteratorType &nit,
           const KernelIteratorType kernelBegin,
           const KernelIteratorType kernelEnd)
{
  unsigned int i;
  PixelType max = NumericTraits<PixelType>::NonpositiveMin();
  PixelType temp;

  KernelIteratorType kernel_it;

  for (i=0, kernel_it=kernelBegin; kernel_it<kernelEnd; ++kernel_it, ++i)
    {
    // if structuring element is positive, use the pixel under that element
    // in the image
    if (*kernel_it > 0)
      {
      // note we use GetPixel() on the SmartNeighborhoodIterator to
      // respect boundary conditions
      temp = nit.GetPixel(i);

      if (temp > max)
        max = temp ;
      }
    }

  return max ;
}
```

ITK

```
Errc PDilatation( const Imc2duc &ims, Imc2duc &imd, int numSE, int halfsize )
{
  Point2d p,p1,p2;
  Imc2duc::ValueType maxX,maxY,maxZ;

  Img2duc *imse = MSE2D(numSE,halfsize);
  if (!imse) {
    std::cerr << "Error: Bad structuring element"<< std::endl;
    return FAILURE;
  }
  Point2d shift((*imse).Size()/2);

  for (p.y=0;p.y<ims.Height();p.y++)
    for (p.x=0;p.x<ims.Width();p.x++) {
      maxX=ims.X[p];
      maxY=ims.Y[p];
      maxZ=ims.Z[p];

      for (p1.y=0;p1.y<imse->Height();p1.y++)
        for (p1.x=0;p1.x<imse->Width();p1.x++) {
          p2=p+p1-shift;
          if (ims.Hold(p2) && (*imse)[p1] && Greater(ims.X[p2],ims.Y[p2],im
                                                    s.Z[p2],maxX,maxY,maxZ)){
            maxX = ims.X[p2];
            maxY = ims.Y[p2];
            maxZ = ims.Z[p2];
          }
        }
      imd.X[p] = maxX;
      imd.Y[p] = maxY;
      imd.Z[p] = maxZ;
    }

  delete imse;
  return SUCCESS;
}
```

Pandore

```
int32_t ldilat( struct xvimage *f, struct xvimage *m,
                int32_t xc, int32_t yc)
{
  register int32_t i, j, k, l;
  int32_t
    rs = rowsize(f), cs = colsize(f), N = rs * cs,
    rsm = rowsize(m), csm = colsize(m),
    x, y, t;
  uint8_t *M = UCHARDATA(m), *F = UCHARDATA(f), *H;
  int32_t sup;

  H = (uint8_t *)calloc(1,N*sizeof(char));
  for (x = 0; x < N; x++) H[x] = F[x];

  for (y = 0; y < cs; y++)
  for (x = 0; x < rs; x++)
  {
    sup = NDG_MIN;
    for (j = 0; j < csm; j += 1)
    for (i = 0; i < rsm; i += 1)
    {
      t = (int32_t)M[j * rsm + i];
      if (t)
      {
        l = y + j - yc;
        k = x + i - xc;
        if ((l >= 0) && (l < cs) && (k >= 0) && (k < rs) && ((int32_t)H[l * rs + k] + t > sup))
          sup = (int32_t)H[l * rs + k] + t;
      }
    }
    F[y * rs + x] = (uint8_t)min(sup, NDG_MAX);
  }

  free(H);
  return 1;
}
```

Pink

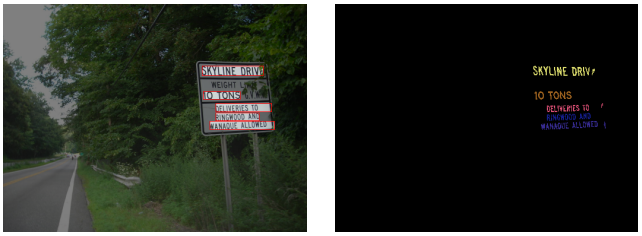
```
Volume* numDilate(Volume* volume, int connectivity)
{
  Volume* dilated;
  long int *oB, oFP, oPbL, oLbS;
  int ix, nx, iy, ny, iz, nz, i;
  U8BIT_t *ptVolume, *ptDilated, *pt, max;

  setBorderWidth ( volume, -1 );
  oB = offsetBox ( volume, connectivity );
  oFP = offsetFirstPoint ( volume );
  ptVolume = data_U8BIT ( volume ) + oFP;
  dilated = duplicateVolumeStructure ( volume, "dilated" );
  ptDilated = data_U8BIT ( dilated ) + oFP;

  setBorderLevel ( volume, 0 );
  getSize ( volume, &nx, &ny, &nz );
  oPbL = offsetPointBetweenLine ( volume );
  oLbS = offsetLineBetweenSlice ( volume );
  for ( iz = 0; iz < nz; iz++ )
    {
      for ( iy = 0; iy < ny; iy++ )
        {
          for ( ix = 0; ix < nx; ix++ )
            {
              pt = ptVolume++;
              max = *pt;
              for ( i = 0; i < connectivity; i++ )
                {
                  pt += oB[i];
                  if ( *pt > max )
                    max = *pt;
                }
              *ptDilated++ = max;
            }
          ptVolume += oPbL;
          ptDilated += oPbL;
        }
      ptVolume += oLbS;
      ptDilated += oLbS;
    }

  free ( oB );
  return ( dilated );
}
```

TIVOLI

Démos disponibles

MILENA est une bibliothèque sert à faire du traitement d'images de façon effective. À titre d'exemples, deux démonstrations de "vraies" problématiques de traitement d'images, réalisées avec MILENA, sont en ligne :

– l'extraction de texte dans les images naturelles



http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/TextInPics

– et la dématérialisation d'images de documents (ci-dessous, l'entrée à gauche et le PDF obtenu en sortie à droite)

http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/PageSnR

D.2.2 *Contributeurs*

La bibliothèque MILENA n'existerait pas sans contributeurs [1]. Je ne peux pas ne pas les citer donc voici un instantané de leur liste :

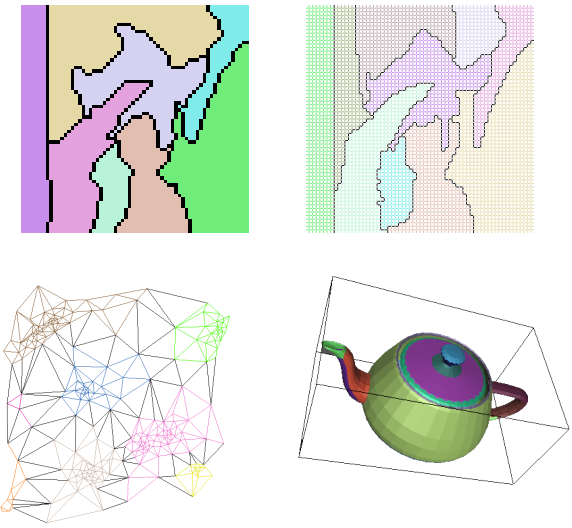
Akim Demaille — Alexandre Abraham — Alexandre Duret-Lutz — Anthony Pinagot — Astrid Wang — Benoît Sigoure — Caroline Vigouroux — Clément Faure — Christophe Berger — Christopher Chedeau — Damien Thivolle — David Lesage — Dimitri Papadopoulos-Orfanos — Edwin Carlinet — Emmanuel Turquin — Etienne Folio — Fabien Freling — Francis Maes — Frédéric Bour — Geoffroy Fouquier — Giovanni Palma — Guillaume Duhamel — Guillaume Lazzara — Heru Xue — Ignacy Gawedzki — Jean Chalard — Jean-Baptiste Mouret — Jean-Sébastien Mouret — Jérôme Darbon — Johan Seland — Julia Faurie — Kristoffer Gleditsch — Loïc Fosse — Ludovic Perrine — Matthieu Garrigues — Michaël Strauss — Nicolas Ballas — Nicolas Burrus — Nicolas Pouillard — Nicolas Tisserand — Nicolas Widynski — Niels Van Vliet — Pierre-Yves Strub — Quentin Hocquet — Quôc Peyrot — Raphaël Poss — Renaud François — Roland Levillain — Réda Dehak — Rémi Coupet — Simon Odou — Simon Nivault — Sylvain Berlemont — Thomas Kielbus — Thomas Moulard — Tristan Croiset — Ugo Jardonnet — Vincent Berruchon — Vivien Delmon — Yann Jacquelet — Yann Régis-Gianas — Yoann Fabre — Yongchao Xu.

Cette liste est tenue à jour à l'URL :

http://www.lrde.epita.fr/cgi-bin/twiki/view/Olena/Contributors

D.3 ENFIN DES IMAGES...

...peu communes, issues d'un même algorithme (code unique), puisque vous avez été sages !



1. Un grand merci à eux !

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES CODES SOURCE

BIBLIOGRAPHIE

Nota bene : cette section bibliographique vient compléter les références des articles portés en annexe, de la section B.1 à la section C.4 (pages 106 à 219).

BIBLIOGRAPHIE

[1] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 2000.

[2] Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss. A static c++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL), at ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, USA, October 2003.

[3] M. K. Carter, K. M. Crennell, E. Golton, R. Maybury, A. Bartlett, S. Hammarling, and R. Oldfield. The design and implementation of a portable image processing algorithms library in FORTRAN and C. In *Proceedings of the 3rd IEE International Conference on Image Processing and its Applications*, pages 516–520, 1989.

[4] James O. Coplien. Curiously recurring template patterns. In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

[5] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming : Methods, Tools, and Applications*. ACM Press / Addison-Wesley Publishing Co., New York, USA, 2000.

[6] Jérôme Darbon, Thierry Géraud, and Alexandre Duret-Lutz. Generic implementation of morphological image operators. In *Mathematical Morphology, Proceedings of the 6th International Symposium (ISMM)*, pages 175–184, Sydney, Australia, April 2002.

[7] Jérôme Darbon, Thierry Géraud, and Patrick Bellot. Generic algorithmic blocks dedicated to image processing. In *Proceedings of the Workshop of the European Conference on Object-Oriented Programming (ECOOP)*, Oslo, Norway, June 2004.

[8] M.R. Dobie and P.H. Lewis. Data structures for image processing in C. *Pattern Recognition Letters*, 12(8) :457–466, 1991.

[9] Marcos Cordeiro d'Ornellas and Rein van den Boomgaard. The state of art and future development of morphological software towards generic algorithms. *International Journal of Pattern Recognition and Artificial Intelligence*, 17(2) :231–256, 2003.

[10] Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Generic design patterns in C++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 189–202, San Antonio, TX, USA, January-February 2001. USENIX Association.

[11] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, 1998.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, New York, NY, 1995.

[13] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremia Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17 :145–205, March 2007.

[14] Thierry Géraud. *Segmentation des structures internes du cerveau en imagerie par résonance magnétique 3D*. PhD thesis, École Nationale Supérieure des Téélécommunications, Paris, June 1998. In French.

[15] Thierry Géraud. Ruminations on Tarjan's union-find algorithm and connected operators. In *Mathematical Morphology : 40 Years On, Proceedings of the 4th International Symposium (ISMM)*, Computational Imaging and Vision Series, pages 105–116. Springer–Verlag, 2005.

[16] Thierry Géraud and Alexandre Duret-Lutz. Generic programming redesign of patterns. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 283–294, Irsee, Germany, July 2000. UVK, Univ. Verlag, Konstanz.

[17] Thierry Géraud and Roland Levillain. Semantics-driven genericity : A sequel to the static C++ object-oriented programming paradigm. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL), at European Conference on Object-Oriented Programming (ECOOP)*, Paphos, Cyprus, July 2008.

[18] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic programming and high-performance libraries. *International Journal of Parallel Programming*, 33 :145–164, June 2005.

[19] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek. Algorithm specialization in generic programming : Challenges of constrained generics in C++. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 272–282, Ottawa, Ontario, Canada, June 2006. ACM Press.

[20] Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors. *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-41090-2.

[21] Dennis C. Koelma and Arnold W.M. Smeulders. An image processing library based on abstract image data-types in C++. In Springer-Verlag, editor, *Proceedings of the 8th International Conference on Image Analysis and Processing*, volume 974 of *Lecture Notes in Computer Science*, pages 96–102, 1995.

[22] C. Kohl and J. Mundy. The development of the Image Understanding Environment. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, pages 443–447, 1994.

[23] K. Konstantinides and J.L. Rasure. The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3) :243–252, 1994.

[24] U. Köthe. *Handbook of Computer Vision and Applications – 3 : Systems and Applications*, chapter Reusable Software in Computer Vision. Academic Press, 1998.

[25] Michael S. Landy, Yoav Cohen, and George Sperling. HIPS : A UNIX-based image processing system. *Computer Vision, Graphics, and Image Processing*, 25(3) :331–347, March 1984.

[26] Roland Levillain, Thierry Géraud, and Laurent Najman. Milena : Write generic morphological algorithms once, run on many kinds of images. In *Mathematical Morphology, Proceedings of the 6th International Symposium (ISMM)*, volume 5720 of *Computational Imaging and Vision Series*, pages 295–306. Springer–Verlag, 2009.

[27] Roland Levillain, Thierry Géraud, and Laurent Najman. Why and how to design a generic and efficient image processing framework : The case of the Milena library. In *Proc. of the IEEE International Conference on Image Processing (ICIP)*, Hong Kong, September 2010.

[28] Roland Levillain, Thierry Géraud, and Laurent Najman. Writing reusable digital geometry algorithms in a generic image processing framework. In *Proc. of the Workshop on Applications of Digital Geometry and Mathematical Morphology (WADGMM)*, Istanbul, Turkey, August 2010.

[29] Bertrand Meyer. Genericity versus inheritance. In *ACM SIGPLAN Notices, Proceedings of the Conference on Object Oriented Programming Systems Languages and Aplications (OOPSLA)*, volume 21, pages 391–405, 1986.

[30] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala : A comprehensive step-by-step guide*. Artima, 2 edition, 2010.

[31] D. Paulus, J. Hornegger, and H. Niemann. *Handbook of Computer Vision and Applications – 3 : Systems and Applications*, volume 3, chapter Software Engineering for Image Processing and Analysis, pages 77–103. Academic Press, 1998.

[32] Jim Piper and Denis Rutovitz. Data structures for image processing in a C language and UNIX environment. *Pattern Recognition Letters*, 3 :119–129, March 1985.

[33] G.X. Ritter, J.N. Wilson, and J.L. Davidson. Image Algebra : an overview. *Computer Vision, Graphics, and Image Processing*, 49(3) :297–331, 1990.

[34] Bjarne Stroustrup. Evolving a language in and for the real world : C++ 1991-2006. In *Proceedings of the 3rd ACM SIGPLAN conference on History of Programming Languages (HOPL)*, volume 4, pages 1–59. ACM, 2007.

[35] Steven L. Tanimotoa. Advances in software engineering and their relations to pattern recognition and image processing. *Pattern Recognition*, 15(3) :113–120, 1982.

[36] *TIVOLI : Traitement d'Images VOLumIques*. Télécom ParisTech, Paris, France, 2010. URL https://trac.telecom-paristech.fr/trac/project/tivoli/wiki.

[37] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates : The Complete Guide*. Addison-Wesley, 2003.

[38] Todd L. Veldhuizen. C++ templates are Turing complete. Technical report, Indiana University, 2003.