Verification formelle et optimisation de l'allocation de registres

Benoît Robillard

▶ **To cite this version:**

Benoît Robillard. Verification formelle et optimisation de l'allocation de registres. Architectures Matérielles [cs.AR]. Conservatoire national des arts et metiers - CNAM, 2010. Français. NNT : 2010CNAM0730 . tel-00562318

HAL Id: tel-00562318
https://theses.hal.science/tel-00562318

Submitted on 3 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

École Doctorale EDITE

LABORATOIRE CEDRIC, équipes CPR et OC

THÈSE présentée par :

Benoît ROBILLARD

soutenue le : 30 Novembre 2010

pour obtenir le grade de : Docteur du Conservatoire National des Arts et Métiers

Spécialité : Informatique

VERIFICATION FORMELLE ET OPTIMISATION DE L'ALLOCATION DE REGISTRES

THÈSE DIRIGÉE PAR :

BLAZY Sandrine	Professeur, Université Rennes 1
DUBOIS Catherine	Professeur, ENSIIE
SOUTIF Éric	Maître de conférences, CNAM

RAPPORTEURS :

DARTE Alain	Directeur de recherche, CNRS
DE WERRA Dominique	Professeur honoraire, EPFL
PAULIN-MOHRING Christine	Professeur, Université Paris-Sud

JURY :

LEROY Xavier	Directeur de recherche, INRIA, Président
DARTE Alain	Directeur de recherche, CNRS
DE WERRA Dominique	Professeur honoraire, EPFL
PAULIN-MOHRING Christine	Professeur, Université Paris-Sud
BLAZY Sandrine	Professeur, Université Rennes 1
DUBOIS Catherine	Professeur, ENSIIE
SOUTIF Éric	Maître de conférences, CNAM

Remerciements

Il est tant de personnes que je tiens à remercier, tant de personnes qui m'ont accompagné de mes premiers pas à aujourd'hui, quelques jours avant la soutenance. J'ai ici la chance de pouvoir les remercier et, même si je sais que mes mots sont inaptes à transcrire mes sentiments, j'espère que ces quelques lignes exprimeront au mieux ce que je dois à chacun d'eux.

Il est tout d'abord les personnes sans qui cette thèse n'aurait jamais vu le jour, mes enseignants, devenus ensuite mes collègues de recherche. Ils ont su me faire découvrir et aimer des domaines dont je n'imaginais l'existence et susciter en moi le goût de la recherche. Il m'est impossible de ne pas remercier Catherine Dubois, qui la première m'a fait découvrir la spécification formelle et la preuve de programme, ainsi que Sandrine Blazy, Julien Forest, Xavier Urbain et Pierre Courtieu qui m'ont transmis une part de leur savoir dans ce domaine. Je dois tout autant remercier Marie-Christine Costa, Alain Faye et Eric Soutif, pour m'avoir initié à cette merveille qu'est la théorie des graphes, Alain Billionnet, Sourour Elloumi, Christophe Picouleau et Frédéric Roupin, pour m'avoir fait découvrir les passionnantes et multiples facettes de la recherche opérationnelle.

Je me dois également de remercier ceux qui m'ont conduit à avoir l'envie de devenir chercheur. Le premier fut Jean-Marc Mota qui, avec Catherine Dubois, m'a encadré lors d'un stage portant sur la formalisation de la modélisation géométrique et qui, avant même que je ne songe à continuer dans la recherche, m'a fait faire mes premiers pas dans le monde de la recherche. Je tiens également à remercier ceux qui ont façonné mon sujet de stage de fin d'études, devenu par la suite mon sujet de thèse, spécialement pour moi. Il n'y a aucun doute que ma motivation et le plaisir que j'ai pris à travailler durant ces trois ans résultent de mon intérêt pour les problèmes que j'ai eu à traiter. Merci donc à Sandrine Blazy, Marie-Christine Costa, Catherine Dubois, Xavier Leroy et Eric Soutif d'avoir imaginé ce sujet de thèse que j'apprécie tant.

Ce cheminement me conduit naturellement à remercier plus particulièrement mes encadrants de thèse, Sandrine Blazy, Catherine Dubois et Eric Soutif pour tout le temps qu'ils m'ont accordé, pour m'avoir admirablement guidé et avoir accordé leur confiance aux idées que j'ai pu leur proposer. La gentillesse dont ils ont fait preuve avec moi est fantastique. Ces trois ans passés à leurs côtés fut un plaisir. Je ne pouvais simplement pas rêver meilleurs encadrants.

Je remercie chaleureusement Alain Darte, Dominique De Werra et Christine Paulin-Mohring de m'avoir fait l'honneur de lire et juger mes travaux. Ce manuscrit ne serait pas ce qu'il est si je n'avais pas bénéficié de leurs précieuses remarques.

Merci à Xavier Leroy, qui me fait le plaisir de présider le jury de cette thèse, et avec qui j'ai pu partager des discussions toujours très intéressantes, ainsi que de succulents mezze.

Je remercie tous mes collègues des équipes *Conception et Programmation Raisonnées* et *Optimisation Combinatoire* de laboratoire CEDRIC pour tous les bons moments passés ensemble et toutes les discussions enrichissantes que nous avons eues lors de réunions d'équipe ou simplement au jour le jour. Vous cotoyer aura été un plaisir de chaque jour. Je remercie tout autant tous mes enseignants et collègues de l'ENSIIE, trop nombreux pour être cités.

Il en est certains qui ont du me supporter plus souvent que les autres : je remercie Matthieu Carlier, Picnic (aussi connu sous le pseudonyme de Pierre-Nicolas), Starman (aussi connu sous le pseudonyme William) et Vincent de la bonne humeur quotidienne qui régnait dans notre bureau. J'ai passé de très bons moments avec vous, si nombreux qu'il est inconcevable de les énumérer. Nul doute que ce n'est que le début. Une pensée tout aussi particulière pour mes collègues doctorants que j'ai vu un peu moins souvent, mais avec qui j'ai passé d'excellents moments également : merci à Amélie, Hélène, Matthieu Trampont et Nicolas. Je n'oublies évidemment pas Zaynah, qui est venue terminer la rédaction de sa thèse chez nous et avec qui j'ai passé de très bons moments. J'ai appris beaucoup à ses côtés. Sa faculté à comprendre et analyser parfaitement les moindres sous-entendus d'une conversation et les comportements humains sont stupéfiants.

Il est enfin ceux qui ont bien voulu me supporter les soirs et week-ends, car il n'y a pas que le travail qui occupe ces créneaux : Anne-So, DraZel, Elsa, Gueu, Mad, Matthieu, Mimi, Nabal, Nemi, Olz, Osi, Orz, Picnic, Starman TiffTiff, Titange et Yass. Que de bons moments passés avec vous pour lesquels je suis heureux de pouvoir vous remercier maintenant.

Enfin, je n'ai pas de mots assez forts pour remercier les deux femmes de ma vie, ma mère, à qui je dois tout, et Céline, pour qui je ferais tout.

Résumé

La prise de conscience générale de l'importance de vérifier plus scrupuleusement les programmes a engendré une croissance considérable des efforts de vérification formelle de programme durant cette dernière décennie. Néanmoins, le code qu'exécute l'ordinateur, ou code exécutable, n'est pas le code écrit par le développeur, ou code source. La vérification formelle de compilateurs est donc un complément indispensable à la vérification de code source.

L'une des tâches les plus complexes de compilation est l'allocation de registres. C'est lors de celle-ci que le compilateur décide de la façon dont les variables du programme sont stockées en mémoire durant son exécution. La mémoire comporte deux types de conteneurs : les registres, zones d'accès rapide, présents en nombre limité, et la pile, de capacité supposée suffisament importante pour héberger toutes les variables d'un programme, mais à laquelle l'accès est bien plus lent. Le but de l'allocation de registres est de tirer au mieux parti de la rapidité des registres, car une allocation de registres de bonne qualité peut conduire à une amélioration significative du temps d'exécution du programme.

Le modèle le plus connu de l'allocation de registres repose sur la coloration de graphe d'interférence-affinité. Dans cette thèse, l'objectif est double : d'une part vérifier formellement certains algorithmes connus d'allocation de registres par coloration de graphe, et d'autre part définir de nouveaux algorithmes optimisants pour cette étape de compilation.

Nous montrons tout d'abord que l'assistant à la preuve Coq est adéquat à la formalisation d'algorithmes d'allocation de registres par coloration de graphes. Nous procédons ainsi à la vérification formelle en Coq d'un des algorithmes les plus classiques d'allocation de registres par coloration de graphes, l'Iterated Register Coalescing (IRC), et d'une généralisation de celui-ci permettant à un utilisateur peu familier du système Coq d'implanter facilement sa propre variante de cet algorithme au seul prix d'une éventuelle perte d'efficacité algorithmique. Ces formalisations nécessitent des réflexions autour de la formalisation des graphes d'interférence-affinités, de la traduction sous forme purement fonctionnelle d'algorithmes impératifs et de l'efficacité algorithmique, la terminaison et la correction de cette version fonctionnelle. Notre implantation formellement vérifiée de l'IRC a été intégrée à un prototype du compilateur CompCert.

Nous avons ensuite étudié deux représentations intermédiaires de programmes, dont la forme SSA, et exploité leurs propriétés pour proposer de nouvelles approches de résolution optimale de la fusion, l'une des optimisations opérées lors de l'allocation de registres dont l'impact est le plus fort sur la qualité du code compilé. Ces approches montrent que des critères de fusion tenant compte de paramètres globaux du graphe d'interférence-affinité,

tels que sa largeur d'arbre, ouvrent la voie vers de nouvelles méthodes de résolution potentiellement plus performantes.

Mots clés : allocation de registres, coloration de graphes, vérification formelle

Table des matières

Chapitre 1

Introduction

J'ai grandi dans une période de révolution technologique dont l'informatique fut la principale actrice. Enfant, j'écrivais des textes sur une vieille machine à écrire, précieux outil sur lequel je n'ose imaginer rédiger ma thèse aujourd'hui. Les ordinateurs n'étaient pas monnaie courante dans les ménages à cette époque. Chacun de mes passages hebdomadaires au centre commercial était l'occasion de jeter un regard envieux à ces merveilleuses machines. Je me surprenais souvent à rêver d'en posséder une. Puis, peu à peu, avoir un ordinateur est devenu envisageable, possible, commun. Aujourd'hui, l'informatique est partout. Tout ce qu'elle apporte, facilité, confort, rapidité, en ont fait une alliée précieuse de notre société, jusque dans les domaines les plus critiques, mettant en jeu vies humaines ou données confidentielles. L'avionique, le nucléaire, et la médecine en sont des exemples phares.

De l'importance des enjeux mis en cause est naturellement apparue la volonté de se convaincre du bien fondé de la confiance accordée aux applications informatiques. Ces préoccupations ont donné naissance à des méthodes de développement et de vérification : les *méthodes formelles*. Celles-ci reposent sur des théories et des notations mathématiques permettant à la fois de *spécifier formellement* le programme à vérifier et de *prouver* à l'aide de l'ordinateur que son implantation respecte bien toutes les propriétés décrites par sa spécification. On parle alors d'implantation *correcte* vis-à-vis de la spécification et de programme *vérifié formellement*.

Les méthodes formelles sont reconnues par des standards de référence, si bien que le septième et dernier niveau de confiance des Critères Communs [Gro06] n'est accordé qu'aux applications conçues grâce à elles. En avionique, la norme DO178B, et plus encore la future norme DO178C, font également la part belle aux méthodes formelles [RTC09]. La prise de conscience générale de l'importance de vérifier plus scrupuleusement les programmes et la maturité des outils voués à cette tâche ont engendré une croissance considérable des efforts de vérification formelle de programme durant cette dernière décennie.

Les programmes sont aujourd'hui principalement écrits dans des langages de haut niveau qui confèrent aux développeurs confort et portabilité : le développeur peut écrire son programme dans un langage donné, sans se préoccuper des caractéristiques matérielles de la machine sur laquelle le programme sera exécuté. Néanmoins, le code qu'exécute l'ordinateur, ou *code exécutable*, n'est pas le code écrit par le développeur, ou *code source*. La traduction du programme source en programme exécutable, nommée *compilation*, doit

en effet accomplir de nombreuses tâches telles que traduire les instructions du *langage source* (celui du développeur) par des instructions du *langage cible* (celui du processeur) ou définir la gestion des données en mémoire. Par conséquent, la compilation peut elle aussi introduire de nouveaux bogues, invalidant ainsi toute démarche de vérification réalisée sur le code source. La revue de code manuelle a longtemps été, et reste encore parfois, la parade pour se protéger de ces menaces : des programmeurs relisent le code assembleur afin de s'assurer de son adéquation avec le code source. Cette pratique possède un second intérêt majeur. Un seul compilateur est capable de compiler, via une chaîne de compilation commune vers différentes *architectures cibles*. La revue de code permet d'implanter au niveau assembleur certaines optimisations dépendantes de l'architecture cible qui ne sont pas prises en charge par le compilateur. Cependant, cette approche reste trop rudimentaire pour pouvoir assurer la sûreté du compilateur. En revanche, le programme qui accomplit la compilation, le *compilateur*, peut tout à fait être soumis à une vérification formelle, complétant ainsi la chaîne de vérification formelle du code source jusqu'au code exécutable. En ce sens, la vérification formelle de compilateurs apparaît comme un complément indispensable à la vérification de code source.

La compilation est un processus si complexe qu'elle est classiquement décomposée en une séquence de transformations, appelées *passes de compilation*, afin d'évoluer progressivement du langage source au langage cible. Cette décomposition en une succession de passes permet de voir un compilateur comme un enchaînement de mini-compilateurs qui réalisent chacun une passe. Vérifier formellement un compilateur peut alors se réduire à vérifier formellement individuellement ses passes : si toutes les passes préservent la sémantique du programme, alors, par transitivité, le compilateur la préserve également. Néanmoins, vérifier formellement les passes d'un compilateur réaliste demeure un problème délicat et titanesque, bien qu'identifié et étudié depuis plus de quarante ans. Il l'est d'autant plus que les compilateurs dits *optimisants* utilisés aujourd'hui comprennent de nombreuses passes d'optimisation, le plus souvent dépendantes de l'architecture cible, destinées à améliorer les performances du code produit. Ainsi, un compilateur tel que GCC intègre plus de cent passes pour transformer un programme source en programme exécutable.

Les critères d'optimisation d'un compilateur peuvent en outre varier d'un système à un autre. On distingue en particulier la *compilation statique* de la *compilation dynamique*. La compilation statique consiste à compiler un programme une fois pour toutes, puis l'exécuter autant de fois que voulu. C'est la forme de compilation la plus classique. De plus, c'est la forme qui peut atteindre les meilleurs résultats qualitatifs puisqu'aucune contrainte théorique de temps ou de capacité ne la limite. Ainsi, certains systèmes embarqués nécessiteux de posséder du code très optimisé peuvent utiliser des techniques poussées et coûteuses sans inconvénient autre que le temps pris par la compilation. C'est par exemple le cas de certaines applications utilisées dans l'avionique. La compilation dynamique consiste à compiler l'exécutable juste avant son utilisation. Cette forme de compilation, souvent appelée compilation à la volée (ou JIT-compilation) est intimement liée à l'usage de bytecode. Le bytecode est une représentation intermédiaire des programmes, non exécutable mais pouvant être interprétée par une machine virtuelle. Ne faire qu'interpréter le code permet de s'affranchir des particularités des différentes architectures cibles sur lesquelles le programme est susceptible d'être exécuté. Des langages comme PHP ou Python utilisent largement ce principe. Le bytecode peut également être compilé vers du code exécutable,

juste avant son exécution, en fonction de l'architecture cible et du programme à compiler : c'est la compilation à la volée. Les applications écrites en Java font généralement usage de ce procédé ; certaines applications embarquées également. Dans ce dernier cadre d'application, la compilation à la volée est soumise à des contraintes algorithmiques fortes, en particulier en termes d'énergie et d'espace, lesquels sont très souvent limités sur des composants embarqués.

L'une des passes de compilation contenant le plus d'optimisations et les plus cruciales et complexes est l'*allocation de registres*. C'est lors de celle-ci que le compilateur décide de la façon dont les variables d'un programme seront stockées en mémoire durant son exécution. La mémoire comporte deux types de conteneurs : les *registres*, zones d'accès rapide, présents en nombre limité, et la *pile*, de capacité supposée suffisamment importante pour héberger toutes les variables d'un programme, mais à laquelle l'accès est bien plus lent. Le but de l'allocation de registres est de tirer au mieux parti de la rapidité des registres, car une allocation de registres de bonne qualité peut conduire à une amélioration significative du temps d'exécution du programme compilé. Un compilateur désireux de produire du code efficace doit donc intégrer un algorithme d'allocation de registres performant, en particulier capable quand c'est possible d'affecter à un même registre plusieurs variables. En outre, ce problème est soumis à des contraintes, dont la plus essentielle de toutes : préserver la sémantique du programme. La conjonction de ces contraintes, des critères d'optimisation et de la combinatoire élevée des solutions possibles rend l'allocation de registres si complexe que cette passe est généralement résolue de façon heuristique. La voie de la résolution optimale de l'allocation de registres, notamment via la programmation linéaire, a également été ouverte et mérite d'être explorée au vu des performances toujours grandissantes des solveurs de programmation linéaire.

L'objectif de cette thèse est double : vérifier formellement des heuristiques et définir des algorithmes optimaux d'allocation de registres performants pour compilateurs statiques.

1.1 L'allocation de registres

1.1.1 L'allocation de registres en quelques mots

L'une des fonctionnalités qui rendent désormais indispensable le développement dans les langages de haut niveau est l'existence de variables en lieu et place des zones mémoires. Avant l'allocation de registres, le compilateur suppose disposer d'un nombre infini de registres appelés *pseudo-registres*. Usuellement, chacun de ces pseudo-registres contient une variable du programme source. L'allocation de registres doit *réaliser* ces pseudo-registres, *i.e.* leur associer un emplacement mémoire réel : un registre ou un emplacement de pile.

L'allocation de registres dépend grandement de l'architecture sur laquelle le code est destiné à être exécuté. Il est donc naturel qu'elle fasse partie des dernières passes effectuées par un compilateur avant la génération de code, d'autant plus que les passes de compilation effectuées avant elle peuvent pour la plupart créer du code, lequel doit évidemment être traité par l'allocation de registres. Le séquencement des passes de compilation liées à l'allocation de registres peut cependant diverger d'un compilateur à un autre, en fonction des optimisations intégrées par le compilateur et des architectures cibles. La passe de planifica-

tion des instructions (ou *instruction scheduling*), qui décide dans quel ordre les instructions sont effectuées par le processeur, peut par exemple être traitée en même temps que l'allocation de registres afin d'optimiser à la fois la gestion de la mémoire et les temps de latence du processeur, mais la précède généralement. Cette dissociation permet de ne pas considérer les problèmes liés à l'interdépendance de ces deux passes qu'il est déjà difficile de résoudre séparément. Nous considérons donc dans ce manuscrit une séquence d'instructions figée, c'est-à-dire que la planification des instructions a été établie préalablement à l'allocation de registres. De même, afin de bien circonscrire notre périmètre d'étude, nous considérons une allocation de registres non interprocédurale, *i.e.* où chaque programme n'est qu'une fonction.

Plus précisément, l'allocation de registres prend en entrée un programme écrit en langage de transfert de registres (où `RTL` pour *Register Transfer Language*). Les opérations de ce langage sont de bas niveau, proches de l'assembleur. Le résultat de l'allocation de registres est un programme écrit en pseudo-assembleur où les pseudo-registres du programme `RTL` ont été réalisés. La figure 1.1 présente cinq instructions de ce langage de sortie. Les trois instructions `load` , `store` et `move` , permettent de déplacer les variables d'un emplacement mémoire à un autre. Les instructions `li` et `add`, permettant respectivement d'écrire une valeur entière dans un registre et d'additionner des valeurs présentes en registres, servent à illustrer nos propos par quelques exemples.

Instruction	Effet
`load Rx @p`	charge dans le registre `Rx` la valeur présente à l'adresse de pile `@p`
`store Rx @p`	écrit à l'adresse de pile `@p` la valeur présente dans le registre `Rx`
`move Rx Ry`	écrit dans le registre `Rx` la valeur présente dans le registre `Ry`
`li Rx i`	écrit dans le registre `Rx` l'entier `i`
`add Rx Ry`	écrit dans le registre `Rx` la somme des valeurs présentes dans `Rx` et `Ry`

FIG. 1.1 – Quelques instructions du langage de sortie de l'allocation de registres.

L'exemple présenté figure 1.2 illustre un programme C et deux allocations de registres potentielles de celui-ci. La première allocation de registres se contente d'affecter à chaque variable un registre différent. Cette solution est en pratique insatisfaisante, puisque le nombre de variables stockées en registres ne peut jamais excéder le nombre de registres. Autrement dit, cette heuristique place des variables en pile dès lors que le nombre de registres de l'architecture sur laquelle le programme vient à être exécuté est plus petit que le nombre de variables. Sur des architectures ne disposant que de six ou huit registres, cette solution est inappropriée. Cependant, certains registres ne sont utilisés qu'une fois puis délaissés. Il est judicieux de réutiliser ceux-ci dès que possible afin de stocker d'autres variables, comme opéré dans la seconde allocation de registres.

Bien qu'une utilisation intelligente des registres permette de diminuer le nombre de registres nécessaire pour stocker toutes les variables d'un programme, il est parfois impossible de ne pas recourir à la pile. Par exemple, si l'on suppose disposer de deux registres seulement, l'allocation de registres du programme précédent requiert des instructions d'écriture en pile, `store` , et de lecture de la pile, `load` , comme illustré figure 1.3.

programme C	une allocation de registres possible pour le programme	une autre allocation de registres possible
`int a = 9 ;`	`li R1 9`	`li R1 9`
`int b = 5 ;`	`li R2 5`	`li R2 5`
`int c = a + b ;`	`move R3 R1`	`move R3 R1`
	`add R3 R2`	`add R3 R2`
`int d = b + c ;`	`move R4 R2`	`add R2 R3`
	`add R4 R3`	
`int e = c + d ;`	`move R5 R3`	`add R3 R2`
	`add R5 R4`	
`int res = a + e ;`	`move R6 R1`	`add R1 R3`
	`add R6 R5`	

FIG. 1.2 – Deux exemples d'allocation de registres. La première se contente de faire correspondre un registre à chaque variable. Ainsi, `a`, `b`, `c`, `d` et `e` sont respectivement affectées aux registres `R1`, `R2`, `R3`, `R4`, `R5` et `R6`. La seconde allocation de registres réutilise les registres `R1`, `R2` et `R3` et n'utilise que ces trois registres. Ainsi `R1` contient `a` puis `res`, `R2` contient `b` puis `d` et `R3` contient `c` puis `e`.

programme C	allocation de registres possible pour trois registres ou plus	allocation de registres possible pour deux registres
`int a = 9 ;`	`li R1 9`	`li R1 9`
`int b = 5 ;`	`li R2 5`	`li R2 5`
		`store R1 @a`
`int c = a + b ;`	`move R3 R1`	`add R1 R2`
	`add R3 R2`	
`int d = b + c ;`	`add R2 R3`	`add R2 R1`
`int e = c + d ;`	`add R3 R2`	`add R2 R1`
		`load R1 @a`
`int res = a + e ;`	`add R1 R3`	`add R1 R2`

FIG. 1.3 – Un programme et deux allocations de registres de celui-ci selon que le nombre de registres est supérieur ou inférieur à trois. Dans le premier cas, les registres suffisent à stocker les valeurs de toutes les variables, dans le second cas il est nécessaire de stocker la valeur d'une des variables en pile, ici la variable `a`, puis de la charger depuis la pile plus tard.

1.1.2 Quatre composantes de l'allocation de registres

L'on distingue communément quatre composantes de l'allocation de registres : le vidage (ou *spilling*), la fusion (ou *coalescing*), l'affectation des registres (ou *register assignment*) et l'insertion des instructions `load` et `store` (ou insertion de *spill code*).

1.1.2.1 Le vidage

Le vidage est la phase qui détermine quelles variables sont stockées en registres à tout moment de l'exécution du programme, ainsi que les instructions `load` et `store` devant être générées, sans pour autant décider dans quel registre sera stockée chaque variable. Le but est de générer le moins d'instructions `load` et `store` possible, eu égard au coût de celles-ci.

1.1.2.2 La fusion

La fusion se focalise sur les instructions `move` . Ces instructions jouent un rôle prédominant au cours de l'allocation de registres. Une instruction `move Rx Rx` est inutile et peut être supprimée du code source, réduisant ainsi le nombre d'instructions du code, et, par là même, le temps d'exécution de celui-ci. La fusion a pour but d'exploiter cette propriété pour supprimer un maximum d'instructions `move` en affectant le même emplacement mémoire aux deux opérandes de ces instructions. Supprimer des instructions `move` peut paraître anecdotique, car les instructions de type `x := y` (dont elles sont la traduction pseudo-assembleur directe) sont relativement rares dans le code source, mais ne l'est pas du tout puisque de nombreuses instructions `move` sont générées par les différentes passes de compilation, en particulier les optimisations. L'importance de la fusion se ressent donc nettement sur l'efficacité du code produit. La fusion de deux variables n'est cependant pas toujours profitable. Imposer que deux variables partagent un même registre peut en effet impliquer de vider en pile plus de variables, ce qui est en général plus néfaste pour le code produit. Jusqu'au début des années 2000, le vidage et la fusion étaient systématiquement traités simultanément ; la tendance actuelle sépare ces deux étapes afin de traiter plus finement chacune d'entre elles en raison de la faible perte qualitative imputée à cette séparation observée empiriquement et de la facilité d'intégration et de maintenance des algorithmes d'allocation de registres qui en découle.

1.1.2.3 L'affectation des registres

L'affectation des registres décide à quels emplacements, *i.e.* registres ou zones de pile, sont placées les variables *in fine*. Celle-ci s'appuie sur les décisions du vidage et de la fusion et est généralement opérée en même temps que la fusion.

1.1.2.4 L'insertion de spill code

L'insertion de spill code matérialise les décisions prises durant les phases de vidage, de fusion et d'affectation des registres. Celles-ci sont traduites en ajouts d'instructions `load` et `store` pour chaque accès à une variable vidée en pile. L'allocation de registres n'est donc pas seulement une opération de décision, mais également une opération de modification du code source qui doit à ce titre préserver la sémantique du programme. L'ajout de ces instructions peut d'ailleurs invalider l'allocation de registres précédemment calculée. En effet, la plupart des instructions du langage cible exigent de leurs opérandes d'être présentes en registres (voire dans des registres spécifiques). Une variable vidée en pile doit donc être chargée en registre avant son utilisation. Si au moment de ce chargement aucun

registre n'est disponible pour l'héberger alors l'allocation de registres est considérée comme invalide. Deux solutions existent pour résoudre ce problème.

La première est de procéder à une allocation de registres itérative. Les variables vidées en pile sont copiées dans des variables temporaires juste avant chacune de leurs utilisations. Ces variables temporaires sont marquées afin de ne jamais pouvoir être placées en mémoire, et l'allocation de registres recommence avec cette contrainte supplémentaire. Cette méthode, appelée *reconstruction* fonctionne plutôt bien en pratique, mais la façon exacte de gérer ces contraintes et des questions plus théoriques, telles que la terminaison de tels procédés, restent très peu détaillées dans la littérature.

La seconde option est de conserver deux registres servant de zone de transfert entre les registres et la pile. Les variables vidées en pile sont chargées dans ces registres, et uniquement dans ces registres, au moment de leurs utilisations. Cette approche est moins efficace que la première. En effet, la première méthode essaie de trouver à chaque fois que besoin est un registre disponible pour stocker temporairement une variable placée en pile (donc au plus deux simultanément, puisque les opérations du langage pseudo-assembleur de sortie sont unaires ou binaires) tandis que la deuxième monopolise deux registres pour le même usage. La deuxième est cependant plus simple à mettre en place et assure qu'un seul calcul d'allocation de registres est suffisant.

Un facteur important à prendre en compte est le nombre de registres présents sur le processeur. Deux types d'architectures sont alors à différencier. Les architectures CISC (ou *Complex Instruction Set Computer*) possèdent un large nombre d'instructions, parfois complexes, et peu de registres. L'architecture x86, par exemple, ne dispose que de huit registres. Sur de telles architectures, seule la première approche est viable. Les architectures RISC (ou *Reduced Instruction Set Computer*) possèdent quant à elles un nombre d'instructions réduit et un plus grand nombre de registres. Les architectures `PPC` ou `ARM` , par exemple, disposent respectivement de seize et trente-deux registres. Sur ces architectures, la seconde approche peut amplement suffire. Les travaux présentés dans cette thèse se concentrent sur les architectures RISC, même si la plupart des résultats peuvent être étendus à des architectures CISC.

1.1.3 Allocation de registres par coloration de graphe

Raisonner directement sur le programme `RTL` pour construire une allocation de registres est délicat en raison des nombreux facteurs à considérer. Il est donc naturel d'avoir recours à des modèles plus condensés et intuitifs. De toutes les modélisations existantes, celle reposant sur la coloration de graphes est la plus connue et la plus étudiée.

Le modèle d'allocation de registres par coloration de *graphe d'interférence* a été unanimement adopté pour la compilation statique depuis les articles fondateurs de Chaitin au début des années 80 [CAC+81, Cha82]. Ces articles, en plus de montrer comment modéliser l'allocation de registres via la coloration de *graphe d'interférence* et proposer un algorithme de résolution, prouvent que tout graphe peut être le graphe d'interférence d'un programme. Ceci implique en particulier que le problème d'allocation de registres est NP-difficile . Il n'existe donc pas d'algorithme polynomial permettant de trouver une solution optimale pour tout graphe, sauf si P = NP. Une résolution heuristique, comme celle proposée par

Chaitin, apparaissait donc comme la seule solution viable à l'époque. Cette preuve sera ensuite reprise en détail dans [BDGR06]. Les auteurs reviennent sur les hypothèses trop restrictives effectuées par Chaitin. En relâchant ces hypothèses, le problème d'allocation de registres reste NP-complet dans quasiment tous les cas. Cependant, les causes de complexité apparaissent plus clairement et, contrairement à ce que laisse entendre la preuve de Chaitin, trouver une quelconque coloration valide du graphe n'est pas nécessairement la source principale de cette complexité.

Le problème de coloration de graphe d'interférence permet facilement d'encoder les contraintes de l'allocation de registres. Les sommets du graphe sont les variables du programme, les couleurs sont les registres, et deux sommets sont liés par une arête, appelée *interférence*, si les variables qu'ils représentent ne peuvent être affectées à un même registre. Deux sommets liés par une interférence ne doivent donc pas être colorés de la même couleur. Ce modèle a par la suite évolué afin de modéliser également les instructions `move` présentes dans le programme. Celles-ci sont représentées par un second type d'arête, appelées *affinités*. Ces affinités peuvent de plus être pondérées de sorte à représenter par exemple le nombre d'instructions `move` liant les deux variables. Si les deux sommets liés par une affinité se voient attribuer la même couleur, alors toute instruction `move` les liant pourra être supprimée du code source. On dit dans ce cas que cette affinité est satisfaite. En outre, ce modèle permet de prendre en compte les éventuelles contraintes liées à l'architecture processeur forçant certaines variables à être présentes dans un registre donné. Il suffit pour cela de précolorer le sommet représentatif de la variable contrainte avec la couleur qui représente le registre dans lequel celle-ci doit être placée.

Le problème d'allocation de registres revient alors à chercher la K-coloration du graphe respectant la précoloration qui maximise la somme des poids des affinités satisfaites, où K désigne dans tout ce manuscrit le nombre de registres allouables. Interférences et affinités obéissent donc à des objectifs antagonistes qu'il faut concilier. Nous ne parlons alors plus de graphe d'interférence, mais de *graphe d'interférence-affinité*. La figure 2.5 présente un exemple de graphe d'interférence-affinité, ainsi qu'une allocation de registres valide pour trois registres.

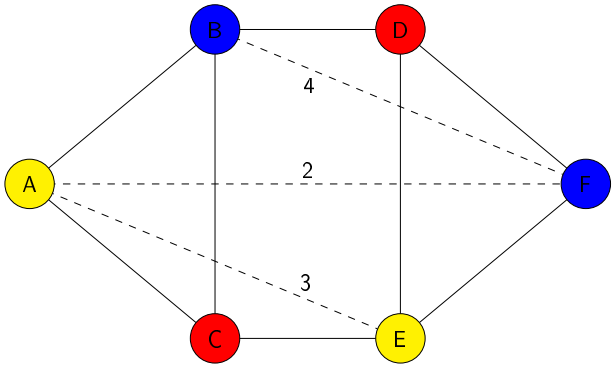


FIG. 1.4 – Un exemple de graphe d'interférence-affinité. Les arêtes en trait plein sont les interférences, celles en pointillés sont les affinités. La 3-coloration du graphe correspond à une allocation de registres satisfaisant les deux affinités de plus grands poids.

Si le nombre de registres disponibles est insuffisant, certaines variables ne peuvent être logées en registres ; les sommets correspondants ne sont pas colorés. Le vidage correspond donc à déterminer le plus grand ensemble de sommets possiblement colorable avec K couleurs. La fusion consiste quant à elle à fusionner autant que possible des sommets liés par une affinité afin que toute coloration leur affecte la même couleur. L'affectation des registres est déterminée par une coloration du graphe.

La simplicité et l'élégance de ce modèle ont engendré de nombreux travaux autour de celui-ci. Certains de ces travaux ont permis de pallier quelques imperfections du modèle, notamment via l'introduction du découpage des durées de vie (ou *live-range splitting*) [BDEO97, CS98], d'autres de définir de nouveaux algorithmes plus fins que celui proposé par Chaitin [CH84, BCT94, GA96, PM98, ONI+10], de gérer des contraintes matérielles spécifiques à certaines architectures [SE02, SRH04], ou encore de prendre en compte la structure du programme et les fragments de code les plus exécutés [CK91, CDE05, GGP03].

1.1.4 La cinquième composante : le découpage des durées de vie

Le modèle par coloration de graphe défini par Chaitin sur-approxime les contraintes réelles qui pèsent sur l'allocation de registres. Cette vérité éclate dès que l'on remarque que chaque variable est assignée à un et un seul emplacement mémoire durant toute l'exécution. Les méthodes ont donc évolué afin de supprimer cette lacune, notamment grâce au découpage des durées de vie [BDEO97, CS98]. Cette technique consiste à copier une variable v en une variable v' afin de les considérer ensuite comme deux variables différentes plutôt qu'une seule. Il s'agit donc d'une modification du programme et non du procédé de construction du graphe d'interférence-affinité. Plus la stratégie de découpage est agressive, *i.e.* plus elle introduit de copies, et plus la sur-approximation des contraintes est fine [CS98, AG01]. Il est d'ailleurs possible de complètement contrecarrer le défaut du modèle en ajoutant des copies de toutes les variables après chaque instruction, ce qui revient à autoriser chaque variable à changer d'emplacement après chaque instruction. La figure 1.5 illustre un exemple de découpage des durées de vie dans un programme et l'impact de celui-ci sur le graphe d'interférence-affinité. Le découpage permet dans ce cas de n'utiliser que deux registres au lieu de trois pour héberger les variables du programme.

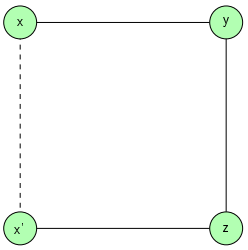
L'utilisation du découpage est profitable à l'allocation de registres mais possède deux faiblesses. La première réside dans le fait qu'ajouter des copies de variables rend encore plus difficile l'étape de fusion. La seconde est que choisir comment découper les durées de vie pour profiter des bienfaits du découpage est encore une question sans réponse définitive, bien que diverses stratégies ont déjà été proposées.

1.1.5 Allocation de registres et forme SSA

Le concept de découpage souligne l'impact fondamental de la représentation du programme sur la qualité du modèle graphique d'allocation de registres. La représentation intermédiaire sous forme SSA est celle qui a été le plus couronnée de succès pour l'allocation de registres.

Un programme est en forme SSA (ou *Single State Assignment Form*) [RWZ88, AWZ88,

```
x := 3;
y := x + x;
x := x + y;
z := x*x;
y := x - 3;
x := y*z;
```




```
x := 3;
y := x + x;
x := x + y;
x' := x;
z := x'*x';
y := x' - 3;
x' := y*z;
```

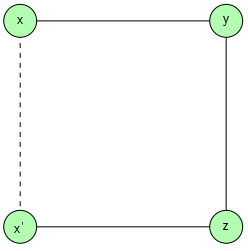


FIG. 1.5 – Un programme et son graphe d'interférence-affinité avant et après découpage de la durée de vie de la variable `x`. Avant découpage, le modèle impose de disposer de trois emplacements pour les variables du programme. Si seulement deux registres sont disponibles, l'une des trois variables est vidée en pile. Après découpage, il est tout à fait possible de stocker les trois variables dans seulement deux registres, au prix d'une instruction `move` supplémentaire.

CFR+91, App98b] si chaque variable est définie textuellement (ou encore, statiquement) une seule fois. Ceci n'interdit cependant pas qu'une variable soit dynamiquement définie plus d'une fois, comme ce peut être le cas dans le corps d'une boucle. Une telle forme de programme permet à des variables qui partagent le même nom mais ne contiennent jamais les mêmes valeurs d'être clairement identifiées comme des entités distinctes.

Nombre de compilateurs optimisants intègrent depuis quelques années une représentation SSA de leur programme, laquelle est propice aux analyses et aux optimisations classiques, comme la propagation de constantes par exemple. Cette forme a donc fait l'objet d'un intérêt particulier ces dernières années et s'est révélée avoir un impact conséquent sur l'allocation de registres. Le premier avantage par rapport à la forme originale du programme est le gain de précision. En effet, si $\{v_1, \ldots, v_n\}$ sont des variables du programme sous forme SSA qui partageaient un même nom dans le programme original, alors la forme SSA permet à ces variables d'être affectées à différents emplacements mémoire, contrairement à la forme originale. Plus important encore, le graphe d'interférence d'un programme sous forme SSA appartient à la famille des *graphes triangulés* [HGG06, HG06, Bou08, Hac07, BVI07], ce qui rend possible la coloration des graphes d'interférence-affinité de tels programme

avec un nombre minimal de couleurs en temps linéaire [Wes00]. Il devient donc possible de colorer le graphe d'interférence-affinité avec K couleurs si une K-coloration existe. Le vidage demeure néanmoins NP-difficile , mais cette découverte a permis de concevoir des heuristiques dédiées aux programmes sous forme SSA plus efficaces pour le vidage [HGG06, HG06, Hac07] et a mené la recherche en allocation de registres à se concentrer sur l'étape de fusion [And03, GH07, BDR08, BVI07].

1.1.6 Allocation de registres et découpage extrême

Une autre forme de programme intéressante pour l'allocation de registres est la *forme élémentaire*. Celle-ci consiste à réaliser un découpage des durées de vie après chaque instruction, permettant ainsi de décider de l'emplacement de chaque variable à chaque instant. Cette démarche a pour la première fois été introduite par Appel et George au sein de leur processus d'optimisation du vidage et de la fusion par programmation linéaire [AG01]. Les auteurs proposent de résoudre le vidage de façon optimale via un programme linéaire et de résoudre ensuite la fusion. Leur approche pour le vidage est instructionnelle, au sens où ils considèrent quelles instructions `load` et `store` doivent être générées avant chaque instruction du programme. Ce processus fonctionne bien pour le vidage mais se heurte à la combinatoire du problème de fusion : l'introduction d'un emplacement mémoire et d'une copie par variable et par instruction rend le problème gigantesque. Leurs résultats sont ainsi peu satisfaisants pour la fusion par programmation linéaire et cette dernière est finalement traitée par une heuristique de coloration de graphe. Cependant les graphes qu'ils doivent considérer sont immenses, en raison de la stratégie de découpage extrême que les auteurs ont adoptée.

Ce sont ces travaux qui permettent à Appel et George de déduire que la séparation des phases de vidage et de fusion, alors une nouveauté, ne déprécie que très peu la qualité globale de l'allocation de registres bien qu'une perte de qualité théorique existe, puisque la fusion dépend d'un vidage fixé. Ainsi, résoudre de façon optimale la fusion dans les graphes issus de leur méthode de vidage permettrait de construire des allocations de registres d'excellente qualité, en raison de la précision du modèle. C'est en ce sens qu'ils créent l'Optimal Coalescing Challenge [AG00], un jeu de 474 instances de fusion très particulières dont le nombre de sommets varie entre quelques dizaines et près de trente mille, issus de l'intégration de leur algorithme dans le compilateur SML. Les premiers résultats optimaux pour ces instances furent produits par Grund et Hack en 2007 [GH07], via la programmation linéaire. Ils résolvent ainsi 471 des 474 instances, non sans mal pour certaines d'entre elles, et concluent qu'à l'instar des résultats obtenus sur la forme SSA, exploiter les propriétés des instances de l'OCC semble la voie la plus adaptée pour les résoudre plus efficacement.

1.1.7 Trois problèmes, trois solutions

L'impact de la forme du programme sur la résolution de l'allocation suggère deux idées qui sont à la genèse de nos approches de résolution de l'allocation de registres par coloration de graphe. La première est que la méthode de résolution employée doit être orientée par la forme du programme et ses propriétés, puisque la précision du modèle dépend grandement de la forme du programme. Aussi, il est primordial que l'effort de résolution soit en accord

avec la représentation du programme : résoudre un problème grossièrement modélisé par des méthodes exponentielles exactes telles que la programmation linéaire est un gâchis ; résoudre grossièrement un modèle très précis en est un autre. Nous n'aborderons donc dans ce manuscrit non pas un problème, celui de l'allocation de registres, mais trois : un pour les programmes quelconques, un pour les programmes sous forme SSA et un pour les programmes sous forme élémentaire. L'étude de ces deux dernières formes est guidée par la seconde idée, laquelle soutient que les affinités doivent être considérées plus globalement qu'elles ne le sont actuellement dans la littérature pour arriver à des solutions de meilleure qualité, comme l'ont été les interférences lors des avancées liées à la forme SSA.

1.2 Vérification formelle et allocation de registres

1.2.1 Quelques méthodes formelles

Les méthodes formelles regroupent diverses techniques permettant d'assurer des propriétés sur les programmes ayant pour point commun de reposer sur des théories mathématiques.

Le *model-checking* [Cla08, QS08] consiste à abstraire le système afin d'en modéliser et vérifier toutes les exécutions possibles. C'est une technique énumérative dont la principale limitation est la taille du modèle. Un compromis doit être trouvé entre la qualité de l'abstraction et son nombre d'états possibles. Plus l'abstraction est fine et plus il y a d'états, ce qui rend difficile la vérification exhaustive des exécutions ; moins l'abstraction est fine et plus il y a de chances que celle-ci introduise des comportements qui ne vérifient pas la propriété désirée.

L'*analyse statique* [CC77] abstrait également le système mais cette fois afin de calculer une sur-approximation des exécutions possibles par calcul symbolique. Il est par exemple simple de vérifier grâce à l'analyse statique si un programme est susceptible de réaliser un dépassement de tableau, *i.e.* de demander la valeur d'une case qui n'existe pas dans un tableau, en calculant une sur-approximation des indices de tableau consultés.

La *vérification formelle* apporte des démonstrations mathématiques, automatiques ou assistées par l'utilisateur, du respect de l'implantation vis-à-vis de la *spécification*. Par exemple, pour un algorithme de tri de liste, la vérification formelle consiste à écrire l'algorithme de tri et à montrer que la liste retournée est une permutation triée de la liste paramètre. Cette méthode confère un niveau de confiance inégalé mais est parfois lourde à mettre en place. Les difficultés que l'on peut rencontrer lors de la vérification formelle d'algorithmes se situent sur trois plans : la spécification, qui doit être fidèle au système décrit et peut donc se révéler complexe, l'implantation, qui peut soulever des problèmes d'efficacité algorithmique ou de terminaison, et la preuve, qui doit lier spécification et implantation. Ce tryptique spécification, implantation, preuve, doit être conçu comme une seule entité, en fonction des objectifs que l'on se fixe. Désirer une grande efficacité de l'implantation induit en général une preuve complexe. Vouloir simplifier la preuve conduit à perdre l'efficacité algorithmique. Un compromis entre la qualité algorithmique du programme et la difficulté de sa preuve est donc à définir avant toute démarche de vérification formelle.

1.2.2 Le système Coq

Dans cette thèse, nous ne parlerons que de vérification formelle au sein du système Coq [Coq, BC04]. Pour un exposé plus large sur les différents assistants à la preuve, tels que Isabelle/HOL [NPW02], Mizar [Muz93], ACL2 [KMM00] ou Twelf [PS99a], et une comparaison de ceux-ci, le lecteur pourra se référer à [Wie06].

Le système Coq est un assistant à la preuve interactif qui permet de décrire à la fois des programmes et des propriétés mathématiques. Cette double fonction est en fait le reflet de l'isomorphisme de Curry-Howard qui établit le lien existant entre logique intuitionniste et calcul. Ce système peut être utilisé à deux fins, souvent confondues mais pourtant très différentes : soit en tant qu'environnement de preuve de théorème, soit en tant qu'environnement de développement et de vérification formelle. C'est cette dernière utilisation que nous faisons du système Coq .

Nous allons maintenant définir les aspects de ce système les plus utilisés dans la suite de ce manuscrit. Afin de ne pas alourdir le manuscrit, nous nous contentons ici de définitions informelles et intuitives. Une description plus complète des principaux rouages de ce système est fournie en annexe. Nous invitons les lecteurs qui ne sont pas familiers avec ce système à se référer à cette description si le besoin s'en fait ressentir.

Le système Coq dispose d'un langage fonctionnel, fondé sur le calcul des constructions inductives [CH88, PM93], permettant de définir de façon inductive à la fois des structures de données et des propriétés. D'un point de vue calculatoire, le langage dont dispose le système Coq est très proche du langage Ocaml . Dans un but de vulgarisation, nous pourrions même décrire un développement Coq comme le produit d'un développement Ocaml et de propriétés portant sur ce développement. Il est ainsi possible de définir des données, des propriétés, des fonctions de calcul sur ces données ou ces propriétés, ou encore d'architecturer des développements grâce au système de modules de Coq , lequel permet aussi bien de définir des interfaces abstraites que des implantations concrètes. Ces modules contiennent eux aussi des données et des propriétés.

La cohabitation des données et des propriétés au sein d'un même formalisme permet de définir des structures de données complexes comprenant à la fois des données et des propriétés portant sur ces données. Ces données sont dites de type dépendant. Par exemple, le type des listes ne contenant que des entiers pairs est un type dépendant, contenant une liste et la propriété de parité des éléments de la liste, car le type de l'un de ses éléments constitutifs, ici la preuve, dépend de l'un des autres, ici la liste. Pour construire une liste de ce type, il faut définir la liste, puis prouver qu'elle ne contient effectivement que des entiers pairs. Raisonner sur de telles structures est donc plus naturel, car elles modélisent plus précisément les données que l'on souhaite implanter, mais également plus difficile.

Les définitions inductives des données permettent de raisonner sur ces données ou ces propriétés par induction, grâce à des principes d'induction automatiquement générés au moment de leurs définitions. Pour reprendre l'exemple des listes d'entiers, prouver par induction qu'une propriété `P` est vraie pour toute liste revient à prouver que la propriété est vraie pour la liste vide, et que pour toute liste `l` qui vérifie `P`, la liste `a :: l` résultant de l'ajout d'un élément `a` à `l` vérifie `P`. Ces preuves sont réalisées via des scripts de commandes appelées tactiques [Del00], ce qui simplifie l'implantation des preuves et leur confère une

lisibilité accrue.

L'une des autres forces du système Coq est un mécanisme, appelé extraction [PM89, Let04], permettant de générer du code certifié à partir des développements Coq . L'extraction produit, à partir d'un programme écrit en Coq , un programme équivalent. Les preuves ne sont pas extraites, car elles font partie du monde logique, et non du monde calculatoire. Cette extraction peut être effectuée vers différents langages, mais nous utiliserons ce mécanisme seulement afin de générer le code Ocaml de nos implantations.

1.2.3 Validation *a posteriori*

La validation *a posteriori* [Sam75, PSS98, Nec00] est une méthode de vérification formelle consistant à vérifier chacune des exécutions d'un programme. Cette vérification est confiée à un autre programme, le *validateur*. Cette méthode pallie le principal désagrément que peut rencontrer un développeur lors de la vérification formelle d'un programme : rester bloqué sur un problème trop difficile pour être prouvé formellement en temps raisonnable. En revanche, un déni du validateur implique un arrêt de l'exécution du programme au sein duquel l'algorithme est intégré. La validation *a posteriori* est particulièrement adaptée à deux situations fréquemment rencontrées.

La première situation est celle dans laquelle l'effort à produire pour valider le programme est largement moindre que celui à produire pour le prouver. Dans ce cas, une économie non négligeable peut être effectuée grâce à la validation *a posteriori*. Il ne faut cependant pas céder de façon abusive à l'attrait de cette simplicité, sans quoi il devient tentant de n'effectuer que de la validation et de multiplier ainsi les sources d'échec potentiel du programme.

La seconde est le recours à un morceau de programme externe. La plupart du temps, les algorithmes validés sont écrits par le même développeur dans des langages de programmation très expressifs afin de s'accorder davantage de confort. Dans d'autres cas, il s'agit de l'utilisation de code écrit par un tiers. Un solveur de programmation linéaire est, par exemple, un outil trop gros pour être redéveloppé, mais auquel il est tentant de faire appel. La seule méthode de vérification formelle envisageable est alors la validation *a posteriori*.

Ses forces ont permis à la validation *a posteriori* de s'imposer ces dernières années comme un outil précieux de vérification. Pour résumer, la validation simplifie le travail de vérification formelle au détriment de trois autres points. Tout d'abord, le validateur peut détecter des faux positifs, des programmes qu'il refuse à tort de valider. Autrement dit, il n'est pas nécessairement complet. Ensuite, le programme qui calcule la solution peut contenir un bogue ce qui entraîne un travail supplémentaire de débogage le cas échéant. Enfin, le validateur doit être exécuté à chaque fois que l'algorithme non prouvé est appelé, ce qui peut avoir un coût en terme de temps d'exécution, alors que la vérification formelle apporte un tampon définitif. En outre, il est parfois aussi difficile de prouver un validateur que de prouver correcte l'implantation de l'algorithme. En contrepartie, la validation permet d'aborder des algorithmes qu'il serait très ambitieux de prouver formellement. Les travaux de Tristan [Tri09] portant sur la validation d'optimisations poussées de compilation comme la planification des instructions en sont des exemples. Les validateurs décrits dans ces travaux sont eux-mêmes des programmes compliqués, si bien que ceux-ci ont été

soumis à une vérification formelle. Ainsi, l'exécution d'un programme n'est validée que si le résultat qu'il renvoie est prouvé correcte.

En conclusion, le choix de la méthode la plus appropriée est donc laissé à l'appréciation du développeur en fonction du niveau de sûreté désiré et de l'effort estimé pour y accéder. C'est pourquoi durant cette thèse nous avons appliqué la preuve de programme à certains algorithmes et préconisé la validation *a posteriori* pour d'autres.

1.2.4 Comparaison vérification/validation par l'exemple

Afin d'illustrer comment fonctionne chacune de ces approches, prenons comme exemple le problème de vérification formelle d'un algorithme de division euclidienne.

La vérification formelle consiste dans ce cas à écrire une fonction *divide* et à prouver que celle-ci respecte la spécification attendue, *i.e.* la véracité des théorèmes suivants :

$$\forall (a, b, q, r) \in \mathbb{Z}^4, b \neq 0 \wedge divide(a, b) = (q, r) \Rightarrow a = b \cdot q + r \wedge |r| < |b|$$

où $|n|$ est la valeur absolue de l'entier relatif n.

La validation consiste quant à elle à écrire une fonction *check_division* qui prend en paramètre un quadruplet d'entiers relatifs (a, b, c, d) et à prouver les théorèmes suivants :

$$\forall (a, b, q, r) \in \mathbb{Z}^4, b \neq 0 \wedge check\_division(a, b, q, r) = true \Rightarrow a = b \cdot q + r \wedge |r| < |b|$$

Dans le cadre de la preuve de programme, le problème consiste à écrire l'algorithme et prouver sa correction. Ce n'est pas difficile car l'algorithme n'est pas compliqué, mais recourir à la validation *a posteriori* est encore plus simple : le validateur ne fait que réaliser une multiplication, une addition et deux comparaisons, l'une pour l'égalité $a = b \cdot q + r$ et l'autre pour la relation $|r| < |b|$.

1.2.5 Le projet CompCert

La vérification formelle de compilateurs consiste à établir un théorème de préservation sémantique. Plusieurs théorèmes sont envisageables. Dans CompCert , le théorème est :

Pour tout programme source, si le programme source a une sémantique bien définie et si la compilation ne signale pas d'erreur, alors le programme cible produit a le même comportement que le programme source [Ler06].

Notons que le théorème de préservation sémantique insiste sur l'absence d'erreur signalée par le compilateur. En effet, une compilation de fichier qui échoue est fâcheuse mais incomparablement moins dommageable qu'une erreur de compilation non diagnostiquée. C'est ce dernier type de comportement qui doit être banni et qui est donc naturellement visé par le théorème. Autrement dit, le théorème de préservation sémantique ne garantit

rien pour les faux positifs. Bien entendu, il faut autant que possible que le compilateur ne refuse de compiler un programme que si ce programme n'est pas syntaxiquement correct.

Différentes méthodes de vérification formelle ont été appliquées à des compilateurs de langages plus ou moins simples, voire des langages jouets. La bibliographie [Dav03] regroupe la plupart des travaux effectués dans ce domaine avant 2003.

Le projet CompCert [Com, Ler09, BDL06, Ler06], au sein duquel prend place ma thèse, consiste à spécifier, développer et vérifier formellement un compilateur réaliste du langage C vers les architectures `PPC` et `ARM` dans le système Coq . Ce langage et ces architectures ont été choisis car le compilateur a pour but d'être utilisé dans les systèmes embarqués critiques qui reposent souvent sur des architectures `PPC` ou `ARM` . Autour de cette épine dorsale peuvent se greffer des extensions, permettant de prendre de nouveaux langages d'entrée comme cela a été fait pour mini-ML [Dar09], ou est en cours pour Concurrent Cminor [Hob08]. Le code Ocaml du compilateur est généré automatiquement à partir des 60 000 lignes de développement Coq grâce au mécanisme d'extraction. Ce compilateur est aujourd'hui opérationnel et incorpore la plupart des optimisations classiques de compilation, ce qui lui permet de produire du code exécutable dont l'efficacité est proche de celle du compilateur GCC au premier niveau d'optimisation. Le schéma général de ce compilateur est présenté figure 1.6.

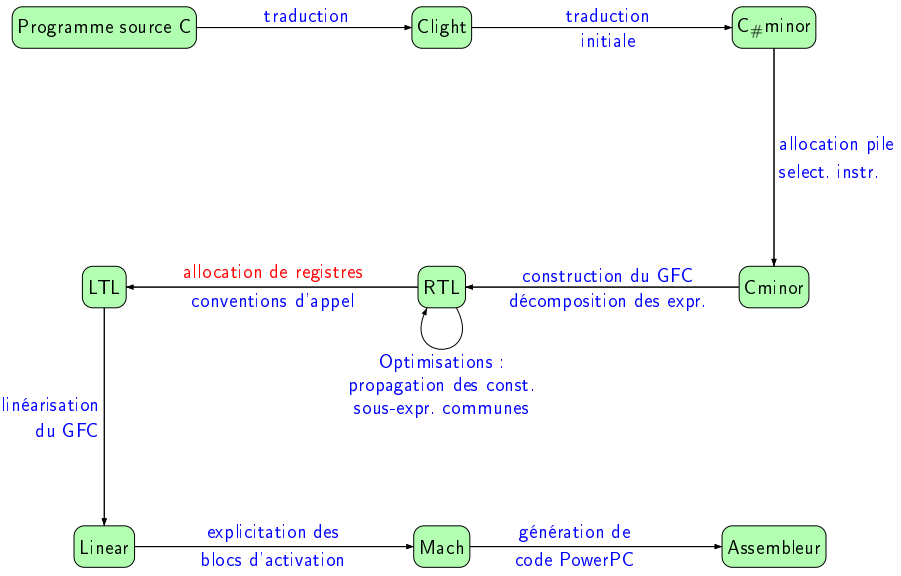


FIG. 1.6 – Processus de compilation de CompCert.

1.2.6 Vérification formelle d'algorithmes d'allocation de registres

L'allocation de registres du compilateur CompCert repose sur la coloration de graphe. Celle-ci est opérée par une heuristique classique et avancée d'allocation de registres, l'*Iterated Register Coalescing* (IRC) [GA96], dont une implantation impérative écrite en Ocaml est vérifiée par validation *a posteriori*. Cette heuristique est en effet particulièrement adaptée au nombre important de registres dont sont munies les architectures RISC, telles que les processeurs pour `PPC` et `ARM` (plus de seize), et aux courtes fonctions des systèmes embarqués critiques [GA96]. En revanche, cette heuristique est peu adaptée pour les architectures de type CISC, lesquelles disposent donc de peu de registres (huit ou moins), et que le découpage des durées de vie est poussé. En effet, l'heuristique a tendance à vider trop de variables en pile et ne pas satisfaire assez d'affinités.

L'allocation de registres est l'une des rares passes de CompCert qui sont seulement validées. L'immense majorité des autres passes sont en effet prouvées correctes directement au sein du système Coq . En outre, la validation ne concerne pas toute l'allocation de registres, mais seulement la coloration du graphe d'interférence-affinité. La figure 1.7 explicite ce qui est prouvé et ce qui est validé au sein de l'allocation de registres.

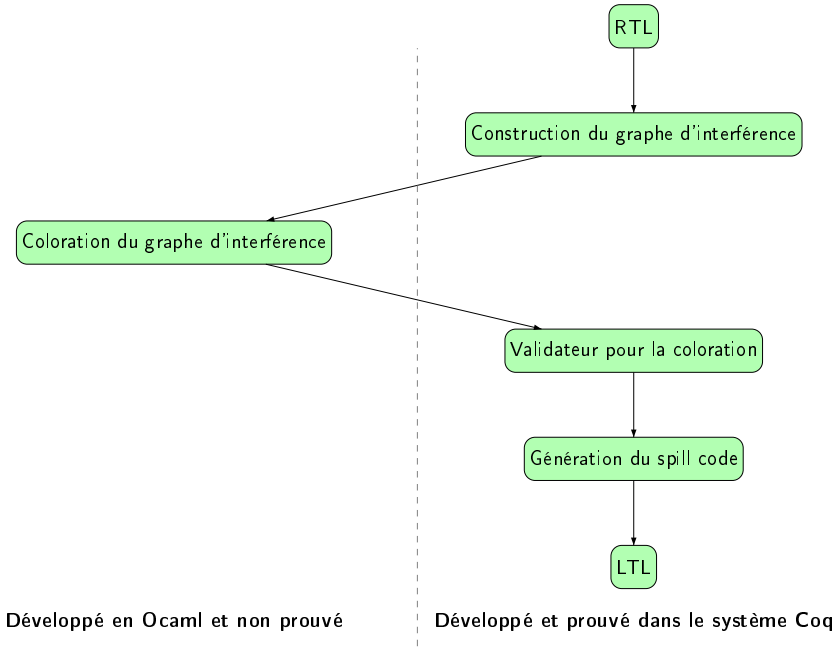


FIG. 1.7 – Schéma de l'allocation de registres de CompCert.

Le recours à la validation pour la coloration du graphe permet de minimiser les raisonnements nécessaires sur les graphes d'interférence-affinité. La formalisation des graphes d'interférence-affinité de CompCert est donc beaucoup plus simple que celle proposée dans ce manuscrit. Ceux-ci sont définis uniquement à partir d'ensembles d'interférences et d'affinités. Chaque fonction possède un seul graphe d'interférence-affinité modélisant à

la fois l'allocation de registres pour les variables entières et les variables flottantes. Après construction de ce graphe, la main est donnée à l'implantation impérative Ocaml de l'IRC. Le résultat de l'exécution de cette heuristique est soumis au validateur. Celui-ci contrôle deux propriétés :

1. la coloration est valide, *i.e.* associe bien aux extrémités de chaque interférence des emplacements mémoire différents ;
2. les emplacements mémoire assignés aux extrémités de toutes les arêtes du graphe sont légaux, *i.e.* sont des registres allouables ou des zones de pile allouables.

Si la validation du résultat échoue, la compilation échoue. Sinon, l'allocation de registres est construite. Pour ce faire, deux registres sont destinés à servir de temporaires pour l'insertion du spill code. Ceci implique d'une part qu'il n'est nécessaire de n'appeler qu'une fois l'IRC, et d'autre part que la preuve de correction de la génération du spill code est indépendante de la vérification de l'algorithme de coloration de graphe. Autrement dit, CompCert possède un algorithme de génération de spill code qui est prouvé correct si une coloration (validée) du graphe lui est passée en paramètre.

Le travail présenté dans cette thèse utilise l'algorithme d'insertion de spill code de CompCert afin de se concentrer uniquement sur la vérification formelle d'algorithmes de coloration de graphe d'interférence-affinité, puisque la correction de l'allocation de registres en découle. Prouver correct un algorithme d'allocation de registres revient donc à prouver le théorème suivant.

Pour tout graphe d'interférence-affinité G, si la précoloration des sommets de G est une K-coloration partielle de G, alors l'algorithme renvoie une K-coloration partielle de G qui ne modifie pas les couleurs attribuées aux sommets précolorés.

Cette année, Rideau et Leroy [RL10] ont proposé un validateur pour l'allocation de registres complète de CompCert , c'est-à-dire jusqu'à l'insertion du spill code. Leur validateur opère directement sur les programmes avant et après l'allocation de registres et vérifie une certaine correspondance entre ceux-ci permettant d'établir la préservation de la sémantique au cours de l'allocation de registres.

Hormis l'allocateur de registres de CompCert , peu de travaux se sont intéressés à la correction de l'allocation de registres.

Le premier travail qui nous est connu propose un système de types dédié à l'allocation de registres, ainsi qu'un algorithme d'allocation de registres correct par construction [Oho04], mais les instructions considérées sont celles d'un petit langage, et cette démarche semble difficilement applicable pour un compilateur réaliste tel que CompCert .

Une autre approche consiste à utiliser l'analyse statique pour définir une analyse de flot de données capable de vérifier statiquement l'allocation de registres [HCS06]. Cette méthode s'appuie sur des analyses de dépendances des données avant et après allocation de registres et permet ainsi de vérifier la validité des emplacements mémoire alloués.

Enfin, il existe un langage dédié à l'allocation de registres, ainsi qu'un système de types [NPP07]. La sûreté du typage de ce système de types a récemment été prouvée en

Twelf. Le but de ce travail est de fournir un cadre général permettant de comparer différentes stratégies d'allocation de registres. Néanmoins ce travail n'a, à notre connaissance, pas donné suite à la vérification formelle d'algorithmes efficaces d'allocation de registres.

1.2.7 Vérification formelle d'algorithmes de théorie des graphes

Les problèmes les plus proches de celui de la vérification formelle d'algorithmes d'allocation de registres par coloration de graphe sont finalement plutôt à chercher du côté des formalisations de notions de théorie des graphes. Toutefois, malgré leur usage plus que fréquent en informatique et la maturité de leur théorie, la vérification formelle de programmes ou théorèmes reposant sur les graphes n'a fait l'objet que de très peu de travaux. Seule une infime partie de la théorie des graphes a été décrite au sein d'assistants à la preuve et soumise à la vérification.

Quelques trop rares travaux sont dévolus à la spécification des concepts de base de théorie des graphes. En 1994, Chou formalise en HOL quelques notions simples et usuelles de théorie des graphes [Cho94] telles que les graphes non orientés, les chaînes ou les arbres. Ces travaux s'ensuivent de la formalisation des graphes planaires [YNHT95] et d'algorithmes de recherche en profondeur et en largeur [YTH+98], toujours en HOL. En 2001, Duprat spécifie les mêmes notions que Chou pour les graphes orientés et non orientés. Cette formalisation permet de raisonner sur les graphes, mais pas de calculer sur les graphes, puisque ceux-ci sont implantés comme des relations inductives. Mizar est probablement l'assistant à la preuve au sein duquel les travaux sur les graphes sont les plus aboutis. En plus des concepts déjà évoqués, la bibliothèque de graphes de cet assistant à la preuve inclut des formalisations plus complexes, telles que celle des graphes triangulés.

Les travaux précités se focalisent principalement sur la formalisation des graphes et non sur l'algorithmique de graphes. D'autres recherches portent naturellement sur cette algorithmique et les problèmes polynomiaux de théorie des graphes. Le problème le plus étudié est le plus que classique problème de plus court chemin. En 1990, Fleury prouve les algorithmes de Dijsktra et de Floyd [Fle90]. Par la suite, Parent reprend ces travaux lorsqu'elle conçoit la première version de la construction `Program` [Par95]. En 1998, Filliâtre et Paulin prouvent l'algorithme de Floyd dans le système Coq grâce à l'ancêtre de Why [Fil], une tactique appelée `Correctness`. Leur implantation suit un style impératif ; les graphes sont représentés par des matrices. Un autre algorithme pour ce même problème, l'algorithme de Dijkstra, est formalisé et prouvé correct à la fois dans Mizar [Che03] et dans ACL2 [MZ05]. Une fois de plus, Mizar devance ses concurrents avec la formalisation d'autres algorithmes tels que l'algorithme de Ford-Fulkerson de recherche de flot maximal, de l'algorithme LexBFS de reconnaissance des graphes triangulés ou l'algorithme de Prim de recherche d'arbre couvrant minimum. Ce dernier algorithme est également prouvé correct en B [JRA03].

Enfin, les assistants à la preuve ont été utilisés pour décrire des preuves de théorèmes importants et complexes de théorie des graphes, en particulier autour de la coloration et du fameux théorème des quatre couleurs. En 1879, Kempe [Kem79] prouve le théorème des quatre couleurs en utilisant un algorithme proche de celui défini par Chaitin un siècle plus tard. Hélas, son raisonnement est erroné ; il faudra onze ans aux mathématiciens pour déceler cette faille. Près d'un siècle plus tard, en 1976, Appel et Haken utilisent un

ordinateur pour fournir la première preuve de ce théorème [AH76]. Ce travail marque la première utilisation d'une machine pour clore un problème ouvert. Cependant, des parties majeures de la preuve restent uniquement écrites à la main.

Plus récemment, les problèmes de coloration de graphe planaire ont été formalisés dans des assistants à la preuve modernes, comme Isabelle/HOL ou Coq . Bauer et Nipkow formalisent les graphes planaires et prouvent le théorème des cinq couleurs [BN02], dont la preuve est éminemment plus simple que celle du théorème des quatre couleurs. En effet, la preuve erronée du théorème des quatre couleurs de Kempe est de fait une preuve valide du théorème des cinq couleurs. Gonthier et Werner proposent quant à eux la première preuve entièrement mécanisée du théorème des quatre couleurs grâce à une formalisation des hypercartes, une généralisation des graphes [Gon05, Gon08]. Leur preuve de près de 60 0000 lignes inclut des algorithmes implantés dans le système Coq sur lesquels ils raisonnent. Cependant, ceux-ci utilisent l'assistant Coq comme un prouveur de théorème et non comme un environnement de développement, dans le sens où leur seul but est de prouver un théorème.

1.3 Contributions

Le modèle d'allocation de registres le plus largement répandu pour la compilation statique, et auquel est largement consacrée cette thèse, repose sur la coloration du graphe d'interférence-affinité. Nous abordons en réalité trois problèmes d'allocation de registres très différents : le problème d'allocation de registres d'un programme quelconque, d'un programme sous forme SSA et d'un programme sous forme élémentaire.

Le chapitre 2 décrit un aperçu de l'état de l'art en allocation de registres par coloration de graphe. Les trois chapitres suivants abordent la vérification formelle de l'allocation de registres par coloration de graphe de programmes quelconques. Les deux derniers chapitres se concentrent sur le problème de fusion pour les programmes sous forme SSA et sous forme élémentaire. Plus précisément :

Le chapitre 3 présente la formalisation des graphes d'interférence-affinité utilisée dans les chapitres 4 et 5 pour implanter et vérifier formellement les algorithmes de coloration. Cette spécification est présentée via une interface, laquelle a également été implantée en Coq , prouvant ainsi sa cohérence. Le tout procure une bibliothèque correcte par construction des graphes d'interférence-affinité.

Le chapitre 4 est dévolu à la vérification formelle en Coq de l'IRC et à son intégration au sein d'un prototype du compilateur CompCert . Nous proposons une implantation fonctionnelle de référence de cet algorithme, initialement publié avec quelques erreurs, qui pourra être utilisée pour réaliser des allocations de registres de bonne qualité sur des graphes d'interférence-affinité modélisés comme décrit par Chaitin [CAC+81]. En effet, le caractère NP-difficile du problème de coloration et le manque de précision des contraintes encodées par le graphe d'interférence-affinité suggèrent une résolution heuristique, plutôt qu'une résolution exacte et chronophage.

Le chapitre 5 dépeint la formalisation en Coq d'un algorithme abstrait d'allocation de registres généralisant divers algorithmes de référence, comme l'algorithme de Chaitin ou

l'IRC. Cet algorithme abstrait, inspiré des similitudes existant entre l'expérience décrite dans le chapitre 4 et mes travaux de master, peut simplement être instancié par un utilisateur peu familier du système Coq pour créer des algorithmes d'allocation de registres par coloration de graphe et prouver facilement à la fois leur terminaison et leur correction. C'est également un moyen de comparer facilement tous ces algorithmes au sein d'un même environnement.

Le chapitre 6 décrit quant à lui des démarches d'optimisation plus poussées pour la fusion dans les graphes triangulés et fait apparaître l'importance d'un facteur global du graphe d'interférence-affinité jusque là inexploité, la *largeur d'arbre*. Nous prouvons en particulier que si cette largeur d'arbre est bornée par le nombre de registres moins un, alors il est optimal de séparer les phases de fusion et d'affectation des registres, ce qui n'est pas vrai dans le cas général. Nous en déduisons une extension pour les programmes sous forme SSA permettant de résoudre la fusion de façon optimale par programmation linéaire. Nous montrons que l'affectation des registres est ensuite équivalente à un problème de coloration classique de graphe triangulé et peut donc être résolue de façon optimale en temps polynomial par un algorithme classique comme la coloration gourmande. L'algorithme, ses fondements et les structures de données qu'il manipule étant très complexes, nous préconisons la validation *a posteriori* pour vérifier formellement l'algorithme.

Enfin, le chapitre 7 s'intéresse à la plus précise des modélisations par coloration de graphe, celle reposant sur le découpage extrême. Cette stratégie de découpage engendre aussi bien des résultats de meilleure qualité que des problèmes plus difficiles, en particulier pour la fusion puisque le nombre de copies ajoutées est gigantesque. Nous présentons une nouvelle approche optimale permettant de résoudre plus efficacement la fusion dans ce contexte et donnant des indices pour déterminer une stratégie de découpage des durées de vie idéale, *i.e.* n'introduisant quasi aucune copie inutile. Comme pour l'algorithme du chapitre précédent, la vérification formelle ne peut raisonnablement être assurée que par validation.

Chapitre 2

Fondements de l'allocation de registres

L'allocation de registres est une passe de compilation où l'optimisation tient un rôle primordial. Ce chapitre décrit plus précisément quels sont les problèmes d'optimisation soulevés durant l'allocation de registres et leur incidence sur le programme compilé. Nous y dépeignons tous les tenants de notre étude ainsi que l'état de l'art actuel en allocation de registres par coloration de graphe.

2.1 Les contraintes de l'allocation de registres

2.1.1 Structure des programmes RTL

L'allocation de registres prend comme paramètre un programme `RTL` . Dans ce langage, un programme est représenté sous forme de *graphe de flot de contrôle*.

Définition 1 (Graphe de flot de contrôle)**.** *Le graphe de flot de contrôle d'un programme est le graphe orienté dont les sommets sont les instructions du programme, et chaque arc est une transition possible entre deux instructions.*

A chaque instruction est associé un point, appelé *point de programme* qui précède immédiatement l'instruction. Ces points représentent les endroits du programme où il est possible de raisonner sur les valeurs des variables ; grossièrement, partout sauf durant les instructions.

Ainsi, la figure 2.1 représente le graphe de flot de contrôle d'une fonction de calcul de la suite de Fibonacci. Pour plus de lisibilité, la syntaxe présentée est celle du langage C et non celle d'un pseudo-assembleur.

Classiquement, un graphe de flot de contrôle repose sur trois structures particulières : les branchements, les jonctions, et les blocs de base.

Définition 2 (Branchement)**.** *Un branchement est un sommet du graphe de flot de contrôle possédant au moins deux successeurs.*

```
        int fib (int n){
(p1)    if (n == 0)
(p2)     return 0;
        else {
(p3)     int acc1 = 0;
(p4)     int acc2 = 1;
(p5)     int i = 0;
(p6)     int tmp = 0;
(p7)     while (i < n){
(p8)      tmp = acc1 + acc2;
(p9)      acc1 = acc2;
(p10)     acc2 = tmp;
(p11)     i = i + 1;
(p12)     }
         }
(p13)    return acc2;
        }
```

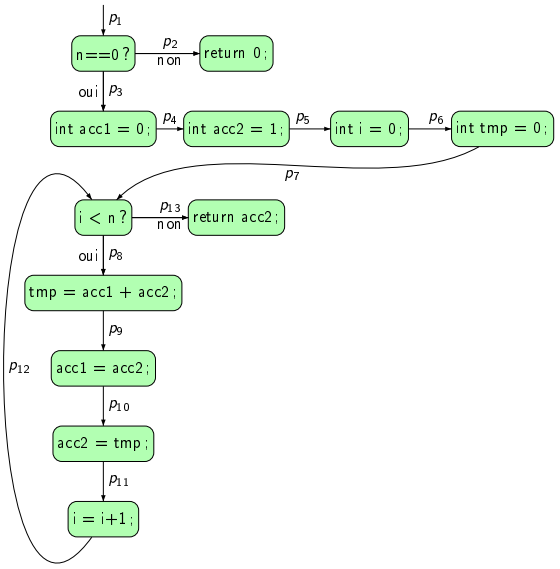


FIG. 2.1 – Un programme et son graphe de flot de contrôle.

Définition 3 (Jonction). *Une jonction est un sommet du graphe de flot de contrôle possédant au moins deux prédécesseurs.*

Définition 4 (Bloc de base). *Un bloc de base est une suite maximale d'instructions sans branchement ni jonction.*

Il est possible de condenser un graphe de flot de contrôle en ne représentant que ces structures. Le graphe de flot de contrôle de la figure 2.1 devient alors comme décrit figure 2.2.

Tout graphe de flot de contrôle possède également une forme textuelle. A chaque instruction sont associés un ou plusieurs *points de programmes* qui la précèdent et lui succèdent. Pour le programme précédent, le graphe de flot de contrôle devient comme décrit figure 2.3.

Comme usuellement en allocation de registres, nous ne considérons que des programmes dits *stricts*, *i.e.* des programmes dont les variables sont proprement initialisées avant d'être utilisées.

Définition 5 (Variable tuée par une instruction). *Une variable v est dite tuée par une instruction i si i affecte une valeur à v.*

Définition 6 (Variable engendrée par une instruction). *Une variable v est engendrée par une instruction i si i utilise la valeur de v.*

Définition 7 (Programme strict). *Un programme est dit strict si pour toute variable v, et toute instruction i qui engendre v, chaque chemin statique de l'entrée du programme à i comporte (au moins) une instruction qui tue v.*

```
        int fib (int n){
(p1)    if (n == 0)
(p2)     return 0;
        else {
(p3)     int acc1 = 0;
(p4)     int acc2 = 1;
(p5)     int i = 0;
(p6)     int tmp = 0;
(p7)     while (i < n){
(p8)      tmp = acc1 + acc2;
(p9)      acc1 = acc2;
(p10)     acc2 = tmp;
(p11)     i = i + 1;
(p12)     }
         }
(p13)    return acc2;
        }
```

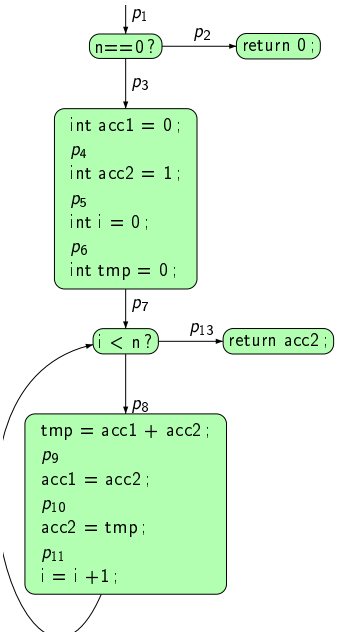


FIG. 2.2 – Un programme et son graphe de flot de contrôle condensé. Un sommet représente désormais un branchement, une jonction ou un bloc de base.

La figure 2.4 présente un exemple de programme non strict. Toute utilisation est bien dynamiquement précédée d'une définition, mais pas statiquement. En effet, le chemin passant dans la branche gauche du premier `if` et dans la branche droite du second contient une utilisation de la variable `x` mais pas de définition de celle-ci. Le fait que le chemin n'est pas exécutable en pratique n'importe pas dans la définition.

2.1.2 Durée de vie et interférences

Une utilisation fine des registres nécessite de définir précisément ce qu'il est possible, et surtout ce qu'il est impossible de faire, durant l'allocation de registres. Cette notion capitale de l'allocation de registres est captée par le concept d'interférence.

Définition 8 (Interférence)**.** *Deux variables interfèrent si celles-ci ne peuvent être simultanément stockées dans le même registre.*

En d'autres termes, deux variables interfèrent s'il existe un point de programme où elles coexistent et ont des valeurs distinctes. Cette relation d'interférence étant en pratique inexploitable, on lui préfère généralement des sur-approximations simples à calculer.

L'approche classique consiste à considérer que deux variables interfèrent si leurs portées syntaxiques s'intersectent en des points de programmes où ces variables sont *vivantes*, *i.e.* où leurs valeurs sont utiles à la suite du programme. Il faut donc savoir quand une variable est vivante.

Définition 9 (Variable vivante)**.** *Une variable v est dite vivante en un point de programme p s'il existe un chemin statique de p vers une instruction qui engendre v ne contenant pas*

```
        int fib (int n){
(p1)    if (n == 0)
(p2)     return 0;
        else {
(p3)     int acc1 = 0;
(p4)     int acc2 = 1;
(p5)     int i = 0;
(p6)     int tmp = 0;
(p7)     while (i < n){
(p8)      tmp = acc1 + acc2;
(p9)      acc1 = acc2;
(p10)     acc2 = tmp;
(p11)     i = i + 1;
(p12)     }
         }
(p13)    return acc2;
        }
```

(p_1)	n == 0 ?	$\rightarrow$	p_3, p_2
(p_2)	return 0 ;		
(p_3)	int acc1 = 0 ;		
(p_4)	int acc2 = 1 ;		
(p_5)	int i = 0 ;		
(p_6)	int tmp ;		
(p_7, p_{12})	i < n ?	$\rightarrow$	p_8, p_{13}
(p_8)	tmp = acc1+acc2 ;		
(p_9)	acc1 = acc2 ;		
(p_{10})	acc2 = tmp ;		
(p_{11})	i = i + 1 ;	$\rightarrow$	p_{12}
(p_{13})	return acc2 ;		

FIG. 2.3 – Un programme et son graphe de flot de contrôle condensé sous forme textuelle. Les points de programme donnés avant une instruction sont ceux qui la précèdent, ceux donnés après le symbole $\rightarrow$ lui succèdent. Une instruction précédée de plusieurs points de programme est donc une jonction ; une instruction suivie de plusieurs points de programmes est un branchement. Dans ce cas, le premier point de programme qui lui succède est celui atteint en cas de réussite du test de branchement, le second est celui atteint en cas d'échec. Si aucun point de programme n'est spécifié comme successeur, c'est qu'il s'agit du point de programme de la ligne suivante.

d'instruction qui tue v. On appelle durée de vie de v l'ensemble des points de programmes où v est vivante.

Définition 10 (Sur-approximation des interférences par intersection de durées de vie)**.** *Deux variables interfèrent si leurs durées de vie s'intersectent.*

Cette définition est en pratique bien plus utile, puisque les durées de vie de toutes les variables du programme peuvent être calculées grâce, notamment, à un algorithme de point fixe [App98a]. Cependant, cette sur-approximation est trop brutale, en particulier en présence d'instructions `move` . En effet, les opérandes de ces instructions possèdent la même valeur, au moins pour un temps, et peuvent donc *a priori* partager un même emplacement durant cette période. Il est néanmoins possible de tenir compte de cette particularité grâce au lemme suivant.

Proposition 1 (Caractérisation de l'intersection des durées de vie)**.** *Dans un programme strict, les durées de vie de deux variables v_1 et v_2 s'intersectent si la durée de vie de l'une contient une instruction qui tue l'autre.*

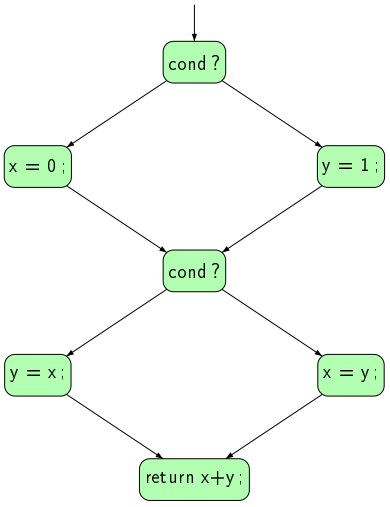


FIG. 2.4 – Un programme non strict.

Ainsi, la notion d'interférence, telle que définie par Chaitin [CAC+81], est finalement la suivante.

Définition 11 (Interférence de Chaitin)**.** *Dans un programme strict, deux variables v_1 et v_2 interfèrent si et seulement si la durée de vie de v_1 contient une instruction qui tue v_2 autre que* `move` v_2 v_1 *ou bien la durée de vie de v_2 contient une instruction qui tue v_1 autre que* `move` v_1 v_2.

Sauf mention contraire, c'est cette définition de l'interférence qui sera utilisée dans la suite de ce manuscrit.

Rappelons qu'en plus de ne pas interférer, deux variables liées par une instruction `move` sont liées par une affinité et doivent donc préférablement être stockées dans un même registre.

En résumé, l'allocation de registres obéit à une règle stricte régie par les interférences et à un critère d'optimisation guidé par les affinités.

2.2 Techniques d'allocation de registres

L'allocation de registres fut dans un premier temps considérée *localement*, *i.e.* au niveau des blocs de base, rendant ainsi le problème plus facilement abordable, même si celui-ci reste NP-difficile comme montré en 2000 [FCL00]. Cependant, les recherches se sont rapidement tournées vers une allocation de registres globale et vers le modèle de coloration de graphe [AC76, CAC+81].

2.2.1 Coloration de graphe d'interférence-affinité

Le graphe d'interférence-affinité permet d'encoder aisément les contraintes de l'allocation de registres d'un programme. Notons que puisque les interférences expriment des contraintes fortes sur les registres, alors que les affinités n'expriment qu'une préférence, une interférence l'emporte sur une affinité. Ainsi, deux sommets qui interfèrent ne sont jamais considérés comme étant liés par une affinité. Plus concrètement, nous formalisons les graphes d'interférence-affinité comme suit.

Définition 12 (Graphe d'interférence-affinité et graphe d'interférence)**.** *Le graphe d'interférence-affinité d'un programme prog est un triplet $G = (V, I, A)$, où :*

- *$V(G)$ est l'ensemble des sommets du graphe. Chaque sommet correspond à une variable de prog ;*
- *$I(G)$ est l'ensemble des arêtes d'interférence de G. Chaque arête représente une interférence entre les variables que ses extrémités représentent ;*
- *$A(G)$ est l'ensemble des affinités de G. Chaque affinité représente une instruction `move` dont les opérandes sont les variables que les extrémités de cette affinité représentent.*
- *$A(G) \cap I(G) = \emptyset$*

Afin d'alléger les notations $V(G)$, $I(G)$ et $A(G)$ seront simplement dénotés par V, I et A tant que cette convention n'est pas source de confusion.
De plus, nous appellerons $G_I = (V, I)$ graphe d'interférence et $G_A = (V, A)$ graphe d'affinité.

Les colorations et K-colorations de graphe sont quant à elle définies comme suit.

Définition 13 (Coloration et K-coloration de graphe)**.** *Une coloration propre d'un graphe d'interférence-affinité (simplement désignée par coloration par la suite) est une fonction qui à chaque sommet du graphe associe une couleur et telle que les couleurs associées aux deux extrémités d'une interférence sont différentes. Étant donné un entier naturel K, une K-coloration d'un graphe d'interférence-affinité est une coloration utilisant au plus K couleurs.*

La figure 2.5 présente un programme issu de [App98a] qui nous servira de programme de référence par la suite, ainsi que la durée de vie de chacune de ses variables, son graphe d'interférence-affinité et une coloration optimale de ce graphe pour $K = 4$.

Si le nombre de registres disponibles est insuffisant, certaines variables ne peuvent être logées en registres ; les sommets correspondants ne sont pas colorés. Le résultat que nous visons est donc une K-coloration partielle et non une K-coloration.

Définition 14 (Coloration et K-coloration partielle)**.** *Une coloration partielle d'un graphe d'interférence-affinité est une fonction qui à chaque sommet du graphe associe au plus une couleur et telle que les couleurs associées aux deux extrémités d'une interférence sont différentes. Étant donné un entier naturel K, une K-coloration partielle d'un graphe d'interférence-affinité est une coloration partielle utilisant au plus K couleurs.*

Le modèle d'allocation de registres par coloration de graphe permet en outre de facilement modéliser des contraintes supplémentaires portant sur l'allocation de registres, telles

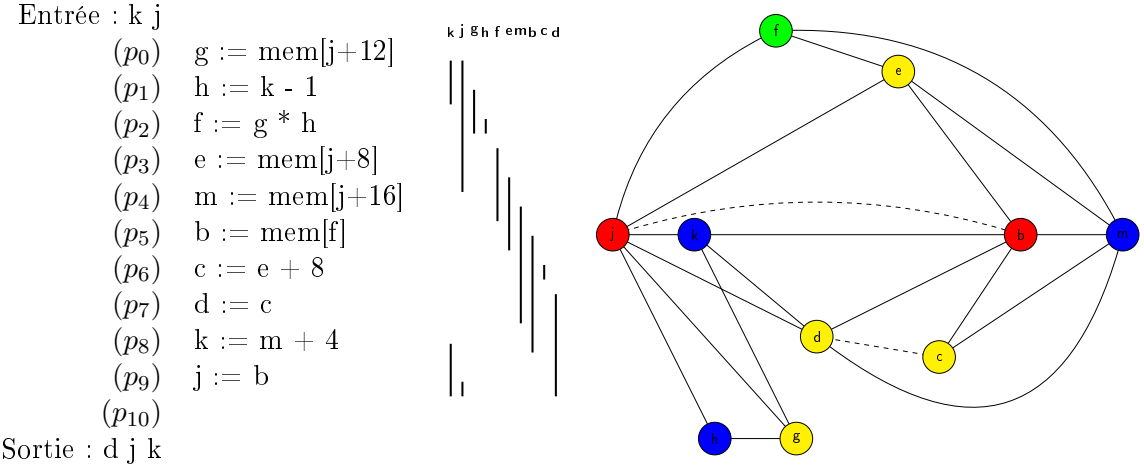


FIG. 2.5 – Un programme, les durées de vie de ses variables, et son graphe d'interférence-affinité. Les variables k et j sont supposées vivantes en entrée et les variables d, k et j sont supposées vivantes en sortie. La coloration du graphe d'interférence-affinité proposée est optimale pour quatre registres ou plus, puisque chaque sommet est coloré et que les deux affinités sont satisfaites.

que les conventions d'appels, qui peuvent imposer que les paramètres du programme sont placés dans des registres dédiés [App98a]. Il suffit pour les satisfaire de précolorer un sommet représentant un paramètre avec la couleur du registre dédié aux paramètres devant l'accueillir. Ensuite, si plusieurs sommets sont précolorés avec une même couleur, il est d'usage de tous les fusionner afin d'éviter la multiplication des sommets précolorés, puisque ces derniers induisent des cas particuliers parfois pénibles et ardus à traiter.

2.2.2 Programmation linéaire

Le problème d'allocation de registres par coloration de graphes étant NP-complet , tout processus de résolution optimal valable pour tout programme repose sur un algorithme exponentiel. Une des méthodes les plus connues, si ce n'est la plus connue, est la programmation linéaire.

Définition 15 (Programme linéaire)**.** *Un programme linéaire est un problème d'optimisation consistant à minimiser (ou maximiser) une fonction linéaire, dite fonction objectif, sous des contraintes de ses variables (égalités ou inégalités). Il s'agit donc de problèmes de la forme :*

$$(P) \begin{cases} Min & f(x) \\ & x \in X \subseteq \mathbb{R}^n \end{cases}$$

où X désigne l'ensemble des solutions admissibles (ou région admissible) et n désigne le nombre de variables. Si les variables sont à valeur dans {0,1}, le programme est dit en variables bivalentes ; si seulement certaines des variables sont à valeur dans {0,1}, le programme est dit en variables mixtes. Les contraintes qui imposent que des variables sont à valeur dans {0,1} sont appelées contraintes d'intégrité.

La figure 2.6 est un exemple de formulation permettant de déterminer une K-coloration de graphe d'interférence-affinité optimale. Les seules variables du modèle sont les variables x_{ic} qui valent 1 si et seulement si le sommet i est coloré avec la couleur c. Le poids d'une affinité (i, j) est donnée par le paramètre w_{ij}. L'objectif est de maximiser la somme des poids des affinités qui ont la même couleur, autrement dit la fonction $\sum_{(i,j)\in A} w_{ij}x_{ic}x_{jc}$, ou encore, de façon équivalente, de minimiser l'opposé de cette fonction. Trois séries de contraintes assurent que la solution est bien une K-coloration partielle. La série (1) implique que chaque sommet possède (au plus) une couleur. La série (2) impose le respect des contraintes liées aux interférences : deux sommets liés par une interférence ne peuvent avoir la même couleur. Enfin, la série (3) stipule que les variables x_{ic} sont des variables bivalentes.

$$(PC)\begin{cases} Min & -\sum_{(i,j)\in A} w_{ij}x_{ic}x_{jc} & & \\ s.c. & & & \\ \forall i \in \{1,\ldots,n(G)\}, & \sum_{c=1}^{K} x_{ic} & \leq 1 & (1) \\ \forall (i,j) \in I,\ \forall c \in \{1,\ldots,K\}, & x_{ic} + x_{jc} & \leq 1 & (2) \\ \forall i \in \{1,\ldots,n(G)\},\ \forall c \in \{1,\ldots,K\}, & x_{ic} \in \{0,1\} & & (3) \end{cases}$$

FIG. 2.6 – Une formulation par programmation linéaire permettant de calculer une K-coloration.

La première tentative de modélisation de l'allocation de registres par la programmation linéaire fut celle de Goodwin et Wilken [GW96], plus tard améliorée par Fu et Wilken [FW02]. Les auteurs modélisent directement les instructions `load` et `store` , ainsi que les états de la mémoire en chaque point de programme. Ce modèle réalise donc toute l'allocation de registres via un seul programme linéaire. Les résultats de cette expérience furent bons mais pas suffisamment pour suppléer les approches plus classiques, en raison du temps de calcul nécessaire pouvant aller jusqu'à plusieurs heures et donc jugé trop long.

En 2001, Appel et George utilisèrent également la programmation linéaire, mais cette fois uniquement pour la phase de vidage, leurs tentatives pour la fusion s'étant avérées peu fructueuses [AG01]. Ce modèle, inspiré des travaux décrits dans [GW96] et adapté à une architecture CISC, permet de résoudre efficacement le vidage dans ce contexte. Pour la fusion, les auteurs proposent d'utiliser le modèle par coloration de graphe et deux processus de résolution : l'utilisation de l'IRC, qui a le défaut de ne pas construire des solutions optimales et donc de contrarier la recherche de solutions optimales visée, et un programme linéaire qui produit des solutions optimales mais dans des temps beaucoup trop importants. Il faut noter que si le programme linéaire de résolution du vidage modélise les instructions `load` et `store` à ajouter, celui de résolution de la fusion a l'originalité de modéliser le problème de coloration de graphe.

En 2007, Barik *et al.* ont proposé une nouvelle formulation de l'allocation de registres par programmation linéaire [BGG+07]. La principale force de cette formulation est de pouvoir prendre en compte les accès bit à bit que peuvent supporter certaines architectures. Le

principal défaut de ce modèle est encore une fois le temps de calcul difficilement prévisible que peut prendre le solveur de programmation linéaire.

Enfin, en 2007, Grund et Hack ont proposé un algorithme de coupes (une méthode reposant sur la programmation linéaire) pour la fusion [GH07], que nous détaillons dans le paragraphe 2.3.6. Ces derniers, comme Appel et George [AG01] modélisent la coloration de graphe. Les résultats qu'ils obtiennent sont très satisfaisants et permettent de résoudre de façon optimale 471 des 474 instances de l'OCC, le jeu de tests proposé en 2001 par Appel et George [AG00].

2.2.3 Linear scan

Si les modèles présentés ci-dessus sont bien adaptés à la compilation statique, l'essor de la compilation à la volée dans certains types de systèmes embarqués ou dans les applications web a conduit au développement de nouvelles techniques, et en particulier l'inspection linéaire (ou *linear scan*) [PEK97, THS98, PS99b]. Comme l'indique son nom, l'idée est de linéariser le code, *i.e.* de le réduire à un bloc de base, et d'appliquer une heuristique d'optimisation locale, *i.e.* dédiée à un bloc de base. Cette approche est très rapide et peu coûteuse en mémoire, contrairement aux algorithmes reposant sur la coloration de graphe : la construction du graphe seule est déjà trop onéreuse en mémoire pour ce type de compilateurs. Cette technique a ensuite évolué afin de pallier les défauts de sur-approximation qu'elle engendre. Wimmer et Mössenbock ont par exemple introduit une technique de découpage des durées de vie afin de limiter les sur-approximations trop grossières opérées lors de la linéarisation du code [WM05]. Récemment, Sarkar et Barik ont proposé un nouvel algorithme reposant sur le même principe qu'ils présentent comme une approche alternative à l'allocation de registres par coloration de graphe [SB07].

2.2.4 Autres travaux

Certains travaux sont sortis des sentiers battus pour proposer de nouveaux modèles d'allocation de registres. Parmi ceux-ci, nous retiendrons particulièrement deux types de travaux.

2.2.4.1 Approche par multiflots

Le premier modèle qui mérite d'être cité repose sur des problèmes de multiflot. Les auteurs modélisent tout d'abord une allocation de registres locale [KG05] puis étendent ce modèle à une allocation de registres globale [KG06]. La résolution proposée est heuristique, via un algorithme qui construit une solution simple puis l'améliore incrémentalement. Ceci permet de moduler la qualité de l'allocation de registres calculée en fonction du temps laissé à l'algorithme. Nous exploiterons par la suite les liens très forts existant entre les problèmes de flots et le problème d'allocation de registres, notamment dans le chapitre 6.

2.2.4.2 Approche par puzzle

La seconde série de travaux, récemment introduite par Palsberg et Pereira, utilise la résolution par puzzle (ou *puzzle solving*) [QP08, PP10]. Les auteurs décrivent une nouvelle approche, dont les deux grandes forces sont de produire une solution localement optimale et de gérer l'aliasing de registres, *i.e.* la capacité de certains registres à contenir aussi bien deux variables simple précision (deux flottants simple précision par exemple) qu'une variable double précision (un flottant double précision par exemple). Cette solution s'avère générique et facilement instantiable sur différentes architectures.

2.3 Allocation de registres par coloration de graphe

La majeure partie de ce manuscrit est consacrée à l'allocation de registres par coloration de graphe. Cette partie décrit en détails les principales phases de l'allocation de registres par coloration de graphe ainsi que les notions de théorie des graphes liées à l'allocation de registres. Les principaux algorithmes utilisés durant la suite de cette étude, comme l'IRC ou l'algorithme de fusion par coupes de Grund et Hack y sont également présentés.

2.3.1 Le vidage

Le vidage consiste à partitionner l'ensemble des variables du programme en deux, les variables présentes en mémoire et celles présentes en registres, et ce à tout moment de l'exécution du programme. Cependant, un sommet représentant une variable, vider une variable en pile revient à la vider en pile partout et à insérer des instructions `load` et `store` avant et après chacune de ses utilisations : c'est le vidage global (ou *spill everywhere*). Le problème de vidage par coloration de graphe est donc légèrement dévié de son but initial, lequel est de générer aussi peu d'instructions `load` et `store` que possible.

Comme évoqué précédemment, et étant donné un graphe d'interférence-affinité, le vidage consiste à trouver un ensemble de sommets de cardinal maximal qui peut être coloré avec K couleurs, *i.e.* trouver le plus grand sous-graphe K-colorable. Les sommets peuvent également être pondérés pour fournir une évaluation de la pénalité subie par leur vidage. Le problème ramené au graphe est donc plus généralement un problème de recherche de sous-graphe K-colorable de poids maximum.

Définition 16 (Sous-graphe ou graphe induit)**.** *Soit $G = (V, E)$ un graphe. Le graphe induit par $S \subseteq V$ est le graphe $G' = (S, E')$ tel que E' est la restriction de E aux arêtes dont les deux extrémités appartiennent à S.*

Définition 17 (Problème de sous-graphe K-colorable maximum)**.** *Le problème de sous-graphe K-colorable maximum consiste à déterminer un ensemble de sommets de poids maximum S tel que le graphe induit par S est K-colorable.*

Pour raisonner sur la qualité du vidage, il faut considérer deux facteurs : le nombre de sommets n'appartenant pas au sous-graphe K-colorable calculé, correspondant chacun à une variable vidée en pile, et la taille du spill code généré en conséquence. Puisque nous utiliserons toujours le même algorithme de génération de spill code, nous n'optimiserons

que la phase de calcul du sous-graphe K-colorable maximum. Aussi, un vidage sera dit optimal si le sous-graphe K-colorable calculé est de poids maximum.

Proposition 2 (Vidage optimal par coloration de graphe)**.** *Le problème de vidage optimal est exactement le problème de sous-graphe K-colorable de poids maximum.*

Malheureusement, déterminer un vidage optimal de cette façon est un problème difficile, puisque le problème de sous-graphe K-colorable de poids maximum est NP-complet . Pire encore, déterminer s'il est nécessaire de vider en pile au moins une variable est un problème NP-complet , puisqu'il revient à déterminer le nombre chromatique du graphe d'interférence.

Définition 18 (Nombre chromatique d'un graphe)**.** *On appelle nombre chromatique d'un graphe G, et on note $\chi(G)$, le nombre de couleurs minimal nécessaire pour réaliser une coloration de G.*

Proposition 3 (Nécessité du vidage et nombre chromatique)**.** *Si K est le nombre de registres allouables, déterminer si le vidage est nécessaire revient à établir si $K \leq \chi(G_c)$.*

2.3.2 Algorithme de vidage de Chaitin

La difficulté du problème sous-graphe K-colorable maximum poussa Chaitin à définir une heuristique pour le vidage. Celle qu'il proposa repose sur un lemme de Kempe datant de 1879 [Kem79] et la notion de sommet trivialement colorable.

Définition 19 (Voisinage d'un sommet)**.** *Le voisinage d'interférence (respectivement d'affinité) d'un sommet s dans un graphe G, noté $N(s, G)$ (respectivement $N_a(s, G)$), est l'ensemble des sommets liés à s par une interférence (respectivement une affinité).*

Définition 20 (Degré d'un sommet dans un graphe)**.** *Le degré d'interférence (respectivement d'affinité) d'un sommet s dans un graphe G, noté $\delta(s, G)$ (respectivement $\delta_a(s, G)$), est le cardinal de $N(s, G)$ (respectivement $N_a(s, G)$).*

Lemme 1 (Sommet trivialement colorable)**.** *Soit G un graphe d'interférence-affinité. Si s est un sommet de G de degré d'interférence strictement inférieur à K, alors G est K-colorable si et seulement si le graphe induit par $V - \{s\}$ est K-colorable. On dit que s est un sommet trivialement colorable [Kem79].*

Démonstration. Si s est un sommet de degré d'interférence strictement inférieur à K, alors toute K-coloration du graphe induit par $V - \{s\}$ peut être augmentée en une coloration de G puisqu'au plus $K - 1$ couleurs sont interdites pour s. L'autre implication est immédiate puisque toute coloration d'un graphe est une coloration de tous ses graphes induits. Ainsi, le graphe est K-colorable si et seulement si le graphe induit par $V - \{s\}$ l'est. □

L'heuristique de vidage de Chaitin se contente alors de supprimer les sommets trivialement colorables du graphe d'interférence, en les empilant au fur et à mesure dans une pile de coloration, jusqu'à ce que le graphe soit vide ou qu'aucun sommet ne soit trivialement colorable. Dans le second cas, une variable correspondant à un sommet restant est désignée

pour être vidée en pile (mémoire), puis une nouvelle phase d'empilement des sommets trivialement colorables commence. Une fois que le graphe est vide, les sommets sont colorés dans l'ordre inverse de leur élimination ; les sommets marqués pour être vidés en pile ne sont pas colorés. Cette phase de coloration correspond à l'affectation des registres.

La figure 2.7 illustre l'application de l'algorithme de Chaitin [CAC+81, Cha82] au graphe de référence. Plusieurs solutions existent pour décider dans quel ordre les sommets sont empilés et les couleurs attribuées. D'autres choix auraient parfaitement pu mener à une coloration qui ne satisfait aucune des deux affinités, voire à une coloration qui laisse deux sommets non colorés. Diverses améliorations de l'algorithme de Chaitin ont d'ailleurs introduit des heuristiques pour choisir les sommets à placer en pile ou les couleurs à attribuer [Bri92, BCT94, GA96].

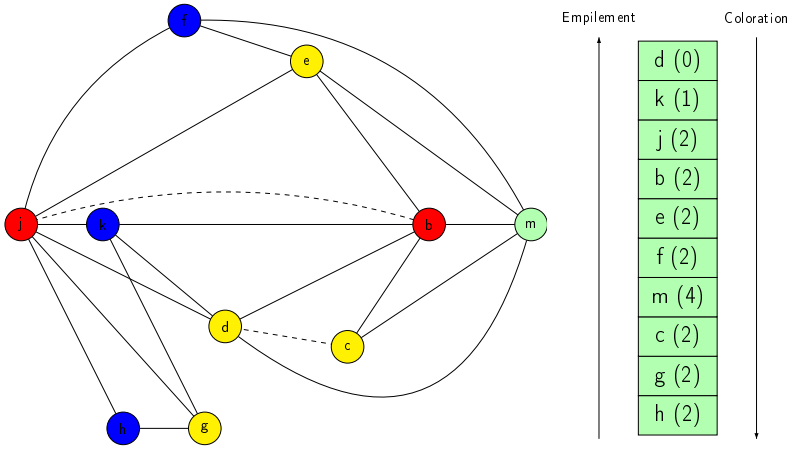


FIG. 2.7 – Illustration de l'heuristique de vidage de Chaitin pour $K = 3$. Les degrés indiqués à côté de la pile sont les degrés d'interférence des sommets au moment de leur empilement. Un degré d'interférence supérieur ou égal à K signifie que la variable est vidée en pile.

L'algorithme de Chaitin fut ensuite amélioré par Briggs, via la notion de vidage optimiste [Bri92, BCT94]. Ce dernier remarqua que le sommet correspondant à une variable désignée pour être placée pile peut parfois être coloré. Les algorithmes qui vérifient si une couleur reste disponible pour les sommets marqués comme devant être vidés en pile opèrent un vidage dit *optimiste* tandis que les autres algorithmes opèrent un vidage dit *pessimiste*. La figure 2.8 illustre un graphe simple pour lequel l'heuristique de Chaitin génère un stockage en mémoire inutile qui peut être évité grâce au vidage optimiste.

2.3.3 Le découpage des durées de vie

Le dialogue avec la pile étant très coûteux pour le code produit, il est primordial de le limiter autant que possible. L'idée motrice du découpage des durées de vie est de subdiviser la durée de vie d'une variable en plusieurs plus petites, et d'affiner ainsi les interférences et affinités liant les variables du programme. En pratique, il suffit de copier une variable v en une variable v' ; les contraintes d'interférence qui pèsent sur v et v' sont alors plus faibles que celles portant initialement sur v. En contrepartie, v et v' sont liées par une

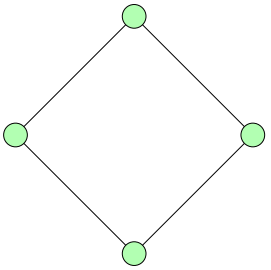


FIG. 2.8 – Un exemple de graphe simple pour lequel l'heuristique de Chaitin génère un stockage en pile inutile, pour $K = 2$. Aucun sommet n'est trivialement colorable puisque chaque sommet est de degré 2. En revanche l'intégration du vidage optimiste remédie au problème.

affinité. Ainsi, v et v' peuvent être affectées à des emplacements différents pour le modique coût d'une instruction `move` afin d'éviter le lourd coût d'instructions `load` et `store` . Le découpage des durées de vie permet ainsi d'affiner le modèle par coloration de graphe. C'est d'ailleurs pour ce modèle qu'il a été introduit [CS98]. Il arrive également qu'une même variable ait une durée de vie composée de plusieurs intervalles distincts. C'est par exemple le cas des variables k et j dans l'exemple de référence (cf. figure 2.5, p. 43).

Le concept de découpage des durées de vie est séduisant et puissant mais recèle une difficulté majeure. Il est très difficile de clairement définir comment subdiviser les durées de vie de façon efficace. Ainsi, de nombreuses façons de fractionner les durées de vie ont été proposées [CH84, Bri92, KH93, LG97, BDEO97, CS98, PM98, AG01, QP08]. Théoriquement, plus les durées de vie sont petites, plus l'approximation des interférences réelles est fine [CS98, AG01]. Empiriquement, ce résultat est vérifié, mais le bénéfice engrangé lors du vidage peut être contrebalancé par le grand nombre d'instructions `move` ajoutées, lesquelles doivent être considérées durant la fusion [BDEO97, CS98, AG01]. Un bon équilibre entre la fusion et le découpage des durées de vie, que l'on peut voir comme deux opérations duales, est donc indispensable.

2.3.4 La fusion

Pour supprimer une instruction `move` du code source, il suffit de satisfaire une affinité. La fusion a donc pour but de satisfaire un maximum d'affinités sans pour autant à nouveau créer le besoin de vider en pile des variables. En effet, il faut toujours garder à l'esprit qu'une instruction `load` ou `store` est bien plus coûteuse qu'une instruction `move` . Si une affinité est satisfaite, les sommets liés par cette affinité peuvent être fusionnés, ce qui a tendance à faire augmenter les degrés des sommets, mais qui peut également contribuer à rendre de nouveaux sommets trivialement colorables [GA96]. On retrouve en particulier dans la littérature trois types de fusion : la fusion *agressive*, la fusion *conservative* et la fusion *optimiste*.

La fusion agressive La fusion agressive [Cha82] vise à satisfaire autant d'affinités que possible. Cette forme de fusion a été la première utilisée. Son principal défaut est que les opérations successives de fusion mènent très souvent à faire augmenter le nombre chromatique du graphe, et contribuent ainsi à vider en pile des variables supplémentaires.

Définition 21 (Problème de fusion agressive)**.** *Soit G un graphe d'interférence-affinité. Le problème de fusion agressive consiste à déterminer un ensemble d'affinités pouvant toutes être simultanément satisfaites de poids maximal ; autrement dit, un ensemble d'affinités de poids maximal tel qu'il existe une coloration de G qui satisfait chacune des affinités de cet ensemble.*

La fusion optimiste La fusion optimiste [PM98, PM04] consiste à effectuer une fusion agressive, puis à annuler certaines étapes de fusion réalisées si le graphe ne parvient pas à être coloré avec les K couleurs disponibles. En ce sens, la fusion optimiste est une amélioration de la fusion agressive. D'un autre point de vue, la fusion optimiste est un mélange de fusion agressive et de découpage des durées de vie. Annuler une fusion revient en effet à réaliser une étape de découpage.

La fusion conservative La fusion conservative [BCT94] est une stratégie qui vise à assurer *a priori* qu'aucun vidage supplémentaire ne deviendra nécessaire suite à la fusion. C'est la stratégie la plus courante aujourd'hui. Cette méthode n'introduit donc pas de vidage en pile et de découpage supplémentaire, mais a parfois tendance à pêcher par son caractère trop prudent. Ainsi, le problème de fusion conservative pour un graphe K-colorable est le suivant.

Définition 22 (Problème de fusion conservative)**.** *Soit G un graphe d'interférence-affinité K-colorable. Le problème de fusion conservative consiste à déterminer un ensemble d'affinités de poids maximal tel qu'il existe une K-coloration de G qui satisfait toutes les affinités de cet ensemble.*

En pratique, déterminer si une affinité peut être satisfaite tout en préservant la K-colorabilité du graphe est en général un problème NP-complet [BDR07a] et les critères utilisés sont des conditions suffisantes qui s'avèrent souvent trop restrictives. Deux critères sont cependant retenus : les critères de Briggs [Bri92, BCT94] et de George [GA96].

Théorème 1 (Critère conservatif de Briggs)**.** *Si $N(x, G) \cup N(y, G)$ contient strictement moins de K sommets non trivialement colorables, alors la fusion de x et y est conservative [BCT94].*

Théorème 2 (Critère conservatif de George)**.** *Si $N(x) \subseteq N(y)$ et que x ou y est précoloré, alors la fusion de x et y est conservative [GA96].*

Aucun de ces deux critères n'est théoriquement meilleur que l'autre. En effet, le critère de Briggs fait intervenir le nombre de registres disponibles tandis que le critère conservatif de George n'en dépend pas. L'alliance des deux critères est généralement adoptée. Une étude détaillée de ces critères et de leurs performances est effectuée dans [Hai05].

2.3.5 IRC

L'IRC [GA96] est une heuristique d'allocation de registres largement utilisée aussi bien dans le monde académique que dans l'industrie. Cette heuristique est fondée sur des concepts préexistants, principalement tirés de l'heuristique de vidage de Chaitin et de l'heuristique de fusion conservative de Briggs [BCT94]. Les forces de cette heuristique sont sa simplicité, son intelligent séquencement des opérations, et la bonne qualité générale des allocations qu'elle génère.

La première phase de l'IRC repose sur quatre opérations, qu'elle effectue selon l'ordre de priorité suivant :

1. la *simplification* consiste à empiler un sommet trivialement colorable qui n'a aucune affinité, comme dans l'heuristique de vidage de Chaitin ;
2. la *fusion* consiste à fusionner deux sommets x et y liés par une affinité s'ils satisfont le critère de Briggs ou celui de George. Le sommet résultant de cette fusion est nommé x et y est empilé dans le but de se voir être coloré comme x ;
3. le *gel* consiste à abandonner l'espoir de satisfaire les affinités d'un sommet trivialement colorable afin de pouvoir le rendre éligible pour la simplification. Néanmoins, les affinités du sommet pourront éventuellement être satisfaites ;
4. le *vidage* correspond au vidage optimiste. Le sommet élu pour cette opération est également empilé.

L'ordre dans lequel les opérations sont effectuées peut être considéré comme un ordre de bénéfice décroissant. La simplification est algorithmiquement peu coûteuse et ne détériore pas la qualité de la solution. L'opération de fusion est un peu plus coûteuse algorithmiquement et conduit généralement à une bonne allocation de registres. Le gel est algorithmiquement peu coûteux mais pessimiste pour les affinités. Enfin, le vidage est l'opération à éviter mais parfois inéluctable.

Durant la deuxième phase de l'IRC, qui correspond à l'affectation des registres, les sommets empilés sont dépilés en ordre inverse et colorés, comme dans l'heuristique de vidage de Chaitin. Cette étape correspond donc à l'affectation des registres. Enfin, si un vidage est généré, et qu'il n'a pas été décidé de garder deux registres pour dialoguer avec la pile, la reconstruction est lancée : le graphe d'interférence-affinité est mis à jour et la procédure relancée. Le schéma complet de l'algorithme, tel que décrit dans l'article introductif est décrit figure 2.9.

2.3.6 L'algorithme de coupes de Grund et Hack

L'algorithme de coupes de Grund et Hack [GH07] permet de résoudre de façon optimale la fusion et l'affectation des registres d'un seul tenant. Le vidage est supposé avoir déjà été réalisé, ce qui implique que le graphe d'interférence-affinité à colorer est K-colorable.

2.3.6.1 Les algorithmes de coupes

Pour définir ce qu'est un algorithme de coupes il faut savoir que la résolution d'un programme linéaire repose sur la résolution de sa relaxation continue, *i.e.* le programme où

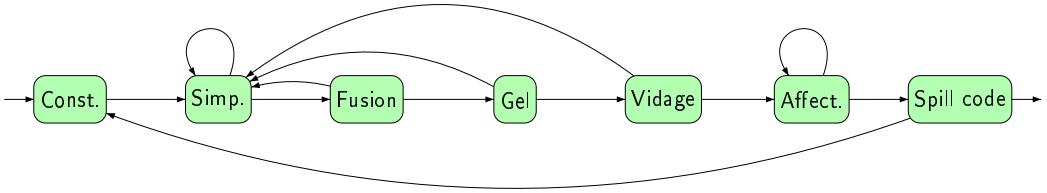


FIG. 2.9 – Représentation schématique de l'IRC. Les opérations d'élimination des sommets et des arêtes (simplification, fusion, gel et vidage) sont d'abord réalisées. Les sommets sont ensuite colorés selon l'ordre inverse de leur élimination. Enfin, le spill code est intégré, le graphe d'interférence-affinité est recalculé et l'algorithme est relancé si les contraintes d'interférence sont violées par l'insertion du spill code.

les contraintes d'intégrité ne sont pas prises en compte (on dit alors que ces contraintes sont relâchées). La région admissible de la relaxation continue d'un programme linéaire P est donc un polyèdre contenant la région admissible de P. Les algorithmes de coupes ajoutent de nouvelles contraintes à P, appelées coupes, dans le but de réduire la région admissible de la relaxation continue tout en préservant la région admissible de P, comme schématisé figure 2.10. La résolution du problème s'en trouve accélérée car le solveur n'ajoute au programme P que les coupes qui lui sont utiles.

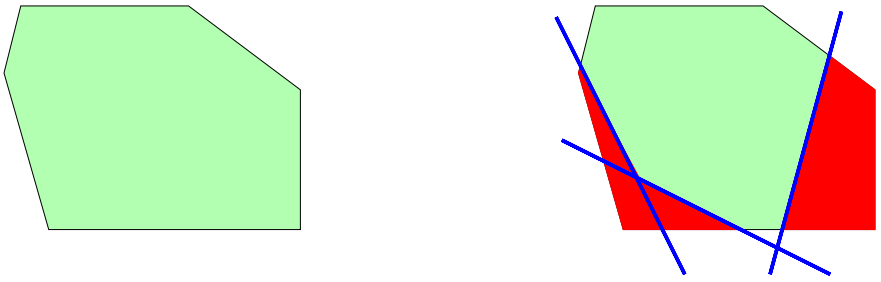


FIG. 2.10 – Schématisation de l'effet des coupes (en bleu) sur la région admissible de la relaxation continue (en vert). La zone rouge correspond à la partie de la région admissible pour la relaxation continue ne respectant pas les coupes.

Il faut noter que les coupes sont des contraintes additionnelles qui ne sont aucunement nécessaires à la correction du programme linéaire. Leur seul but est d'accélérer la résolution.

2.3.6.2 Programmation linéaire

Le formulation par programmation linéaire proposée par Grund et Hack, donné figure 2.11, modélise la fusion par coloration de graphe. Les auteurs utilisent pour ce faire deux types de variables bivalentes :

– les variables x_{ic} qui valent 1 si et seulement si le sommet i est de couleur c ;

– les variables y_e qui valent 1 si et seulement si e est une affinité dont les extrémités sont de couleurs différentes.

$$(PF)\begin{cases} Min & \sum_{(i,j)\in A} w_{ij}y_{ij} & & & \\ s.c. & & & & \\ \forall i \in \{1,\dots,n(G)\}, & \sum_{c=1}^{K} x_{ic} & = & 1 & (1) \\ \forall (i,j)\in I,\ \forall c\in\{1,\dots,K\}, & x_{ic}+x_{jc} & \leq & 1 & (2) \\ \forall (i,j)\in A,\ \forall c\in\{1,\dots,K\}, & x_{ic}-x_{jc} & \leq & y_{ij} & (3) \\ \forall i\in\{1,\dots,n(G)\},\ \forall c\in\{1,\dots,K\}, & x_{ic}\in\{0,1\} & & & (4) \\ \forall (i,j)\in A & y_{ij}\in\{0,1\} & & & (5) \end{cases}$$

FIG. 2.11 – Programme linéaire de fusion proposé par Grund et Hack.

La formulation indique qu'il faut minimiser la somme des poids des affinités non satisfaites sous quatre séries de contraintes. Les séries (1) et (2) stipulent que chaque sommet doit être coloré et que les extrémités de toute interférence doivent avoir des couleurs différentes. Ce sont les inégalités classiques de coloration. La série (3) spécifie le lien entre les variables x et y : une variable y_{ij} vaut 0 si et seulement si $x_{ic} = x_{jc}$ pour toute couleur c. Enfin les deux dernières séries contiennent les contraintes d'intégrité pour les variables x et y.

2.3.6.3 Coupes de chaîne

Grund et Hack proposent plusieurs séries de coupes pour leur programme linéaire. Nous ne détaillons ici que la série la plus efficace d'après les auteurs, puisqu'elle constitue à elle seule plus de 95% des coupes effectivement utilisées par le solveur.

Définition 23 (Chaîne)**.** *Une chaîne est une liste d'arêtes consécutives, i.e. telle que la première extrémité de la $(n+1)$-ème arête est la seconde extrémité de la n-ème. Une chaîne est dite élémentaire si aucun sommet n'y apparaît deux fois.*

Définition 24 (Coupe de chaîne)**.** *Soient deux sommets s_1 et s_2 tels que (s_1, s_2) est une interférence. Pour toute chaîne $(e_1, \dots, e_l)$ d'affinités entre s_1 et s_2, l'inégalité suivante est valide :*

$$\sum_{i=1}^{l} y_{e_i} \geq 1$$

La figure 2.12 est un exemple de coupe de chaîne.

2.4 Static Single Assignment (SSA)

Les années 80 ont été marquées par le modèle d'allocation de registres par coloration de graphe ; les années 90 ont été marquées par l'intégration du découpage des durées de

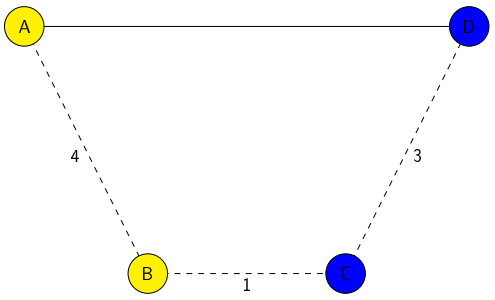


FIG. 2.12 – Une coupe de chaîne. Les affinités (A, B),(B, C) et (C, D) ne peuvent être simultanément satisfaites puisque sinon l'interférence liant A à D serait violée.

vie et le développement des techniques de fusion ; les années 2000 ont été marquées par les conséquences émanant de l'utilisation de la forme SSA des programmes sur l'allocation de registres.

Nous détaillons ici le théorème le plus important pour l'allocation de registres sous forme SSA, lequel montre que le graphe d'interférence d'un programme sous forme SSA appartient à la classe des graphes triangulés. Il en résulte que ce graphe peut être coloré avec K couleurs en temps polynomial s'il est K-colorable.

2.4.1 Forme SSA d'un programme

La forme SSA naît des travaux de Rosen et Alpern [RWZ88, AWZ88] portant sur les études de valeurs redondantes ou égales dans des programmes et les optimisations leur étant liées. Celle-ci n'est cependant rendue pratiquement utilisable que trois ans plus tard par le premier algorithme de mise sous forme SSA de programmes [CFR+91].

Définition 25 (Forme SSA d'un programme)**.** *Un programme est sous forme SSA si chaque variable est textuellement définie une seule fois [CFR+91].*

Intuitivement, la forme SSA évite toute redéfinition d'une variable, ce qui correspond à associer à chaque variable un rôle unique. Ainsi, les analyses de programme sur une forme SSA sont souvent plus fines que sur le programme initial. La figure 2.13 présente une forme SSA du programme qui nous sert de fil rouge.

L'inconvénient majeur de la forme SSA est le besoin d'unifier les noms des variables, à certaines jonctions par exemple. Cette opération est réalisée par une fonction virtuelle, appelée Φ-fonction [CFR+91].

La figure 2.14 illustre un exemple de programme pour lequel la mise sous forme SSA fait appel à une Φ-fonction. Celle-ci permet d'unifier en une variable v_3 la variable v_1, copie de v issue de la branche gauche du programme, et la variable v_2, copie de v issue la branche droite.

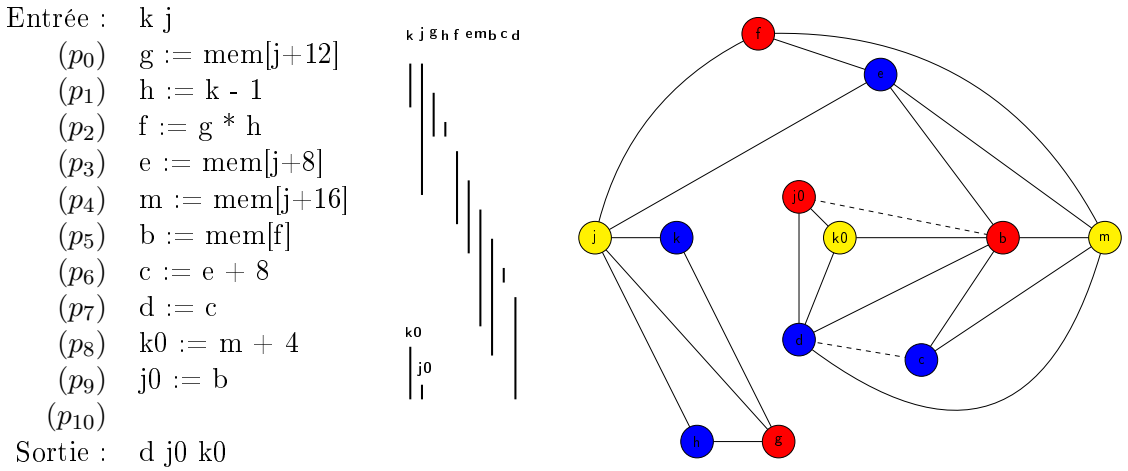


FIG. 2.13 – Le programme de référence sous forme SSA, les durées de vie de ses variables et une coloration optimale pour 3 registres ou plus. Il n'est plus nécessaire de disposer de 4 registres comme initialement car l'utilisation de la forme SSA permet aux variables k et k0 d'une part et j et j0 d'autre part d'être placées dans des registres différents.

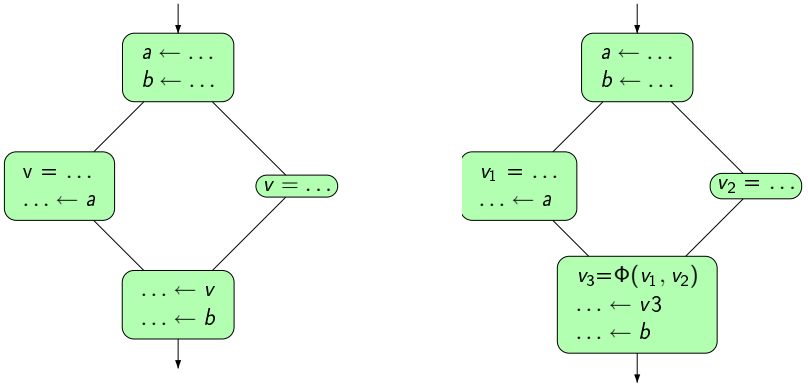


FIG. 2.14 – Un programme et sa forme SSA. Une Φ-fonction est ajoutée à la jonction afin d'unifier les différentes copies de la variable v.

2.4.2 Les graphes triangulés

Définition 26 (Graphes triangulés)**.** *Un graphe est dit triangulé si tout cycle de longueur supérieure ou égale à quatre contient une corde, i.e. une arête liant deux sommets non adjacents du cycle.*

Cette définition des graphes triangulés n'est pas manipulable en pratique. *A contrario*, deux importantes caractérisations des graphes triangulés sont largement utilisées. La première s'appuie sur le concept d'ordre d'élimination simpliciale [FG69].

Définition 27 (Clique, clique maximale, clique maximum)**.** *Une clique d'un graphe est un graphe induit tel que toute paire de sommets est liée par une arête. Une clique est dite maximale si elle n'est incluse dans aucune autre clique. Une clique est dite maximum si elle*

possède un nombre de sommets maximal. Étant donné un graphe G, on dénote par $\omega(G)$ le nombre de sommets d'une clique maximum.

Définition 28 (Sommets simpliciaux et ordre d'élimination simpliciale)**.** *Soit $G = (V, E)$ un graphe. Un sommet s est simplicial dans G si le graphe induit par les voisins de s est une clique. Un ordre d'élimination simpliciale d'un graphe G est un ordonnancement $(v_1,\ldots,v_n)$ de $V(G)$ tel que pour tout i de $\{1,\ldots,n\}$ v_i est simplicial dans le graphe induit par $(v_i,\ldots,v_n)$.*

Proposition 4 (Caractérisation des graphes triangulés I)**.** *Un graphe est triangulé si et seulement s'il admet un ordre d'élimination simpliciale [Gav72].*

La figure 2.15 présente un exemple de graphe triangulé, ainsi qu'un ordre d'élimination simpliciale de celui-ci. Trouver un tel ordre pour un graphe, s'il en existe, est un problème polynomial. Ainsi, déterminer si un graphe est triangulé est un problème polynomial, sur lequel nous reviendrons ultérieurement.

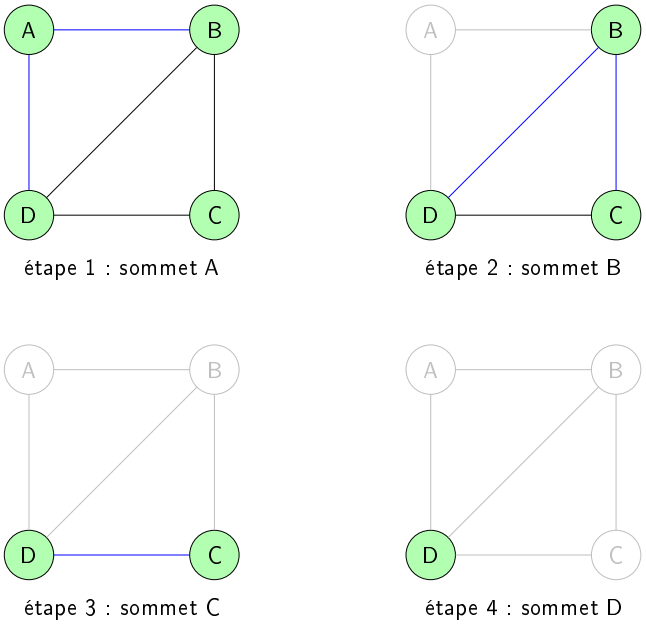


FIG. 2.15 – Un graphe triangulé et la construction d'un de ses ordres d'élimination simpliciale, l'ordre (A B C D), étape par étape.

Les ordres d'élimination simpliciale ont été utilisés avec succès pour résoudre des problèmes NP-complets dans le cas général mais polynomiaux dans les graphes triangulés. Un exemple qui nous intéresse particulièrement est le problème de coloration minimale d'un graphe.

Définition 29 (Coloration minimale d'un graphe)**.** *Le problème de coloration minimale d'un graphe G consiste à trouver une coloration utilisant un nombre minimal de couleurs, i.e. $\chi(G)$ couleurs.*

Si le problème de coloration minimale est NP-complet dans le cas général, il devient polynomial dans les graphes triangulés [Gav72]. En effet, les graphes triangulés font partie de la classe des graphes parfaits, *i.e.* la classe des graphes pour laquelle le nombre chromatique de tout sous-graphe (y compris donc le graphe lui-même) est égal à la taille de la plus grande clique dudit sous-graphe. Or, connaître un ordre d'élimination simpliciale d'un graphe G permet de trouver en temps polynomial toutes ses cliques maximales, et donc $\omega(G)$. Il est même possible de trouver une coloration minimale en temps linéaire si un ordre d'élimination simpliciale est connu.

Les graphes triangulés peuvent également être caractérisés en tant que famille de graphes d'intersections [RS86].

Définition 30 (Graphe d'intersection d'une famille de graphes)**.** *Le graphe d'intersection d'une famille de graphes F est le graphe tel que les sommets sont les éléments de F et deux sommets F_i et F_j sont liés par une arête si les graphes F_i et F_j s'intersectent.*

Définition 31 (Graphe connexe et arbre)**.** *Un graphe est dit connexe si toute paire de sommets peut être reliée par une chaîne. Un arbre est un graphe connexe sans cycle, i.e. sans chaîne reliant un sommet à lui-même.*

Proposition 5 (Caractérisation des graphes triangulés II)**.** *Un graphe est triangulé si et seulement s'il est le graphe d'intersection d'une famille de sous-arbres d'un arbre.*

L'exemple de la figure 6.5 illustre un graphe triangulé ainsi qu'une représentation en tant que graphe d'intersection de sous-arbres d'un arbre. Cette forme de décomposition a rapidement mené à la définition de nouveaux algorithmes reposant le plus souvent sur des algorithmes de parcours d'arbre.

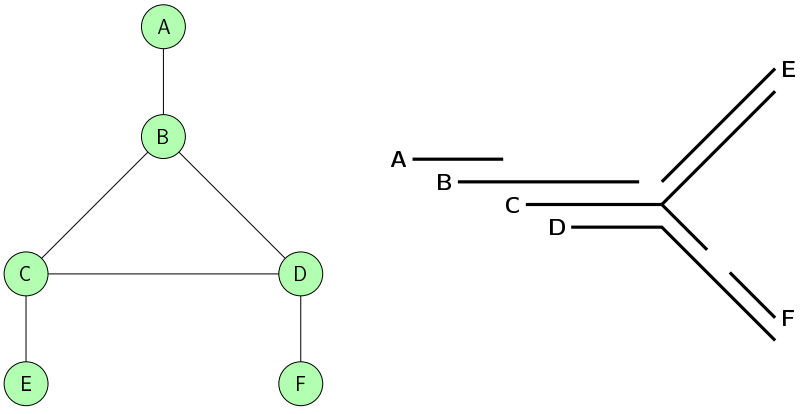


FIG. 2.16 – Un graphe triangulé et une représentation en tant que graphe d'intersection des sous-arbres d'un arbre.

2.4.3 Propriété de dominance

Le théorème 5 portant sur triangulation des graphes d'interférence sous forme SSA passe par une relation connue portant sur les instructions du programme appelée ordre de dominance [LT79, GT04].

Définition 32 (Relation de dominance). *Une instruction s domine une instruction s' si tout chemin de l'entrée du programme à s' passe par s. Cette relation est un ordre sur les instructions du programme [LT79].*

Définition 33 (Graphe de dominance). *Le graphe de dominance d'un programme est le diagramme de Hasse de la relation de dominance, i.e. le graphe orienté tel que (s, s') appartient au graphe si s domine s' et tel que les arcs de transitivité sont supprimés.*

Définition 34 (Propriété de dominance). *Un programme satisfait la propriété de dominance s'il existe une instruction (correspondant nécessairement à l'entrée du programme) qui domine toutes les autres.*

Théorème 3 (Arbre de dominance et forme SSA). *Le graphe de dominance d'un programme strict sous forme SSA satisfaisant la propriété de dominance est un arbre.*

Théorème 4 (Arbre de dominance et durées de vie). *Sous forme SSA, les durées de vie des variables d'un programme strict satisfaisant la propriété de dominance sont des sous-arbres de l'arbre de dominance.*

La figure 2.17 présente l'arbre de dominance de la forme SSA du programme présenté figure 2.14.

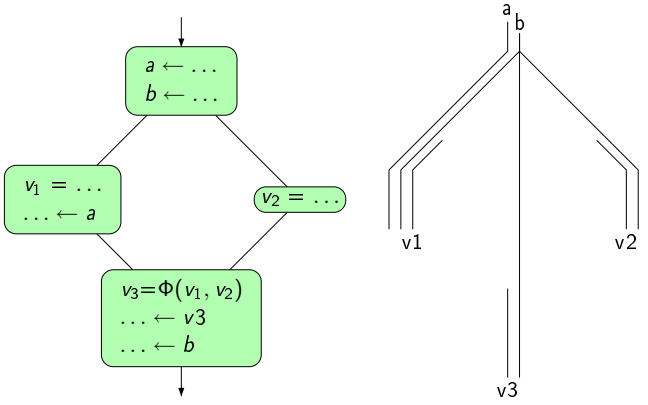


FIG. 2.17 – Un programme sous forme SSA et les durées de vie de ses variables. Ces derniers forment des sous-arbres de l'arbre de dominance.

2.4.4 Forme SSA et graphes triangulés

L'intérêt majeur de la forme SSA pour l'allocation de registres réside dans le théorème suivant, indépendamment publié par Bouchez, Darte, Guillon et Rastello d'une part et Goos et Hack d'autre part en 2005 [BDGR05, Hac05], mais dont la première preuve est réalisée par Darte dès 2002.

Théorème 5 (Graphes d'interférence sous forme SSA)**.** *Si un programme strict satisfaisant la propriété de dominance est sous forme SSA, alors son graphe d'interférence est triangulé.*

Démonstration. Le graphe d'interférence est le graphe d'intersection des durées de vie des variables. Or, d'après le théorème 4 ces durées de vie sont des sous-arbres de l'arbre de dominance du programme. Le graphe d'interférence est donc triangulé. □

Le théorème implique en particulier qu'il est possible de connaître le nombre minimal de registres nécessaires pour stocker toutes les variables du programme, puisque celui-ci n'est autre que le nombre chromatique d'un graphe triangulé. En outre, il devient facile de caractériser toutes les cliques maximales du graphe d'interférence : chacune correspond à l'ensemble des variables vivantes en un point de programme [Bou08].

Ce résultat a eu des conséquences fondamentales sur la façon d'aborder l'allocation de registres. En premier lieu, nombre de résultats considérés comme immuables sont devenus obsolètes. La preuve de NP-complétude de Chaitin en est un exemple : décider si vider au moins une variable est nécessaire devient un problème polynomial en forme SSA, mais décider quelles variables vider en mémoire reste NP-difficile [BDR07b, YG87].

En second lieu, de nouveaux algorithmes s'appuyant sur les graphes triangulés ont pu être définis.

Le premier algorithme d'allocation de registres dédié aux graphes d'interférence parfaits, une surclasse des graphes triangulés, est apparu en 2003 [And03], avant même la première publication du théorème 5. En effet, Andersson remarque empiriquement que nombre de programmes ont un graphe d'interférence parfait et utilise donc pour la première fois un algorithme capable de calculer une coloration minimale. Deux ans plus tard, deux algorithmes d'allocation de registres apparaissent [PP05, BPM05]. Chacun d'eux réalise une coloration minimale grâce à un algorithme optimal connu de coloration puis résout la fusion de façon heuristique. Cependant, les approches diffèrent par le fait que Palsberg et Pereira font l'observation du caractère triangulé des graphes d'interférence, tandis que Brisk prouve que les graphes d'interférence de programmes sous forme SSA sont parfaits. De leur côté, Bouchez, Darte et Rastello prouvent que les graphes d'interférence de programme sous forme SSA sont triangulés, remettant ainsi en question tous les acquis jusqu'alors établis, en particulier les résultats de complexité [BDGR05, BDGR06, BDR07b, BDR07a]. En parallèle, Hack et Goos prouvent le même résultat et s'en servent d'appui pour définir une nouvelle heuristique [HGG06]. En continuant sur cette voie, Hack définit une nouvelle heuristique de fusion plus performante [HG08], tandis que Bouchez, Darte et Rastello établissent un nouveau critère de fusion conservatif spécifique aux graphes d'interférence triangulés ainsi qu'une heuristique reposant sur celui-ci [BDR07a, BDR08]. Ce dernier résultat améliore la qualité de la fusion dans les graphes de programmes sous forme SSA mais ne peut

être appliqué itérativement puisqu'une fusion peut briser le caractère triangulé du graphe d'interférence.

Une constante se dégage de ces travaux dédiés aux graphes d'interférence triangulés : la séparation du vidage et de la fusion. Cette séparation en deux phases permet de simplifier l'allocation de registres de programmes sous forme SSA. En effet, le vidage est plus précis dans ces graphes en vertu de la triangulation du graphe d'interférence, puisque calculer le nombre chromatique et une coloration minimale (en terme de nombre de couleurs) sont des problèmes polynomiaux. De plus, chacune des deux phases seule reste NP-difficile à résoudre dans les graphes triangulés [BDR07b, BDR07a, Bou08]. Ainsi, les découpler apparaît comme une nécessité pour aller vers des solutions de meilleure qualité. Cette intuition est d'ailleurs confirmée par les expérimentations pratiques puisque les résultats obtenus via cette séparation sont meilleurs que ceux reposant sur un algorithme mêlant les deux phases [KG09].

2.5 Découpage extrême

La stratégie de découpage extrême est apparue dans divers travaux sans forcément être mise en avant : les méthodes de résolution par programmation linéaire de Goodwin et Wilken [GW96] et Appel et George [AG01] en font en effet usage puisque ces techniques déterminent l'emplacement mémoire de chaque variable en chaque point de programme. Les travaux de Palsberg et Pereira par résolution de puzzle en font quant à eux nommément usage [QP08, PP10].

2.5.1 Graphes élémentaires

Les méthodes de découpage extrême décrites par Appel et George d'une part, et Pereira d'autre part, diffèrent légèrement. Les graphes élémentaires ont été introduits par Pereira [QP08]. Leur processus de construction est simple : toutes les variables vivantes en un point de programme y sont copiées en parallèle. Les copies parallèles permettent d'éviter de créer de fausses interférences dues à la sur-approximation des contraintes par le graphe d'interférence-affinité. La définition des graphes élémentaires est la suivante.

Définition 35 (Graphe élémentaire d'un programme)**.** *Soit un programme prog et la transformation suivante, qui transforme prog en prog′.*
Soient p_1 et p_2 deux points de programme séparés par une instruction i et $v_0^1, \ldots, v_n^1$ les variables vivantes en p_1. Si l'instruction i est de la forme $v = v_j^1$ op v_k^1 alors i est remplacée comme suit :

$$\bullet p_1$$

$$v_0^2 \leftarrow v_0^1 \parallel v_1^2 \leftarrow v_1^1 \parallel \ldots \parallel v_n^2 \leftarrow v_n^1$$

$$v := v_j^2 \ op \ v_k^2$$

$$\bullet p_2$$

Le graphe élémentaire de prog est le graphe d'interférence-affinité de prog′.

La figure 2.18 présente le graphe élémentaire du programme de référence. Dans un graphe élémentaire, chaque instruction est représentée par l'intersection de deux cliques. La première de ces cliques représente les variables vivantes au point de programme qui précède l'instruction, la seconde les variables vivantes au point de programme qui suit l'instruction. Prenons comme exemple l'instruction `h := k1-1`, présente entre les points de programme p_1 et p_2. Les variables vivantes `j1`, `g0` et `k1` sont vivantes avant l'instruction (et forment une clique d'interférences) et les variables `h`, `g0` et `j1` sont vivantes après cette instruction (et forment elles aussi une clique d'interférences). L'enjeu est aujourd'hui d'exploiter cette structure spécifique afin de définir des algorithmes appropriés à ces graphes de taille parfois gigantesque.

2.5.2 Graphes éclatés

Les graphes construits selon le procédé décrit par Appel et George [AG01] sont quant à eux nommés graphes éclatés et construits de la manière suivante.

Définition 36 (Graphe éclaté d'un programme)**.** *Soit un programme prog et la transformation de programme suivante, qui transforme prog en prog′.*

Soient p_1 et p_2 deux points de programme de prog séparés par une instruction i et $v_0^1, \ldots, v_n^1$ les variables vivantes en p_1.

1. *Si aucune nouvelle variable n'est définie et si l'instruction i est de la forme $v_j^1 = v_j^1$ op v_k^1 alors toutes les variables vivantes en p_1 sont copiées en parallèle et la nouvelle variable est calculée à partir des valeurs des copies. L'instruction i est donc remplacée comme suit.*

$$\bullet p_1$$

$$v_0^2 \leftarrow v_0^1 \parallel v_1^2 \leftarrow v_1^1 \parallel \ldots \parallel v_n^2 \leftarrow v_n^1$$

$$v_j^2 := v_j^2 \ op \ v_k^2$$

$$\bullet p_2$$

2. *Sinon, une nouvelle variable v est définie. Dans ce cas, la définition de la nouvelle variable et les copies de toutes les variables vivantes en p_1 et p_2 sont effectuées en parallèle afin de ne pas introduire de fausse interférence entre la nouvelle variable et les variables vivantes en p_1 qui ne sont plus vivantes en p_2. L'instruction i est donc remplacée comme suit.*

$$\bullet p_1$$

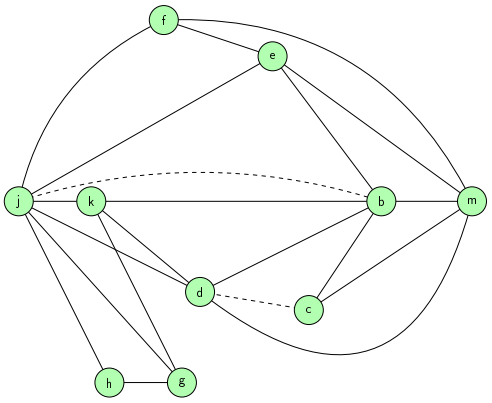
$$v_0^2 \leftarrow v_0^1 \parallel v_1^2 \leftarrow v_1^1 \parallel \ldots \parallel v_n^2 \leftarrow v_n^1 \parallel v := v_j^1 \ op \ v_k^1$$

$$\bullet p_2$$

Entrée : k j

(p_0) g := mem[j+12]
(p_1) h := k - 1
(p_2) f := g * h
(p_3) e := mem[j+8]
(p_4) m := mem[j+16]
(p_5) b := mem[f]
(p_6) c := e + 8
(p_7) d := c
(p_8) k := m + 4
(p_9) j := b
(p_{10})

Sortie : d j k



Entrée : k j

(p_0) k0 := k ‖ j0 := j
g := mem[j0+12]
(p_1) j1 := j0 ‖ g0 := g ‖ k1 := k0
h := k1 - 1
(p_2) j2 := j1 ‖ g1 := g0 ‖ h0 := h
f := g1 * h0
(p_3) f0 := f ‖ j3 := j2
e := mem[j3+8]
(p_4) e0 := e ‖ f1 := f0 ‖ j4 :=j3
m :=mem[j4+16]
(p_5) e1 := e0 ‖ m0 := m ‖ f2 := f1
b :=mem[f2]
(p_6) b0 := b ‖ m1 := m0 ‖ e2 := e1
c := e2 + 8
(p_7) b1 := b0 ‖ m2 := m1 ‖ c0 := c
d := c0
(p_8) b2 := b1 ‖ d0 := d ‖ m3 :=m2
k2 := m3 + 4
(p_9) d1 := d0 ‖ k3 := k2 ‖ b3 :=b2
j5 := b3
(p_{10})

Sortie : d1 j5 k3

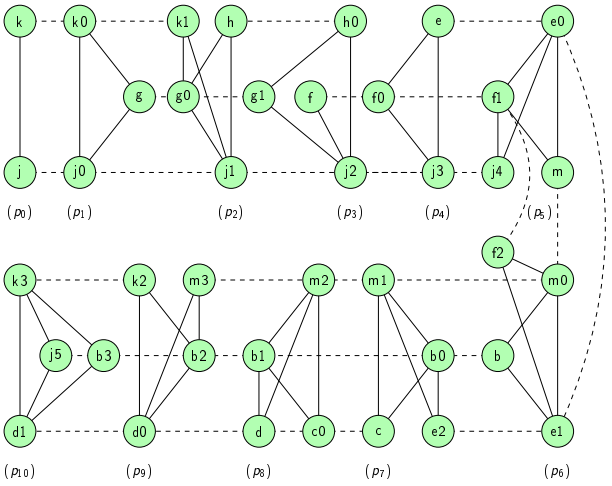
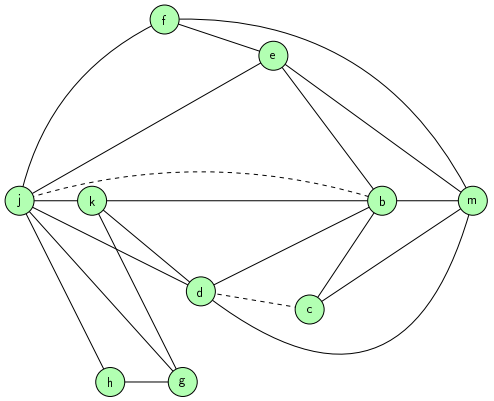


FIG. 2.18 – Le programme de référence, son graphe d'interférence-affinité, sa forme élémentaire telle que définie par Palsberg et Pereira et son graphe éclaté.

Entrée : k j
(p_0) g := mem[j+12]
(p_1) h := k - 1
(p_2) f := g * h
(p_3) e := mem[j+8]
(p_4) m := mem[j+16]
(p_5) b := mem[f]
(p_6) c := e + 8
(p_7) d := c
(p_8) k := m + 4
(p_9) j := b
(p_{10})
Sortie : d j k



Entrée : k j
(p_0) k0 := k ‖ j0 := j ‖ g := mem[j+12]
(p_1) j1 := j0 ‖ g0 := g ‖ h := k0 - 1
(p_2) j2 := j1 ‖ f := g0 * h
(p_3) f0 := f ‖ j3 := j2 ‖ e := mem[j2+8]
(p_4) e0 := e ‖ f1 := f0 ‖ m :=mem[j3+16]
(p_5) e1 := e0 ‖ m0 := m ‖ b :=mem[f1]
(p_6) b0 := b ‖ m1 := m0 ‖ c := e1 + 8
(p_7) b1 := b0 ‖ m2 := m1 ‖ d := c
(p_8) b2 := b1 ‖ d0 := d ‖ k1 := m2 + 4
(p_9) d1 := d0 ‖ k2 := k1 ‖ j4 := b2
(p_{10})
Sortie : d1 j4 k2

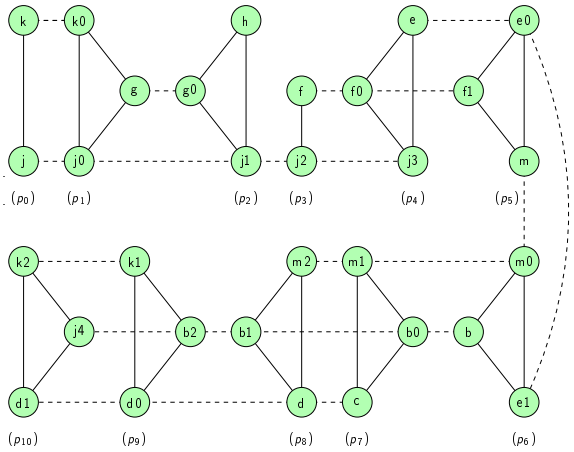


FIG. 2.19 – Le programme de référence, son graphe d'interférence-affinité, sa forme élémentaire telle que définie par Appel et George et son graphe éclaté.

Le graphe éclaté de prog est le graphe d'interférence-affinité de prog′.

La figure 2.19 présente le graphe éclaté du programme de référence. Cette fois, chaque instruction est représentée par une clique d'interférence (ce qui est prouvé plus tard dans ce manuscrit, chapitre 7) regroupant les variables vivantes en sortie de cette instruction. La définition du graphe est donc plus complexe que pour les graphes élémentaires, mais sa structure est également plus simple. L'enjeu est le même que pour les graphes élémentaires, à ceci près qu'il est certainement un peu plus facile de raisonner dans un premier temps sur les graphes éclatés. C'est d'ailleurs le leitmotiv du chapitre 7.

2.6 Bilan

Ce chapitre introduit toutes les connaissances actuelles sur lesquelles sont fondées nos démarches d'allocation de registres. Les principaux paradigmes algorithmiques utilisés pour résoudre ce problème y sont présentés avant de se concentrer sur celui auquel est largement

consacrée cette thèse : la coloration de graphe d'interférence-affinité. Nous détaillons pour ce dernier modèle en quoi consistent les étapes fondamentales de l'allocation de registres, à savoir le vidage, le découpage des durées de vie, la fusion, l'affectation des registres et l'insertion de spill code. Nous illustrons comment celles-ci peuvent être traitées au travers de la description de trois algorithmes : l'algorithme de Chaitin, l'IRC et l'algorithme de coupes de Grund et Hack. Vient ensuite une introduction à la forme SSA et aux propriétés que possèdent les graphes d'interférence-affinité de programmes sous forme SSA. La plus importante de toutes réside dans le caractère triangulé des graphes sous forme SSA. Cette propriété récemment découverte a complètement fait évoluer les problématiques d'allocation de registres, rendant obsolètes certains résultats considérés jusqu'alors immuables. Enfin, ce chapitre illustre une autre forme de programme, la forme élémentaire, conduisant à un modèle d'allocation de registres encore plus précis que la forme SSA mais également plus délicat à résoudre. Cette dernière forme, issue d'une stratégie de découpage extrême, induit des propriétés sur les graphes d'interférence-affinité que nous pensons important d'exploiter à l'image de ce qui se fait actuellement grâce à l'utilisation de la forme SSA.

Chapitre 3

Formalisation des graphes d'interférence-affinité

Formaliser en Coq des algorithmes d'allocation de registres par coloration de graphe requiert de formaliser les graphes d'interférence-affinité. Les graphes font partie de ces structures de données centrales à l'informatique pour lesquelles nombre de représentations existent, chacune ayant ses forces et ses inconvénients, mais dont la manipulation est loin d'être aisée. Cependant, aucune bibliothèque de graphes extractible n'existe dans la bibliothèque standard du système Coq , puisque les graphes sont généralement représentés comme des relations mathématiques et non comme des structures de données.

Ce chapitre introduit la formalisation des graphes d'interférence-affinité utilisée pour spécifier les algorithmes d'allocation de registres par la suite. Nous en définissons tout d'abord une axiomatisation, sous forme d'une interface contenant à la fois les données constitutives des graphes et leurs propriétés, dédiée à simplifier l'utilisation de ces graphes puis présentons une implantation de cette interface représentant les graphes d'interférence-affinité non pondérés comme le sont ceux de CompCert . Cette implantation permet de valider la cohérence de l'interface et de disposer d'une bibliothèque efficace, correcte et extractible des graphes d'interférence-affinité. Ce tout procure deux bibliothèques de graphes d'interférence-affinité : une bibliothèque Coq qui nous a permis de spécifier et prouver corrects des algorithmes d'allocation de registres par coloration de graphe, et une bibliothèque Ocaml , extraite à partir de la première, utilisable pour les développements et correcte par construction.

Le développement Coq des graphes d'interférence-affinité est consultable en ligne à l'adresse `http://www.ensiie.fr/~robillard/CoqInterferenceGraphs/`.

3.1 La structure de données

Nous définissons les graphes d'interférence-affinité dans le système Coq comme un type de module `InterferenceGraph`. La formalisation proposée colle à la définition des graphes d'interférence-affinité donnée dans ce manuscrit (cf. Définition 12, page 42), classiquement utilisée pour spécifier les graphes. Aussi, nous les représentons sous la forme d'un ensemble

de sommets et de deux ensembles d'arêtes. Cette modélisation a la force d'être très expressive et de permettre ainsi de spécifier élégamment et simplement les notions requises par la plupart des algorithmes d'allocation de registres reposant sur la coloration de graphes d'interférence-affinité. C'est selon nous la représentation abstraite la plus intuitive et la plus expressive des graphes d'interférence-affinité. C'est pour cette raison que nous l'avons choisie. Ce choix n'impose absolument pas que l'implantation suive cette voie, ce qui n'est d'ailleurs pas le cas ; la spécification et l'implantation sont totalement indépendantes. La figure 3.1 donnée à la fin de cette section est la description Coq de la partie de l'interface dédiée à la représentation des données (sommets, arêtes) présentée ci-après.

3.1.1 Les sommets

Le module `InterferenceGraph` est paramétré par un module `Vertex` représentant le type ordonné des sommets du graphe. Imposer que les sommets forment un type ordonné ne constitue pas une perte de généralité de notre spécification et permet d'utiliser les bibliothèques efficaces d'ensembles prédéfinies dans Coq .

La fonction `V` du module associe à tout graphe son ensemble de sommets. En outre, un prédicat permet de spécifier qu'un sommet appartient à un graphe. Ce dernier, noté $\in_s$ par la suite, est défini comme l'appartenance du sommet à l'ensemble des sommets du graphe. $v \in_s g$ si et seulement si le sommet v appartient à g.

(spec_in_graph) $v \in_s g =_{\text{def}} v \in (\texttt{V}\ g)$

Les graphes d'interférence-affinité ont la particularité de disposer de sommets précolorés. Nous supposons disposer d'une fonction booléenne `is_precolored` qui détermine si un sommet est précoloré ou non. Cette fonction est un paramètre du développement puisque le type `Vertex.t` des sommets en est lui-même un.

3.1.2 Les arêtes

Nous suivons une approche similaire pour les arêtes. Cependant celles-ci nécessitent plus de travail, puisqu'il faut d'abord définir le module `Edge` qui les représente, alors que le module `Vertex` est un paramètre du développement. Comme pour les sommets, nous avons besoin que les arêtes forment un type ordonné afin de pouvoir utiliser les bibliothèques d'ensembles.

Une arête est représentée par un triplet `(x,y,w)` où `x` et `y` sont les extrémités de l'arête et `w` est le poids optionnel de l'arête. Nous définissons une égalité sur les arêtes permettant de considérer indifféremment les arêtes `(x,y,w)` et `(y,x,w)`. Pour ce faire, nous passons par une forme normale des arêtes. Une arête `(x,y,w)` est en forme normale si `x` est plus petit que `y`, ce que l'on note $x <_s y$. Deux arêtes e_1 et e_2 sont égales, et on note $e_1 =_a e_2$, si leurs formes normales sont égales. Nous utilisons une fonction `ordered_edge` afin de normaliser les arêtes ; celle-ci permute les extrémités de l'arête si la seconde extrémité est plus petite que la première et se comporte comme l'identité sinon.

(ordered_lt_spec) $\texttt{snd\_end}\ e <_s \texttt{fst\_end}\ e \Rightarrow \texttt{ordered\_edge}\ e = \texttt{permute}\ e$

(ordered_notlt_spec) $\neg(\texttt{snd\_end}\ e <_s \texttt{fst\_end}\ e) \Rightarrow \texttt{ordered\_edge}\ e = e$

où `fst_end` et `snd_end` sont respectivement les accesseurs à la première et à la seconde extrémité d'une arête. Un troisième accesseur, nommé `get_weight`, permet de connaître le poids d'une arête.

Afin de distinguer les interférences des affinités, nous convenons qu'une interférence n'est pas pondérée, ce qui est représenté par un poids inexistant $\emptyset$, tandis qu'une affinité de poids `x` est pondérée par un poids $\lfloor x \rfloor$. Une affinité non pondérée est considérée comme de poids nul ; elle a donc pour poids $\lfloor 0 \rfloor$. Cette convention permet de ne manipuler qu'un type de données pour les arêtes et donc de factoriser toutes les preuves valables à la fois pour les interférences et les affinités. Deux prédicats `Aff` et `Interf` permettent de spécifier si une arête est une affinité ou une interférence, en fonction de son poids.

Ceci fait, la démarche suivie pour les sommets est applicable pour les arêtes. Les fonctions `AE` et `IE` permettent de définir respectivement les affinités et les interférences, et un prédicat, noté $\in_a$ par la suite, spécifie l'appartenance d'une arête au graphe. $e \in_a g$ si et seulement si l'arête e appartient au graphe g :

(spec_in_graph_edge) $e \in_a g =_{\text{def}} e \in (\texttt{IE}\ g) \vee e \in (\texttt{AE}\ g)$

Pour compléter la spécification, la propriété (*spec_AE*) (respectivement (*spec_IE*)) modélise que `AE g` (respectivement `IE g`) contient exactement les affinités (respectivement les interférences) de `g`.

(spec_AE) $e \in (\texttt{AE}\ g) \Leftrightarrow \texttt{Aff}\ e \wedge e \in_a g$

(spec_IE) $e \in (\texttt{IE}\ g) \Leftrightarrow \texttt{Interf}\ e \wedge e \in_a g$

Le module `Edge` contient également un prédicat `Incident` $e\ v$ qui est vérifié si l'arête e est incidente à v, *i.e.* si v est l'une des extrémités de e. Ce prédicat est central à la spécification de nombreuses propriétés sur les graphes, ce qui nous a poussé à l'intégrer directement dans le module `Edge`.

(spec_incident) $\texttt{Incident}\ e\ v =_{\text{def}} (\texttt{fst\_end}\ e) =_s v \vee (\texttt{snd\_end}\ e) =_s v$

(spec_in_graph) $v \in_s g =_{\text{def}} v \in (\texttt{V}\ g)$
(ordered_lt_spec) $\texttt{snd\_end}\ e <_s \texttt{fst\_end}\ e \Rightarrow \texttt{ordered\_edge}\ e = \texttt{permute}\ e$
(ordered_notlt_spec) $\neg(\texttt{snd\_end}\ e <_s \texttt{fst\_end}\ e) \Rightarrow \texttt{ordered\_edge}\ e = e$
(edge_eq) $e =_a e' =_{\text{def}} \texttt{ordered\_edge}\ e = \texttt{ordered\_edge}\ e'$
(spec_incident) $\texttt{Incident}\ e\ v =_{\text{def}} (\texttt{fst\_end}\ e) =_s v \vee (\texttt{snd\_end}\ e) =_s v$
(spec_aff) $\texttt{Aff}\ e =_{\text{def}} \exists w, \texttt{get\_weight} =_o \lfloor w \rfloor$
(spec_interf) $\texttt{Interf}\ e =_{\text{def}} \texttt{get\_weight} =_o \emptyset$
(spec_AE) $e \in (\texttt{AE}\ g) \Leftrightarrow \texttt{Aff}\ e \wedge e \in_a g$
(spec_IE) $e \in (\texttt{IE}\ g) \Leftrightarrow \texttt{Interf}\ e \wedge e \in_a g$
(spec_in_graph_edge) $e \in_a g =_{\text{def}} e \in (\texttt{IE}\ g) \vee e \in (\texttt{AE}\ g)$

FIG. 3.1 – Récapitulatif de la spécification des sommets et des arêtes du graphe.

3.2 Les propriétés de base des graphes d'interférence-affinité

Les graphes d'interférence-affinité obéissent à des propriétés usuelles de graphes, si classiques qu'elles sont fréquemment oubliées par les développeurs et sources de nombreux bogues. C'est d'autant plus vrai lorsque, comme ici, deux types d'arêtes sont à considérer. Ces propriétés constituent des invariants pour les graphes ; toute opération modifiant un graphe doit les préserver. La figure 3.2 fournie en fin de section regroupe les spécifications Coq de ces trois propriétés.

La première de celles-ci stipule que les extrémités de toute arête du graphe doivent appartenir au graphe :

(edge_end) $e \in_a g \Rightarrow (\texttt{fst\_end}\ e) \in_s g \wedge (\texttt{snd\_end}\ e) \in_s g$

La deuxième propriété indique que le graphe est sans boucle, *i.e.* qu'aucun sommet n'est lié à lui-même par une arête. Cette propriété est particulièrement importante dans notre cas, puisque l'existence d'une telle configuration peut induire qu'aucune coloration du graphe n'existe, si l'arête en question est une interférence. S'il s'agit d'une affinité, cette arête ne recèle pas d'intérêt non plus. Il est donc préférable d'interdire une telle éventualité dans notre périmètre d'étude. Des graphes autres que des graphes d'interférence-affinité peuvent avoir besoin de telles arêtes.

(edge_diff) $e \in_a g \Rightarrow (\texttt{snd\_end}\ e) \neq_s (\texttt{fst\_end}\ e)$

La troisième propriété est plus délicate à écrire. Cette dernière spécifie que deux sommets du graphe ne peuvent être liés que par au plus une arête. En effet, deux interférences sont équivalentes à une seule ; deux affinités sont remplacées par une seule affinité dont le poids est la somme des deux ; deux sommets ne peuvent être liés à la fois par une affinité et une interférence puisque l'interférence a la priorité sur l'affinité.

La propriété peut être reformulée sous une forme plus simple à exprimer : deux arêtes ayant les mêmes extrémités doivent être égales. L'utilisation d'ensembles permet ensuite d'assurer que deux arêtes égales n'ont qu'un représentant dans le graphe. Afin de clarifier et alléger la spécification, nous avons recours à une notion d'égalité faible d'arêtes pour la définir. Deux arêtes e_1 et e_2 sont faiblement égales, et on note $e_1 \simeq e_2$, si elles ont les mêmes extrémités. D'où la spécification :

(simple_graph) $e_1 \in_a g \wedge e_2 \in_a g \wedge e_1 \simeq e_2 \Rightarrow e_1 =_a e_2$

(edge_end) $e \in_a g \Rightarrow (\texttt{fst\_end}\ e) \in_s g \wedge (\texttt{snd\_end}\ e) \in_s g$
(edge_diff) $e \in_a g \Rightarrow (\texttt{snd\_end}\ e) \neq_s (\texttt{fst\_end}\ e)$
(simple_graph) $e_1 \in_a g \wedge e_2 \in_a g \wedge e_1 \simeq e_2 \Rightarrow e_1 =_a e_2$

FIG. 3.2 – Définition Coq des trois invariants des graphes d'interférence-affinité.

3.3 Relation de voisinage et degrés des sommets

Viennent ensuite les notions de voisinage, de degré, de sommet de bas degré et de sommet lié par au moins une affinité. La figure 3.3 décrite en fin de section est la spécification de toutes ces notions dans l'interface des graphes d'interférence-affinité.

A partir des arêtes du graphe, il est facile de définir l'interférence et la liaison d'affinité de deux sommets comme l'existence d'une interférence ou d'une affinité liant ces sommets.

(spec_interfere) `Interfere` $x\ y\ g =_{\text{def}} (x, y, \emptyset) \in_a g$

(spec_prefere) `Prefere` $x\ y\ g =_{\text{def}} \exists w, (x, y, \lfloor w \rfloor) \in_a g$

Si ces prédicats permettent de spécifier aisément nombre de propriétés des graphes d'interférence-affinité, ils ne se montrent pas très pratiques d'un point de vue plus calculatoire : les algorithmes d'allocation de registres s'appuient plus souvent sur la notion de voisinage. L'IRC utilise par exemple les sommets de bas degré, *i.e.* les sommets dont le voisinage d'interférence contient moins de K sommets, et les sommets sans affinité, *i.e.* dont le voisinage d'affinité est vide.

Nous spécifions donc le voisinage d'interférence d'un sommet v dans un graphe g, noté $N(v, g)$ et son degré d'interférence, noté $\delta(v, g)$. $\delta(v, g)$ est défini comme le cardinal de $N(v, g)$. La fonction de calcul du cardinal est fournie par la bibliothèque sur les ensembles. Le voisinage d'affinité et le degré d'affinité, quant à eux notés $N_a(v, g)$ et $\delta_a(v, g)$, sont définis de façon analogue.

(in_interf) $x \in N(y, g) \Leftrightarrow$ `Interfere` $x\ y\ g$

(in_aff) $x \in N_a(y, g) \Leftrightarrow$ `Prefere` $x\ y\ g$

Les notions de sommet de bas degré et de sommet lié par une affinité sont alors axiomatisables. Celles-ci sont représentées via des fonctions booléennes `has_low_degree` et `move_related`, et non des prédicats. Ce choix permet de les utiliser par la suite au sein des algorithmes afin de décider si un sommet est de bas degré ou non, ou si un sommet est lié par une affinité ou non.

Un sommet est de bas degré si son degré d'interférence est strictement inférieur à K.

(spec_low_degree) `has_low_degree` $g\ K\ x = true \Leftrightarrow$ `interf_degree` $g\ v < K$

La fonction `move_related` pourrait être définie de façon analogue, comme suit.

(move_related_abort) `move_related` $g\ x = true \Leftrightarrow$ `aff_degree` $g\ v < 1$

Cette définition a l'intérêt d'être similaire à celle de la fonction `has_low_degree`. Les deux fonctions pourraient alors être abstraites en une même fonction, dégageant ainsi une factorisation de développement et de preuves. Cette voie est séduisante mais a été mise de côté au profit d'une définition plus efficace de la fonction `move_related` puisque celle-ci est susceptible d'être appelée à maintes reprises. La solution adoptée consiste à vérifier si le voisinage d'affinité du sommet considéré est non vide, ce qui est bien plus rapide que de calculer son cardinal et le comparer avec 1.

(spec_move_related) `move_related` $g\ x = true \Leftrightarrow N(v, g) \neq \emptyset$

(spec_interfere) `Interfere` $x\ y\ g =_{\text{def}} (x, y, \emptyset) \in_a g$
(spec_prefere) `Prefere` $x\ y\ g =_{\text{def}} \exists w, (x, y, \lfloor w \rfloor) \in_a g$
(in_interf) $x \in N(y, g) \Leftrightarrow$ `Interfere` $x\ y\ g$
(in_aff) $x \in N_a(y, g) \Leftrightarrow$ `Prefere` $x\ y\ g$
(spec_aff_degree) `aff_degree` $g\ v =$ `cardinal` $N_a(v, g)$
(spec_interf_degree) `interf_degree` $g\ v =$ `cardinal` $N(v, g)$
(spec_low_degree) `has_low_degree` $g\ K\ x = true \Leftrightarrow$ `interf_degree` $g\ v < K$
(spec_move_related) `move_related` $g\ x = true \Leftrightarrow N(g, v) \neq \emptyset$

FIG. 3.3 – Partie de l'interface des graphes d'interférence-affinité liée aux notions de voisinage, de degré, de sommets de bas degré et de sommets liés par une affinité.

3.4 Les fonctions de modification des graphes

Notre interface spécifie les fonctions sur les graphes les plus couramment utilisées lors de l'allocation de registres. Celles-ci sont au nombre de trois :

1. Une fonction `remove_vertex` qui supprime un sommet d'un graphe ;
2. Une fonction `delete_affinities` qui supprime toutes les affinités incidentes à un sommet ;
3. Une fonction `merge` qui fusionne deux sommets liés par une affinité.

3.4.1 Suppression d'un sommet

La fonction `remove_vertex` prend comme paramètres un sommet v et un graphe g et renvoie le graphe g' résultant de la suppression de v dans g. Nous axiomatisons cette fonction via deux propriétés.

(In_remove_vertex) $x \in_s g' \Leftrightarrow x \in_s g \wedge x \neq_s v$
(In_remove_edge) $e \in_a g' \Leftrightarrow (e \in_a g \wedge \neg$`Incident` $e\ v)$

La propriété **(In_remove_vertex)** indique que les sommets de g' sont ceux de g, moins v. La propriété **(In_remove_edge)** est le pendant de la propriété **(In_remove_vertex)** pour les arêtes : les arêtes de g' sont celles de g, moins celles incidentes à v.

3.4.2 Suppression des affinités incidentes à un sommet

La fonction `delete_affinities` prend comme paramètres un sommet v et un graphe g et renvoie le graphe g' résultant de la suppression des affinités incidentes à v dans g. Nous axiomatisons cette fonction via deux propriétés également.

(In_delete_affinities_vertex) $x \in_s g' \Leftrightarrow x \in_s g$
(In_delete_affinities_edge) $e \in_a g' \Leftrightarrow e \in_a g \wedge \neg($`Aff` $e \wedge$ `Incident` $e\ v)$

La propriété **(In_delete_affinities_vertex)** stipule que l'ensemble des sommets du graphe est stable pour l'opération `delete_affinities`. Son équivalent pour les arêtes, la propriété **(In_delete_affinities_edge)**, indique que les arêtes de g' sont les arêtes de g qui ne sont pas des affinités incidentes à v.

3.4.3 Fusion de deux sommets liés par une affinité

La fonction de l'interface la plus délicate à spécifier est la fonction `merge`. Si $e = (x, y, w)$ est une affinité du graphe g, `merge` $e\, g$ retourne le graphe g' où x et y ont été fusionnés en un seul sommet. Nous considérons arbitrairement que x est le sommet résultant de la fusion. Cette décision n'a aucune incidence sur l'évolution du graphe mais se répercute sur la spécification de la fonction. Décider que le sommet résultant de la fusion est y conduirait à une spécification analogue, où x et y échangeraient leur rôles. Les sommets de g' sont donc les sommets de g, moins y. Cette opération requiert de modifier les extrémités des arêtes incidentes à y et de les rediriger vers x. Pour ce faire, nous introduisons une transformation d'arêtes appelée *redirection*. Intuitivement, la fonction `merge` transforme toute arête incidente à y en sa redirection vers x dans g', puis supprime le sommet y.

Plus formellement, soit $e' = (a, b, w)$ une arête. La redirection de e' de c à d, dénotée par $e'_{[c \to d]}$ est l'arête telle que toute occurrence de c dans les extrémités de e' est remplacée par d. Nous ne considérons cependant pas le cas où les deux extrémités de e' sont c, puisque les graphes d'interférence-affinité sont sans boucle. $e'_{[c \to d]}$ est donc définie comme suit :

1. $(a, b)_{[c \to d]} =_{\text{def}} (d, b)$ si $a =_s c$
2. $(a, b)_{[c \to d]} =_{\text{def}} (a, d)$ si $b =_s c$
3. $(a, b)_{[c \to d]} =_{\text{def}} (a, b)$ si $a \neq_s c \wedge b \neq_s c$

La fonction `merge` appliquée à l'affinité (x, y, w) et au graphe g est finalement spécifiée comme suit :

(In_merge_vertex) $v \in_s g' \Leftrightarrow v \in_s g \wedge v \neq_s y$

(In_merge_interf_edge) $e' \in (\texttt{IE}\ g) \Rightarrow e'_{[y \to x]} \in (\texttt{IE}\ g')$

(In_merge_aff_edge) $e' \in (\texttt{AE}\ g) \wedge \neg \texttt{Interfere}\ (\texttt{fst\_end}\ e'_{[y \to x]})(\texttt{snd\_end}\ e'_{[y \to x]})\, g' \wedge$
$e \neq e' \Rightarrow \texttt{Prefere}\ (\texttt{fst\_end}\ e'_{[y \to x]})\, (\texttt{snd\_end}\ e'_{[y \to x]})\, g'.$

(In_merge_edge_inv) $e' \in_a g' \Rightarrow \exists e'' \in_a g, e' \simeq e''_{[y \to x]} \wedge \texttt{same\_type}\ e'\ e''$

La propriété **(In_merge_vertex)** indique que les sommets de g' sont ceux de g, moins y. Les propriétés **(In_merge_interf_edge)** et **(In_merge_aff_edge)** axiomatisent l'évolution des arêtes : toute interférence est transformée en sa redirection ; une affinité est transformée en sa redirection si les extrémités de la redirection n'interfèrent pas. Il est nécessaire de prendre cette précaution afin de conserver la propriété **(simple_graph)** (cf. figure 3.2, page 68). La figure 3.4 présente les deux configurations dans lesquelles la redirection d'une affinité est éclipsée par une interférence.

Enfin, la propriété **(In_merge_inv)** spécifie qu'une arête de g' est faiblement égale à la redirection d'une arête de g. L'usage de l'égalité faible permet de factoriser toutes les configurations aboutissant à la présence d'une affinité dans le graphe. En effet, connaissant

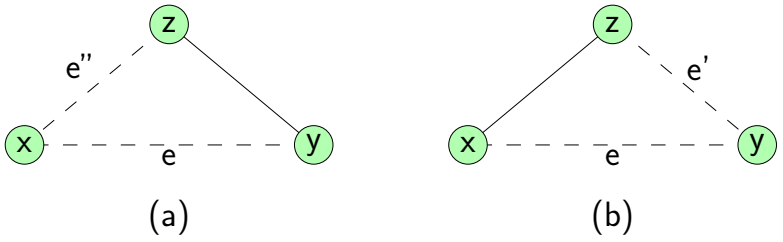


FIG. 3.4 – Deux configurations menant à la disparition d'une affinité au profit d'une interférence après fusion de x et y. Avant fusion x et z sont liés par une affinité dans le cas (a) et par une interférence après fusion. Dans le cas (b), ce sont y et z qui sont liés par une affinité qui est éclipsée par l'interférence entre x et z après la fusion.

uniquement g', il est impossible de savoir si l'affinité présente dans g' est une ancienne affinité de g, une affinité de g qui a été redirigée, ou une affinité résultant de l'addition d'une affinité et de la redirection d'une seconde affinité. Les trois possibilités sont présentées figure 3.5. Le prédicat `same_type a e'` précise que a et e sont soit toutes deux des affinités, soit toutes deux des interférences.

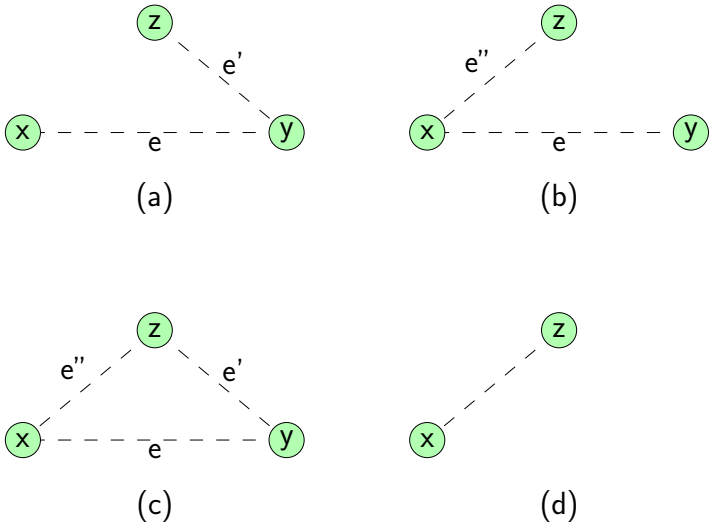


FIG. 3.5 – Trois configurations (a), (b) et (c) menant à une configuration identique (d) après fusion de x et y.

Nous avons évoqué que la fonction `merge` est appliquée à une affinité du graphe. Ainsi, l'application de la fonction `merge` à une arête e dans un graphe g requiert une preuve que e appartient à g et une preuve que e est une affinité, d'où son type dépendant donné ci-dessous.

```
Parameter merge :
∀ (e : Edge.t) (g : t), (In_graph_edge e g) → (Aff e) → t.
```

3.5 Implantation des graphes d'interférence-affinité

Pour pouvoir extraire les graphes, il est nécessaire d'implanter une structure de données extractible respectant leur interface. C'est ce que nous avons fait, en nous limitant toutefois aux graphes non pondérés, puisque les graphes construits dans CompCert ne sont pas pondérés. Considérer les graphes pondérés aurait nécessité un travail supplémentaire au niveau de l'implantation. Il aurait en particulier fallu utiliser des structures différentes pour représenter les affinités et les interférences. Aussi, au vu de nos besoins et du temps nécessaire pour implanter les graphes pondérés, nous avons préféré nous limiter aux graphes non pondérés.

3.5.1 Choix d'implantation

Nombre de solutions sont envisageables pour implanter les graphes. Parmi les représentations possibles, les plus courantes reposent sur des ensembles de sommets et d'arêtes, des matrices d'adjacence, creuses ou non, ou des listes d'adjacence. Chacune de ces possibilités recèle des forces et des faiblesses qui doivent guider nos choix. Le contexte particulier de la vérification formelle dans le système Coq induit lui aussi des spécificités dont il faut tenir compte, comme les bibliothèques disponibles.

Notre interface repose sur des ensembles de sommets et d'arêtes car c'est la représentation qui nous a semblé la plus intuitive et la plus expressive si l'on pense aux graphes d'interférence-affinité à un haut niveau d'abstraction. Cependant, ce n'est pas le choix que nous avons retenu pour l'implantation. L'usage intensif des voisinages de sommets et l'aspect local des fonctions utilisées dans les algorithmes d'allocation de registres, au sens où seule une partie très restreinte du graphe est modifiée à chaque appel de fonction intermédiaire, indiquent que les voisinages et degrés d'un sommet doivent pouvoir être consultés et modifiés très efficacement ; ce n'est pas le cas si le graphe est représenté par des ensembles de sommets et d'arêtes. Les listes et matrices d'adjacence sont plus efficaces sur ce point. La faible densité des graphes d'interférence-affinité, *i.e.* le faible nombre moyen de voisins de chaque sommet, nous a finalement poussé à choisir les listes d'adjacence.

Afin de rendre plus efficace les consultations nous n'utilisons pas des listes d'adjacence mais des ensembles d'adjacence. Notre graphe est représenté par un ensemble de sommets et deux fonctions d'association indexées par les sommets : la première fonction d'association lie à chaque sommet du graphe ses voisins d'affinité, la seconde lie à chaque sommet ses voisins d'interférence.

3.5.2 La structure de données

Les choix présentés ci-dessus sont implantés par la structure présentée figure 3.6. Celle-ci concilie à la fois des données (`vertices`, `imap` et `amap`) et des preuves de pro-

priétés portant sur ces données (`endpoints_imap`, `endpoints_amap`, `sym_imap`, `sym_amap`, `not_eq_endpoints_imap` et `not_eq_endpoints_amap`).

```
Record t : Type := Make_Graph {
(* Les sommets du graphe *)
vertices : VertexSet.t;
(* La fonction d'association des sommets et
   leurs voisins d'interférence *)
imap : VertexMap.t VertexSet.t;
(* La fonction d'association des sommets et
   leurs voisins d'affinité *)
amap : VertexMap.t VertexSet.t;
(* Le domaine des fonctions d'association est
   l'ensemble des sommets du graphe *)
endpoints_imap : ∀ x, VertexMap.In x imap ⇔
                        VertexSet.In x vertices;
endpoints_amap : ∀ x, VertexMap.In x amap ⇔
                        VertexSet.In x vertices;
(* Les fonctions d'association sont symétriques *)
sym_imap : ∀ x y, VertexSet.In x (adj_set y imap) ⇒
                  VertexSet.In y (adj_set x imap);
sym_amap : ∀ x y, VertexSet.In x (adj_set y amap) ⇒
                  VertexSet.In y (adj_set x amap);
(* Le graphe est simple *)
simple_graph : ∀ x y, VertexSet.In x (adj_set y imap) ∧
                      VertexSet.In x (adj_set y amap) ⇒
                      False;
(* Le graphe est sans boucle *)
not_eq_endpoints_imap : ∀ x y,
                          VertexSet.In x (adj_set y imap) ⇒
                          ¬Vertex.eq x y
not_eq_endpoints_amap : ∀ x y,
                          VertexSet.In x (adj_set y amap) ⇒
                          ¬Vertex.eq x y
}.
```

FIG. 3.6 – Structure de données représentant un graphe dans notre implantation Coq.

Le champ `vertices` de la structure contient l'ensemble des sommets du graphe. Pour plus d'efficacité, nous avons choisi d'utiliser la représentation des ensembles s'appuyant sur des arbres AVL [AVL62], une version d'arbre binaires de recherche équilibrés, disponible dans le système Coq .

Le champ `imap` contient une fonction associant à chaque sommet du graphe l'ensemble de ses voisins d'interférence. Ces fonctions d'associations sont elles aussi représentées par des arbres AVL. Chaque nœud de cet arbre contient un sommet et l'arbre AVL représentant

l'ensemble des voisins de ce sommet. Le champ `amap` est le pendant de `imap` pour la relation d'affinité. Puisque nous n'utilisons que des graphes non pondérés, il n'est pas nécessaire d'utiliser une structure plus complexe afin de stocker les poids des affinités. La figure 3.7 présente un exemple de graphe et de la structure de données qui le représente.

Le champ `endpoints_imap` contient une preuve de la propriété qui stipule que les sommets ayant une image dans la fonction d'association représentative des interférences `imap` sont exactement les sommets du graphe. La preuve `endpoints_amap` établit la propriété équivalente pour la fonction d'association `amap`.

Le champ `sym_imap` contient une preuve de la propriété indiquant que les interférences sont symétriques, *i.e.* que si x est un voisin de y alors y est un voisin de x. La définition du prédicat `Symmetric` est la suivante.

```
Definition Symmetric m := ∀ x y,
VertexSet.In x (adj_set y m) ⇒ VertexSet.In y (adj_set x m).
```

Le champ `sym_amap` contient la propriété analogue pour les affinités. Notre choix s'est arrêté sur une représentation symétrique afin de simplifier la recherche du voisinage d'un sommet. En effet, il aurait été possible de choisir une représentation asymétrique, qui consisterait à associer à chaque sommet tous ses voisins plus grands que lui. En pratique, cela aurait rendu difficile la consultation des voisinages puisque trouver ses voisins plus petits serait laborieux et coûteux. Pour ces raisons, une représentation symétrique est mieux adaptée.

Le champ `simple_graph` contient une preuve que le graphe est simple. Cette propriété est en effet exigée dans la spécification de l'interface. Les fonctions d'association permettent aisément de formuler cette propriété : un sommet ne doit pas à la fois appartenir au voisinage d'affinité et au voisinage d'interférence d'un même sommet.

Enfin, le champ `not_eq_endpoints_imap` contient une preuve que les extrémités de toute interférence du graphe sont différentes. Nous la spécifions en imposant que tout voisin d'interférence ou d'affinité x d'un sommet y n'est pas égal à x. La propriété `not_eq_endpoints_imap` est la propriété analogue pour `amap`.

Au final, cette structure de données permet d'accéder au voisinage d'affinité ou d'interférence d'un sommet en temps logarithmique puisque les fonctions d'association sont représentées par des arbres AVL. Elle répond donc au critère prédominant que nous avions identifié avant de définir cette implantation, à savoir, la rapidité de calcul des voisins d'un sommet. C'est là la force principale de notre implantation et la raison de son choix.

3.5.3 Lien entre spécification et implantation

La spécification des graphes d'interférence-affinité utilise des ensembles d'arêtes, ce qui lui permet d'être très expressive, tandis que leur implantation utilise des arbres AVL, ce qui confère à cette dernière une plus grande efficacité algorithmique. Le pont reliant ces deux représentations doit être construit au moment de l'implantation des graphes d'interférence-affinité. Nous définissons dans ce but une fonction `map_to_set` capable de transformer les arbres `amap` et `imap` en ensembles d'arêtes. La spécification de cette fonction est la suivante.

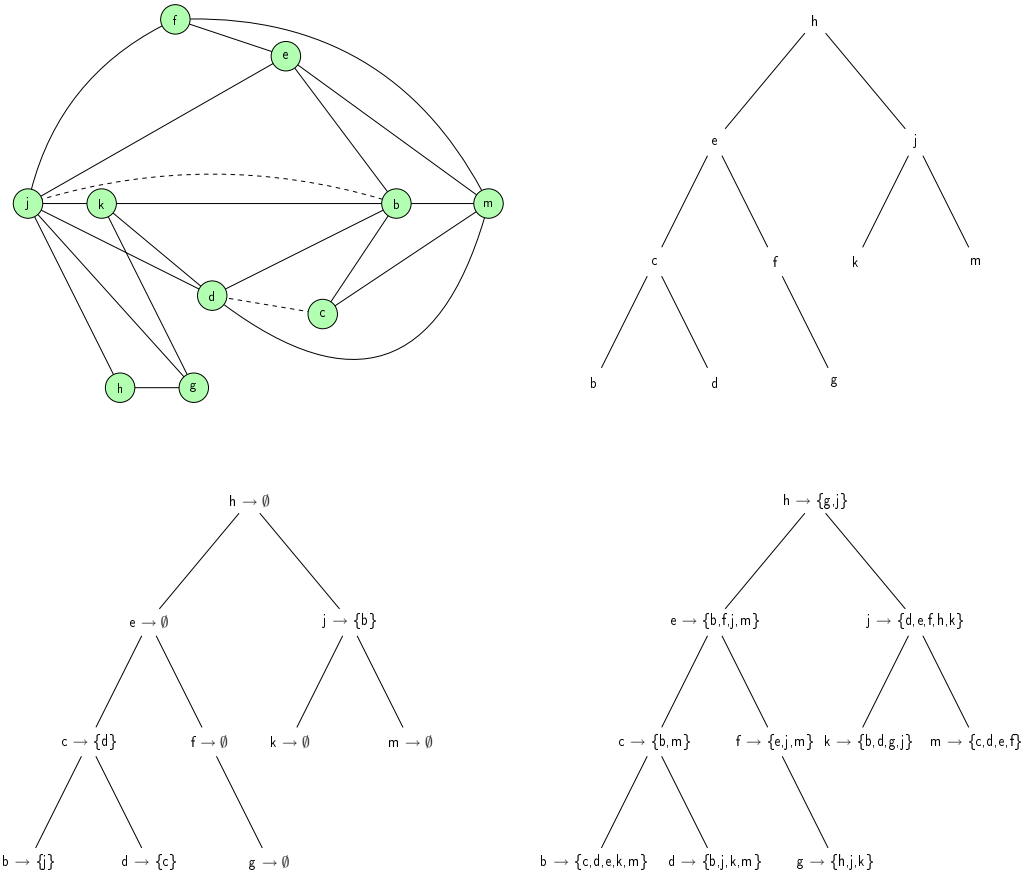


FIG. 3.7 – Un graphe (a), l'arbre AVL des sommets du graphe (b), l'arbre AVL des affinités du graphe (c) et l'arbre AVL des interférences du graphe (d). Les ensembles notés entre accolades correspondent à des ensembles qui sont également représentés par des arbres AVL, mais que nous n'avons pu exhiber sous cette forme ici pour des raisons de lisibilité.

(map_to_set_spec) `Symmetric` $m \Rightarrow$
$((x, y, w) \in (\texttt{map\_to\_set}\ m\ w) \Leftrightarrow y \in (\texttt{adj\_set}\ x\ m)))$

Cette fonction suppose que les fonctions d'association sont symétriques, ce sans quoi il est impossible de prouver l'implication de gauche à droite et donc d'établir le lien entre un graphe et son implantation. Cette supposition n'est pas limitative puisque les deux fonctions d'association auxquelles nous voulons appliquer cette fonction sont symétriques. L'implantation de cette fonction est présentée figure 3.8.

Celle-ci consiste à parcourir l'ensemble des sommets du graphe, via la fonctionnelle `VertexMap.fold`, et ajouter à chaque étape l'ensemble `adding_edges y mapy s` des arêtes incidentes au sommet courant `y` à l'ensemble ainsi déjà construit `s`.

Prouver la correction de l'implantation de la fonction `map_to_set` n'est pas chose facile, en raison de l'utilisation de la fonctionnelle `VertexMap.fold` et de ses limitations. Nous avons dû enrichir les propriétés actuelles de la bibliothèque portant sur cette fonction,

```
Definition adding_edges y mapy s w :=
VertexSet.fold (fun z s' → EdgeSet.add (y,z,w) s')
               mapy s.

Definition map_to_set map w :=
VertexMap.fold (fun y mapy s → adding_edges y mapy s)
               map EdgeSet.empty.
```

FIG. 3.8 – Implantation des fonctions `map_to_set` et `adding_edges` faisant le lien entre la structure qui implante les graphes, les fonctions d'association sous forme d'arbres AVL, et la spécification des graphes, décrite via des ensembles d'arêtes.

lesquelles n'étaient pas suffisantes pour nos raisonnements. En effet, la spécification de la fonction `fold`, donnée figure 3.9, ne contient qu'une propriété établissant le lien entre celle-ci et la fonction `fold_left` de la bibliothèque de listes.

```
Parameter fold_1 :
∀ (A : Type) (i : A) (f : key → elt → A → A),
fold f m i = fold_left (fun a p → f (fst p) (snd p) a)
                       (elements m) i.
```

FIG. 3.9 – Spécification de la fonction `fold` de la bibliothèque des fonctions d'associations (FMap). Si `m` est une fonction d'association, la liste `elements m` contient tous les couples (clé,valeur) de `m`.

Raisonner sur la fonction `VertexMap.fold` revient donc à raisonner sur la fonction `fold_left`. Le meilleur moyen de raisonner sur cette dernière est de procéder par induction sur la liste des éléments passée en paramètre. Supposons que nous devons prouver une propriété de la forme ∀ `l a, P (fold_left f l x)`, où `P` est une propriété portant sur une liste, `f` une fonction sur les listes, `l` une liste d'éléments et `x` un élément de base, appelé accumulateur. En procédant par induction sur la liste `l`, deux sous-buts sont à prouver :

1. `P(fold_left f nil x)`, qui se réduit en `P x`
2. ∀ `a l, P(fold_left f (a :: l) x)`, réductible en ∀ `a l, P (fold_left f l (f x a))`, avec l'hypothèse d'induction ∀ `l, P (fold_left f l x)`.

Le premier but est généralement simple à prouver. Le second, en revanche, est souvent difficile en raison de la modification de l'accumulateur qui empêche d'appliquer l'hypothèse d'induction. Toutefois, ce problème peut être contourné en invoquant des propriétés de commutativité et d'associativité de la fonction `f`. En particulier, si `f` est de type `A →B →A`, et vérifie la propriété ∀ `(x : A) (y z : B), f ((f x y) z) = f ((f x z) y)`, alors l'égalité `fold_left f (a :: l) x = fold_left f (fold_left f l x) a` fait apparaître le terme `fold_left f l x` ce qui permet généralement de conclure via l'hypothèse d'induction. Si l'on considère maintenant une fonction d'association qui à chaque sommet associe un ensemble de sommets (comme `imap` ou `amap`), et la fonction d'ajout d'un sommet

```
Variable A B : Type.

Variable Aeq : A → A → Prop.
Variable Aeq_refl : ∀ x, Aeq x x.
Variable Aeq_trans : ∀ x y z, Aeq x y ⇒ Aeq y z ⇒ Aeq x z.

Inductive eq_option : option A → option A → Prop :=
|None_eq : eq_option ∅ ∅
|Some_eq : ∀ s s', Aeq s s' ⇒ eq_option ⌊s⌋ ⌊s'⌋.

Definition EqualMap m1 m2 :=
∀ x, eq_option (find x m1) (find x m2).

Lemma fold_left_assoc_map : ∀ l (f : t A → B → t A) x h,
(∀ (y z : B) s, EqualMap (f (f s y) z) (f (f s z) y)) ⇒
(∀ e1 e2 a, EqualMap e1 e2 ⇒ EqualMap (f e1 a) (f e2 a)) ⇒
EqualMap (fold_left f (h :: l) x) (f (fold_left f l x) h).
```

FIG. 3.10 – Définition du lemme `fold_left_assoc_map` permettant de simplifier les raisonnements sur la fonction `fold_left`.

au voisinage de chaque sommet, alors le lemme à prouver pour exploiter ce principe est ∀ `x y`, `VertexSet.add x (VertexSet.add y s)` = `VertexSet.add y (VertexSet.add x s)`.

Malheureusement, l'usage de l'égalité de Leibniz (ou structurelle) = est trop restrictif. Ces deux ensembles sont égaux pour l'égalité extensionnelle sur les ensembles, notée $=_{vs}$, qui établit que deux ensembles sont égaux s'ils ont les mêmes éléments, mais pas pour l'égalité =. Nous parons cette complication via une égalité `EqualMap`, laquelle nous permet de définir le lemme `fold_left_assoc_map`, présenté figure 3.10, et ainsi de raisonner plus aisément sur la fonction `fold_left` et, par conséquent, sur la fonction `VertexMap.fold`. A noter qu'une propriété supplémentaire sur `EqualMap` est requise.

Nous sommes alors en mesure de prouver la correction de la fonction `map_to_set`. Une fois celle-ci établie, nous pouvons prouver les propriétés de base des graphes d'interférence-affinité décrites dans la section 3.2 grâce aux preuves intégrées à la structure de graphe. En effet, la propriété **(edge_end)** découle directement des preuves `endpoints_imap` et `endpoints_amap`, tout comme **(edge_diff)** découle de `not_eq_endpoints_imap` et `not_eq_endpoints_amap`, ou comme **(simple_graph)** découle de `simple_graph`.

3.6 Fonctions de transformation des graphes

La complexité de la structure de données, pourtant réduite aux propriétés les plus essentielles, induit également la complexité des fonctions de modification des graphes. Chacune de ces fonctions doit non seulement mettre à jour l'ensemble des sommets et les fonctions d'association des sommets et leurs voisins mais aussi établir une preuve de

chacune des propriétés du graphe. Il faut en outre prouver que l'implantation de chacune des fonctions vérifie bel et bien la spécification de cette fonction décrite dans l'interface. En conséquence l'effort de développement est ici important. L'implantation complète des graphes d'interférence-affinité représente au final environ 3600 lignes de code : 800 lignes de développement et spécification des propriétés et 2800 lignes de preuve.

3.6.1 Suppression d'un sommet

Considérons un sommet v d'un graphe g. Pour retirer v de g, il faut le supprimer de l'ensemble des sommets de g, retirer toutes les occurrences de ce sommet des deux fonctions d'association représentant les interférences et les affinités et prouver que les propriétés de la structure de données sont préservées. La spécification de la fonction `remove_map` chargée de cette tâche est donc la suivante.

(remove_map_in) `Symmetric` $m \Rightarrow$
$(y \in_{map} (\texttt{remove\_map}\ m\ x) \Leftrightarrow (y \neq_s x \wedge y \in_{map} m)$

(remove_map_adj_set) `Symmetric` $m \wedge y \neq x \Rightarrow$
$(\texttt{adj\_set}\ y\ (\texttt{remove\_map}\ m\ x)) =_{vs} (\texttt{VertexSet.remove}\ x\ (\texttt{adj\_set}\ y\ m))$

(remove_map_adj_set_v) `Symmetric` $m \Rightarrow (\texttt{adj\_set}\ x\ (\texttt{remove\_map}\ m\ x)) =_{vs} \emptyset$

En ce qui concerne l'implantation, retirer v de l'ensemble des sommets de g est simple, il suffit de le retirer d'un ensemble. Retirer toutes les arêtes incidentes à v est plus délicat, puisque les occurrences de v sont disséminées dans la structure. La façon la plus efficace de retirer toutes ces occurrences consiste à parcourir l'ensemble des voisins de v pour retirer v de l'ensemble des voisins de chacun d'eux, puis de supprimer v de la fonction d'association. Plus exactement, pour chaque voisin x de v, on associe à x l'ensemble de ses voisins, moins v, puis l'on supprime v de la fonction d'association. La mise à jour des fonctions d'association est le rôle de la fonction `remove_incident`, dont la spécification est la suivante.

(remove_incident_in) `Symmetric` $m \Rightarrow (y \in_{map} m \Leftrightarrow y \in_{map} (\texttt{remove\_incident}\ m\ x))$

(remove_incident_adj_set) `Symmetric` $m \Rightarrow$
$(\texttt{adj\_set}\ y\ (\texttt{remove\_incident}\ m\ r)) =_{vs} (\texttt{VertexSet.remove}\ r\ (\texttt{adj\_set}\ y\ m))$

Les implantations de ces deux fonctions sont détaillées figure 3.11. L'implantation de la fonction `remove_incident` suit parfaitement sa description : l'ensemble `adj_set x m` est parcouru et, pour chaque sommet `y` lui appartenant, `x` est supprimé de l'ensemble `adj_set y m`. La fonction `remove_map` se contente de supprimer le sommet `x` après avoir appliqué la fonction `remove_incident`.

Cette fonction est ensuite appliquée aux fonctions d'association `imap g` et `amap g`. Reste à prouver les propriétés nécessaires à la construction du graphe. Nous ne détaillerons pas ces preuves ici. Le corps de la fonction `remove_vertex` est donné figure 3.12. L'ensemble de sommets et les fonctions d'association sont mis à jour et les propriétés de la structure de graphes sont prouvées.

```
Definition remove_incident_adj m x adj :=
VertexSet.fold
(fun y → VertexMap.add y (VertexSet.remove x (adj_set y m)))
adj m.

Definition remove_incident m x :=
remove_incident_adj m x (adj_set x m).

Definition remove_map x m :=
VertexMap.remove x (remove_incident m x).
```

FIG. 3.11 – Implantations des fonctions `remove_incident` et `remove_map`.

```
Definition remove_vertex v g :=
Make_Graph (VertexSet.remove v (V g))
           (imap_remove (imap g) v)
           (imap_remove (amap g) v)
           (endpoints_in_remove_vertex_imap v g)
           (endpoints_in_remove_vertex_amap v g)
           (simple_graph_remove_vertex_map v g)
           (sym_imap_remove_vertex v g)
           (sym_amap_remove_vertex v g)
           (not_eq_endpoints_remove_vertex_imap v g)
           (not_eq_endpoints_remove_vertex_amap v g).
```

FIG. 3.12 – Implantation de la fonction `remove_vertex`.

3.6.2 Suppression des affinités incidentes à un sommet

La fonction `delete_affinities` se comporte à très peu de choses près comme la fonction `remove_vertex` pour supprimer les affinités du graphe. En effet, si v est le sommet dont les affinités doivent être supprimées, la première étape consiste à parcourir les voisins de v pour retirer leur lien avec v de `amap`. C'est exactement la même chose que fait la fonction `map_remove` au travers de la fonction `remove_incident`. Ensuite v est associé à l'ensemble vide dans `amap`. Cette association peut être problématique si le sommet v n'appartient pas au graphe. En effet, associer à v un ensemble de voisins, même vide, permet de prouver que v appartient au graphe, ce qui n'est pas le cas. C'est pourquoi il faut préalablement tester si v appartient ou non au graphe. Mais il est possible de faire mieux en se contentant de vérifier si le voisinage d'affinité de v dans le graphe est vide ou non. S'il l'est, alors il suffit de ne rien modifier. Sinon, le sommet v appartient forcément au graphe et on peut donc lui associer l'ensemble vide comme voisinage d'affinité sans risque. La fonction de mise à jour de la fonction d'association `amap` est décrite figure 3.13.

```
Definition amap_delete_affinities v g :=
let m := amap g in
let adj := adj_set v m in
if (VertexSet.is_empty adj) then m
else VertexMap.add v VertexSet.empty
                   (remove_incident_adj v m adj).
```

FIG. 3.13 – Implantation de la fonction de suppression des affinités incidentes à un sommet `v` de la fonction d'association représentative des affinités dans un graphe `g`.

Les preuves nécessaires à la construction du graphe sont similaires à celles de la fonction `remove_vertex`. La structure commune, reposant sur la fonction `remove_incident` permet une réutilisation du développement et surtout des preuves. L'implantation de la fonction `delete_affinities` est donnée figure 3.14.

```
Definition delete_affinities v g :=
Make_Graph (V g)
           (imap g)
           (amap_delete_affinities v g)
           (endpoints_in_delete_affinities_imap v g)
           (endpoints_in_delete_affinities_amap v g)
           (simple_graph_delete_affinities v g)
           (sym_imap_delete_affinities v g)
           (sym_amap_delete_affinities v g)
           (not_eq_endpoints_delete_affinities_imap v g)
           (not_eq_endpoints_delete_affinities_amap v g).
```

FIG. 3.14 – Implantation de la fonction `delete_affinities`.

3.6.3 Fusion de deux sommets liés par une affinité

La fonction `merge` est naturellement la plus ardue à implanter : spécification complexe implique ici implantation complexe. Soit $e = (x, y)$ l'affinité à fusionner.

La modification de l'ensemble des sommets du graphe est simple, il suffit d'en supprimer y.

La fonction de mise à jour des interférences, présentée figure 3.15, est décomposée en trois étapes :

1. Les interférences incidentes à y sont supprimées de la fonction d'association via la fonction `remove_incident` (ligne 5) ;
2. Pour tout voisin v de y, y est remplacé par x dans le voisinage de v (lignes 6-13) ;
3. Le voisinage de x est modifié et défini comme l'union des voisinages d'interférence de x et y (lignes 14-15).

```
1 : Definition imap_merge e g :=
2 : let inty := adj_set (snd_end e) (imap g) in
3 : let intx := adj_set (fst_end e) (imap g) in
4 : let addy := VertexSet.diff inty intx in
5 : let delete_snd := remove_incident (imap g) (snd_end e) in
6 : let redirect_m :=
7 : VertexSet.fold (
8 : fun y m' →
9 :   VertexMap.add
10:    y
11:    (VertexSet.add (fst_end e) (adj_set y delete_snd))
12:    m')
13: addy delete_snd
14: in VertexMap.add (fst_end e) (VertexSet.union intx addy)
15:                  (VertexMap.remove (snd_end e) redirect_m).
```

FIG. 3.15 – Définition de la fonction de mise à jour de la fonction d'association représentant les interférences du graphe au moment de la fusion d'une affinité `e`.

Raisonner sur cette fonction n'est pas aisé et assez lourd, puisque la priorité est donnée à l'efficacité algorithmique. Afin d'alléger les preuves, nous avons défini une nouvelle fonction `merge_int_set` qui calcule le voisinage d'un sommet après fusion de l'affinité `e`, puis avons prouvé que ce voisinage est égal à celui obtenu grâce à la fonction `imap_merge`, dont l'implantation, modulo quelques notations, est présentée figure 3.16. Nous raisonnons ensuite sur la fonction `imap_merge` au travers de la fonction `merge_int_set`.

Pour les affinités, la mise à jour est plus ardue puisque certaines affinités sont éclipsées par des interférences. Il ne faut donc rediriger vers x depuis y que les affinités de la forme (y, z) où z n'interfère pas avec x. De plus, il faut supprimer les affinités de la forme (x, z) où y interfère avec z. A cette modification près, le corps de la fonction `amap_merge` de mise à jour de la fonction d'association représentant les affinités est identique à celle

```
Definition merge_int_set e g v :=
let inty := adj_set (snd_end e) (imap g) in
let intx := adj_set (fst_end e) (imap g) in
let addy := VertexSet.diff inty intx in
let intv := adj_set v (imap g) in
if v == (fst_end e)
  then VertexSet.union intx addy
  else if v == (snd_end e)
    then VertexSet.empty
    else if v ∈ addy
      then VertexSet.add (fst_end e)
                         (VertexSet.remove (snd_end e) intv
      else VertexSet.remove (snd_end e) intv.

Lemma merge_int_equiv : ∀ e g v,
(adj_set v (imap_merge e g)) =vs (merge_int_set e g v).
```

FIG. 3.16 – Définition d'une fonction `merge_int_set` de calcul des voisinages après fusion d'une affinité `e`, équivalente à la fonction `imap_merge`. La fonction `RegFacts.Props.In_dec v addy` teste si `v` appartient à `y`. Le lemme `merge_int_equiv` est voué à simplifier la preuve.

décrite pour les interférences. De plus, nous utilisons également une fonction intermédiaire `merge_aff_set` équivalente à `amap_merge` pour réaliser les preuves.

Le code de la fonction `merge` finalement obtenu est donné figure 3.17.

```
Definition merge e g (Hin : In_graph_edge e g)
                     (Haff : Aff e) :=
 Make_Graph (VertexSet.remove (snd_end e) (V g))
            (imap_merge e g)
            (amap_merge e g )
            (endpoints_imap_merge e g Hin)
            (endpoints_amap_merge e g Hin)
            (simple_graph_merge e g)
            (sym_imap_merge e g Hin Haff)
            (sym_amap_merge e g Hin)
            (not_eq_endpoints_imap_merge e g Hin Haff)
            (not_eq_endpoints_amap_merge e g).
```

FIG. 3.17 – Implantation de la fonction `merge`.

3.7 Bilan

Définir une structure efficace de graphes en Coq est un problème difficile auquel nous avons répondu en deux temps.

Tout d'abord, nous avons défini une interface de ceux-ci utilisant la représentation classique, simple et expressive usuelle : un ensemble de sommets et deux ensembles d'arêtes. Cette représentation se veut modulaire, puisque le type des sommets est un paramètre des graphes. Nous avons ensuite formalisé quelques propriétés des graphes simples non orientés et les notions d'interférence, d'affinité et de voisinage d'un sommet, centrales aux algorithmes d'allocation de registres. La dernière partie de cette interface concerne les fonctions de modification usuelles de graphes d'interférence-affinité : le retrait d'un sommet, la fusion d'une affinité et la suppression des affinités incidentes à un sommet. Le tout compose une interface de taille raisonnable, extensible et modulaire des graphes d'interférence-affinité.

Dans un second temps, nous fournissons une implantation de cette interface représentant les graphes non pondérés. Cette implantation implique deux conséquences majeures :

1. l'existence d'une implantation prouve que l'interface présentée est cohérente ;
2. les structures de données utilisées sont extractibles, ce qui permet de disposer non seulement d'une bibliothèque Coq de graphes d'interférence-affinité mais aussi d'une bibliothèque Ocaml correcte par construction.

Chapitre 4

Vérification formelle de l'IRC

Le recours à la validation *a posteriori* pour la vérification formelle de l'implantation de l'IRC dans le compilateur CompCert est du à trois facteurs :

1. La description impérative de l'algorithme, inadéquate au langage purement fonctionnel de Coq ;
2. La simplicité de la validation, en opposition à la complexité de l'implantation Coq et de la preuve de correction ;
3. Le manque de bibliothèque extractible dédiée aux graphes dans le système Coq .

Implanter directement dans le système Coq cet algorithme requiert donc d'apporter des réponses à ces trois points. Le dernier point a été traité dans le chapitre précédent, les deux premiers sont l'objet de ce chapitre. Ces points sont d'autant plus épineux que l'IRC fut dans un premier temps décrit de façon incorrecte ; divers bogues furent reportés et l'algorithme corrigé en conséquence.

Nous proposons une implantation fonctionnelle de référence de l'IRC, expliquons de façon informelle en quoi cette implantation est équivalente à une version itérative et montrons formellement en Coq la correction totale de celle-ci, *i.e.* que l'algorithme termine et retourne une coloration valide du graphe d'interférence-affinité étendant la précoloration [BRA10].

Le développement Coq complet de cet algorithme est consultable en ligne à l'adresse `http://www.ensiie.fr/~robillard/IRC`.

4.1 Spécification de l'algorithme

L'IRC prend deux paramètres : un graphe d'interférence-affinité, contenant éventuellement quelques sommets précolorés, et une palette de couleurs, représentant les registres machines allouables. Le résultat est une coloration partielle du graphe d'interférence-affinité qui utilise uniquement la palette de couleurs et qui préserve la précoloration.

Plus formellement, notre spécification indique en premier lieu que la fonction qui implante l'algorithme, appelée `IRC`, retourne un résultat de type `coloring`, défini comme suit :

Definition `coloring := Vertex.t → option Color.t`

Le type `Coloring` représente les fonctions partielles des sommets vers les couleurs. Pour être une coloration partielle, cette fonction doit de plus vérifier trois prédicats. Tout d'abord, deux sommets liés par une interférence doivent avoir des couleurs différentes :

(col_1) $e \in (\texttt{IE}\ g)\ \wedge \texttt{IRC}\ g\ p\ (\texttt{fst\_end}\ e)\ =_o\ \lfloor x \rfloor \wedge \texttt{IRC}\ g\ p\ (\texttt{snd\_end}\ e)\ =_o\ \lfloor y \rfloor \Rightarrow x\ \neq_s\ y$

où $=_o$ désigne l'égalité sur le type `option Color.t` déduite de celle sur les couleurs.

Ensuite, tout sommet coloré appartient effectivement au graphe d'interférence :

(col_2) $\texttt{IRC}\ g\ p\ x\ =_o\ \lfloor y \rfloor \Rightarrow x\ \in_s\ g$

Ce prédicat n'est pas absolument nécessaire, mais il simplifie la preuve de l'algorithme par la suite et ne constitue en aucun cas une limitation, puisqu'il est toujours possible de restreindre le domaine d'une fonction aux sommets du graphe.

Enfin, toute couleur affectée à un sommet doit appartenir à la palette de couleurs :

(col_3) $\texttt{IRC}\ g\ p\ x\ =_o\ \lfloor y \rfloor \Rightarrow y \in p$

La conjonction de ces trois prédicats forme le prédicat **(proper_col)**.

En plus de construire une coloration partielle, l'algorithme doit préserver la précoloration `precoloring` g :

(precol) $\texttt{precoloring}\ g\ p\ x = \lfloor y \rfloor \Rightarrow \texttt{IRC}\ g\ p\ x\ =_o\ \lfloor y \rfloor$

En résumé, notre preuve de correction consiste à prouver que la fonction `IRC` de type `graph` →`palette` →`coloring` termine et renvoie une solution vérifiant le prédicat **(proper_col)** et la propriété **(precol)**.

4.2 Description de l'implantation

4.2.1 Point de départ

L'implantation de l'IRC dont s'inspire notre travail est une description fonctionnelle en SML d'Andrew W. Appel orientée vers la preuve, *i.e.* pensée pour être prouvée correcte (mais malgré tout erronée), qui sert de support à son cours de compilation. De là est née cette collaboration avec lui autour de la formalisation et l'implantation dans Coq de l'IRC.

L'implantation SML précitée suit de près la version originale de l'algorithme mais apporte des informations clés sur les difficultés de ce développement dans un système tel que Coq .

La première de ces informations est la définition des fonctions de simplification, de fusion, de gel et de vidage comme quatre fonctions mutuellement récursives. Une telle description n'est pas souhaitable pour implanter l'algorithme dans le système Coq puisque la preuve de terminaison s'en trouverait compliquée. Il nous faudra donc contourner cette difficulté.

La deuxième information est l'utilisation d'une fonction `getAlias` afin de déterminer quel sommet impose la couleur d'un autre après une fusion. Par exemple, si (x, y) est une affinité qui doit être fusionnée, l'implantation décide que `getAlias` $y = x$. Cependant, si (z, y) est fusionnée, les alias de y (et donc de x) deviennent z. Il s'agit donc de calculer une fermeture transitive des fusions appliquées. Cette solution introduit de nombreux appels à la fonction `GetAlias` qu'il semble ardu de tracer et qui induisent certainement des difficultés au niveau de la preuve. Nous éviterons donc de recourir à une fonction de ce type.

La troisième porte sur la mise à jour des ensembles de sommets de bas degré et des sommets liés par une affinité, lesquels sont constamment maintenus à jour durant l'algorithme. Celle-ci pourrait être à première vue améliorée, car trop de calculs sont réalisés, en particulier au moment de la fusion, dans laquelle un bogue est d'ailleurs apparu.

La quatrième est l'utilisation que fait Andrew W. Appel d'une forme normale des arêtes pour décider qu'un sommet précoloré est toujours la première extrémité d'une arête. Ce principe permet de casser des symétries, puisqu'il permet de considérer indifféremment l'arête (x, y, w) et l'arête (y, x, w), et doit donc être conservé. Il est d'ailleurs intégré à l'interface des graphes que nous avons présentée.

Ces remarques, en plus de la structure globale de l'algorithme qui semble adaptée à la preuve, permettent de partir sur des bases saines pour construire l'implantation de l'algorithme puisque les sources de difficultés principales sont d'ores et déjà repérées.

4.2.2 Ensembles de travail et structure dédiée

L'IRC repose sur quatre opérations qui sont appelées par ordre de priorité en fonction des degrés des sommets du graphe et de leurs affinités. Afin d'améliorer le temps d'exécution de l'algorithme, il est primordial de classer les sommets du graphe dans des ensembles de travail en fonction de leurs propriétés. Ces ensembles sont au nombre de quatre, un par opération possible :

1. `spillWL` est défini comme l'ensemble des sommets de haut degré non précolorés, *i.e.* l'ensemble des sommets candidats au vidage.
2. `freezeWL` est défini comme l'ensemble des sommets de bas degré, liés par une affinité et non précolorés, *i.e.* l'ensemble des sommets candidats au gel.
3. `simplifyWL` est défini comme l'ensemble des sommets de bas degré, sans affinité et non précolorés, *i.e.* l'ensemble des sommets candidats à la simplification.
4. `movesWL` est défini comme l'ensemble des affinités du graphe, *i.e.* l'ensemble des arêtes candidates à la fusion.

Les propriétés caractéristiques de ce quadruplet d'ensembles peuvent être considérées comme un invariant que nous appelons par la suite `WL_invariant`. L'efficacité et la correction de l'algorithme reposent sur le respect de cet invariant, dont la spécification formelle est la suivante.

(spec_invariant) `WL_invariant` $g\ K$ (`spillWL`,`freezeWL`,`simplifyWL`,`movesWL`) $\Leftrightarrow$
$(\forall x, x \in$ (`spillWL`) $\Leftrightarrow$ `has_low_degree` $g\ K\ x = false \wedge x \notin$ (`precolored` g)) $\wedge$
$(\forall x, x \in$ (`freezeWL`) $\Leftrightarrow$ `has_low_degree` $g\ K\ x = true \wedge$

$$\begin{array}{ll} & \texttt{move\_related}\ g\ x = true \wedge x \notin (\texttt{precolored}\ g)) \wedge \\ (\forall x, x \in (\texttt{simplifyWL}) \Leftrightarrow & \texttt{has\_low\_degree}\ g\ K\ x = true \wedge \\ & \texttt{move\_related}\ g\ x = false \wedge x\ \in_s\ g \wedge x \notin (\texttt{precolored}\ g)) \wedge \\ (\forall e, e \in (\texttt{movesWL}) & \Leftrightarrow \texttt{Aff}\ e \wedge e\ \in_a\ g) \end{array}$$

A noter que dans la spécification de l'invariant, un sommet qui appartient à l'un des quatre ensembles de travail doit appartenir au graphe. Ceci est spécifié explicitement pour l'ensemble `simplifyWL` mais est implicite pour les ensembles `spillWL`, car tout sommet de haut degré appartient au graphe, et `freezeWL`, car tout sommet lié par une affinité appartient également au graphe.

Les ensembles de travail peuvent être facilement calculés en parcourant le graphe. Ceux-ci sont ensuite incrémentalement mis à jour lors de chaque modification du graphe. Il est donc nécessaire de parcourir entièrement le graphe une seule fois, au moment de l'initialisation de l'algorithme, et non à chaque fois que le graphe est modifié, ce qui améliore significativement la rapidité d'exécution de l'algorithme.

Une fois les ensembles de travail initialisés, l'algorithme fait appel à une fonction auxiliaire dont le seul argument est une structure, appelée `irc_graph`, contenant le graphe `gph`, les ensembles de travail `wl`, la palette `pal`, le nombre de couleurs `K`, une preuve que (`WL_invariant gph pal wl`) est vérifié et une preuve que `K` est le cardinal de `pal`. Conserver `K` dans la structure n'est pas indispensable mais évite de le recalculer trop fréquemment. La définition exacte de cette structure, ainsi que sa construction sont décrites figure 4.1.

```
Record irc_graph := Make_IRC_Graph {
gph : Graph.t;
wl : WL;
pal : VertexSet.t;
k : nat;
Hwl : WL_invariant gph pal wl;
Hk : VertexSet.cardinal pal = k }.

Definition graph_to_IRC_graph g palette :=
let K := VertexSet.cardinal palette in
let wl := init_WL g K in
Make_IRC_Graph g wl palette K
        (WL_invariant_init g K wl) (refl_equal K).

Definition IRC :=
let g' := graph_to_IRC_graph g palette in (IRC_aux g').
```

FIG. 4.1 – La structure `irc_graph` et l'initialisation de l'algorithme. La structure est construite à partir du graphe et de la palette uniquement. Elle est ensuite passée comme paramètre à la fonction `IRC_aux` qui implante le corps de l'algorithme.

4.2.3 Description de l'algorithme

Bien que la description usuelle de l'IRC soit itérative, cet algorithme pourrait parfaitement être défini sous forme récursive. En fait, l'IRC est loin d'être le seul algorithme d'allocation de registres dans ce cas. La plupart des algorithmes procédant par empilement et dépilement des sommets du graphe peuvent aisément être réécrits sous une forme fonctionnelle. Cette forme est même plus condensée et plus naturelle à écrire puisqu'il n'est nullement nécessaire de gérer la pile : la récursivité assure d'elle même que l'ordre de dépilement est l'inverse de l'ordre d'empilement. Ces préceptes conduisent à l'implantation suivante :

```
1  : Function IRC_aux g {measure IRC_measure} : Coloring :=
2  : match simplify g with
3  : | ⌊(r,g')⌋ → available_color g r (IRC_aux g')
4  : | ∅       → match coalesce g with
5  :   | ⌊(e,g')⌋ → complete_col e (IRC_aux g')
6  :   | ∅       → match freeze g with
7  :     | ⌊g'⌋    → IRC_aux g'
8  :     | ∅       → match spill g with
9  :       | ⌊(r,g')⌋ → available_color g r (IRC_aux g')
10 :       | ∅       → precoloring g
11 :       end
12 :     end
13 :   end
14 : end.
```

FIG. 4.2 – Implantation Coq de la fonction `IRC_aux`.

L'en tête de la fonction `IRC_aux` utilise le mot-clé **Function** ainsi qu'une information {**measure** `IRC_measure`} pour des raisons liées à la terminaison de cette fonction. Nous reviendrons en détail sur ce point dans la section 4.6 de ce chapitre.

L'algorithme fonctionne comme suit. S'il existe un sommet de bas degré sans affinité et non precoloré alors la fonction de simplification `simplify` est appelée (lignes 2 et 3) : la simplification est réalisée, le sommet `r` soumis à la simplification et le graphe modifié `g`'sont renvoyés, la coloration de `g`' est faite via un appel récursif à `IRC` et le sommet `r` est coloré.

Sinon, s'il est possible de satisfaire une affinité alors la fonction `coalesce` de fusion est appelée (lignes 4 et 5) : la fusion est effectuée, l'affinité `e` soumise à la fusion et le graphe modifié `g`' sont renvoyés, la coloration de `g`' est calculée par appel récursif et, enfin, les deux extrémités de `e` sont colorées de la même couleur.

Sinon, s'il existe un sommet de bas degré lié par une affinité et non précoloré alors la fonction `freeze` est appelée (lignes 6 et 7) : le gel est opéré, le sommet `r` soumis au gel et le graphe modifié `g`' sont renvoyés, le sommet `r` est gelé et la coloration de `g`' est construite par appel récursif.

Sinon, s'il ne reste que des sommets de haut degré parmi les sommets non précolorés du

graphe alors l'on procède au vidage (lignes 8 et 9). Le vidage est similaire à la simplification à ceci près que la fonction de coloration `available_color` peut échouer à retourner une couleur disponible pour le sommet `r` soumis au vidage.

Enfin, s'il ne reste plus de sommet non précoloré dans le graphe alors la précoloration (`precoloring g`) est renvoyée (ligne 10).

4.3 Fonctions de mise à jour de la structure

Quatre fonctions auxiliaires sont appelées par l'algorithme afin d'évaluer les opérations possibles et, si besoin est, de mettre à jour la structure `irc_graph`. Ces fonctions sont donc partielles et renvoient toutes $\emptyset$ en cas d'échec.

(`simplify` g) qui simplifie un sommet v et retourne $\lfloor (v, g') \rfloor$, où g' est le graphe résultant du retrait de v de g.

(`coalesce` g) qui recherche une affinité e de g pouvant être satisfaite et fusionne ses extrémités, menant ainsi au graphe g', et renvoie $\lfloor (e, g') \rfloor$.

(`freeze` g) qui supprime les affinités incidentes à un sommet de bas degré lié par une affinité et non précoloré v et retourne $\lfloor g' \rfloor$, où g' désigne le graphe ainsi obtenu.

(`spill` g) qui supprime le sommet de haut degré et non précoloré v de moindre coût et renvoie $\lfloor (v, g') \rfloor$, où g' est le graphe résultant de la suppression de v de g.

Comme évoqué précédemment, chaque fonction possède deux rôles : vérifier que l'opération est possible, tâche qui se résume la plupart du temps à consulter les ensembles de travail, et mettre à jour la structure par appel à une autre fonction postfixée par `_irc`.

La figure 4.3 présente le corps de la fonction `simplify`.

```
1 : Definition simplify g : option (Register.t * irc_graph) :=
2 : let simplifyWL := get_simplifyWL (wl g) in
3 : let HWL := Hwl g in
4 : match any_vertex simplifyWL as v
    return (any_vertex simplifyWL = v ⇒
            option (Register.t * irc_graph))
    with
5 : | ⌊r⌋ → fun H : any_vertex simplifyWL = ⌊r⌋ →
6 :                 let Hinw := any_vertex_in H in
7 :               ⌊(r, simplify_irc r g Hinw)⌋
8 : | ∅ → fun H : any_vertex simplifyWL = ∅ → ∅
9 : end
10: (refl_equal (any_vertex simplifyWL)).
```

FIG. 4.3 – Définition de la fonction `simplify`.

La première étape consiste à accéder à l'ensemble de travail `simplifyWL` et à la preuve que les ensembles de travail vérifient l'invariant `WL_invariant` (lignes 2 et 3). Ensuite, l'on

recherche n'importe quel sommet `r` de l'ensemble de travail `simplifyWL` par appel à la fonction `any_vertex` (ligne 4). La construction **match** ... **as** ... **return** est utilisée afin de disposer des hypothèses de filtrage `any_vertex_simplifyWL` $= \lfloor$`r`$\rfloor$ et `any_vertex simplifyWL` $= \emptyset$.

Dans le cas où un sommet `r` est candidat à la simplification, celle-ci est opérée (lignes 5 à 7). La ligne 6 procède à la construction de l'hypothèse `Hinw`. Cette preuve est l'un des arguments passés ensuite à la fonction `simplify_irc`. Celle-ci est indispensable pour procéder à une mise à jour appropriée des ensembles de travail et prouver que l'invariant `WL_invariant` est préservé par la simplification.

Le résultat renvoyé est finalement $\lfloor$(`r`, `simplify_ irc r g Hinw`)$\rfloor$.

Si aucun n'est candidat à la simplification $\emptyset$ est renvoyé.

La fonction `simplify_irc` effectue quant à elle quatre tâches :

1. elle met à jour le graphe en appelant la fonction `remove_vertex` de notre bibliothèque ;
2. elle transfère la palette `pal`, `K` et la preuve de la propriété `Hk` indiquant que `K` est le cardinal de `pal` ;
3. elle construit incrémentalement les ensembles de travail du nouveau graphe en fonction de ceux de l'ancien ;
4. elle prouve que le nouveau graphe et les nouveaux ensembles de travail sont liés par l'invariant `WL_invariant`.

La figure 4.4 présente le corps de la fonction `simplify_irc`, qui est chargée de cette tâche pour le cas de la simplification.

```
Definition simplify_irc r ircg Hinw :=
Make_IRC_Graph (remove_vertex r (gph ircg))
               (simplify_wl r ircg (k ircg))
               (pal ircg)
               (k ircg)
               (Inv_simplify_wl r ircg Hinw)
               (Hk ircg).
```

FIG. 4.4 – Définition de la fonction `simplify_irc`.

Les arguments de `simplify_irc` sont le sommet `r` à simplifier, un `irc_graph ircg`, et une preuve `Hinw` que `r` appartient à l'ensemble des candidats à la simplification dans `g`. A partir de tout cela, le sommet `r` est supprimé du graphe par appel à la fonction `remove_vertex`, les ensembles de travail sont mis à jour via une fonction `simplify_wl` et les autres données sont transmises inchangées. Une preuve que les nouveaux ensembles de travail vérifient l'invariant pour le nouveau graphe est construite par la fonction `Inv_simplify_wl` et la preuve que `K` est le cardinal de `pal` est simplement transmise. Le tout forme l'`irc_graph` résultant de la simplification de `r` dans `ircg`. Les fonctions `simplify_wl` et `Inv_simplify_wl`, dont les définitions sont complexes, seront l'objet de la partie 4.5 de ce chapitre.

La décomposition et la forme des fonctions auxiliaires pour les trois autres opérations sont analogues.

4.4 Fonctions de mise à jour de la coloration

La phase de dépilement de l'algorithme correspond à la coloration incrémentale du graphe, *i.e.* à l'affectation des registres. Initialement, la coloration est réduite à la précoloration imposée par les contraintes architecturales. Ensuite, l'algorithme colore les sommets un à un jusqu'à ce que tous soient colorés ou vidés en pile. La coloration est construite grâce à trois fonctions.

La fonction `precoloring` associe à chaque sommet du graphe précoloré sa couleur. Il est nécessaire que cette précoloration soit correcte pour qu'elle puisse être étendue par l'algorithme en une coloration correcte.

La fonction `available_color` recherche une couleur qu'il est possible d'affecter à un sommet donné afin d'étendre une coloration fixée. Cette fonction est partielle puisqu'elle peut échouer à trouver une nouvelle couleur, dans le cas où toutes les couleurs de la palette sont interdites. La correction de cette fonction tient en un lemme :

(available_coloring_spec) $\texttt{proper\_col}\ col\ g\ pal \wedge v \in_s g \Rightarrow \texttt{proper\_col}\ (\texttt{available\_color}\ g\ v\ col)\ g\ pal$

La dernière fonction de construction de la coloration est la fonction `complete_col` qui permet d'assigner aux deux extrémités d'une affinité choisie pour être fusionnée la même couleur. La correction de cette fonction tient en deux lemmes :

(complete_coloring_snd) $e \in_a g \Rightarrow (\texttt{complete\_col}\ e\ col)\ (\texttt{snd\_end}\ e) =_o (\texttt{complete\_col}\ e\ col)\ (\texttt{fst\_end}\ e)$

(complete_coloring_any) $\texttt{snd\_end}\ e \neq_s x \Rightarrow (\texttt{complete\_col}\ e\ col)\ x =_o col\ x$

Contrairement à ce qui aurait pu être attendu, la correction de cette fonction ne spécifie pas que la coloration résultante doit être correcte. Ceci est tout à fait normal : la fonction se contente de compléter la coloration si une arête a été désignée pour être fusionnée ; le fait que la coloration ainsi obtenue soit correcte ou non dépend uniquement du critère de fusion employé.

4.5 Mise à jour des ensembles de travail

Le cœur de l'IRC se situe dans la mise à jour des ensembles de travail et la preuve de préservation de l'invariant. Notre implantation est pensée pour mettre à jour efficacement et rapidement les ensembles de travail, en n'opérant que les modifications strictement nécessaires, contrairement à la définition de référence actuelle de l'algorithme [App98a] qui fait une mise à jour complète des voisinages des sommets touchés par les opérations : par exemple, si le sommet v est retiré du graphe alors tous ses voisins de bas degré (après suppression de v) sont examinés et reclassés. Comme nous le verrons, certains de ces voisins vont être retirés d'un ensemble de travail pour y être insérés à nouveau. Nous désirons interdire ce type d'opération inutile.

Cette partie détaille comment les mises à jour des ensembles de travail sont réalisées pour chaque opération et prouve leur correction. Cette démarche se fait à chaque fois en

trois temps. Tout d'abord nous définissons l'évolution des voisinages d'affinité et d'interférence des sommets pour chaque opération. Ensuite, nous déduisons de l'évolution des voisinages l'évolution des fonctions caractéristiques que sont les fonctions `has_low_degree` et `move_related`. Ce sont en effet en fonction des résultats de ces fonctions que les sommets sont classés. Enfin, nous décrivons comment sont construits les nouveaux ensembles de travail à partir des anciens.

4.5.1 Simplification et vidage

La simplification et le vidage sont des opérations équivalentes en termes de graphe : un sommet est désigné pour être supprimé. Ce point commun permet de factoriser toute une partie du développement même si l'évolution des ensembles de travail diffère. Nous considérons par la suite un graphe g, un sommet v de g et $g' = \texttt{remove\_vertex}\ v\ g$.

4.5.1.1 Evolution des voisinages des sommets

Le voisinage d'interférence (respectivement d'affinité) de tout sommet différent de v dans g' est obtenu par suppression de v dans le voisinage de ce sommet dans g.

Lemme 2 (Voisinage des sommets)**.** *Soit* $x \neq v$.
$N(x, g') = N(x, g) - \{v\}$.
$N_a(x, g') = N_a(x, g) - \{v\}$.

Ce résultat induit deux corollaires immédiats pour les voisinages, en fonction de si v appartient au voisinage du sommet considéré ou non.

Si x est un voisin de v alors son voisinage dans g' est obtenu par suppression de v de son voisinage dans g.

Corollaire 1 (Voisinage des voisins de v)**.** *Soit* $x \neq v$.
$x \in N(v, g) \Rightarrow N(x, g') = N(x, g) - \{v\}$.
$x \in N_a(v, g) \Rightarrow N_a(x, g') = N_a(x, g) - \{v\}$.

Si x n'est pas un voisin de v alors son voisinage dans g' est son voisinage dans g.

Corollaire 2 (Voisinage des autres sommets)**.** *Soit* $x \neq v$.
$x \notin N(v, g) \Rightarrow N(x, g') = N(x, g)$.
$x \notin N_a(v, g) \Rightarrow N_a(x, g') = N_a(x, g)$.

Ces corollaires induisent eux-mêmes deux nouveaux corollaires pour les degrés.

Si x est un voisin de v alors son degré dans g' est obtenu en retranchant 1 de son degré dans g.

Corollaire 3 (Degrés des voisins de v)**.** *Soit* $x \neq v$.
$x \in N(v, g) \Rightarrow \delta(x, g') = \delta(x, g) - 1$.
$x \in N_a(v, g) \Rightarrow \delta_a(x, g') = \delta_a(x, g) - 1$.

Si x n'est pas un voisin de v alors son degré dans g' est égal à son degré dans g.

Corollaire 4 (Degrés des autres sommets)**.** *Soit* $x \neq v$.
$x \notin N(v,g) \Rightarrow \delta(x,g') = \delta(x,g)$.
$x \notin N_a(v,g) \Rightarrow \delta_a(x,g') = \delta_a(x,g)$.

4.5.1.2 Evolution des fonctions caractéristiques

Une fois l'évolution des degrés parfaitement comprise, l'évolution des fonctions caractéristiques `has_low_degree` et `move_related` devient aisée.

Tout sommet sans affinité $x \neq v$ de g est sans affinité dans g'.

Lemme 3 (Sommets sans affinités)**.** *Soit* $x \neq v$.
`move_related` $g\ x\ =\ false\ \Rightarrow$ `move_related` $g'\ x\ =\ false$

Tout sommet lié par une affinité $x \neq v$ de g est sans affinité dans g' si et seulement s'il a pour seul voisin d'affinité v.

Lemme 4 (Sommets devenant sans affinité)**.** *Soit* $x \neq v$.
(`move_related` $g\ x\ =\ true\ \wedge$ `move_related` $g'\ x\ =\ false$) $\Leftrightarrow$
$x \in\ N_a(v,g)\ \ \wedge\ \ \delta_a(x,g) = 1$.

Tout sommet de bas degré $x \neq v$ de g est également de bas degré dans g'.

Lemme 5 (Sommets de bas degré)**.** *Soit* $x \neq v$.
`has_low_degree` $g\ K\ x\ = true\ \Rightarrow$ `has_low_degree` $g'\ K\ x\ =\ true$

Tout sommet de haut degré $x \neq v$ de g est de bas degré dans g' si et seulement s'il est un voisin de v de degré d'interférence K.

Lemme 6 (Sommets devenant de bas degré)**.** *Soit* $x \neq v$.
`has_low_degree` $g\ K\ x\ =\ true\ \wedge$ `has_low_degree` $g'\ K\ x\ =\ false\ \Leftrightarrow$
$x \in\ N(v,g)\ \ \wedge\ \ \delta(x,g) = K$.

4.5.1.3 Evolution des ensembles de travail

Nous sommes maintenant en mesure d'écrire l'algorithme de mise à jour des ensembles de travail puisque le comportement des fonctions caractéristiques est clairement défini. Cette mise à jour demeure néanmoins quelque peu complexe.

Soit $wl = (\texttt{spillWL}, \texttt{freezeWL}, \texttt{simplifyWL}, \texttt{movesWL})$ un quadruplet d'ensembles de travail tel que l'invariant (`WL_invariant` g *palette* wl) est vérifié. Afin d'alléger la suite, nous notons $IN(v,g)$ l'ensemble des voisins d'interférence de v non précolorés et qui ont un degré égal à K.

(def_IN) $\ x\ \in\ IN(v,g)\ \Leftrightarrow x \in N(v,g) \wedge \delta(x,g) = K \wedge$ `is_precolored` $x\ =\ false$

Ces sommets correspondent à ceux qui doivent être transférés de l'ensemble des candidats au vidage vers les ensembles de candidats à la simplification ou au gel, puisqu'ils sont de haut degré dans g mais de bas degré dans g', comme le stipule le lemme 6. Ceux liés par une affinité dans g' doivent être classés parmi les candidats au gel, les autres étant

classés parmi les candidats à la simplification. Pour effectuer ce classement, nous définissons $(IN_{mr}(v,g), IN_{nmr}(v,g))$ comme la partition des sommets de $IN(g)$ telle que $IN_{mr}(v,g)$ et $IN_{nmr}(v,g)$ contiennent respectivement les sommets liés par une affinité dans g et les sommets sans affinité dans g.

(def_INmr) $x \in IN_{mr}(v,g) \Leftrightarrow x \in IN(g) \wedge \texttt{move\_related}\ x\ g = true$

(def_INnmr) $x \in IN_{nmr}(v,g) \Leftrightarrow x \in IN(g) \wedge \texttt{move\_related}\ x\ g = false$

Nous définissons de façon analogue des ensembles spécifiques pour les sommets devenant sans affinité dans g' alors qu'ils étaient liés par une affinité dans g. $PN(v,g)$ désigne l'ensemble des voisins d'affinité de v, de bas degré, non précolorés et de degré d'affinité égal à 1 dans g. Ces sommets devront être déplacés de l'ensemble des candidats au gel à l'ensemble des candidats à la simplification.

(def_PN) $x \in PN(g) \Leftrightarrow x \in N_a(v,g) \wedge \delta_a(x,g) = 1 \wedge \texttt{is\_precolored}\ x = false$

Le quadruplet d'ensembles de travail $wl' = (\texttt{spillWL}', \texttt{freezeWL}', \texttt{simplifyWL}', \texttt{movesWL}')$ résultant de la suppression du sommet v de g peut finalement être calculé à partir du quadruplet $wl = (\texttt{spillWL}, \texttt{freezeWL}, \texttt{simplifyWL}, \texttt{movesWL})$ d'ensembles de travail de g par la séquence d'étapes suivante :

1. Les sommets de $IN(v,g)$ sont supprimés de `spillWL`.
2. Les sommets de $IN_{mr}(v,g)$ sont ajoutés à `freezeWL`.
3. Les sommets de $IN_{nmr}(v,g)$ sont ajoutés à `simplifyWL`.
4. Les sommets de $PN(v,g)$ sont supprimés de l'ensemble des candidats au gel résultant de l'étape 2, et ajoutés à l'ensemble de candidats à la simplification obtenu à l'étape 3.
5. Les affinités incidentes à v sont supprimées de `movesWL`.
6. Le sommet v est supprimé de l'ensemble auquel il appartient.

La correction de ce processus réside dans le théorème suivant.

Théorème 6. `WL_invariant` g' *palette* wl'.

4.5.1.4 Instantiation pour la simplification et le vidage

La procédure proposée dans le paragraphe précédent fonctionne lorsque n'importe quel sommet est retiré du graphe. Elle peut cependant être améliorée selon que le sommet à supprimer est candidat à la simplification ou au vidage. Dans le premier cas, nous savons que le sommet v appartient à l'ensemble des candidats à la simplification ; il est par conséquent non précoloré, de bas degré et sans affinité. Dans le second cas, v est candidat au vidage ; il est de haut degré et non précoloré.

Le cas le plus intéressant est celui de la simplification. Puisque v est dans ce cas sans affinité, l'ensemble $PN(v,g)$ est vide, ce qui supprime l'opération 4 du processus décrit ci-dessus. De plus, v appartient à l'ensemble des candidats à la simplification, ce qui permet de préciser la dernière opération. Le processus devient donc :

1. Les sommets de $IN(v, g)$ sont supprimés de `spillWL`.
2. Les sommets de $IN_{mr}(v, g)$ sont ajoutés à `freezeWL`.
3. Les sommets de $IN_{nmr}(v, g)$ sont ajoutés à `simplifyWL`.
4. Les affinités incidentes à v sont supprimées de `movesWL`.
5. Le sommet v est supprimé de l'ensemble des candidats à la simplification résultant de l'étape 3.

La spécialisation pour le cas du vidage est plus directe, seule la dernière opération est spécialisée : v est retiré de l'ensemble des candidats au vidage.

4.5.2 Fusion

Nous considérons désormais l'opération de fusion. Soit g un graphe, $e = (x, y, w)$ une affinité élue pour la fusion et $g' =$ `merge` $e\ g$ le graphe résultant de la fusion de x et y dans g.

Des trois opérations réalisées sur le graphe, celle-ci est la plus délicate à manipuler. Cette difficulté provient en partie de l'utilisation d'un critère plus perfectionné pour déterminer si une affinité peut être fusionnée. Classiquement, deux critères sont utilisés :

1. Le critère de George [GA96] indique que x et y peuvent être fusionnés si $N(x, v) \subseteq N(y, v)$, et x ou y est précoloré. Ce critère n'a pas été implanté mais ne recèle pas de difficulté particulière. Il suffit de tester que l'ensemble des voisins d'interférence de l'une des extrémités est inclut dans le voisinage d'interférence de l'autre extrémité, ce qui peut être fait directement grâce à la fonction `VertexSet.subset` prédéfinie dans la bibliothèque sur les ensembles. Il faut cependant prendre garde à ce que les deux sommets fusionnés ne soient pas tous deux précolorés.
2. Le critère de Briggs [BCT94] exprime le fait que x et y peuvent être fusionnés si le sommet résultant de leur fusion a strictement moins de K voisins de haut degré. Ceci implique intuitivement que le sommet résultant finira par être candidat à la simplification. Comme pour le critère de George, les deux sommets fusionnés ne doivent pas être tous deux précolorés.

L'implantation Coq du critère de fusion de Briggs est donnée figure 4.5. La fonction `nb_of_significant_degree` permet de compter le nombre de sommets de `new_adj` de degré d'interférence supérieur à K dans g.

Afin de briser la symétrie des arêtes, nous imposons que la seconde extrémité d'une affinité élue pour la fusion ne soit pas précolorée. Ce choix est une conséquence de notre convention de fusion, qui impose que la seconde extrémité doit être supprimée du graphe et colorée comme la première extrémité. Ainsi, si la seconde extrémité d'une affinité qui satisfait le critère de fusion est précolorée, nous permutons les extrémités afin d'obtenir une arête qui satisfait également ce critère et respecte notre convention. Cette opération conduit à une affinité dont la seconde extrémité n'est pas précolorée puisque le critère de fusion requiert que les deux sommets à fusionner ne soient pas tous deux précolorés.

```
Definition conservative_coalescing_criteria e g K :=
if (is_precolored (fst_end e) &&
    is_precolored (snd_end e))
then false
else let new_adj := VertexSet.union
                      (interference_adj (fst_end e) g)
                      (interference_adj (snd_end e) g) in
  if le_lt_dec K (nb_of_significant_degree g K new_adj)
  then false
  else true.
```

FIG. 4.5 – Implantation Coq du critère de Briggs.

4.5.2.1 Evolution des voisinages des sommets

Deux cas sont à distinguer pour caractériser l'évolution des voisinages des sommets lors de la fusion de x et y, selon que l'on considère le sommet x ou un autre.

Le voisinage d'interférence de x est obtenu par union des voisinages d'interférence de x et de y, ce qui n'est pas surprenant au vu de la spécification de la fonction `merge` puisque chaque interférence incidente à y est redirigée vers x.

Lemme 7 (Voisinage d'interférence de x)**.** $N(x, g') = N(x, g) \cup N(y, g)$.

Suivant le même principe, le voisinage d'affinité de x est la différence des unions des voisins d'affinité de x et y et des voisins d'interférence de x et y.

Lemme 8 (Voisinage d'affinité de x)**.** $N_a(x, g') = (N_a(x, g) \cup N_a(y, g)) - (N(x, g) \cup N(y, g))$.

En pratique, nous n'utilisons pas cette représentation du voisinage de x dans g', mais une autre, équivalente et plus rapide à calculer. Celle-ci repose sur le fait que l'intersection des voisinages d'interférence et d'affinité d'un sommet est vide.

Lemme 9 (Voisinage d'affinité de x)**.** $N_a(x, g') = (N_a(x, g) - N(y, g)) \cup (N_a(y, g) - N(x, g))$.

Pour tout autre sommet v du graphe, il faut encore distinguer trois cas, en fonction des relations qui lient le sommet à x et y dans g :

1. Si v n'interfère pas avec y alors son voisinage est inchangé.
2. Si v interfère avec y et avec x alors y est retranché du voisinage de v.
3. Si v interfère avec y mais pas avec x alors x doit être ajouté à son voisinage et y doit en être retranché.

Lemme 10 (Voisinage d'interférence des autres sommets)**.** *Soit* $v \notin \{x, y\}$.

1. $v \notin N(y, g) \Rightarrow N(v, g') = N(v, g)$.
2. $v \in N(y, g) \wedge v \in N(x, g) \Rightarrow N(v, g') = N(v, g) - \{y\}$.
3. $v \in N(y, g) \wedge v \notin N(x, g) \Rightarrow N(v, g') = (N(v, g) - \{y\}) \cup \{x\}$.

Enfin, le voisinage d'affinité de v évolue également en fonction des relations qui lient ce sommet à x et y :

1. Si v n'est pas lié à y, son voisinage d'affinité est inchangé.
2. Si v est lié par une interférence à y et n'est pas lié par une affinité à x alors son voisinage d'affinité est inchangé.
3. Si v est lié par une interférence à y et par une affinité à x alors x doit être retranché du voisinage d'affinité de v.
4. Si v est lié par une affinité à y et par une interférence à x alors y doit être retranché du voisinage d'affinité de v.
5. Si v est lié par une affinité à y et à x alors y doit être retranché des voisins d'affinité de v.
6. Si v est lié par une affinité à y et n'est pas lié à x alors y doit être retranché du voisinage d'affinité de v et x doit y être ajouté.

Lemme 11 (Voisinage d'affinité des autres sommets)**.** *Soit* $v \notin \{x, y\}$.

1. $v \notin N_a(y, g) \wedge v \notin N(y, g) \Rightarrow N_a(v, g') = N_a(v, g)$.
2. $v \in N(y, g) \wedge v \notin N_a(x, g) \Rightarrow N_a(v, g') = N_a(v, g)$.
3. $v \in N(y, g) \wedge v \in N_a(x, g) \Rightarrow N_a(v, g') = N_a(v, g) - \{x\}$.
4. $v \in N_a(y, g) \wedge v \in N(x, g) \Rightarrow N_a(v, g') = N_a(v, g) - \{y\}$.
5. $v \in N_a(y, g) \wedge v \in N_a(x, g) \Rightarrow N_a(v, g') = N_a(v, g) - \{y\}$.
6. $v \in N_a(y, g) \wedge v \notin N(x, g) \wedge v \notin N_a(x, g) \Rightarrow N_a(v, g') = (N_a(v, g) - \{y\}) \cup \{x\}$.

Il en résulte que les degrés d'interférence et d'affinité d'un sommet v quelconque évoluent selon les lois suivantes.

Corollaire 5 (Degré d'interférence des autres sommets)**.** *Soit* $v \notin \{x, y\}$.
$v \in N(x, g) \cap N(y, g) \Rightarrow \delta(v, g') = \delta(v, g) - 1$.
Sinon, $\delta(v, g') = \delta(v, g)$.

Corollaire 6 (Degré d'affinité des autres sommets)**.** *Soit* $v \notin \{x, y\}$.
$v \in (N(x, g) \cap N_a(y, g)) \Rightarrow \delta_a(v, g') = \delta_a(v, g) - 1$.
$v \in (N_a(x, g) \cap N(y, g)) \Rightarrow \delta_a(v, g') = \delta_a(v, g) - 1$.
$v \in (N_a(x, g) \cap N_a(y, g)) \Rightarrow \delta_a(v, g') = \delta_a(v, g) - 1$.
Dans les autres cas, $\delta_a(v, g') = \delta_a(v, g)$.

En revanche, il est impossible de définir le degré de x dans g' en fonction uniquement du degré de x dans g puisqu'il s'agit de déterminer le cardinal de l'union de deux ensembles ($N(x, g)$ et $N(y, g)$) uniquement en fonction des cardinaux de ces ensembles. Calculer l'intersection des deux ensembles, puis son cardinal permettrait de prédire le degré de x après fusion mais est au moins aussi coûteux que de recalculer une fois cette fusion effectuée.

4.5.2.2 Evolution des fonctions caractéristiques

Nous déduisons de l'évolution des voisinages des sommets l'évolution des fonctions caractéristiques.

Tout sommet sans affinité dans g est sans affinité dans g'.

Lemme 12 (Sommet sans affinité). $\texttt{move\_related}\ g\ v\ =\ false\ \Rightarrow$
$\texttt{move\_related}\ g'\ v\ =\ false$

Tout sommet v différent de x et y de g lié par une affinité dans g est sans affinité dans g' si et seulement son degré d'affinité est égal à 1 et soit v est lié par une affinité à x et par une interférence à y, soit v est lié par une affinité à y et par une interférence à x.

Lemme 13 (Sommet devenant sans affinité). *Soit* $v \notin \{x, y\}$.
$\texttt{move\_related}\ g\ v\ =\ true\ \wedge\ \texttt{move\_related}\ g'\ v\ =\ false\ \Leftrightarrow$
$v \in (N_a(x,g) \cap N(y,g)) \cup (N_a(y,g) \cap N(x,g))\ \wedge\ \delta_a(v,g) = 1$

Tout sommet différent de x et y de bas degré dans g est de bas degré dans g'.

Lemme 14 (Sommet de bas degré). *Soit* $v \notin \{x, y\}$.
$\texttt{has\_low\_degree}\ g\ K\ v\ =\ true\ \Rightarrow\ \texttt{has\_low\_degree}\ g'\ K\ v\ =\ true$

Tout sommet de haut degré v différent de x et y de g est de bas degré dans g' si et seulement si v est un voisin d'interférence de x et de y de degré d'interférence K.

Lemme 15 (Sommet devenant de bas degré). *Soit* $v \notin \{x, y\}$.
$(\texttt{has\_low\_degree}\ g\ K\ v\ =\ false\ \wedge\ \texttt{has\_low\_degree}\ g'\ K\ v\ =\ true)\ \Leftrightarrow$
$v \in N(x,g) \cap N(y,g)\ \wedge\ \delta(v,g) = K$

4.5.2.3 Mise à jour des ensembles de travail

Soit $wl = (\texttt{spillWL}, \texttt{freezeWL}, \texttt{simplifyWL}, \texttt{movesWL})$ un quadruplet vérifiant le prédicat ($\texttt{WL\_invariant}\ g\ palette\ wl$). Nous introduisons une fois encore des notations afin d'alléger la description de la mise à jour des ensembles de travail.

Nous notons $L(x, y, g, K)$ l'ensemble des sommets interférant à la fois avec x et y, non précolorés, et de degré exactement K. Ces sommets de haut degré de g sont de bas degré dans g' et doivent à ce titre changer d'ensemble de travail, conformément au lemme 15.

(def_L) $v\ \in\ L(x,y,g,K)\ \Leftrightarrow$
$v\ \in\ N(x,g)\ \cap\ N(y,g)\ \wedge \texttt{is\_precolored}\ v\ =\ false\ \wedge\ \delta(v,g)\ =\ K$

Comme dans le cas de la suppression d'un sommet du graphe, nous partitionnons cet ensemble en deux sous-ensembles contenant les sommets sans affinité d'une part et les sommets liés par une affinité d'autre part.

(def_Lmr) $v\ \in\ L_{mr}(x,y,g,K)\ \Leftrightarrow\ v\ \in\ L(x,y,K,g) \wedge \texttt{move\_related}\ g\ v\ = true$

(def_Lnmr) $v\ \in\ L_{nmr}(x,y,g,K)\ \Leftrightarrow\ v\ \in\ L(x,y,K,g) \wedge \texttt{move\_related}\ g\ v\ = false$

Enfin, nous notons $M(x, y, g, K)$ l'ensemble des sommets non précolorés, de bas degré, ayant exactement une affinité et liés par une affinité à une extrémité de e et par une interférence à l'autre extrémité de e.

(def_M) $v \in M(x,y,g,K) \Leftrightarrow v \in (N(x,g) \cap N_a(y,g)) \cup (N_a(x,g) \cap N(y,g)) \wedge$ `has_low_degree` $g\ K\ v = true \wedge \delta_a(v,g) = 1 \wedge$ `is_precolored` $v = false$

Ces sommets de bas degré ne seront plus liés par une affinité dans g' alors qu'ils l'étaient dans g. Ils devront être transférés de l'ensemble des candidats au gel à l'ensemble des candidats à la simplification.

Soit $wl' = (\texttt{spillWL}', \texttt{freezeWL}', \texttt{simplifyWL}', \texttt{movesWL}')$ le quadruplet obtenu à partir de wl par la procédure suivante :

1. Les sommets de $L(x, y, g, K)$ sont supprimés de `spillWL`.
2. Les sommets de $M(x, y, g, K)$ sont supprimés de `freezeWL`.
3. Les sommets de $L_{mr}(x, y, g, K)$ sont ajoutés à l'ensemble de candidats au gel résultant de l'étape 2.
4. Les sommets de $L_{nmr}(x, y, g, K)$ et $M(x, y, g, K)$ sont ajoutés à l'ensemble des candidats à la simplification obtenue après l'étape 1.
5. Pour tout sommet v de $N_a(x, g) \cap N(y, g)$ l'affinité liant v à x est supprimée de `movesWL`.
6. Pour tout sommet v de $N_a(y, g) - N(x, g)$ une affinité (v, x) est ajoutée à l'ensemble des candidats à la fusion obtenu après l'étape 5.
7. Chaque affinité incidente à y est supprimée de l'ensemble des candidats à la fusion résultant de l'étape 6.
8. Si x n'est pas précoloré alors x est classé dans l'ensemble de travail approprié, en fonction de son degré d'interférence et de ses éventuelles affinités.
9. Si le sommet x (respectivement y) n'est pas précoloré, celui-ci est supprimé de l'ensemble des candidats au vidage résultant de l'étape 1 s'il est de haut degré dans g ou de l'ensemble des candidats au gel résultant de l'étape 3 s'il est de bas degré dans g.

Cette procédure est correcte, *i.e.* construit un quadruplet qui vérifie l'invariant pour g' si wl vérifie l'invariant pour g.

Théorème 7. `WL_invariant` g' *palette* wl'.

4.5.3 Gel

Nous considérons enfin le gel d'un sommet v dans un graphe g, lequel conduit au graphe $g' =$ `delete_affinities` $v\ g$.

4.5.3.1 Evolution des voisinages des sommets

Etant donné que cette opération ne modifie pas les interférences du graphe, les voisinages d'interférence restent inchangés.

Lemme 16 (Voisinage d'interférence). $N(x, g') = N(x, g)$.

D'où le corollaire direct pour les degrés :

Lemme 17 (Degré d'interférence). $\delta(x, g') = \delta(x, g)$.

En revanche, il faut distinguer deux cas pour le voisinage d'affinité : le sommet v et les autres. Pour le sommet v, le résultat est simple : son voisinage d'affinité est vide dans g' et son degré d'affinité devient donc nul.

Lemme 18 (Voisinage d'affinité de v). $N_a(v, g') = \emptyset$

Lemme 19 (Voisinage d'affinité de v). $\delta_a(v, g') = 0$

Le voisinage d'affinité dans g' de tout sommet différent de v est obtenu en supprimant v du voisinage d'affinité dans g.

Lemme 20 (Voisinage des autres sommets). *Soit* $x \neq v$.
$N_a(x, g') = N_a(x, g) - \{v\}$.

Ce lemme induit deux corollaires, en fonction de l'appartenance du sommet au voisinage d'affinité de v.

Si x est un voisin d'affinité de v dans g alors v doit effectivement être retiré du voisinage d'affinité.

Corollaire 7 (Voisinage des voisins de v). *Soit* $x \neq v$.
$x \in N_a(v, g) \Rightarrow N_a(x, g') = N_a(x, g) - \{v\}$.

Si x n'est pas un voisin d'affinité de v dans g alors le voisinage d'affinité de x est inchangé.

Corollaire 8 (Voisinage des autres sommets). *Soit* $x \neq v$.
$x \notin N_a(v, g) \Rightarrow N_a(x, g') = N_a(x, g)$.

De ces corollaires résulte l'évolution du degré d'affinité des sommets autres que v.

Corollaire 9 (Degré d'affinité des voisins de v). *Soit* $x \neq v$.
$x \in N_a(v, g) \Rightarrow \delta_a(x, g') = \delta_a(x, g) - 1$.

Corollaire 10 (Degré d'affinité des autres sommets). *Soit* $x \neq v$.
$x \notin N_a(v, g) \ \Rightarrow \delta_a(x, g') = \delta_a(x, g)$.

4.5.3.2 Evolution des fonctions caractéristiques

Comme nous l'avons fait auparavant pour les autres opérations, nous déduisons des évolutions des degrés l'évolution des fonctions caractéristiques.

Tout sommet sans affinité de g est sans affinité dans g'.

Lemme 21 (Sommets sans affinité). `move_related` $g\, x = false \Rightarrow$ `move_related` $g'\, x = false$

Tout sommet lié par une affinité x différent de v est sans affinité dans g' si et seulement si x est un voisin d'affinité de v de degré d'affinité 1.

Lemme 22 (Affinités des voisins de v)**.** *Soit $x \neq v$.*
`move_related` $g\,x = true \wedge$ `move_related` $g'\,x = false \Leftrightarrow x \in N_a(v,g) \wedge \delta_a(x,g) = 1$

Tout sommet qui n'est pas un voisin d'affinité de v est lié par une affinité dans g' si et seulement s'il est lié par une affinité dans g.

Lemme 23 (Conservation des affinités des autres sommets)**.** *Soit $x \neq v$.*
$x \notin N_a(v,g) \Rightarrow$ `move_related` $g'\,v =$ `move_related` $g\,v$

Puisque les voisinages et degrés d'interférence sont inchangés, un sommet est de bas degré dans g' si et seulement s'il est de bas degré dans g.

Lemme 24 (Sommets de bas degré)**.** *Soit x un sommet.*
`has_low_degree` $x\ g\ K\ =$ `has_low_degree` $x\ g'\ K$.

4.5.3.3 Mise à jour des ensembles de travail

Soit $wl = (\texttt{spillWL}, \texttt{freezeWL}, \texttt{simplifyWL}, \texttt{movesWL})$ un quadruplet vérifiant l'invariant (`WL_invariant` g *palette* wl).

Nous notons $D(v,g,K)$ l'ensemble des voisins d'affinité de v de bas degré, non précolorés et de degré d'affinité égal à 1 dans g. Ces sommets, candidats au gel dans g deviennent candidats à la simplification dans g'.

(def_D) $x \in D(v,g,K) \Leftrightarrow x \in N_a(v,g) \wedge$ `is_precolored` $x = false\ \wedge$
$\delta_a(x,g) = 1\ \wedge$ `has_low_degree` $g\,K\,x\ =\ true$

Soit $wl' = (\texttt{spillWL}', \texttt{freezeWL}', \texttt{simplifyWL}', \texttt{movesWL}')$ le quadruplet d'ensembles de travail construit à partir de $wl = (\texttt{spillWL}, \texttt{freezeWL}, \texttt{simplifyWL}, \texttt{movesWL})$ suivant la procédure ci-dessous.

1. v est supprimé de `freezeWL` et ajouté à `simplifyWL`.
2. Les sommets de $D(v,g,K)$ sont supprimés des candidats au gel résultant de l'étape 1.
3. Les sommets de $D(v,g,K)$ sont ajoutés à l'ensemble des candidats à la simplification obtenu après l'étape 1.
4. Les affinités incidentes à v sont supprimées de `movesWL`.

Le quadruplet wl' vérifie l'invariant pour g'.

Théorème 8. `WL_invariant` g' *palette* wl'.

4.6 Terminaison

Avant d'assurer que le résultat renvoyé par un algorithme vérifie bien la spécification de l'algorithme, il faut prouver que l'algorithme renvoie bel et bien un résultat. La terminaison

est donc un acteur de premier ordre du processus de vérification formelle. De plus, le système Coq exige une preuve de terminaison pour que l'extraction puisse être réalisée.

La description de référence de l'IRC, comme nombre de descriptions d'algorithmes d'allocation de registres par coloration de graphe fait fi de la terminaison. Il n'est pourtant pas si évident que l'algorithme termine toujours, surtout que la définition de référence fait appel à quatre fonctions qui s'appellent mutuellement. Mettre en exergue un argument de terminaison est donc d'autant plus délicat. Nous avons défini une mesure pour l'IRC, *i.e.* une fonction à valeur dans les entiers naturels qui décroît à chaque appel récursif de l'algorithme et permet ainsi d'en prouver la terminaison.

La mesure que nous proposons est linéaire en le nombre de sommets du graphe et procure une borne supérieure relativement précise du nombre d'appels récursifs à réaliser. Celle-ci est $\mathcal{B}(g) = (2 \times n(g)) - p(g)$, où $n(g)$ est le nombre de sommets non précolorés du graphe et $p(g)$ est le nombre de sommets non précolorés, de bas degré et sans affinité de g. L'on peut également voir plus intuitivement $p(g)$ comme le nombre de candidats à la simplification. Nous avons construit cette mesure en remarquant que des quatre opérations pouvant être appliquées, toutes sauf le gel font décroître le nombre de sommets du graphe. Or, le gel ne peut être appliqué qu'au plus une fois par sommet lié par une affinité, et donc au plus $n(g) - p(g)$ fois. Il peut donc y avoir au plus $n(g)$ appels récursifs pour les opérations de simplification, de fusion et de vidage et au plus $n(g) - p(g)$ opérations de gel, ce qui conduit à la borne $\mathcal{B}(g)$.

Nous prouvons que $\mathcal{B}(g)$ décroit strictement à chaque appel récursif. Cette preuve s'appuie sur l'évolution des ensembles de travail et sur les preuves des lemmes suivants.

Lemme 25 (Décroissance de la mesure par retrait de sommet)**.** *Soit v un sommet non précoloré de g et $g' =$* `remove_vertex` *$v\,g$. Alors, $\mathcal{B}(g') < \mathcal{B}(g)$.*

Lemme 26 (Décroissance de la mesure par fusion d'affinité)**.** *Soit $e = (x, y, w)$ une arête de g choisie par le critère de fusion et g' le graphe résultant de la fusion de x et y dans g. Alors, $\mathcal{B}(g') < \mathcal{B}(g)$.*

Lemme 27 (Décroissance de la mesure par gel d'un sommet)**.** *Soit v un sommet candidat pour le gel dans un graphe g et g' le graphe résultant de cette opération. Alors, $\mathcal{B}(g') < \mathcal{B}(g)$.*

Il suffit alors de raisonner par induction sur les appels de l'algorithme `IRC_aux` pour en prouver la terminaison. En effet, seules cinq possibilités sont envisageables :

1. Un appel récursif en cas de simplification ;
2. Un appel récursif en cas de fusion ;
3. Un appel récursif en cas de gel ;
4. Un appel récursif en cas de vidage ;
5. Le renvoi de la précoloration.

La fonction de précoloration terminant toujours, il ne reste plus qu'à prouver que chaque appel récursif décroit bien selon la mesure.

Théorème 9 (Terminaison de l'implantation)**.** *Soit g un graphe d'interférence-affinité. Si* `IRC_aux` *g fait un appel à* `IRC_aux` *g', alors $\mathcal{B}(g') < \mathcal{B}(g)$. En conséquence, le nombre d'appels récursifs de* `IRC_aux` *g est borné supérieurement par $\mathcal{B}(g)$ et* `IRC_aux` *g renvoie toujours un résultat, i.e. termine toujours.*

4.7 Correction

La preuve de correction de l'IRC exige, comme l'indique la spécification, que si la précoloration du graphe est valide alors l'algorithme retourne une coloration valide du graphe qui étend cette précoloration.

Théorème 10 (Correction de l'implantation)**.** *Si* `precoloring` *g est une coloration valide, alors* `IRC_aux` *g retourne une coloration valide de g qui préserve la précoloration.*

Comme pour la terminaison, la correction est prouvée par induction sur les appels récursifs. La preuve de chacun des quatre premiers cas est immédiate grâce aux lemmes de correction des fonctions `precoloring`, `available_color` et `complete_col` (cf. page 92). Il s'agit en effet à chaque fois de prouver que colorer un sommet supplémentaire est correct, ce qui est simple puisque le sommet est choisi en fonction de conditions bien définies qui assurent justement la pérennité de la coloration. Le dernier cas est l'hypothèse du théorème.

4.8 Evaluation expérimentale

4.8.1 Caractéristiques du développement

Le code Coq de notre implantation de l'IRC est composé d'environ 600 lignes de fonctions et définitions. Ce code est complété par les énoncés de théorèmes et scripts de preuve qui constituent en tout environ 4800 lignes : 3300 lignes (soit environ 110 lemmes) portent sur les preuves de la mise à jour incrémentale des ensembles de travail, 300 lignes (soit 17 lemmes) sont consacrées à la preuve de terminaison, 650 lignes (soit 22 lemmes) sont vouées à la preuve de correction et 550 lignes (soit 55 lemmes) sont dévolues aux propriétés des graphes d'interférence-affinité. Au total, le ratio preuve/code est donc d'environ 8, ce qui est comparable au ratio du compilateur CompCert [Ler06]. Les chiffres ci-dessus soulignent la difficulté de la mise à jour incrémentale des ensembles de travail. Cependant, la preuve de terminaison est courte parce qu'elle s'appuie fortement sur cette mise à jour. Il faut donc nuancer la disproportion de la preuve.

4.8.2 Implantation dans CompCert

Un prototype de CompCert utilisant notre allocateur de registres a été mis au point. L'intégration de cet allocateur a nécessité de transformer les graphes existant dans CompCert en les nôtres, ce après quoi l'allocateur prend la main, puis de prouver que les deux propriétés décrites dans CompCert (que se contentent de vérifier le validateur de CompCert) sont établies. Contrairement à CompCert , qui n'utilise qu'un graphe d'interférence-affinité, nous avons choisi de dissocier registres entiers et registres flottants afin d'améliorer les performances, comme il est d'usage dans les stratégies de type «diviser pour mieux régner». Nous obtenons donc deux colorations qu'il faut combiner pour obtenir la coloration du graphe de CompCert . Le processus complet d'intégration de notre allocateur dans CompCert est schématisé figure 4.6.

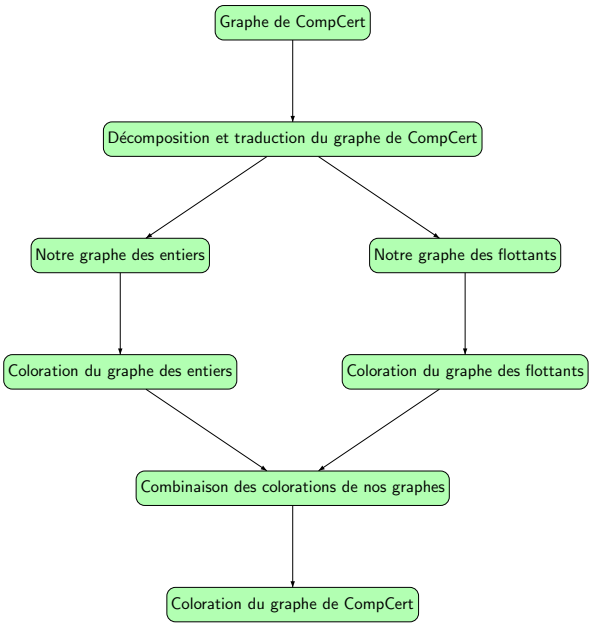


FIG. 4.6 – Description de l'intégration de notre allocateur de registres au sein de CompCert.

4.8.3 Evaluation du code extrait

Nous avons comparé l'extraction de notre développement avec l'allocateur de registres de CompCert . La comparaison est faite sur les instances de test de CompCert , dont les caractéristiques sont données figure 4.7. Ces instances vont de 50 à 300 lignes de code C . Les fonctions sont courtes, comme il est d'usage dans le domaine de l'embarqué critique qui, rappelons le, est le domaine d'application visé par le compilateur CompCert. Notre allocateur génère au plus deux graphes par fonction compilée, une pour chaque type de registres, et l'IRC est appliqué séparément sur chacun des graphes. Chaque ligne du tableau 4.7 contient les caractéristiques d'un programme test. Le nombre de graphes à colorer peut être supérieur à deux puisque chaque programme peut contenir un nombre quelconque de fonctions.

Les résultats comparatifs des temps d'exécution des deux allocateurs de registres sont donnés figure 4.8. Les mesures ont été réalisées sur un ordinateur Apple PowerMAC muni d'un processeur PowerPC 970 cadencé à 2.0Ghz et de 6Go de RAM, sous MacOS 10.4.11. Les deux premières colonnes de l'histogramme contiennent les temps d'exécution des deux allocateurs en millisecondes. Notre allocateur n'est pas aussi rapide que celui de CompCert . Ce résultat n'est guère surprenant puisque les structures de données sur lesquelles s'appuie notre implantation, *i.e.* les implantations des ensembles et des fonctions d'association, reposent sur des arbres binaires auxquels les accès sont logarithmiques, tandis que l'implantation Ocaml impérative utilise des tables de hash qui offrent des accès en $O(1)$ en moyenne. Néanmoins, les temps d'allocation de registres demeurent très bons puisque ceux-ci n'excèdent jamais le dixième de seconde sur les instances considérées. Le ralentissement

Instance	Graphes	Variables	Interférences	Affinités
AES cipher	7	113	586	166
Almabench	10	53	310	22
Binary trees	6	23	42	14
Fannkuch	2	50	332	27
FFT	4	72	391	37
Fibonacci	2	17	18	9
Integral	7	12	12	5
K-nucleotide	17	24	74	14
Lists	5	18	33	11
Mandelbrot	2	45	117	17
N-body	9	28	73	10
Number sieve	2	25	53	12
Number sieve bits	3	76	58	12
Quicksort	3	28	116	16
SHA1 hash	8	34	107	15
Spectral test	9	14	35	6
Virtual machine	2	73	214	38
Arithmetic coding	37	31	85	15
Lempel-Ziv-Welch	32	32	127	16
Lempel-Ziv	33	29	92	15

FIG. 4.7 – Caractéristiques des instances de test de CompCert.

observé est donc parfaitement acceptable.

La troisième colonne représente le temps virtuellement obtenu en ajoutant une pénalité logarithmique aux temps d'exécution de la version impérative. En d'autres termes, la dernière colonne est le temps pris par l'allocateur impératif multiplié par $(\log\ n)$, où n est le nombre de sommets du graphe. Ce facteur logarithmique compense l'écart théorique de complexité entre les deux implantations. En pratique, nous observons que le temps pris par notre allocateur est très proche du temps virtuel ainsi calculé. Cette proximité confirme que l'implantation que nous proposons est efficace et pourrait certainement atteindre les mêmes niveaux de performance que l'implantation itérative dans un environnement de programmation moins limité par les structures de données que l'est celui de Coq , par exemple Ocaml ou Haskell.

Enfin, nous avons comparé la qualité du code généré par les deux allocateurs. Conformément à nos attentes, ces codes générés se sont avérés équivalents. Rien d'étonnant puisqu'ils ne sont que deux implantations d'un même algorithme.

4.9 Extensions

Le travail présenté dans ce chapitre apporte un niveau supplémentaire de sûreté à l'allocation de registres et, par conséquent, à la compilation. Décortiquer aussi scrupuleusement

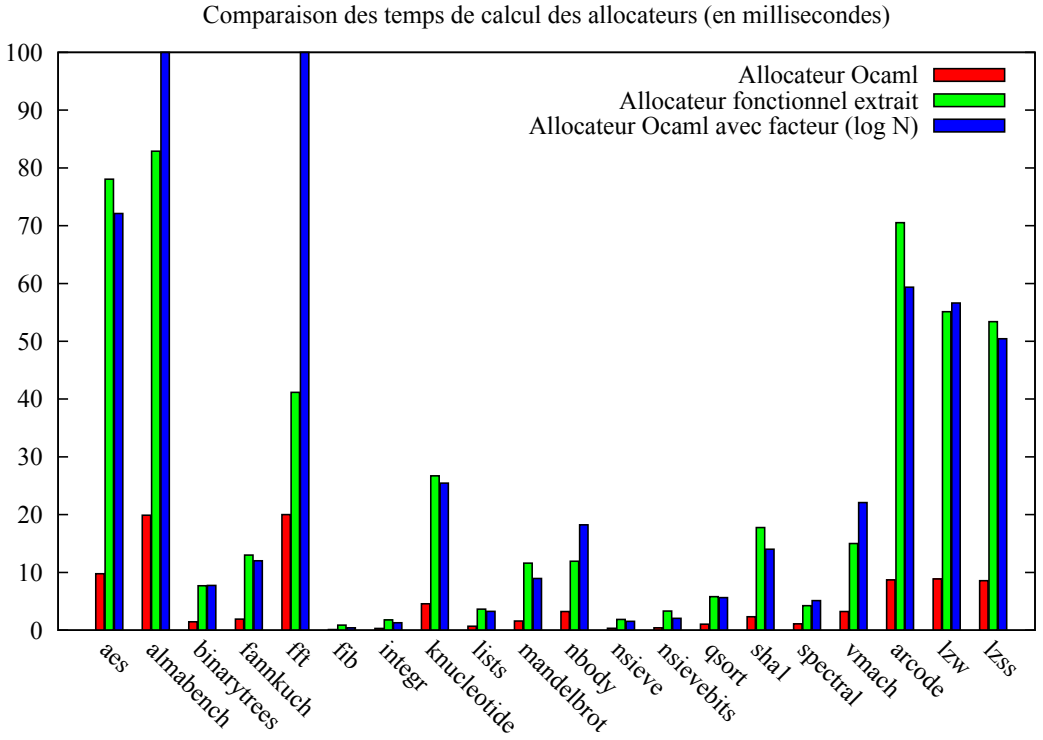


FIG. 4.8 – Comparaison des temps pris par chacun des allocateurs de registres. Afin d'améliorer la lecture, les résultats pour les instances almabench et fft sont tronqués à 100 millisecondes dans la troisième colonne alors que ceux-ci s'élèvent respectivement à 131 et 120 millisecondes.

un algorithme tel que l'IRC nous a également mené à des réflexions plus poussées sur la formalisation de cet algorithme ou, plus généralement, sur la formalisation des algorithmes d'allocation de registres par coloration de graphe. Cette section décrit des extensions possibles de l'IRC. Nous les classons en deux catégories : les améliorations algorithmiques et l'intégration de la reconstruction.

4.9.1 Améliorations algorithmiques

L'IRC est une heuristique d'allocation de registres très classique qui a été étudiée en détails et pour laquelle des optimisations algorithmiques ont été proposées. Parmi celles-ci, deux se détachent de par leur efficacité, mais n'ont pas été implantées dans notre développement en raison des difficultés inhérentes à leur intégration.

4.9.1.1 Gestion des voisinages des sommets précolorés

La première optimisation possible consiste à ne pas conserver les voisinages des sommets précolorés dans le graphe afin de gagner de l'espace et améliorer les temps de consultation

des voisinages. En effet, les voisinages des sommets que nous cherchons sont la plupart du temps ceux des sommets non précolorés. Cependant, deux nouveaux problèmes apparaissent avec cette optimisation.

Tout d'abord, les voisins des sommets précolorés peuvent être nécessaires pour le critère de fusion : une des extrémités de l'affinité évaluée peut être précolorée et il est alors nécessaire de connaître ses voisins pour déterminer le nombre de voisins de haut degré du sommet qui résulterait de la fusion de l'arête. Il en découlerait une grosse perte algorithmique lors de cette opération qui, nous pensons, compense le gain obtenu par ailleurs. Une autre solution consisterait à n'autoriser que la fusion d'affinités n'ayant aucune extrémité précolorée. Cette approche a été rejetée car elle va à l'encontre de l'efficacité de l'algorithme en ne limitant que trop le critère de fusion [Hai05].

D'autre part, ne pas conserver les voisinages des sommets précolorés dans le graphe casserait les propriétés de symétrie des graphes et donc toute une partie des preuves portant sur notre structure. Modifier la structure de données en fonction de l'algorithme aurait évidemment été mal venu. Tout bien considéré, intégrer cette optimisation aurait coûté beaucoup pour un gain espéré faible.

4.9.1.2 Gestion des affinités

La seconde optimisation est sans doute la plus connue pour l'IRC. Il s'agit d'une méthode alternative de gestion de l'ensemble de travail contenant les affinités susceptibles d'être fusionnées. Dans notre implantation, cet ensemble de travail contient toutes les affinités du graphe. L'optimisation consiste à évincer de cet ensemble toute affinité qui a auparavant échoué au test de fusion, puis de rajouter une affinité (x, y) à l'ensemble de travail lorsque l'un des voisins de haut degré de x ou de y devient de bas degré [GA96]. Cette approche évite de tester plusieurs fois la même affinité dans un graphe qui aura peu évolué entre temps, *i.e.* un environnement dans lequel les deux tests ont toutes les chances de mener au même résultat. De nombreuses exécutions du critère de fusion peuvent ainsi être évitées, améliorant de ce fait la vitesse d'exécution de l'algorithme. Une amélioration de cette gestion des affinités a même été proposée par Leung et George [LG07] afin de réaliser encore moins de tests du critère de fusion. Pour ce faire, ils tiennent à jour en permanence pour chaque affinité (x, y) une borne supérieure du nombre de sommets qui empêchent x et y d'être fusionnés. Si la borne chute en dessous de 0, il est possible de fusionner les deux sommets.

Nous n'avons pas implanté cette optimisation car celle-ci brise l'invariant sur les ensembles de travail `WL_invariant`. Il serait néanmoins possible d'affaiblir l'invariant afin de spécifier que les arêtes de l'ensemble de travail forment un sous-ensemble des affinités du graphe. Il faudrait en outre définir une façon efficace de gérer les affinités, aussi bien celles présentes dans l'ensemble de travail que celles qui n'y sont pas. Cependant, un travail de stage portant sur ce sujet a déjà permis d'implanter une première version de cette optimisation dans notre implantation de l'IRC [Ty10]. Cette amélioration de notre implantation consiste à considérer non pas un ensemble de travail pour la fusion, mais deux : l'un est dit actif, dans le sens où il représente les affinités qui sont considérées lors des tests de fusion, et l'autre est dit passif, car les affinités de cet ensemble ne seront ajoutées à l'ensemble actif que si aucune affinité de l'ensemble actif courant ne peut être fusionnée. La taille

du développement consacré à cette optimisation est d'environ mille lignes, soit 15% de notre développement. En termes d'efficacité, aucun comparatif entre les temps d'exécution avec et sans cette optimisation n'a malheureusement pu être réalisé en raison de difficultés d'ordre technique rencontrées lors de l'intégration à CompCert .

4.9.2 Reconstruction

La trame générale de l'IRC incorpore une phase de reconstruction, nécessaire lorsque le spill code inséré crée de nouvelles contraintes qui ne sont pas respectées par l'allocation de registres actuelle. Nous n'avons pas eu à nous soucier de cette partie pour l'allocation de registres de CompCert , puisque, comme expliqué plus tôt, des registres sont dédiés à la gestion du spill code, évitant ainsi tout besoin de reconstruction. Cependant, la reconstruction s'impose comme une perspective prioritaire.

La reconstruction est une phase généralement assez mal décrite, et souvent passée sous silence, dans la littérature. En soi, cette phase consiste à construire un nouveau graphe d'interférence-affinité qui va être coloré. Certains sommets, correspondant aux temporaires utilisés pour dialoguer avec la mémoire sont marqués comme ne pouvant être assujettis au vidage. Ils ne sont cependant pas précolorés.

A première vue, le point le plus épineux à considérer pour la preuve de la phase de reconstruction est sa terminaison. Il faut en effet arriver à prouver que l'algorithme ne va pas procéder indéfiniment à des reconstructions. Pour ce faire, une solution consiste à trouver une fonction qui décroît à mesure des reconstructions. Nous pensons que le nombre de sommets du graphe qui peuvent être désignés pour le vidage est une fonction qui satisfait ces conditions. En effet, les sommets déjà vidés en pile le sont définitivement (ils ne font plus partie des sommets qui peuvent être vidés en pile) et les nouveaux temporaires ajoutés pour dialoguer avec la mémoire sont marqués comme ne pouvant l'être. Néanmoins une étude plus poussée est nécessaire à ce sujet pour prouver qu'il s'agit bien d'une mesure.

Intégrer la reconstruction à notre implantation permettrait de ne plus monopoliser les deux registres dédiés et ainsi d'améliorer la qualité des allocations de registres calculées. Si le besoin ne s'en fait pas ressentir sur l'architecture `PPC` , il commence à l'être sur l'architecture `ARM` et la reconstruction est indispensable dans l'optique de développer de nouveaux portages pour des architectures dotées de peu de registres, comme le x86 par exemple.

4.10 Bilan

Dans ce chapitre, nous avons présenté, formalisé et implanté un algorithme optimisant d'allocation de registres par coloration de graphe : l'IRC. La spécification de cet algorithme a soulevé de nombreuses difficultés aussi bien théoriques que pratiques, comme la transcription de la description impérative en une implantation purement fonctionnelle, la preuve de terminaison de cette implantation, ou son efficacité algorithmique. Nous avons en particulier formalisé très précisément comment mettre à jour les ensembles de travail de façon incrémentale. Cette définition de l'IRC peut être considérée comme une description de référence, parfaitement exempte de bogues, de cet algorithme.

Sur le plan technique, ce travail a requis l'utilisation de fonctionnalités avancées du système Coq telles que les types dépendants, les monades, les foncteurs ou les fonctions récursives non structurelles. L'extraction de notre implantation produit un allocateur de registres écrit en Ocaml que nous avons intégré à un prototype du compilateur CompCert . Celui-ci produit du code compilé équivalent à l'allocateur de registres actuel, mais plus sûr. En contrepartie, le temps pris par la coloration du graphe d'interférence-affinité se voit légèrement allongé, en raison de l'inévitable facteur logarithmique dû aux structures de données purement fonctionnelles que nous utilisons.

Chapitre 5

Extension de la vérification formelle de l'IRC à une classe d'algorithmes

L'IRC est le second algorithme d'allocation de registres que nous avons prouvé, le premier étant un algorithme plus simple reposant sur la coloration gourmande proche de celui de Palsberg et Pereira [PP05] dédié aux graphes SSA [Rob07, BRS08]. De ces deux expériences se sont dégagées des constantes qui sont également partagées par d'autres algorithmes classiques d'allocation de registres, comme par exemple celui de Chaitin. Ce chapitre exploite la structure commune de ces algorithmes afin de formaliser, implanter et prouver correct un algorithme générique d'allocation de registres. Ce dernier peut être paramétré par un utilisateur très simplement, afin de définir un algorithme d'allocation de registres correct par construction et permettre ainsi à de plus nombreux compilateurs de disposer d'une allocation de registres sûre. Il permet en outre de comparer la qualité des allocations de registres construites par différents algorithmes.

Le développement Coq complet de cet algorithme est consultable en ligne à l'adresse `http://www.ensiie.fr/~robillard/generic_allocator`.

5.1 Une structure récursive commune

L'IRC et l'algorithme de coloration gourmande présentent une structure similaire très caractéristique des algorithmes d'allocation de registres. Ce ne sont d'ailleurs pas les seuls : l'algorithme de Chaitin et toutes ses variantes entrent également dans cette classe. En effet, tous ceux-ci utilisent un même éventail de fonctions : une fonction de suppression de sommet, émulant à la fois la simplification ou le vidage, une fonction de suppression d'arête, pour le gel, et une fonction de fusion d'affinités. La trame est toujours la même : appliquer les opérations précitées jusqu'à ce que le graphe soit vide, puis construire la coloration en dépilant les sommets. Ce principe, uniformément décrit impérativement dans la littérature et faisant appel à une pile correspond de fait parfaitement à une description récursive. La méthodologie adoptée pour passer de la version itérative de l'IRC à notre description fonctionnelle peut donc *a priori* être appliquée à d'autres algorithmes classiques d'allocation de registres.

Pour mieux s'en rendre compte, il suffit de remarquer que la description de l'IRC (cf. figure 4.2.3, page 89) est encore très abstraite. Plusieurs autres algorithmes d'allocation de registres, comme celui de Chaitin [CAC+81], celui de Briggs [BCT94], ou encore des algorithmes plus récents comme l'algorithme de coloration de graphe pour programmes sous forme SSA de Palsberg et Pereira [PP05] ou encore celui de fusion par extension de coloration [ONI+10] pourraient tout aussi bien être décrits par ce même schéma, à ceci près que la structure sous-jacente `irc_graph` ne serait pas nécessaire. En fait, n'importe quel algorithme procédant par empilement et dépilement et ne faisant pas appel à d'autres opérations que la simplification, la fusion, le gel et le vidage entrent dans ce schéma. Tout dépend ensuite des conditions de déclenchement de chacune des opérations, qui ne constituent qu'une toute petite partie du développement.

Cette description récursive généraliste nous semble plus naturelle pour implanter toute cette classe d'algorithmes puisque l'empilement et le dépilement des sommets est géré automatiquement par la récursivité. L'aspect impératif demeure néanmoins très présent, même s'il est en partie caché ici.

5.2 Description de l'algorithme générique

5.2.1 L'algorithme

En se fondant sur cette possibilité d'abstraction d'une classe non négligeable d'algorithmes d'allocation de registres, nous avons développé un module Coq , permettant à un utilisateur moins familier avec Coq de n'implanter que les composantes minimales nécessaires à la définition de son algorithme d'allocation de registres, à condition que les opérations appelées ne soient que les quatre précédemment citées. L'algorithme est défini de sorte à laisser les fonctions propres à l'utilisateur et leurs preuves de correction abstraites. Ces fonctions propres à l'utilisateur sont au nombre de neuf et peuvent être classées en deux catégories.

La première catégorie contient les fonctions de déclenchement des opérations de simplification, fusion, gel et vidage. L'utilisateur doit définir quatre fonctions partielles (`fsimp`, `fcoal`, `ffreeze` et `fspill`) permettant respectivement de désigner un sommet qui doit être simplifié, une affinité qui doit être fusionnée, un sommet qui doit être gelé ou un sommet qui doit être vidé en pile.

La seconde catégorie contient les fonctions de coloration du graphe. L'utilisateur doit définir l'ensemble des fonctions de coloration appelées en fonction de l'opération opérée sur le graphe. Ces fonctions sont donc naturellement au nombre de quatre : `colsimp`, `colcoal`, `colfreeze` et `colspill`. Une dernière fonction `precoloring` vient s'ajouter, afin de représenter l'éventuelle précoloration du graphe.

L'implantation de notre algorithme générique d'allocation de registres, nommé `alloc`, donnée figure 5.1 ressemble à celle de l'IRC. Les fonctions `simp`, `coal`, `freeze` et `spill` font respectivement appel aux fonctions `fsimp`, `fcoal`, `ffreeze` et `fspill` définies par l'utilisateur, comme détaillé ultérieurement (cf. section 5.2.2). Les opérations sont effectuées selon l'ordre de priorité habituel : la simplification est prioritaire, la fusion intervient ensuite, le gel apparaît en troisième position et le vidage est réalisé en dernier recours. Toutes ces

fonctions sont suffisantes pour définir de manière précise notre algorithme d'allocation de registres.

```
Function alloc (g : Graph.t) {wf alloc_order}: Coloring :=
match simp g with
| ⌊(v,g')⌋ → colsimp (alloc g') v
| ∅ → match coal g with
    | ⌊(e,g')⌋ → colcoal (alloc g') e
    | ∅ → match freeze g with
        | ⌊(v,g')⌋ → coldel (alloc g') v
        | ∅ → match spill g with
            | ⌊(v,g')⌋ → colspill (alloc g') v
            | ∅ → precoloring g
                  end
            end
      end
end.
```

FIG. 5.1 – Algorithme générique d'allocation de registres par coloration de graphe.

Cependant, il n'est pas obligatoire de recourir aux quatre fonctions. Si nous désirons implanter un algorithme qui ne fait que de la simplification et du vidage, par exemple, il suffit de définir la fonction de simplification comme la fonction qui cherche un sommet de bas degré, les fonctions de fusion et de gel comme des fonctions qui échouent toujours, et la fonction de vidage comme la fonction qui renvoie n'importe quel sommet du graphe.

5.2.2 Construction des fonctions auxiliaires

5.2.2.1 Spécification

Les spécifications des fonctions auxiliaires `simp`, `coal`, `freeze` et `spill` sont restreintes aux propriétés indispensables aux preuves de terminaison et de correction de l'algorithme.

La fonction `simp` doit vérifier deux propriétés : la première composante du couple doit être un sommet du graphe et la seconde le graphe obtenu après suppression de ce sommet du graphe.

(spec_simp_1) `simp` $g = \lfloor (v, g') \rfloor \Rightarrow v \in_s g$

(spec_simp_2) `simp` $g = \lfloor (v, g') \rfloor \Rightarrow g' =$ `remove_vertex` $v\ g$

La fonction `coal` doit renvoyer une affinité appartenant au graphe accompagnée du graphe obtenu après fusion de cette affinité. La fonction `merge` a besoin de deux preuves : l'une pour l'appartenance de l'arête fusionnée au graphe, et l'autre pour exprimer que cette arête est une affinité. Ces preuves sont construites au sein de la fonction `coal`.

(spec_coal_1) `coal` $g = \lfloor (e, g') \rfloor \Rightarrow e \in_a g$

(spec_coal_2) `coal` $g = \lfloor (e, g') \rfloor \Rightarrow$ `Aff` e

(spec_coal_3) `coal` $g = \lfloor (e, g') \rfloor \Rightarrow \exists$ `Hin`, $\exists$ `Haff`, $g' =$ `merge` $e\ g$ `Hin Haff`

La fonction `freeze` doit renvoyer un sommet du graphe lié par au moins une affinité et le graphe résultant du gel de ce sommet.

(spec_freeze_1) `freeze` $g = \lfloor (v, g') \rfloor \Rightarrow$ `move_related` $v\ g\ =\ true$

(spec_freeze_2) `freeze` $g = \lfloor (v, g') \rfloor \Rightarrow g' =$ `delete_affinities` $v\ g$

Enfin, la fonction `spill` doit renvoyer un sommet du graphe et le graphe issu de la suppression de ce sommet du graphe.

(spec_spill_1) `spill` $g = \lfloor (v, g') \rfloor \Rightarrow v\ \in_s\ g$

(spec_spill_2) `spill` $g = \lfloor (v, g') \rfloor \Rightarrow g' =$ `remove_vertex` $v\ g$

5.2.2.2 Implantation

La construction des fonctions auxiliaires `simp`, `coal`, `freeze` et `spill` est analogue à celle présentée lors de la description de l'implantation de l'IRC. Chacune des fonctions fait appel à la fonction définie par l'utilisateur pour trouver un sommet (ou une affinité dans le cas de la fusion) sur lequel peut être appliquée l'opération de simplification, de gel ou de vidage. Le résultat est le couple contenant le sommet (ou l'affinité) sélectionné(e) et le graphe après application de l'opération appropriée.

A la différence de l'implantation de l'IRC, qui nécessitait l'utilisation de types dépendants pour la construction des ensembles de travail, les fonctions `simp`, `freeze` et `spill` n'en font pas usage. En revanche, la fonction `coal` en conserve le besoin puisque la fonction `merge` est de type dépendant alors que les `remove_vertex` et `delete_affinities` ne le sont pas. Il faut donc créer les hypothèses `Hin` et `Haff` de la spécification **(spec_coal_3)**.

Les corps de ces quatre fonctions sont définis figure 5.2. Leur correction repose uniquement sur les fonctions définies par l'utilisateur, ce qui nous permet de déduire la spécification que doit respecter chacune de ces dernières.

5.2.3 Spécification des fonctions de choix de l'utilisateur

Les spécifications des fonctions auxiliaires `fsimp`, `fcoal`, `ffreeze` et `fspill` sont restreintes aux propriétés indispensables pour établir la correction des fonctions `simp`, `coal`, `freeze` et `spill`.

La fonction `fsimp` ne doit donc vérifier qu'une seule propriété : le sommet renvoyé doit appartenir au graphe.

(spec_fsimp) `fsimp` $g = \lfloor v \rfloor \Rightarrow v\ \in_s\ g$

La fonction `fcoal` doit renvoyer une affinité appartenant au graphe.

(spec_fcoal_1) `fcoal` $g = \lfloor e \rfloor \Rightarrow e\ \in_a\ g$

(spec_fcoal_2) `fcoal` $g = \lfloor e \rfloor \Rightarrow$ `Aff` e

La fonction `ffreeze` doit renvoyer un sommet du graphe lié par au moins une affinité.

```
Definition simp g : option (Register.t * Graph.t) :=
match fsimp with
| ⌊r⌋ → ⌊(r, remove_vertex r g)⌋
| ∅ → ∅
end.

Definition coal g : option (Edge.t * Graph.t) :=
match fcoal g as e
return (fcoal g = e ⇒ option (Edge.t * Graph.t)) with
| ⌊edge⌋ → fun H : fcoal g = ⌊edge⌋ →
          ⌊(edge, merge edge g (fcoal1 g edge H)(fcoal2 g edge H))⌋
| ∅ → fun H : fcoal g = ∅ → ∅
end (refl_equal (fcoal g)).

Definition freeze g : option (Register.t * Graph.t) :=
match fdel g with
| ⌊x⌋ → ⌊(x, delete_affinities x g)⌋
| ∅ → ∅
end.

Definition spill g : option (Register.t * Graph.t) :=
match fspill g with
| ⌊r⌋ → ⌊(r, remove_vertex r g)⌋
| ∅ → ∅
end.
```

FIG. 5.2 – Implantation des fonctions auxiliaires.

(spec_ffreeze) `ffreeze` $g = \lfloor v \rfloor \Rightarrow$ `move_related` $v\ g\ =\ true$

Enfin, la fonction `fspill` doit renvoyer un sommet du graphe.

(spec_spill) `fspill` $g = \lfloor v \rfloor \Rightarrow v\ \in_s\ g$

5.2.4 Les fonctions de coloration

Les fonctions de coloration doivent étendre la coloration du graphe de façon correcte, si celles-ci sont appelées dans les conditions nominales d'utilisation, *i.e.* si elles sont appliquées au résultat de la fonction de choix correspondant à la même opération. La précoloration doit être une coloration valide du graphe.

(spec_colsimp) `fsimp` $g = \lfloor v \rfloor \wedge$ `proper_col` col (`remove_vertex` $v\ g$) $pal \Rightarrow$
`proper_col` (`colsimp` $col\ v\ g\ pal$) $g\ pal$

(spec_colcoal) `fcoal` $g = \lfloor e \rfloor \wedge$ `proper_col` col (`merge` $e\ g$ `Hin` `Haff`) $pal \Rightarrow$
`proper_col` (`colcoal` $col\ e\ g\ pal$) $g\ pal$

(spec_coldel) `fdel` $g = \lfloor v \rfloor \wedge$ `proper_col` *col* (`delete_affinities` v g) *pal* $\Rightarrow$
`proper_col` (`coldel` *col* v g *pal*) g *pal*

(spec_colspill) `fspill` $g = \lfloor v \rfloor \wedge$ `proper_col` *col* (`remove_vertex` v g) *pal* $\Rightarrow$
`proper_col` (`colspill` *col* v g *pal*) g *pal*

(spec_precoloring) `proper_col precoloring` g *pal*

Ce sont donc en tout neuf fonctions et dix propriétés qui doivent être implantées par l'utilisateur afin de complètement définir l'algorithme et automatiquement en assurer la terminaison et la correction.

5.3 Terminaison

Dans les études de cas de l'IRC et de la coloration gourmande [Rob07, BRS08], nous avons prouvé la terminaison de chacune des fonctions grâce à une mesure du graphe. Ici, nous utilisons un ordre bien fondé inspiré d'une relation d'ordre connue de théorie des graphes.

Définition 37 (Mineur d'un graphe)**.** *Un graphe G' est un mineur d'un graphe G, noté $G' <_m G$, si G' peut être obtenu à partir de G par une suite finie d'opérations de suppression de sommet, de suppression d'arête ou de fusion de sommets liés par une arête.*

Théorème 11 (Ordre des mineurs)**.** *La relation $<_m$ est une relation d'ordre strict partielle sur les graphes.*

Le fait que la relation $<_m$ ne soit que partielle n'est pas important du moment que les arguments des appels récursifs consécutifs sont comparables. La décroissance des arguments lors de chaque appel récursif est immédiate pour la suppression de sommet (simplification et vidage) et pour la fusion. En revanche, la décroissance pour le gel est évidente mais pas immédiate. C'est pourquoi nous avons défini une nouvelle relation qui colle de plus près à nos besoins.

Définition 38 (Ordre dédié à l'allocation de registres)**.** *Un graphe d'interférence-affinité G' est dit plus petit qu'un graphe d'interférence-affinité G, noté $G' <_r G$, si G' peut être obtenu à partir de G par une suite finie d'opérations de suppression de sommet, de suppression de toutes les affinités incidentes à un sommet qui en possède ou de fusion de sommets liés par une affinité.*

Plus formellement, cet ordre est défini dans Coq par induction et possède trois constructeurs.

1. $\forall\ g\ v,\ v \in_s g \Rightarrow$ `remove_vertex` $v\ g <_r g$
2. $\forall\ g\ v,$ `move_related` $g\ v = true \Rightarrow$ `delete_affinities` $v\ g <_r g$
3. $\forall\ g\ e$ (`Hin` : $e \in_a g$) (`Haff` : `Aff` e), `merge` $e\ g$ `Hin Haff` $<_r g$

Nous avons prouvé qu'il s'agit bien d'un ordre bien fondé dans Coq . Pour ce faire, il est nécessaire de disposer des hypothèses de chaque constructeur. Ce fait est intuitif :

pour que le graphe décroisse effectivement il faut retirer un sommet qui lui appartient, supprimer toutes les affinités incidentes à un sommet du graphe qui en possède au moins une, ou encore fusionner une affinité qui lui appartient. Ces hypothèses sont assurées par les propriétés que l'utilisateur a dû précédemment prouver sur ses fonctions de choix. La preuve de bonne fondation de cet ordre, nommé `alloc_order` dans le développement, utilise une correspondance avec la mesure du graphe définie comme la somme du nombre de sommets et du nombre d'arêtes du graphe.

Théorème 12. *La relation $<_r$ est une relation d'ordre strict partielle bien fondée sur les graphes d'interférence-affinité.*

Démonstration. Soit f la fonction définie sur les graphes d'interférence-affinité par la relation $f(G) = n(G)+m(G)$, où $n(G)$ et $m(G)$ désignent respectivement le nombre de sommets et le nombre d'arêtes de G. Nous allons montrer que $G <_r G'$ implique $f(G) < f(G')$. La preuve est faite par induction sur la construction du prédicat $<_r$. Si G' est obtenu par suppression d'un sommet de G alors $n(G') < n(G)$ et $m(G') \leq m(G)$, d'où $f(G') < f(G)$. Si G' est obtenu par suppression de toutes les affinités incidentes à un sommmet de G qui possède au moins une affinité alors $n(G') = n(G)$ et $m(G') < m(G)$, d'où $f(G') < f(G)$. Si G' est obtenu par fusion des extrémités d'une affinité de G alors $n(G') < n(G)$ et $m(G') < m(G)$, d'où $f(G') < f(G)$. A noter que dans Coq , nous nous sommes contentés de prouver que $m(G') \leq m(G)$ pour ce dernier cas. □

Prouver la décroissance selon l'ordre `alloc_order` est immédiat puisque pour chacun des appels récursifs de l'algorithme `alloc`, un constructeur de l'ordre `alloc_order` peut directement être appliqué. La terminaison de l'algorithme est donc automatiquement assurée. L'utilisateur n'a pas à s'en soucier.

Nous aurions pu utiliser cet ordre pour prouver la terminaison de l'IRC ou de la coloration gourmande. Nous avons choisi de donner une mesure car les ensembles de travail permettaient de prouver simplement la décroissance et car cette mesure apportait une borne de complexité de l'algorithme en fonction de la taille du graphe. Dans le but de généraliser, l'ordre que nous proposons semble plus adapté car la preuve est plus directe et modulaire. Si le besoin d'ajouter de nouvelles opérations autres que le retrait de sommet, le gel ou la fusion des extrémités devait être ajouté, il suffirait d'ajouter un constructeur au type inductif qui définit l'ordre `alloc_order` et de modifier la preuve de bonne fondation afin de tenir compte de ce nouveau constructeur.

5.4 Correction

La preuve est réalisée par induction sur les appels récursifs de l'algorithme, exactement comme pour l'IRC. Chaque cas est une application directe des lemmes de la spécification des fonctions de coloration, si bien que la preuve de correction ne tient qu'en quelques lignes. La figure 5.3 présente l'énoncé Coq du théorème de correction ainsi que son script de preuve.

```
Theorem correct_coloring :
        ∀ g, proper_coloring (algo g) g palette.

Proof.
intro. functional induction (algo g).
apply spec_colsimp. apply (spec_simp_1 _ _ _ e).
  rewrite ←(spec_simp_2 _ _ _ e). auto.
destruct (spec_coal_2 _ _ _ e0). rewrite H.
  apply spec_colco with (H := x). rewrite ←H. auto.
apply spec_colfree. apply (spec_freeze_1 _ _ _ e1).
  rewrite ←(spec_freeze_2 _ _ _ e1). auto.
apply spec_colspill. apply (spec_spill_1 _ _ _ e2).
  rewrite ←(spec_spill_2 _ _ _ e2). auto.
apply spec_precoloring.
Qed.
```

FIG. 5.3 – Enoncé et preuve du théorème de correction de l'algorithme `alloc`.

5.5 Fonctions prédéfinies

Afin de simplifier davantage l'implantation d'algorithmes, nous avons prédéfini quelques unes des fonctions habituellement utilisées pour décider des opérations devant être appliquées ou de la façon dont la coloration doit évoluer.

5.5.1 Les fonctions de choix

Les fonctions de choix usuelles ne sont pas très nombreuses. Elles se focalisent généralement sur les propriétés de bas degré et de liaison d'affinité d'un sommet, ou sur les critères classiques de fusion.

Nous avons défini neuf fonctions de choix pour les sommets en fonction des valeurs des fonctions `has_low_degree` et `move_related` :

1. Une fonction `any_vertex` qui choisit un sommet quelconque ;
2. Une fonction `any_low` qui choisit un sommet de bas degré ;
3. Une fonction `any_high` qui choisit un sommet de haut degré ;
4. Une fonction `any_move` qui choisit un sommet lié par une affinité ;
5. Une fonction `any_nonmove` qui choisit un sommet sans affinité ;
6. Une fonction `any_low_move` qui choisit un sommet de bas degré lié par une affinité ;
7. Une fonction `any_low_nonmove` qui choisit un sommet de bas degré sans affinité ;
8. Une fonction `any_high_move` qui choisit un sommet de haut degré lié par une affinité ;
9. Une fonction `any_high_nonmove` qui choisit un sommet de haut degré sans affinité.

Ces fonctions sont fondées sur une fonction générique `any_vertex_such_that` qui prend en paramètre une fonction booléenne `f` et un graphe `g` et retourne un sommet `v` de `g` tel

que `f v = true`, si un tel sommet existe. La fonction est partielle puisque l'existence d'un tel sommet ne peut être assurée. Sa spécification est donc la suivante.

(spec_any_vertex_such_in) `any_vertex_such_that` $f\ g\ = \lfloor v \rfloor \Rightarrow v\ \in_s\ g$
(spec_any_vertex_such_true) `any_vertex_such_that` $f\ g\ = \lfloor v \rfloor \Rightarrow f\ v\ =\ true$

L'implantation de cette fonction parcourt l'ensemble des sommets du graphe jusqu'à trouver un sommet vérifiant la propriété parmi les sommets de G. La fonction ne termine pas immédiatement mais se contente alors de propager la solution trouvée jusqu'à avoir terminé le parcours de la structure.

```
Definition continuation f v o :=
if (is_none o) then
  if (f v) then ⌊v⌋
  else ∅
else o.

Definition any_elt_such_that (f : t ⇒ bool) s :=
fold (continuation f) s ∅.

Definition any_vertex_such_that (f : Vertex.t ⇒ bool) g :=
any_elt_such_that f (V g).
```

Cette fonction permet à l'utilisateur de définir d'éventuelles autres fonctions de choix, en spécifiant via la fonction `f` comment le sommet doit être choisi. Ces fonctions doivent être accompagnées de leurs preuves de correction, lesquelles découlent de la correction de la fonction `any_vertex_such_that`.

En ce qui concerne la fusion, les critères de Briggs et de George, ainsi que l'alliance de ces deux critères sont prédéfinis.

5.5.2 Les fonctions de coloration

Les fonctions de coloration classiques que nous avons implantées sont au nombre de quatre :

1. Une fonction `available_color` qui permet si possible d'assigner une couleur n'étant actuellement pas affectée à l'un des voisins d'interférence. Cette fonction peut être utilisée pour répondre à une simplification ou un vidage optimiste ;
2. Une fonction `affinity_color` qui choisit la couleur qui satisfait le plus grand nombre d'affinités possible en fonction des couleurs affectées aux sommets voisins. Cette fonction peut être utilisée pour la simplification ou le vidage ;
3. Une fonction `no_color` qui laisse la coloration inchangée, qui peut être utilisée lors d'un gel ;
4. Une fonction `same_color` qui affecte à deux sommets la même couleur, principalement dévolue à la fusion.

Certaines fonctions de coloration, comme `available_color` par exemple, sont correctes de par leur définition. D'autres, comme la fonction `same_color` sont correctes parce qu'appliquées uniquement dans des conditions qui permettent d'établir leur correction.

5.6 Exemples

Pour montrer à quel point il devient facile d'implanter un algorithme d'allocation de registres par coloration de graphe grâce à cette bibliothèque, nous l'avons utilisée pour définir deux algorithmes classiques d'allocation de registres, l'algorithme de Chaitin et l'IRC.

5.6.1 L'algorithme de Chaitin

L'algorithme de Chaitin n'opère que de la simplification et du vidage. Un sommet est candidat à la simplification s'il est de bas degré dans le graphe, sinon il est candidat au vidage. Les fonctions de choix que l'utilisateur doit définir sont donc :

```
Definition fsimp := any_low.

Definition fcoal (g : Graph.t) : (option Edge.t) := ∅.

Definition ffreeze (g : Graph.t) : (option Register.t) := ∅.

Definition fspill := any_vertex.
```

Les preuves de correction pour les quatre fonctions sont immédiates et tiennent en une ligne de script Coq chacune.

Les fonctions de coloration sont inutiles pour le vidage et le gel puisque les fonctions de choix renvoient toujours $\emptyset$. Pour la simplification et le vidage, la fonction de coloration est la même : le sommet est coloré avec une couleur non déjà attribuée à un sommet de son voisinage, s'il en existe une.

```
Definition colsimp := available_color.

Definition colcoal := no_color.

Definition colfree := no_color.

Definition colspill := available_color.
```

Les preuves de correction de chacune de ces fonctions sont également triviales et tiennent en une ligne de script Coq .

L'implantation d'une version formellement vérifiée et extractible en Ocaml de l'algorithme de Chaitin est alors disponible. Il suffit pour cela d'instancier `alloc` avec toutes les variables qui viennent d'être définies, comme présenté ci-dessous.

```
Definition chaitin_heuristic := alloc
                        fsimp fcoal fdel fspill
                        palette precoloring
                        colsimp colcoal colfreeze colspill
                        spec_fsimp spec_fspill
```

```
                    spec_fcoal_1 spec_fcoal_2
                    spec_ffreeze.
```

Cette implantation n'aura nécessité qu'une vingtaine de lignes de preuve et, en tout et pour tout, environ une centaine de lignes à l'utilisateur.

5.6.2 L'IRC

Le second algorithme que nous avons implanté grâce à cette méthode est l'IRC. Dans ce cas, il est nécessaire de définir quatre fonctions de choix. Celles-ci font partie de celles prédéfinies dans l'environnement.

```
Definition fsimp := any_low_nonmove.

Definition fcoal := any_coalescible_affinity.

Definition ffreeze := any_low_move.

Definition fspill := any_vertex.
```

De même, les fonctions de coloration sont des fonctions usuelles et font partie des fonctions que nous avons prédéfinies.

```
Definition colsimp := available_color.

Definition colcoal := same_color.

Definition colfree := no_color.

Definition colspill := available_color.
```

Comme pour l'algorithme de Chaitin, il aura uniquement fallu une centaine de lignes de code Coq pour obtenir une version formellement vérifiée de l'IRC.

5.7 Note sur l'optimisation

Notre bibliothèque permet de définir facilement des algorithmes classiques d'allocation de registres. Cependant, cette facilité se créé au dépend des optimisations internes aux algorithmes, lesquelles ne peuvent facilement être abstraites. Pour l'algorithme de Chaitin, la version obtenue via l'utilisation de notre bibliothèque est proche d'une implantation optimisée. En revanche, l'IRC est un algorithme au sein duquel les optimisations ont une place prépondérante. L'usage et la mise à jour des ensembles de travail détaillés dans le chapitre précédent sont essentiels à l'efficacité de l'algorithme et requièrent un travail conséquent sur les structures de données, les algorithmes et les preuves qui ne peut être intégré directement via notre implantation générique.

5.8 Bilan

Nous avons décrit, implanté et prouvé correct un algorithme générique d'allocation de registres par coloration de graphe abstrayant des algorithmes classiques et avancés d'allocation de registres comme les algorithmes de Chaitin, de Briggs, ou encore l'IRC. Instancier l'algorithme générique en ces algorithmes est d'ailleurs aisé, puisqu'il ne faut qu'une centaine de lignes de code pour ce faire. Cet outil rend accessible à un utilisateur peu familier du système Coq la vérification d'un de ses algorithmes d'allocation de registres, pourvu que celui-ci entre dans le schéma que nous avons considéré. La modularité de l'algorithme permet en outre de comparer des stratégies d'allocation de registres par coloration de graphe très simplement au sein d'un même environnement. Bien sûr, tous les algorithmes ne le suivent pas et le travail à réaliser pour prouver d'autres algorithmes d'allocation de registres ou des versions plus optimisées des mêmes algorithmes reste considérable.

Chapitre 6

Optimisation de la fusion dans les graphes SSA par programmation linéaire

Depuis l'avènement de la forme SSA comme représentation intermédiaire de programme de référence dans les compilateurs et la mise en évidence du caractère triangulé des graphes d'interférence des programmes sous forme SSA [BDGR05, Hac05], la recherche en allocation de registres s'est recentrée sur le problème de fusion et ses différentes variantes [GH07, BVI07, HG08, BDR08] dans ces graphes. Une vaste étude de complexité de la fusion dans les graphes quelconques, les graphes K-colorables, les graphes gloutons K-colorables[1] et dans les graphes triangulés est décrite dans [BDR07a].

Les deux variantes les plus classiques pour la fusion sont la fusion agressive, où le but est de satisfaire un ensemble d'affinités de poids maximal, et la fusion conservative, où l'objectif est de satisfaire un ensemble d'affinités de poids maximal et de conserver la K-colorabilité du graphe d'interférence-affinité. Si ces deux problèmes sont évidemment différents, il existe un point de jonction qui correspond au cas où la fusion agressive préserve la K-colorabilité du graphe. Mieux connaître où cette jonction intervient permettrait d'utiliser des algorithmes plus adaptés. Ceci permettrait en particulier de séparer les phases de fusion et d'affectation aux registres sans perte de qualité théorique de l'allocation de registres. De plus, la fusion conservative, souvent trop prudente, pourrait alors être immédiatement écartée si l'on peut assurer que la fusion agressive lui est équivalente. Assurer l'équivalence de ces deux stratégies de fusion passe indubitablement par l'étude de la configuration globale des affinités du graphe, ce qui laisse à penser qu'il faudrait supplanter l'approche locale de fusion, affinité après affinité, par une approche plus globale.

Nous soutenons au travers de ce chapitre l'idée qu'une approche plus globale de la fusion s'appuyant sur des propriétés des affinités permettrait d'améliorer significativement les méthodes de résolution de ce problème. Considérer la configuration des affinités nous a permis d'identifier une classe de graphes dans laquelle fusions agressive et conservative sont

[1] Un graphe glouton K-colorable est un graphe pouvant être coloré avec K couleurs par l'algorithme de coloration gourmande.

équivalentes. Nous présentons ce résultat, analysons comment celui-ci peut être étendu aux graphes SSA, et en déduisons une formulation optimale de la fusion conservative dans ces graphes par programmation linéaire.

6.1 De l'impact de la configuration des affinités

Considérer localement les affinités conduit à une approche gloutonne qui peut se révéler relativement peu efficace en pratique. Afin de bien mettre en avant ce fait, nous présentons deux preuves de complexité de la fusion conservative dans des graphes très simples à colorer, puisque leurs graphes d'interférence sont bipartis, *i.e.* 2-colorables. La complexité du problème ne peut dans ce cas provenir que des affinités.

6.1.1 Preuve de complexité pour $K \geq 3$ dans les graphes d'interférence bipartis

Nous devons avant tout mentionner qu'un résultat très proche a auparavant été prouvé dans [BDR07a]. Néanmoins, ce n'est pas tant le résultat que la façon d'y arriver, légèrement différente et plus détaillée ici, qui nous intéresse.

Définition 39 (Forêt d'étoiles)**.** *Une étoile est un arbre tel qu'il existe un sommet qui est une extrémité de toute arête.*
Une forêt d'étoiles est un graphe dont chaque composante connexe est une étoile.

Théorème 13 (Complexité pour $K \geq 3$ dans les graphes d'interférence bipartis)**.** *La fusion conservative est NP-difficile pour $K \geq 3$ et des poids unitaires, même si le graphe d'interférence est un arbre et que le graphe d'affinité est une forêt d'étoiles.*

Démonstration. La preuve est réalisée par réduction du problème de K-coloration de graphe. Il nous faut donc montrer que tout problème de K-coloration de graphe peut être réduit en un problème équivalent de fusion conservative dans un graphe d'interférence-affinité tel que le graphe d'interférence est un arbre et le graphe d'affinité est une forêt d'étoiles. Soit G un graphe à colorer avec K couleurs. Nous supposons sans perte de généralité que G est connexe. Nous définissons la valeur d'affinité d'une coloration comme la somme des poids des affinités satisfaites par celle-ci et construisons un graphe d'interférence-affinité G' tel que G est K-colorable si et seulement si G' a une K-coloration de valeur d'affinité $\nu(G) = m(G) - n(G) + 1$, où $m(G)$ est le nombre d'arêtes de G et $n(G)$ est le nombre de sommets de G.

Soit T un arbre couvrant de G (*i.e.* un arbre dont les sommets sont ceux de G et les arêtes sont un sous-ensemble de celles de G) et $\{e_1, \ldots, e_{\nu(G)}\}$ les arêtes de $E(G) - E(T)$[2]. Pour $i \in \{0, \ldots, \nu(G)\}$, nous définissons G_i comme suit :

- $G_0 = G$
- pour $i \in \{1, \ldots, \nu(G)\}$, G_i est obtenu en supprimant $e_i = (x_i, y_i)$ de G_{i-1}, et en ajoutant un sommet $x_i y_i$ lié par une interférence à x_i et par une affinité de poids unitaire à y_i.

[2]Cet ensemble contient exactement $\nu(G)$ arêtes.

G' est simplement défini comme $G_{\nu(G)}$. La figure 6.1 présente cette réduction sur un exemple.

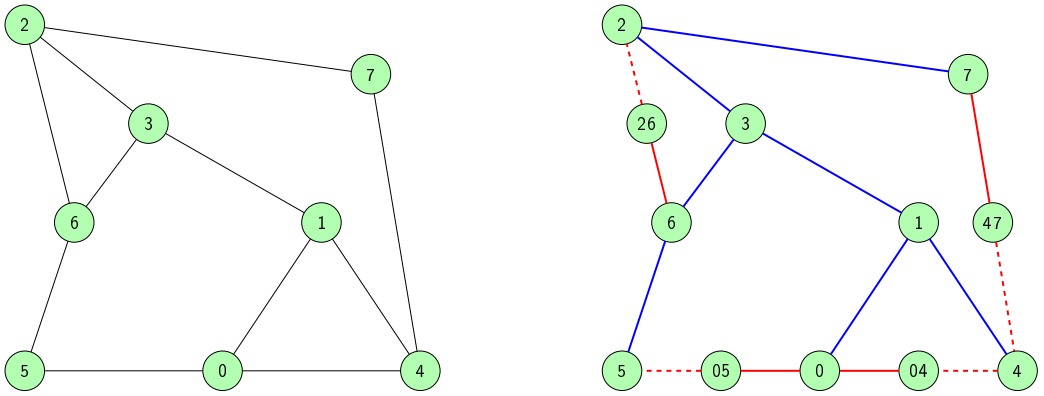


FIG. 6.1 – Exemple de construction de G' (à droite) à partir de G (à gauche). Les arêtes de l'arbre couvrant T sont en bleu tandis que les arêtes ajoutées lors de la réduction sont en rouge.

Lemme 28 (Graphe d'interférence de G')**.** *Le graphe de d'interférence de G' est un arbre.*

Démonstration. Nous prouvons que toute paire de sommets de G' est liée par une unique chaîne élémentaire d'interférences.
Soient x et y deux sommets de G'. Si x et y sont des sommets de G alors il existe une unique chaîne les liant incluse dans l'arbre T. En effet, T est inclus dans le graphe d'interférence de tout graphe G_i, pour $i \in \{1, \ldots, \nu(G)\}$. Si x (et/ou y) est un sommet ajouté par la réduction il suffit de remarquer que le seul voisin d'interférence de x (et/ou y) est un sommet de T, appliquer le raisonnement ci-dessus et ajouter la seule interférence incidente à x (et/ou y). □

Lemme 29 (Graphe d'affinité de G')**.** *Le graphe d'affinité de G' est une forêt d'étoiles.*

Démonstration. Le graphe d'affinité de $G_{\nu(G)}$ est clairement biparti. Chaque affinité a une extrémité qui est un sommet de G et une extrémité qui n'en est pas un. De plus, chaque $x_i y_i$ a exactement une seule affinité qui lui est incidente, ce qui implique que le graphe d'affinité de G' est une forêt d'étoiles. □

Lemme 30 (Valeur des colorations optimales)**.** *G est K-colorable si et seulement si G' a une K-coloration de valeur $\nu(G)$.*

Démonstration. Nous généralisons le résultat pour le prouver par induction sur i. Nous prouvons précisément que pour tout $i \in \{0, \ldots, \nu(G)\}$, G est K-colorable si et seulement si G_i a une K-coloration de valeur d'affinité i.

– $i = 0$: trivial puisque $G = G_0$.

– $i+1$: G_{i+1} a une K-coloration de valeur $i+1$ si et seulement s'il a une K-coloration qui satisfait toutes ses affinités. En particulier, l'affinité $(x_{i+1}y_{i+1}, y_{i+1})$ est satisfaite. Fusionner les extrémités de cette affinité mène au graphe G_i qui, par hypothèse d'induction possède une K-coloration de valeur d'affinité i si et seulement si G est K-colorable. Ainsi, G_{i+1} admet une K-coloration de valeur $i+1$ si et seulement si G est K-colorable.

□

□

Cette réduction s'appuie ici sur le problème de coloration. L'on voit bien que toute la complexité du problème peut facilement être transférée des interférences aux affinités. Considérer uniquement les propriétés globales des interférences, comme c'est le cas actuellement dans toutes les démarches de résolution de la fusion dans les graphes SSA semble donc recéler une faiblesse.

6.1.2 Preuve de complexité pour $K = 2$ dans les graphes d'interférence bipartis

La réduction que nous avons présentée ci-dessus s'appuie sur la complexité du problème de coloration, que l'on croît souvent être la principale source de complexité du problème de fusion. Nous avons donc voulu étudier ce qu'il en était pour $K = 2$, cas dans lequel la coloration de graphe est polynomiale quel que soit le graphe, puisqu'il existe un algorithme complet de recherche de 2-coloration de graphe 2-colorable. La complexité ne peut donc alors être le fruit que des affinités.

Théorème 14 (Complexité pour $K = 2$)**.** *La fusion conservative est NP-difficile pour $K = 2$ dans les graphes 2-colorables.*

Démonstration. La preuve est une réduction de 3SAT, qui est connu pour être NP-difficile . Rappelons tout d'abord la définition de ce problème. Une clause est définie comme une disjonction de littéraux à valeurs booléennes. Une formule est une conjonction de clauses. La problème SAT consiste à déterminer des valeurs des littéraux qui satisfont une formule donnée, *i.e.* qui satisfont chacune des clauses de cette formule. Le problème 3SAT est une particularisation du problème SAT dans laquelle chaque clause possède exactement 3 littéraux.

Nous considérons une instance 3SAT contenant un nombre n de clauses et construisons une instance équivalente de fusion conservative. Nous débutons en construisant deux sommets T et F liés par une interférence et dédiés à représenter les valeurs booléennes vrai et faux. L'idée motrice de la réduction est d'associer à chacune des deux couleurs l'une des valeurs booléennes. Ainsi, si chaque sommet représente un littéral, la couleur d'un sommet donne la valeur de vérité du littéral qu'il représente. Suivant ce principe, nous disons qu'un littéral est satisfait s'il est coloré comme T.

Nous associons à chaque clause C de la formule un gadget tel que toute coloration discrimine si la clause C est satisfaite : si elle l'est, la valeur d'affinité doit être une certaine valeur v_t et sinon, une valeur $v_f < v_t$. Le gadget que nous définissons est en fait la somme

de deux gadgets. Le gadget global a une valeur d'affinité de $v_f = 26$ si aucun littéral de la clause n'est satisfait et de $v_t = 30$ si au moins un littéral est satisfait. Autrement dit, la valeur d'affinité est égale à 30 si la clause est satisfaite et à 26 sinon.

Le premier gadget discrimine les cas où un ou deux littéraux sont satisfaits, induisant dans ce cas une valeur d'affinité de 4, des deux autres cas, où la valeur d'affinité est alors nulle.

Etant donnée une clause C, ce gadget est construit comme suit : pour chaque littéral l_i de C, nous construisons deux sommets (l_i, C) et $(\overline{l_i}, C)$ liés par une interférence, où $\overline{l_i}$ désigne le complément de l_i. Nous ajoutons également certaines affinités de poids 2 entre (l_i, C) et $(\overline{l_j}, C)$ comme décrit figure 6.2 au sein du gadget complet.

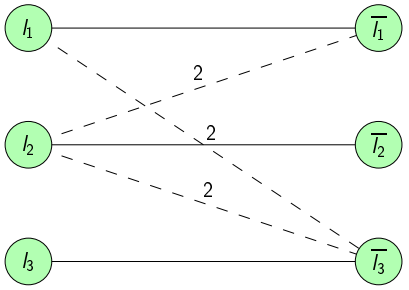


FIG. 6.2 – Le premier gadget. Si on ne considère que 2 couleurs, ce gadget rapporte une valeur d'affinité nulle si tous les sommets de gauche ont la même couleur et une valeur d'affinité de 4 sinon.

Le second gadget discrimine le cas où les trois littéraux sont satisfaits des autres. Le cas échéant, la valeur d'affinité est égale à 30. Sinon, celle-ci est seulement égale à 26.

Etant donnée une clause C, le second gadget requiert de créer un nouveau sommet caractéristique appelé B_C. Ce sommet agit comme un balancier. Sa couleur bascule de celle représentant faux à celle représentant vrai si et seulement si les trois littéraux de C sont satisfaits. Une illustration de ce gadget est présentée figure 6.3. Le gadget final est quant à lui représenté figure 6.4.

Nous disposons maintenant de tous les gadgets, chacun représentant une clause. Le graphe est quasi construit. Reste à prendre garde à assigner à deux sommets (l_i, C) et (l_i, C') la même couleur. Pour assurer cette nécessité, nous ajoutons des affinités dites de cohérence entre ces sommets. Plus précisément nous créons une chaîne d'affinité de cohérence contenant tous ces sommets. Ces affinités de cohérence ont un poids très fort $w = 30n + 1$.

La taille du graphe ainsi construit est clairement polynomiale en le nombre de clauses. Montrons maintenant que résoudre la fusion conservative avec 2 couleurs est équivalent à décider si la formule 3SAT initiale a une solution. Nous prouvons dans un premier temps que toute 2-coloration optimale satisfait toutes les affinités de cohérence.

Lemme 31 (Affinité de cohérence)**.** *Soit l_i un littéral apparaissant dans deux clauses C et C' de l'instance 3SAT. Toute solution optimale du problème de fusion conservative affecte*

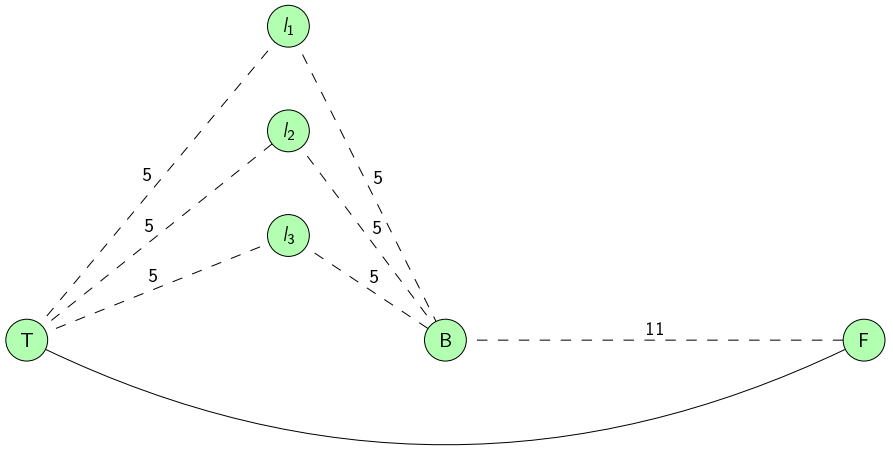


FIG. 6.3 – Le second gadget. Dans toute 2-coloration optimale, B_C est coloré comme T si et seulement si tous les littéraux de la clause sont colorés comme T. Ainsi, toute coloration optimale de ce gadget possède une valeur d'affinité de 30 si les trois littéraux sont satisfaits et de 26 si l'un des littéraux est contraint à être coloré comme F.

la même couleur aux sommets (l_i, C) *et* (l_i, C').

Démonstration. Soient p le nombre d'affinités de cohérence du graphe et $\mathcal{C}$ une coloration de valeur v qui satisfait toutes les affinités de cohérence. Nous avons immédiatement la relation

$$v \geq pw = p(30n + 1)$$

Soit maintenant $\mathcal{C}'$ une coloration de valeur d'affinité v' qui ne satisfait pas toutes les affinités de cohérence. En remarquant que dans le meilleur des cas chaque clause fait augmenter la valeur d'affinité de la solution de 30, nous avons :

$$v' \leq (p-1)w + 30n = (30n+1)p - 1 \leq v - 1$$

Toute coloration optimale satisfait donc toutes les affinités de cohérence. □

Lemme 32 (Valeur optimale)**.** *La formule 3SAT a une solution si et seulement si l'instance de fusion conservative a une solution de valeur* $pw + 30n$.

Démonstration. Remarquons tout d'abord que $pw + 30n$ est une borne supérieure de la valeur de toute solution, puisque le graphe ne contient que n clauses qui peuvent chacune rapporter 30 et p affinités de cohérence qui peuvent chacune rapporter w. De plus, une solution de valeur $pw + 30n$ peut être transformée en une solution de la formule 3SAT et

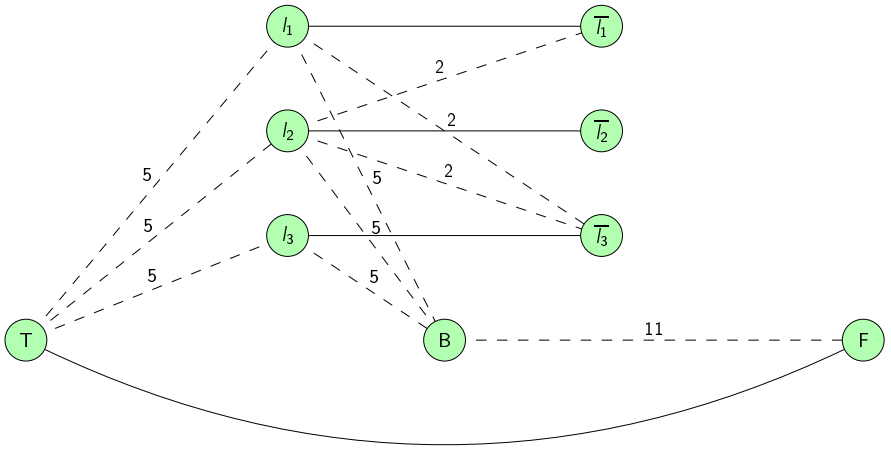


FIG. 6.4 – Le gadget final obtenu par fusion des deux gadgets précédemment décrits. La valeur d'affinité de toute coloration optimale est égale à 30 si au moins un littéral peut être coloré comme T et est égale à 26 sinon.

réciproquement. Les sommets colorés comme T représentent des littéraux satisfaits et ceux colorés comme F les littéraux non satisfaits. □

□

Même si ce problème est NP-difficile , il existe un algorithme très simple et polynomial si le nombre de composantes connexes d'interférence est fixé. En particulier, si le graphe ne contient qu'une composante connexe d'interférence, le problème est polynomial, ce qui est évident puisqu'un tel graphe n'admet que deux 2-colorations.

Théorème 15 (Complexité dans les graphes d'interférence bipartis)**.** *La fusion conservative est polynomiale pour $K = 2$ si le graphe d'interférence est biparti connexe. Plus généralement, il peut être résolu en $O(m + 2^q n)$ où q désigne le nombre de composantes connexes d'interférence du graphe.*

Démonstration. La preuve repose sur un algorithme simple. Nous colorons d'abord chaque composante connexe avec deux couleurs. Pour chaque composante, il n'existe que deux 2-colorations, calculables en $O(m)$. Il suffit ensuite d'énumérer toutes les combinaisons de ces 2-colorations, au nombre de 2^q. Chacune de ces colorations peut être réalisée au pire en $O(n)$. □

Le constat est ici simple : les affinités sont une source de complexité aussi importante, voire plus encore, que la coloration. C'est de ce fait que nous sommes partis pour chercher sous quelles conditions séparer la recherche des affinités satisfaites de la coloration, *i.e.* séparer la fusion de l'affectation aux registres, était envisageable. La réponse à cette question est simple : il faut et il suffit que la fusion agressive soit conservative.

6.2 Préliminaires

6.2.1 Séparation de la fusion et de l'affectation aux registres

Introduisons tout d'abord la terminologie nécessaire à la suite de ce chapitre. L'allègement de notre discours nécessite l'introduction de deux nouveaux concepts, les ensembles d'affinités compatibles et le graphe fusionné d'un tel ensemble.

Tout ensemble d'affinités ne peut pas être satisfait dans son entier. Ce résultat bien connu peut s'observer en présence d'une coupe de chaîne par exemple. Un ensemble d'affinités qui peuvent être simultanément satisfaites sera par la suite dit compatible. Chaque ensemble d'affinités compatible correspond exactement à une fusion admissible pour le graphe.

Définition 40 (Ensemble d'affinités compatible)**.** *Un ensemble d'affinité est dit compatible dans un graphe si toutes les affinités de cet ensemble peuvent être simultanément satisfaites, i.e. ne sont pas en conflit avec une interférence ; autrement dit, s'il existe une coloration du graphe (utilisant un nombre quelconque de couleurs) satisfaisant toutes les affinités de l'ensemble.*

Une fois la fusion opérée, c'est la coloration qui entre en scène. Afin d'alléger notre discours, nous introduisons une seconde définition, celle du graphe fusionné d'un ensemble d'affinités compatible. C'est le graphe qui doit être coloré après la fusion.

Définition 41 (Graphe fusionné d'un ensemble d'affinités compatible)**.** *Le graphe fusionné d'un ensemble d'affinités compatible E est le graphe obtenu après fusion de chacune des extrémités des affinités de E.*

Cette nouvelle terminologie nous permet de redéfinir fusions conservative et agressive sous une forme plus intuitive pour aborder cette étude. Etant donné un graphe d'interférence-affinité, la fusion agressive consiste à déterminer un ensemble d'affinités compatible de poids maximal tandis que la fusion conservative consiste à déterminer un ensemble d'affinités compatible de poids maximal dont le graphe fusionné est K-colorable.

6.2.2 Largeur d'arbre d'un graphe

Nombre de problèmes combinatoires NP-difficiles deviennent polynomiaux dans la classe des graphes triangulés, comme par exemple les problèmes de coloration minimale, de clique maximum ou de stable maximum [Gav72]. De là est apparue l'idée que, pour de tels problèmes, plonger un graphe dans un graphe triangulé puis résoudre le problème dans ce dernier graphe permettrait de calculer des solutions de bonnes qualité pour le premier graphe. C'est ce que nous appellerons le principe de triangulation.

Définition 42 (Triangulation d'un graphe). *Soit $G = (V, E)$ un graphe. Une triangulation de G est un graphe triangulé $G' = (V, E \cup E')$.*

La résolution de problèmes dans les graphes triangulés repose lourdement sur un paramètre essentiel d'un graphe triangulé : sa largeur d'arbre.

Définition 43 (Décomposition arborescente d'un graphe triangulé). *La décomposition arborescente d'un graphe triangulé est la représentation de ce graphe sous forme d'intersection de sous-arbres d'un arbre. Ce dernier arbre est appelé support de la décomposition.*

Définition 44 (Largeur d'arbre d'un graphe triangulé). *La largeur d'une décomposition arborescente $(T, \{T_1, \ldots, T_j\})$ est la taille de l'intersection maximale des sous-arbres de T, moins 1. La largeur d'arbre d'un graphe triangulé est la largeur de n'importe laquelle de ses décompositions arborescentes.*

Par exemple, la largeur d'arbre du graphe de la figure 6.5 est 2.

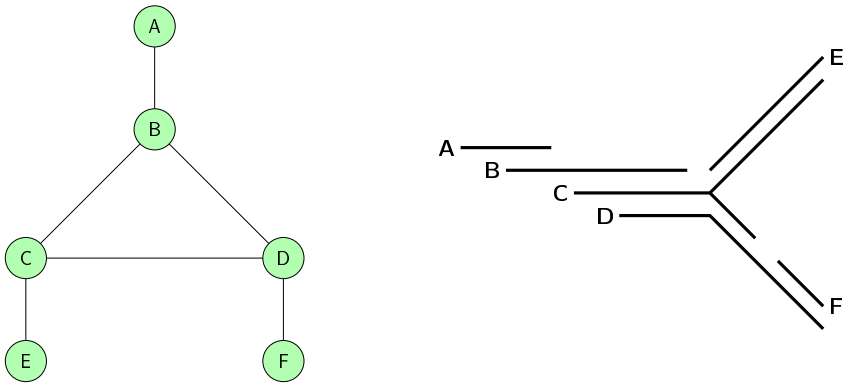


FIG. 6.5 – Un graphe triangulé et l'une de ses décompositions arborescentes. La décomposition indique que la largeur d'arbre du graphe est égale à 2, puisque la plus grande intersection ne contient que trois sous-arbres.

En suivant le principe de triangulation, la définition de la largeur d'arbre est étendue à n'importe quel graphe.

Définition 45 (Largeur d'arbre d'un graphe et p-arbres partiels). *Soit $G = (V, E)$ un graphe. La largeur d'arbre de G, notée $tw(G)$, est le minimum des largeurs d'arbres des triangulations de G.*
Etant donné un entier p, un p-arbre partiel est un graphe de largeur d'arbre au plus p.

Les triangulations de graphe d'interférence-affinité que nous allons définir dans ce chapitre ne font qu'ajouter des affinités au graphe. Ainsi, nous parlerons non pas de graphes d'interférence triangulés, mais de graphes d'interférence-affinité parfaitement triangulés.

Définition 46 (Graphe d'interférence-affinité parfaitement triangulé). *Un graphe d'interférence-affinité $G = (V, I, A)$ est dit parfaitement triangulé si $G' = (V, I \cup A)$ est triangulé.*

6.3 Optimisation dans les $(K-1)$-arbres partiels

Dans cette partie, nous montrons que la fusion agressive est optimale dans les $(K-1)$-arbres partiels. Autrement dit, la fusion agressive est conservative dans ces graphes. Ce résultat induit que, dans cette classe de graphes, la fusion peut être séparée de l'affectation aux registres. De plus, si chacune de ces opérations est résolue de façon optimale, la fusion obtenue est optimale.

Nous prouvons ce résultat en utilisant la relation de mineur. Rappelons qu'un graphe G' est un mineur d'un graphe G si G' peut être obtenu à partir de G par une suite d'applications de l'une des trois opérations suivantes : suppression de sommet, suppression d'arête, et fusion de deux sommets liés par une arête. Lors de la fusion d'une affinité, le graphe résultant est donc un mineur du graphe initial.

De plus, la largeur d'arbre est croissante selon les mineurs.

Théorème 16 (Croissance de la largeur d'arbre [RS86])**.** *Soient deux graphes G et G'. Si $G' <_m G$ alors $tw(G') \leq tw(G)$.*

Cette propriété nous mène au fait que la fusion d'une affinité fait décroître la largeur d'arbre du graphe.

Lemme 33 (Décroissance de la largeur d'arbre par fusion)**.** *Soient G un graphe d'interférence-affinité, e une affinité désignée pour la fusion et G' le graphe obtenu après fusion des extrémités de e dans G. $tw(G') \leq tw(G)$.*

Démonstration. Par définition des mineurs, on a $G' <_m G$. La croissance de la largeur d'arbre implique donc $tw(G') \leq tw(G)$. □

Un second théorème connu vient compléter ce premier. Le nombre chromatique est toujours inférieur à la largeur d'arbre d'un graphe.

Théorème 17 (Nombre chromatique et largeur d'arbre [Wes00])**.** *Soit G un graphe. $\chi(G) \leq tw(G) + 1$.*

Ainsi, nous sommes en mesure de prouver que toute fusion dans un $(K-1)$-arbre partiel est conservative.

Théorème 18 (Equivalence de la fusion agressive et de la fusion conservative dans les $(K-1)$-arbres partiels)**.** *Soit G un graphe $(K-1)$-arbre partiel, e une affinité désignée pour la fusion et G' le graphe obtenu après fusion des extrémités de e dans G. G' est K-colorable.*
Autrement dit, dans tout $(K-1)$-arbre partiel la fusion agressive est conservative.

Démonstration. D'après les théorèmes précédents, $\chi(G') \leq tw(G') + 1 \leq tw(G) + 1 \leq K$. Donc G' est K-colorable. □

Ainsi, fusions conservative et agressive sont équivalentes dans cette classe de graphes. Il suffit donc de résoudre la fusion agressive, plus simple à résoudre et plus performante.

6.4 Résolution optimale de la fusion dans les $(K-1)$-arbres partiels

L'approche en deux temps peut donc être appliquée aux $(K-1)$-arbres partiels. Reste à résoudre chacune de ces deux phases.

6.4.1 Fusion agressive et multicoupe

La fusion agressive consiste à satisfaire un ensemble d'affinités compatible de poids maximal. Ce problème de fusion agressive est en fait équivalent à un problème connu d'optimisation combinatoire, le problème de multicoupe minimum (MCM).

Définition 47 (Le problème de multicoupe minimum)**.** *Soit $G = (V, E)$ un graphe pondéré et $T = \{(s_1, p_1), \ldots, (s_q, p_q)\} \subseteq V \times V$. Le problème de multicoupe minimum (V, E, T) consiste à trouver un ensemble d'arêtes de poids total minimal dont la suppression déconnecte s_i et p_i pour $i \in \{1, \ldots, q\}$.*

Le problème de multicoupe associé à un graphe d'interférence-affinité est simplement défini comme suit :

Définition 48 (Problème de multicoupe associé à un graphe d'interférence-affinité)**.** *Soit $G = (V, I, A)$ un graphe d'interférence-affinité. Le problème de multicoupe associée à G, notée $MC(G)$, est défini comme l'instance de multicoupe minimum (V, A, I).*

Intuitivement, la seule contrainte pesant sur la fusion agressive est d'interdire les coupes de chaînes. Le parallèle entre les deux problèmes est alors immédiat :

Théorème 19 (Affinités non satisfaites et multicoupe)**.** *Soit $G = (V, I, A)$ un graphe d'interférence-affinité.*
L'ensemble des affinités non satisfaites dans une coloration de G forme une solution du problème de multicoupe associé à G. Réciproquement, pour toute solution du problème de multicoupe associé à G, il existe une coloration de G de même valeur d'affinité.

Démonstration. Soient x et y deux sommets liés par une interférence dans G. Ces deux sommets ne sont donc pas colorés de la même couleur. Ainsi, il n'existe pas de chaîne d'affinités satisfaites reliant x à y. L'ensemble des affinités non satisfaites dans la coloration de G forment donc une solution de l'instance de $MC(G)$.
Soit S une multicoupe de $MC(G)$ et $E = A(G) - S$. Les affinités de E forment des composantes connexes. Chacune de ces composantes peut être colorée d'une seule couleur par définition de E. En affectant une couleur différente à chacune de ces composantes, la coloration obtenue est une coloration de G et satisfait exactement les affinités de E. □

La figure 6.6 illustre la correspondance entre l'instance de fusion et le problème de multicoupe associé.

Résoudre la fusion agressive est donc équivalent à résoudre le problème de multicoupe minimum.

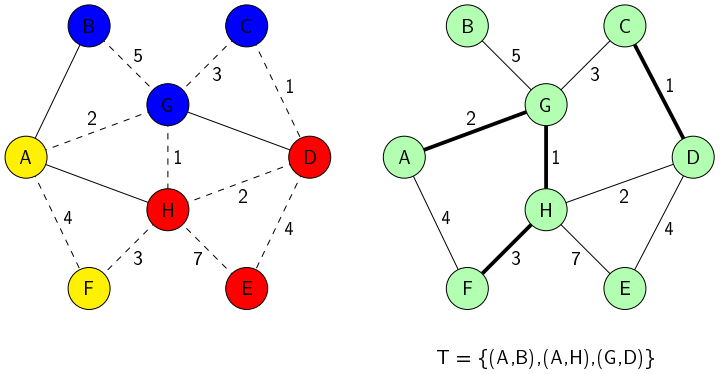


FIG. 6.6 – Un graphe d'interférence-affinité G et $MC(G)$. Le graphe est pourvu d'une 3-coloration optimale. Les arêtes en gras forment une multicoupe optimale et correspondent exactement aux affinités non satisfaites dans G.

6.4.2 Coloration

Le résultat de la fusion agressive est un ensemble d'affinités compatible ayant la particularité d'être maximal.

Définition 49 (Ensemble d'affinités compatible maximal)**.** *Un ensemble d'affinités compatible E est dit maximal dans un graphe G si aucune affinité de G ne peut être ajoutée à E sans briser la compatibilité de l'ensemble.*

Cette singularité implique une propriété intuitive sur le graphe fusionné : celui-ci ne contient que des interférences.

Théorème 20 (Graphe fusionné d'un ensemble compatible maximal)**.** *Soit G un graphe d'interférence-affinité et E un ensemble d'affinités compatible maximal. Le graphe fusionné de E ne contient que des interférences.*

Démonstration. Si le graphe fusionné de E contient une affinité alors celle-ci peut être satisfaite et donc ajoutée à E, ce qui contredit la maximalité de E. Ainsi le graphe fusionné de E ne contient que des interférences. □

Une fois un ensemble d'affinités maximal calculé, le problème est réduit à la coloration du graphe fusionné, dont l'on sait, d'après le théorème 18, qu'il est K-colorable puisque le graphe initial est un $(K-1)$-arbre partiel.

6.4.3 Résolution

Reste à déterminer le procédé de résolution à adopter pour la fusion agressive et pour la coloration. La fusion agressive et la coloration étant chacune NP-difficile , deux voies habituelles sont envisageables : la résolution heuristique ou la résolution exacte par un algorithme exponentiel, comme la programmation linéaire.

La résolution heuristique de la fusion agressive peut passer par des méthodes classiques, telles que la fusion itérative des affinités du graphe par ordre de poids décroissant par exemple, ou des heuristiques connues de multicoupe minimum.

Le théorème 18 garantit que le graphe fusionné résultant est K-colorable. Trouver une K-coloration demeure NP-difficile et une résolution heuristique peut ne pas permettre d'en trouver. Pour être sûr de K-colorer le graphe, et ainsi éviter tout vidage que l'on sait inutile, il faut donc user d'un algorithme exponentiel, tel que l'algorithme de reliement-contraction ou la programmation linéaire.

Quitte à recourir à une méthode exponentielle pour résoudre la coloration, autant en faire de même pour la fusion agressive. Celle que nous proposons s'appuie sur la programmation linéaire et le parallèle entre les problèmes de fusion agressive et de multicoupe minimum. La formulation de la multicoupe minimum est donnée figure 6.7. La variable booléenne y_e vaut 1 si et seulement si l'affinité e appartient à la multicoupe, ou, autrement dit, n'aura pas ses extrémités colorées de la même couleur. L'ensemble $P(i,j)$ contient tous les chaînes élémentaires d'affinités joignant i à j et ne contenant aucune arête d'interférence interne, *i.e.* qui lie deux sommets de la chaîne.

$$(PAG) \begin{cases} Min & \sum_{e \in A} w_e y_e \\ s.c. & \\ \forall (i,j) \in I, \forall p \in P(i,j), & \sum_{e \in p} y_e \geq 1 \\ \forall e \in A, & y_e \in \{0,1\} \end{cases}$$

FIG. 6.7 – Formulation classique du problème de multicoupe minimum.

6.5 Extension aux graphes SSA

La démarche suivie pour les $(K-1)$-arbres partiels peut être étendue aux graphes SSA en passant par une autre voie, celle de l'étude de l'évolution des décompositions arborescentes au cours de la fusion. Nous montrons dans un premier temps comment prouver sous ce nouvel angle que la fusion agressive est conservative dans les $(K-1)$-arbres partiels, puis en déduisons une généralisation pour le cas général, les graphes SSA.

6.5.1 Les $(K-1)$-arbres partiels

Considérons le graphe d'interférence-affinité G de la figure 6.8. Ce graphe n'est pas parfaitement triangulé, mais sa largeur d'arbre est 2. Il existe donc (au moins) une triangulation de G de largeur d'arbre 2. Un exemple de telle triangulation, et l'une de ses décomposition arborescente, sont données figure 6.9.

La fusion de n'importe quelle affinité conduit alors à un graphe dont une décomposition arborescente est facilement calculable. En effet, la fusion de deux sommets correspond à la fusion de deux sous-arbres de la décomposition arborescente. Cette fusion de deux sous-

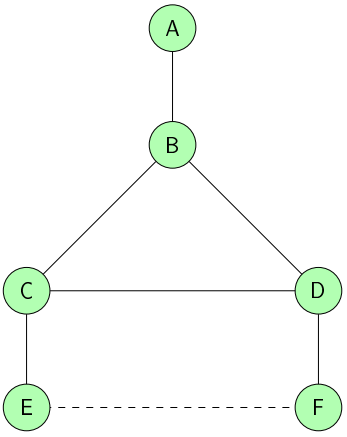


FIG. 6.8 – Le graphe n'est pas parfaitement triangulé. Le cycle $(CDEF)$ est sans corde.

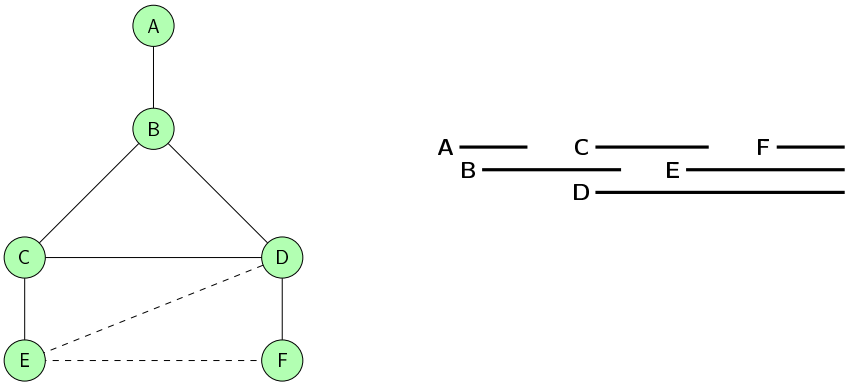


FIG. 6.9 – Une triangulation G' de G (défini figure 6.8) et une décomposition arborescente de G' de largeur d'arbre 2.

arbres mène à une nouvelle décomposition arborescente valide, puisque les deux sous-arbres fusionnés s'intersectent.

Théorème 21 (Evolution des décompositions arborescentes lors d'une fusion)**.** *Soit G un graphe parfaitement triangulé, $e = (x, y)$ une affinité de G, D une décomposition arborescente de G et G' le graphe obtenu par fusion de x et y.*
La décomposition arborescente D' obtenue par fusion des sous-arbres de x et de y dans D est une décomposition arborescente de G'.

Démonstration. Tout d'abord, D' est bien une décomposition arborescente puisque la fusion des sous-arbres de x et y donne bien un sous-arbre du support. Ceci n'est vrai que parce que les sous-arbres de x et y s'intersectent. Il nous reste à montrer que cette décomposition est correcte.
Soient v_1 et v_2 deux sommets de G'. Montrons que les sous-arbres de v_1 et v_2 s'intersectent dans D' si et seulement si v_1 et v_2 sont voisins dans G'.
Supposons d'abord que les sous-arbres de v_1 et v_2 s'intersectent dans D' et montrons que v_1 et v_2 sont voisins dans G'. Si v_1 et v_2 sont différents de xy (le sommet obtenu par fusion de x et y), alors les sous-arbres de v_1 et v_2 s'intersectent dans D. Donc v_1 et v_2 sont voisins dans G et donc dans G'.
Si v_1 est égal à xy alors le sous-arbre de v_2 intersecte soit le sous-arbre de x soit le sous-arbre de y dans G. Ainsi, v_2 est une voisin de x ou de y dans G et est donc un voisin de $xy = v_1$ dans G'. Le cas où v_2 est égal à xy est symétrique.
Supposons maintenant que v_1 et v_2 sont deux sommets voisins dans G'. Montrons que leurs sous-arbres s'intersectent dans D. Si v_1 et v_2 sont différents de xy, alors v_1 et v_2 sont voisins dans G. Leurs sous-arbres s'intersectent donc dans D et, par conséquent, dans D'.
Si v_1 est égal à xy alors v_2 est voisin de x ou de y dans G. Ainsi, le sous-arbre de v_2 intersecte soit celui de x, soit celui de y, dans D. Dans tous les cas, le sous-arbre de v_2 intersecte celui de $xy = v_1$ dans D'. Le cas où v_2 est égal à xy est symétrique.

□

La figure 6.10 illustre l'évolution du graphe et de sa décomposition arborescente pour le cas de la fusion de l'affinité (E, F) dans le graphe G'.

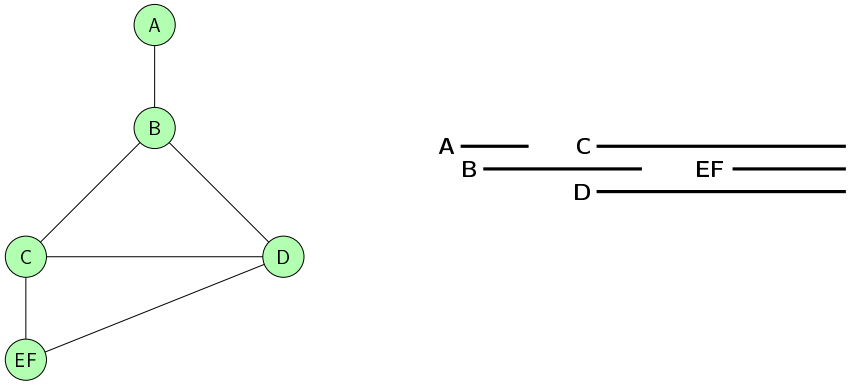


FIG. 6.10 – Le graphe obtenu après fusion de E et F dans G et la décomposition arborescente construite comme décrit durant la preuve du lemme 21.

Le théorème précédent induit deux corollaires importants. Tout d'abord, les graphes parfaitement triangulés sont stables par fusion.

Théorème 22 (Stabilité de la triangulation)**.** *Soit G un graphe parfaitement triangulé, $e = (x, y)$ une affinité de G et G' le graphe obtenu par fusion de x et y.*
G' est parfaitement triangulé.

Démonstration. D'après le théorème précédent, G' admet une décomposition arborescente. Il est donc parfaitement triangulé. □

De plus, il est facile de calculer incrémentalement l'évolution des cliques maximales du graphe.

Définition 50 (Clique dérivée)**.** *Soit G un graphe et G' un graphe obtenu par fusion d'une affinité $e = (x, y)$ de G. Soit C une clique maximale de G. Une clique C' est dite dérivée de C par fusion de e si l'une des propriétés suivantes est vérifiée :*
- *C contient à la fois x et y et $C' = C - \{x, y\} \cup \{xy\}$*
- *C contient x mais pas y et $C' = C - \{x\} \cup \{xy\}$*
- *C contient y mais pas x et $C' = C - \{y\} \cup \{xy\}$*
- *C ne contient ni x ni y et $C' = C$*

où xy désigne le sommet résultant de la fusion de x et y.

Lemme 34 (Cliques dérivées maximales)**.** *Soit G un graphe et G' un graphe obtenu par fusion d'une affinité $e = (x, y)$ de G. Toute clique dérivée d'une clique maximale de G par fusion de e est une clique de G'.*

Démonstration. Conséquence directe du lemme 21 d'évolution des décompositions arborescentes lors d'une fusion. □

Lemme 35 (Cliques maximales)**.** *Soit G un graphe et G' un graphe obtenu par fusion d'une affinité $e = (x, y)$ de G. Toute clique maximale de G' est une clique dérivée d'une clique maximale de G.*

Démonstration. Conséquence directe du lemme 21 d'évolution des décompositions arborescentes lors d'une fusion. □

Ainsi, la largeur d'arbre ne fait que décroître au fur et à mesure des fusions d'affinités, puisque la taille de la plus grande clique décroît. Celle-ci demeure donc inférieure à $(K-1)$ si elle l'est initialement. Nous en arrivons au même point que dans le paragraphe précédent : la fusion agressive est conservative pour les $(K-1)$-arbres partiels.

Lemme 36 (Décroissance de la taille des cliques par dérivation)**.** *Soit G un graphe et G' un graphe obtenu par fusion d'une affinité $e = (x, y)$ de G. Toute clique dérivée d'une clique maximale de G par fusion de e contient moins de sommets que la clique dont elle est dérivée.*

Démonstration. Soit C' une clique dérivée d'une clique maximale C de G. La preuve est triviale par induction sur la construction de la clique dérivée (cf. définition des cliques dérivées). Si C contient x et y, la taille de C' est celle de C moins 1, sinon C et C' ont la même taille. □

Théorème 23 (Décroissance de la largeur d'arbre par fusion)**.** *Soient G un graphe d'interférence-affinité, e une affinité désignée pour la fusion et G' le graphe obtenu après fusion des extrémités de e dans G. $tw(G') \leq tw(G)$.*

Démonstration. Toute clique maximale de G' est dérivée d'une clique de G. La clique dont elle dérive est donc une clique de taille supérieure. Par conséquent, $tw(G') \leq tw(G)$. □

6.5.2 Les graphes SSA

Les résultats de la partie précédente soulignent l'importance de la largeur d'arbre du graphe d'interférence-affinité pour la fusion. Cependant, ce résultat est encore limité à une classe trop restreinte de graphes. Cette partie étend le résultat précédent à une classe plus large de graphes et plus classique, les graphes SSA. Nous en déduisons un nouveau programme linéaire modélisant la fusion conservative dans les graphes SSA.

La partie précédente se focalisait sur les $(K-1)$-arbres partiels. Nous considérons désormais un graphe d'interférence-affinité G K-colorable de largeur d'arbre quelconque dont le graphe d'interférence est triangulé.

L'approche que nous allons suivre consiste intuitivement à construire un ensemble compatible de poids maximal dont le graphe fusionné est K-colorable. Un tel ensemble correspond à une fusion conservative optimale, puisque l'ensemble des affinités satisfaites dans une fusion conservative optimale répond à ce critère. Pour construire cet ensemble, nous allons appliquer la fusion agressive à l'ensemble du graphe et renforcer les contraintes sur les parties du graphe qui font que la largeur d'arbre est supérieure à $(K-1)$ afin de faire chuter la largeur d'arbre du graphe d'interférence-affinité en dessous de $(K-1)$, ce qui permettra d'appliquer les résultats précédents. Si aucune zone de ce type n'existe, la conclusion est immédiate : la fusion agressive peut conduire à une solution optimale. Sinon, ces zones doivent être éliminées.

Le théorème 22 établit que les fusions consécutives d'affinités de G' conservent le caractère parfaitement triangulé du graphe. Faire décroître la largeur d'arbre en dessous de $K-1$ est donc équivalent à faire décroître la taille de chaque clique maximale en dessous de K. Or, nous savons parfaitement déterminer comment évoluent les cliques maximales au cours de la fusion.

Lemme 37 (Cliques de taille supérieure à K)**.** *Soient G un graphe parfaitement triangulé et G' le graphe obtenu par fusion d'une affinité $e = (x, y)$ de G. Toute clique maximale de G' de taille supérieure à K est une clique dérivée d'une clique maximale de G de taille supérieure à K.*

Démonstration. Immédiat par les théorèmes 34 et 36. □

Il en résulte une condition suffisante de K-colorabilité du graphe fusionné d'un ensemble compatible d'affinités.

Théorème 24 (Condition suffisante de colorabilité)**.** *Soient G un graphe parfaitement triangulé, A un ensemble d'affinités compatible de G et G' le graphe fusionné de A. Si pour toute clique maximale C de G, le graphe fusionné de C est une clique de taille inférieure à K alors G' est K-colorable.*

Démonstration. Soit A un ensemble d'affinités compatible dont la fusion fait décroître la taille de chacune des cliques maximales en dessous de K. Il nous suffit de montrer que la

fusion des affinités de A mène à un $(K-1)$-arbre partiel. Le théorème 23 permettra alors de conclure.

Toute clique maximale C de G' est obtenue par une suite de dérivations d'une clique de G de taille supérieure. En conséquence, si G' contient une clique de taille supérieure à K alors l'hypothèse est contredite. G' est un graphe parfaitement triangulé ne contenant pas de clique de taille supérieure à K ; c'est un $(K-1)$-arbre partiel. □

Nous disposons donc d'une condition suffisante pour que toutes les affinités d'un ensemble compatible puissent être satisfaites tout en préservant la K-colorabilité de G'. Cette condition est de fait suffisante pour assurer la K-colorabilité de G puisque toute coloration de G' est une coloration de même valeur de G par construction de G'.

6.6 Résolution optimale dans les graphes SSA

6.6.1 Calcul d'un ensemble compatible optimal par programmation linéaire

Le résultat précédent ouvre la voie au calcul de solutions heuristiques. Par contre, une résolution optimale par programmation linéaire requiert une condition nécessaire et suffisante. Nous spécifions donc dans cette partie une condition nécessaire et suffisante de préservation de la K-colorabilité, puis proposons une formulation par programmation linéaire s'appuyant lourdement sur le lien entre le problème de fusion agressive et celui de multicoupe de poids minimal.

Théorème 25 (Condition nécessaire de colorabilité)**.** *Soit G un graphe parfaitement triangulé dont chaque clique maximale de taille supérieure à K contient une K-clique d'interférences et A un ensemble d'affinités compatible maximal de G. Si A a un graphe fusionné K-colorable alors la fusion de A fait décroître la taille de chacune des cliques maximales de G en dessous de K.*

Démonstration. Chacune des cliques maximales de G est soit de taille inférieure à K, soit réduite à une K-clique d'interférences puisque le graphe fusionné est K-colorable et ne contient que des affinités. □

D'où une condition nécessaire et suffisante de K-colorabilité d'un graphe fusionné d'un graphe parfaitement triangulé.

Théorème 26 (Condition nécessaire et suffisante de K-colorabilité)**.** *Soit A un ensemble d'affinités compatible maximal de G'. Le graphe fusionné de A est K-colorable si et seulement si la fusion de A fait décroître la taille de chaque clique maximale de G' en dessous de K.*

Démonstration. Immédiat par application de la condition nécessaire et de la condition suffisante précédemment décrites, qui sont toutes deux applicables pour G'. □

Le graphe d'interférence-affinité sur lequel on veut résoudre la fusion ne peut pas être passé en paramètre du programme linéaire que nous allons définir puisqu'il nous faut se placer dans le cadre de la condition nécessaire et suffisante de préservation de la K-colorabilité. Nous devons donc construire un second graphe G' qui respecte les conditions d'application de la condition nécessaire et suffisante et tel que résoudre la fusion dans G' équivaut à le résoudre dans G.

Dans un premier temps, nous triangulons G en n'ajoutant que des affinités de poids nul. Cette première transformation permet d'aboutir à un graphe parfaitement triangulé G''. Nous calculons alors les cliques maximales de G'' et les cliques maximales du graphe d'interférence de G. Soit alors C_i une clique maximale de taille α strictement supérieure à K de G'' et $I_i = \{x_1, \ldots, x_p\}$ une clique d'interférences maximale incluse dans C_i. I_i peut être calculée en conservant la plus grande intersection de C_i avec l'une des cliques maximales du graphe d'interférence. Nous complétons ensuite I_i par un ensemble de nouveaux sommets $I'_i = \{x_{p+1}, \ldots, x_K\}$ si $p < K$. Ainsi, I_i est finalement une clique d'interférences de taille K. Les sommets de I'_i sont également liés par des affinités de poids nul à tous les sommets de $C_i - I_i$ et ajoutés à C_i. Ainsi, C_i est une clique. Remarquons que les nouveaux sommets ne sont à chaque fois voisins que des sommets d'une clique maximale de G''. Ainsi, les cliques maximales autres que C_i restent maximales et aucune nouvelle clique n'est maximale. Seule C_i est modifiée. L'opération est répétée pour chaque clique maximale de G'' de taille supérieure à K, menant ainsi au graphe G'. Les cliques maximales de G' sont celles de G'' après modification des C_i.

Théorème 27 (Equivalence de la fusion dans G et G')**.** *Si Col est une K-coloration de G' de valeur v alors la restriction de Col aux sommets de G est une K-coloration de G de valeur v. Inversement, toute K-coloration de G de valeur v peut être étendue en une coloration de G' de valeur de v.*

Démonstration. Immédiat par construction de G'. □

Il ne reste plus qu'à définir un programme linéaire capable de calculer un ensemble d'affinités compatible faisant décroître la taille de chaque clique maximale de G' en dessous de K. Pour ce faire, nous contraignons les sommets de $C_i - I_i$ (où C_i est une clique maximale de taille strictement supérieure à K dans G' et I_i la K-clique d'interférences incluse dans C) à porter la même couleur que l'un des sommets de I_i, ce qui, intuitivement, implique que l'ensemble compatible calculé doit réduire la taille de C_i en dessous de K. Les contraintes sont finalement les suivantes.

$$\forall i, \forall v \in C_i - I_i, \sum_{j \in I_i - N(v)} y_{ij} = |I_i - N(v, G)| - 1$$

Ces contraintes doivent être interprétées comme suit. Soit C_i une clique maximale de G' de taille strictement supérieure à K et v un sommet de $C_i - I_i$. Parmi les $|I_i - N(v, G)|$ affinités liant v à un sommet de I_i, toutes sauf une doivent ne pas être satisfaites.

L'ajout de ces contraintes au programme (PAG) conduit à un nouveau programme linéaire (P), décrit figure 6.11, permettant de calculer un ensemble d'affinités compatible de poids maximal dont le graphe fusionné est K-colorable.

$$(P)\begin{cases} Min & \sum_{e\in A} w_e y_e & & \\ s.c. & & & \\ \forall (i,j)\in I, \forall p\in P(i,j), & \sum_{e\in p} y_e & \geq & 1 \\ \forall a, \forall i \in C_a - I_a, & \sum_{j\in I_a - N(i)} y_{ij} & = & |I_a - N(i)| - 1 \\ \forall e\in A, & y_e \in \{0,1\} & & \end{cases}$$

FIG. 6.11 – Programme linéaire modélisant la fusion.

Nous voulons maintenant prouver que ce programme est correct.

Théorème 28 (Correction du programme (P))**.** *Soit S une solution de (P) et $E = \{e \in A \backslash y_e = 0\}$. S est un ensemble d'affinités compatible dont le graphe fusionné est K-colorable.*

Démonstration. Le théorème 26 nous permet de ne prouver que le fait que E est un ensemble d'affinités compatible dont la fusion fait décroître la taille de chaque clique maximale en dessous de K.
La compatibilité de l'ensemble est assurée par les inégalités de multicoupe, puisque c'est là leur essence même.
Les dernières inégalités assurent que pour chaque clique C_i de taille supérieure à K, tous les sommets sont fusionnés avec au moins l'un des sommets de $C_i - I_i$. Chaque clique de taille strictement supérieure à K est ainsi réduite à une clique de taille exactement K. □

Le programme que nous proposons ne permet pas de déterminer n'importe quel ensemble d'affinités compatible dont le graphe fusionné est K-colorable. Autrement dit, ce programme n'est pas complet. Il est facile de s'en rendre compte en remarquant que ne fusionner aucune affinité préserve la K-colorabilité, mais que les contraintes ne sont pas satisfaites.

Cependant, l'ensemble des solutions admissibles est exactement l'ensemble des ensembles d'affinités compatibles maximaux. C'est donc un sur-ensemble de tous les ensembles d'affinités compatibles optimaux dont le graphe fusionné est K-colorable.

Théorème 29 (Complétude du programme (P))**.** *Soient E un ensemble d'affinités compatibles maximal de G' faisant décroître la taille de chacune des cliques maximales de G' en dessous de K et Y le vecteur tel que y_e vaut 0 si et seulement si e appartient à E.*
Y est une solution de (P).

Démonstration. Nous devons prouver que le vecteur Y vérifie les deux séries de contraintes. E est un ensemble d'affinités compatible donc les inégalités de multicoupe sont satisfaites. Soit C_i une clique maximale de taille supérieure à K. Le graphe fusionné de E est K-colorable et E est maximal, donc chaque sommet de $C_i - I_i$ peut (de par la K-colorabilité) et doit (de par la maximalité) satisfaire une affinité avec l'un des sommets de I_i. La seconde série d'inégalités est donc satisfaite. □

6.6.2 Coloration du graphe fusionné

Le graphe fusionné d'un ensemble compatible est triangulé si le graphe d'interférence-affinité est parfaitement triangulé, comme le prouve le théorème 22. En conséquence, si le graphe d'interférence-affinité est initialement parfaitement triangulé, la phase de coloration peut être résolue de façon optimale en temps polynomial, puisqu'il suffit de colorer un graphe triangulé ne contenant que des interférences. Un algorithme de coloration gourmande suffit à réaliser cette tâche.

6.6.3 Exemples

6.6.3.1 Un premier exemple

Cette partie applique la méthode précédemment décrite à un petit graphe afin de l'illustrer. Nous prenons comme graphe d'interférence-affinité initial le graphe G défini figure 6.12, dont le graphe d'interférence est effectivement triangulé.

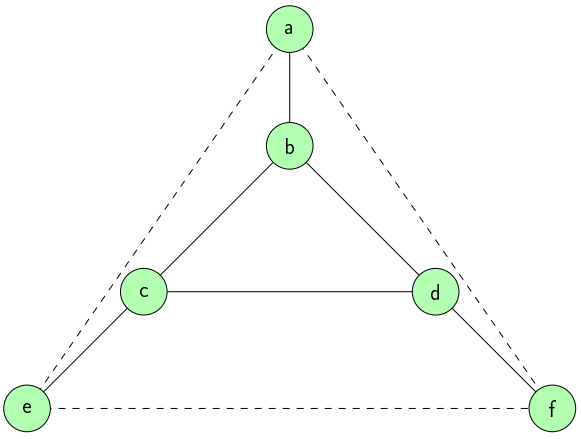


FIG. 6.12 – Le graphe d'interférence-affinité initial G dont le graphe d'interférence est triangulé.

La triangulation G'' n'ajoutant que des affinités est, par exemple, celle définie figure 6.13. G'' est bien une triangulation de G, preuve en est sa décomposition arborescente jointe.

Le parcours de l'arborescence de la figure 6.13 nous permet de déterminer les ensembles de cliques maximales de G'' et du graphe d'interférence de G'', lesquels sont respectivement {{A,B,E,F},{B,C,D,E,F}} et {{A,B},{C,E},{D,F},{B,C,D}}.

A partir de cette étape, le nombre de couleurs disponibles entre en compte. Si l'on considère que K est fixé à 3, alors le graphe G'' possède deux cliques maximales $C_1 = \{A, B, E, F\}$ et $C_2 = \{B, C, D, E, F\}$ de taille strictement supérieure à K. Les cliques

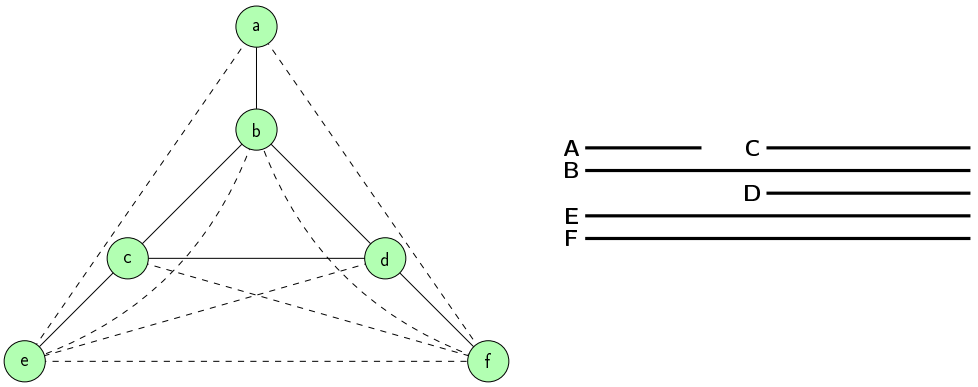


FIG. 6.13 – Une triangulation G'' et une décomposition arborescente du graphe G''.

d'interférence maximales incluses respectivement dans C_1 et C_2 sont $I_1 = \{A, B\}$ et $I_2 = \{B, C, D\}$. Nous devons donc ajouter un sommet S au graphe pour faire grandir la taille de I_1 jusqu'à 3. La figure 6.14 présente finalement le graphe G' qui sert de paramètre au programme linéaire.

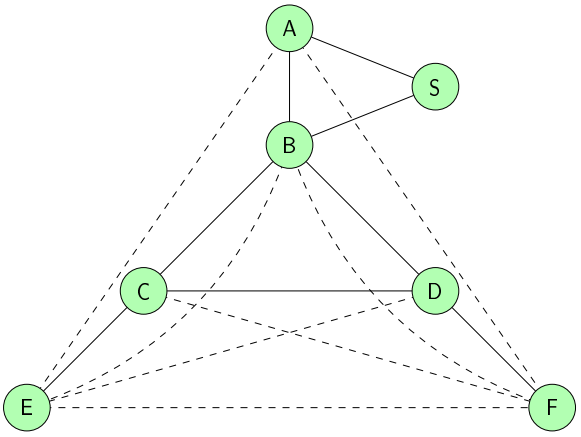


FIG. 6.14 – Le graphe G' passé en paramètre du programme linéaire.

Le programme linéaire (P) instancié avec G' est quant à lui présenté figure 6.18. Ce programme contient :

- 9 variables, chacune vouée à déterminer si une affinité est satisfaite ou non ;
- 20 contraintes de multicoupe permettant d'assurer la compatibilité de l'ensemble des affinités satisfaites ;

– 4 contraintes permettant d'assurer la 3-colorabilité du graphe.

L'algorithme de coupe de Grund et Hack [GH07] possèderait, pour modéliser la fusion de G un programme contenant :

– 3 variables vouées à déterminer si une affinité est satisfaite ou non ;
– 18 variables indiquant si un sommet est coloré d'une couleur ou non ;
– 6 contraintes imposant que chaque sommet est coloré avec une seule couleur ;
– 9 contraintes imposant que les extrémités de chaque interférence sont de couleurs différentes ;
– 18 contraintes liant les variables portant sur les affinités et les variables portant sur les couleurs des sommets ;
– 20 coupes permettant d'accélérer la résolution.

Au total, notre programme linéaire contient donc 9 variables et 24 contraintes contre 21 variables, 33 contraintes et 20 coupes pour le programme linéaire de Grund et Hack.

Ne reste plus qu'à résoudre ce programme linéaire. Dans les faits, il est impossible de satisfaire simultanément deux des affinités de poids non nul. En conséquence, n'importe quelle solution admissible qui satisfait l'une des affinités de la fonction objectif est optimale. C'est le cas de la solution suivante :

$$y_{AE} = 0,\ y_{AF} = 1,\ y_{BE} = 1,\ y_{BF} = 0,\ y_{CF} = 1,\ y_{DE} = 0,\ y_{EF} = 1,\ y_{ES} = 1,\ y_{FS} = 1$$

L'ensemble compatible solution est donc {(A,E),(B,F),(D,E)}. Le graphe fusionné de cet ensemble compatible muni d'une 3-coloration, et la coloration optimale du graphe initial qui en découle sont présentés figure 6.16.

6.6.3.2 Le graphe SSA de référence

Cette partie applique la méthode précédemment décrite au graphe d'interférence-affinité de la forme SSA du programme de référence. Ce graphe, ainsi que les durées de vie de ses variables sont rappelés figure 6.17.

Ce graphe est de fait parfaitement triangulé. Les cliques maximales de G et de G_c sont respectivement $\{\{k,j,g\},\{j,g,h\},\{j,f,e\},\{f,e,m\},\{e,m,b\},\{m,b,c,d\},\{k0,j0,b,d\}\}$ et $\{\{k,j,g\},\{j,g,h\},\{j,f,e\},\{f,e,m\},\{e,m,b\},\{m,b,c\},\{m,b,d\},\{k0,b,d\},\{k0,j0,d\}\}$.

Nous fixons K à 3. Par conséquent, deux cliques maximales sont de taille strictement supérieure à K : $C_1 = \{m,b,c,d\}$ et $C_2 = \{k0,j0,b,d\}$. Les cliques d'interférence maximales incluses dans ces deux cliques sont respectivement $I_1 = \{m,b,c\}$ et $I_2 = \{k0,b,d\}$. Il n'est donc cette fois pas nécessaire d'ajouter de nouveau sommet au graphe. Le graphe paramètre est cette fois G.

Le programme linéaire (P) instancié avec G est présenté figure 6.18. Le programme ne contient aucune inégalité de multicoupe et est donc réduit aux inégalités engendrées par les cliques maximales, soit seulement deux contraintes. Le programme linéaire de Grund et Hack possède, pour le même problème, 38 variables et 75 contraintes.

L'ensemble compatible solution est donc {(c,d),(j0,b)} : les deux affinités du graphe peuvent donc toutes deux être satisfaites. Il suffit ensuite d'appliquer l'algorithme de coloration gourmande pour obtenir une 3-coloration du graphe fusionné et en déduire ainsi une 3-coloration du graphe initial.

$$
(Pex)\ \begin{cases}
Min & y_{AE}+y_{AF}+y_{EF} & \\
s.c. & & \\
(*\ chemins\ AS\ *) & & \\
y_{AF}+y_{FS} & \geq & 1 \\
y_{AE}+y_{ES} & \geq & 1 \\
y_{AE}+y_{EF}+y_{FS} & \geq & 1 \\
(*\ chemins\ BS\ *) & & \\
y_{BF}+y_{FS} & \geq & 1 \\
y_{BE}+y_{ES} & \geq & 1 \\
y_{BE}+y_{EF}+y_{FS} & \geq & 1 \\
(*\ chemins\ AB\ *) & & \\
y_{AE}+y_{BE} & \geq & 1 \\
y_{AE}+y_{EF}+y_{BF} & \geq & 1 \\
y_{AF}+y_{BF} & \geq & 1 \\
y_{AF}+y_{EF}+y_{BE} & \geq & 1 \\
(*\ chemins\ BC\ *) & & \\
y_{BF}+y_{CF} & \geq & 1 \\
y_{BE}+y_{EF}+y_{CF} & \geq & 1 \\
y_{BE}+y_{AE}+y_{AF}+y_{CF} & \geq & 1 \\
(*\ chemins\ BD\ *) & & \\
y_{BE}+y_{DE} & \geq & 1 \\
y_{BF}+y_{EF}+y_{DE} & \geq & 1 \\
y_{BF}+y_{AF}+y_{AE}+y_{DE} & \geq & 1 \\
(*\ chemins\ CD\ *) & & \\
y_{CF}+y_{EF}+y_{DE} & \geq & 1 \\
y_{CF}+y_{AF}+y_{AE}+y_{DE} & \geq & 1 \\
(*\ chemins\ CE\ *) & & \\
y_{CF}+y_{EF} & \geq & 1 \\
(*\ chemins\ DF\ *) & & \\
y_{DE}+y_{EF} & \geq & 1 \\
(*\ clique\ C_1\ *) & & \\
y_{AE}+y_{BE}+y_{ES} & = & 2 \\
y_{AF}+y_{BF}+y_{ES} & = & 2 \\
(*\ clique\ C_2\ *) & & \\
y_{BE}+y_{DE} & = & 1 \\
y_{BF}+y_{CF} & = & 1 \\
\forall e \in A, & y_e \in \{0,1\} &
\end{cases}
$$

FIG. 6.15 – Programme linéaire de fusion instancié pour le graphe G'.

6.7 Triangulation des graphes SSA

Les travaux présentés dans la partie précédente s'appuient fortement sur le calcul d'une triangulation du graphe d'interférence-affinité. Cette partie décrit brièvement une heuristique de triangulation des graphes SSA.

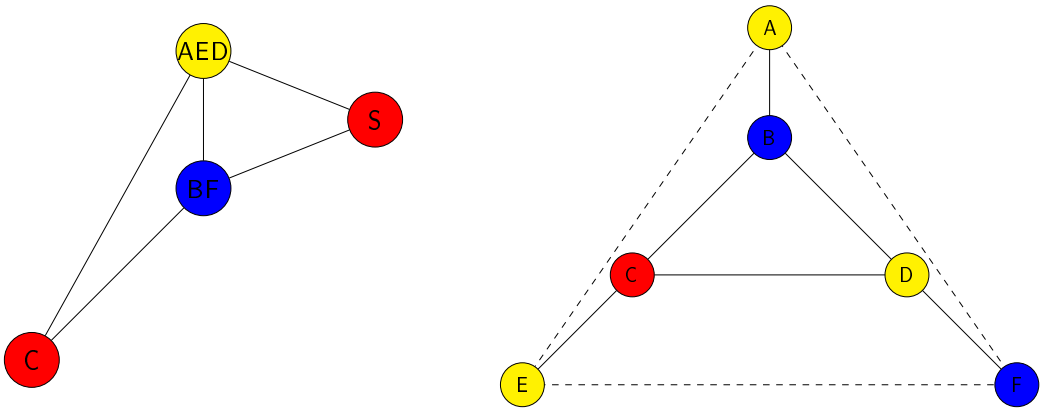


FIG. 6.16 – Le graphe fusionné de l'ensemble compatible optimal obtenu par programmation linéaire. Ce graphe est triangulé et peut, à ce titre, être facilement coloré en temps polynomial.

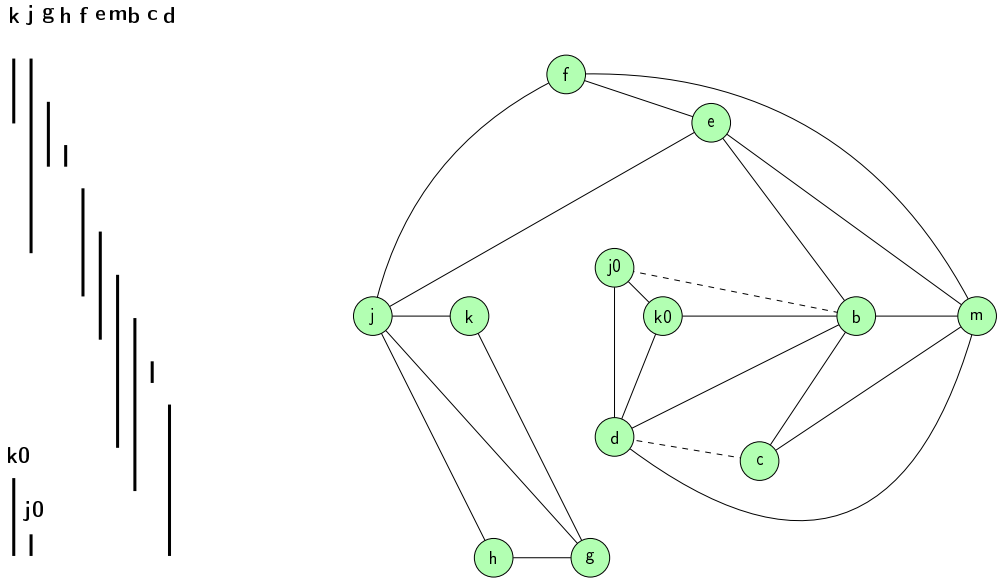


FIG. 6.17 – Le graphe d'interférence-affinité G du programme de référence sous forme SSA et les durées de vie de ses variables.

Si l'on revient aux raisons de la triangulation des graphes SSA, l'on sait que le support de la décomposition du graphe d'interférence est l'arbre de dominance du programme. L'idée de triangulation est simple : toute paire de sommets liés par une affinité doivent voir leurs sous-arbres s'intersecter dans la décomposition arborescente ayant pour support l'arbre de dominance. Nous proposons d'étendre le sous-arbre de chacun des deux sommets

$$(Pref)\begin{cases} Min & y_{cd} + y_{j0b} \\ s.c. & \\ (* \ clique \ C_1 \ *) & \\ y_{cd} & = \quad 0 \\ (* \ clique \ C_2 \ *) & \\ y_{j0b} & = \quad 0 \\ \forall e \in A, & y_e \in \{0,1\} \end{cases}$$

FIG. 6.18 – Programme linéaire de fusion conservative instancié pour le graphe de référence. Les contraintes impliquent directement la solution optimale.

jusqu'à leur plus petit ancêtre commun dans l'arbre de dominance.

Cette méthode a l'intérêt de permettre également de gérer les sommets précolorés. Deux sommets de la même couleur peuvent être liés par une affinité de poids «infini», auquel cas de nombreuses affinités de poids nul doivent être ajoutées à la triangulation en forçant les sous-arbres de ces deux sommets à s'intersecter. Deux sommets de couleurs différentes peuvent de la même façon être liés par une interférence. L'implantation de l'algorithme, qui n'a pas encore été réalisée, permettra de se rendre compte de l'impact de ces sommets précolorés.

L'algorithme de triangulation n'a théoriquement pas besoin d'être optimal, ni même efficace. C'est pourquoi cet algorithme simpliste de triangulation suffit amplement. Les tests expérimentaux permettront de juger de l'importance de la qualité de la triangulation et de ses conséquences sur le programme linéaire.

6.8 Perspectives

6.8.1 Implantation et extensions

Cet algorithme n'a pas encore été implanté. La perspective la plus proche est donc son implantation et son évaluation. Ce travail a été motivé par l'équipe Compsys du LIP (Lyon), avec qui nous avons eu des échanges durant ma thèse, et pour qui améliorer la vitesse de résolution de la fusion dans les graphes SSA s'avère être un enjeu important. Maintenant que l'algorithme est défini, nous souhaitons leur présenter cette approche puis l'intégrer à un compilateur SSA, tels que ceux de STMicroelectronics.

Nous comptons en outre approfondir cette approche de fusion guidée par la largeur d'arbre du graphe d'interférence-affinité et mêlant fusion conservative et fusion agressive pour définir par exemple de nouvelles heuristiques.

6.8.2 Vérification formelle

L'algorithme décrit dans ce chapitre n'a pas été implanté et n'a par conséquent pas été non plus soumis à la vérification formelle. Nous pensons la validation *a posteriori* appropriée à ce type d'algorithme, entre autres en raison des données complexes à manipuler, comme les décompositions arborescentes, et de l'utilisation de la programmation linéaire.

6.8.2.1 Les graphes SSA

Les graphes SSA ont l'importante propriété d'avoir des graphes d'interférence triangulés. Cette propriété offre toute une palette d'outils permettant de définir de nouvelles approches de résolution de l'allocation de registres par coloration de graphe. Cependant, ces outils que sont par exemple les ordres d'élimination simpliciale ou les décompositions arborescentes sont cependant difficiles à manipuler. Les décrire formellement et raisonner dessus semble complexe.

De surcroît, la mise sous forme SSA d'un programme permet d'obtenir des propriétés qu'il serait difficile de formaliser et prouver et, plus encore, d'exploiter. Disposer de l'arbre de dominance, prouver ses liens avec les durées de vie des variables et montrer que la graphe d'interférence d'un graphe SSA est triangulé semble loin d'être aisé. Ce sont pourtant là des perspectives importantes en terme de vérification formelle de l'allocation de registres.

Cependant, devant les difficultés émanant de l'usage de ces méthodes pointues, la validation *a posteriori* apparaît comme une approche saine et appropriée pour la vérification d'algorithmes dédiés aux graphes SSA.

6.8.2.2 La programmation linéaire

Vérifier formellement la programmation linéaire permettrait d'implanter des algorithmes vérifiés de résolution optimale non seulement de l'allocation de registres, mais également de tous les problèmes combinatoires qu'il est intéressant de résoudre grâce à cette méthode.

Implanter un algorithme de résolution de la programmation linéaire dans le système Coq est hors de portée : les solveurs usuels sont des logiciels optimisés reposant sur des algorithmes complexes qu'il serait impossible de concurrencer avec un algorithme écrit dans le système Coq . Il est donc naturel de faire appel aux solveurs existants et, de fait, à la validation *a posteriori*.

Cependant, la résolution du programme linéaire n'est pas le seul passage qui doit être vérifié formellement. Un programme linéaire est une structure de données compliquée censée représenter un problème concret. Prouver de façon interne à Coq la correspondance entre un modèle et le problème qu'il représente est déjà un problème difficile et important pour les utilisateurs de programmation linéaire. Si l'on sait écrire une fonction qui transforme toute solution du programme linéaire en une solution du problème initial et prouver que cette fonction est correcte alors valider *a posteriori* la solution du programme linéaire revient à valider *a posteriori* la solution du problème concret. En outre, cette méthode, dont l'architecture générale est décrite figure 6.19, permettrait de ne limiter les possibilités d'erreur qu'à la résolution du programme linéaire et non à sa définition. Néanmoins, ce ne sont là que des remarques préliminaires à cette tâche d'ampleur. Une étude dédiée à ce sujet est sans nul doute à mener.

6.9 Bilan

La fusion est fréquemment traitée en même temps que l'affectation aux registres, ce qui conduit à un problème très délicat. En effet, deux sources de complexité importantes

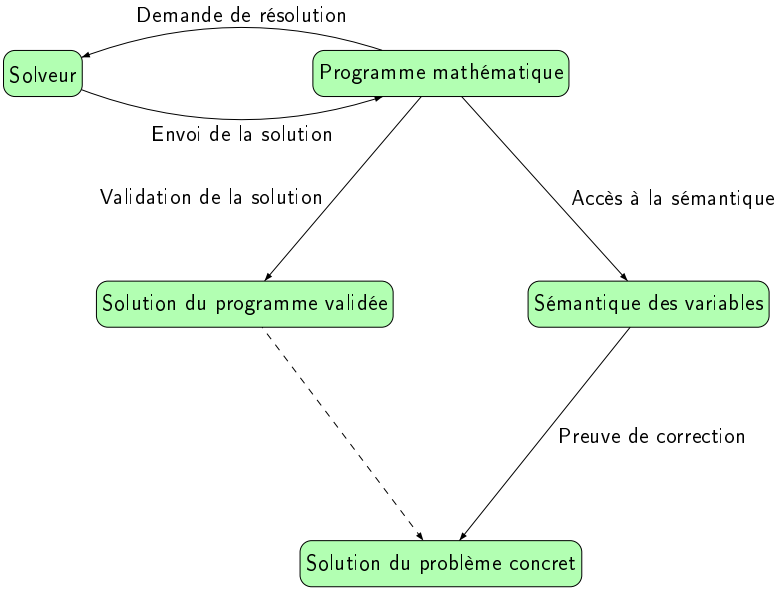


FIG. 6.19 – Schéma de vérification formelle de la programmation linéaire. Le programme linéaire est défini de façon interne à Coq. La résolution du programme est confiée à un solveur externe dont la solution est validée *a posteriori*. Par ailleurs, une fonction de concrétisation de la solution du programme linéaire permet de transformer la solution du programme linéaire en une solution du problème initial. Cette fonction doit être prouvée correcte grâce à la correspondance entre les variables du programme linéaire et les éléments du problème concret.

interviennent alors, la coloration et les affinités. Ce constat nous a naturellement mené à étudier la possibilité de séparer ces deux phases afin de se concentrer sur les sources de complexité l'une après l'autre, sans néanmoins détériorer la qualité des solutions calculables. Sous cette contrainte, les deux phases peuvent être considérées indépendamment sans perte qualitative théorique si et seulement si la fusion agressive est conservative.

Nous nous sommes donc focalisés sur la frontière séparant ces deux stratégies de fusion. Il en résulte que la fusion locale, affinité par affinité, est un frein que nous devons surpasser pour définir de nouvelles heuristiques de fusion plus efficaces. Des propriétés plus globales du graphe, incorporant les propriétés de ses affinités, doivent être exploitées. Cette démarche nous a permis de mettre en exergue une classe de graphes dans lesquels fusions agressive et conservative sont équivalentes, les $(K-1)$-arbres partiels.

Nous avons ensuite pu généraliser ce résultat et définir une nouvelle approche de résolution optimale de la fusion conservative pour les graphes SSA. Cette formulation allie de fait fusions agressive et conservative. Après triangulation du graphe, une fusion agressive est modélisée sur le graphe entier et des contraintes supplémentaires sont ajoutées afin

d'assurer que la largeur d'arbre du graphe finira par chuter en dessous de K. Les résultats obtenus dans les $(K-1)$-arbres partiels permettent ensuite de conclure que le graphe résultant de la fusion est K-colorable et triangulé. L'affectation aux registres, *i.e.* la coloration de ce graphe, peut donc ensuite être traitée de façon optimale en temps polynomial.

La vérification formelle d'un tel algorithme apparaît délicate. En effet, manipuler les décompositions arborescentes ainsi que l'arbre de dominance du programme ou tout simplement mettre sous forme SSA le programme sont des opérations complexes qui doivent faire l'objet d'une étude postérieure. La validation *a posteriori* semble être, à ce jour, la seule solution pérenne pour vérifier un tel algorithme.

Il ressort de cette étude que n'opérer que de la fusion conservative ou que de la fusion agressive, comme c'est le cas de tous les algorithmes d'allocation de registres actuels, n'est pas une solution pérenne. Les deux approches doivent être combinées afin de dépasser les limites aujourd'hui atteintes.

Chapitre 7

Fusion dans les graphes éclatés par recomposition des durées de vie

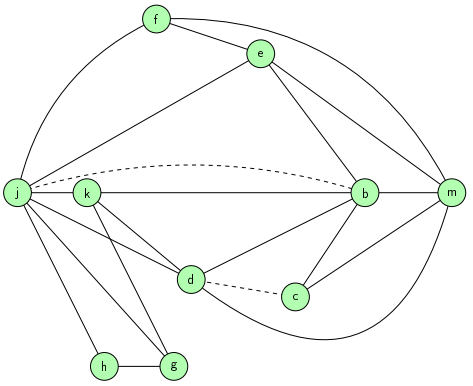
La forme SSA permet au graphe d'interférence-affinité de sur-approximer moins grossièrement les contraintes réelles de l'allocation de registres. Cependant, il faut aller plus loin et considérer l'emplacement de chaque variable à chaque point de programme pour que cette sur-approximation devienne minime. Considérer l'emplacement de chaque variable à chaque point de programme est équivalent à réaliser du découpage extrême, *i.e.* entre chaque instruction. Une démarche si agressive de découpage introduit inévitablement des points de découpage inutiles, au sens où aucune variable ne changera d'emplacement à ce point.

Le but du travail présenté dans ce chapitre est de reconnaître de tels points de découpage dans les graphes éclatés, quantifier leur fréquence, les supprimer et résoudre de façon optimale la fusion. Nous présentons pour cela une condition suffisante de suppression d'un point de découpage conservant l'optimalité et un algorithme de reconnaissance de tels points. Cet algorithme réduit de 80% en moyenne la taille des graphes de l'OCC. La résolution est ensuite effectuée grâce à l'algorithme de coupes de Grund et Hack qui avait déjà permis de résoudre 471 des 474 instances de l'OCC. Notre réduction permet de résoudre en moyenne 300 fois plus rapidement les 471 instances déjà résolues et produit les premiers résultats optimaux des trois dernières instances. Elle donne en outre des indications sur les points de découpage inutiles en vue de définir de nouvelles stratégies de découpage [BR09].

7.1 Recomposition des durées de vie

Le découpage extrême des durées de vie tire sa force de deux facteurs : sa définition facilement implantable et la précision atomique de la gestion des variables. Ce degré de précision implique que le graphe éclaté d'un programme est très proche du flot de données du programme, comme l'illustre l'exemple de la figure 7.1.

Entrée : k j
(p_0) g := mem[j+12]
(p_1) h := k - 1
(p_2) f := g * h
(p_3) e := mem[j+8]
(p_4) m := mem[j+16]
(p_5) b := mem[f]
(p_6) c := e + 8
(p_7) d := c
(p_8) k := m + 4
(p_9) j := b
(p_{10})
Sortie : d j k



Entrée : k j
(p_0) k0 := k ‖ j0 := j ‖ g := mem[j+12]
(p_1) j1 := j0 ‖ g0 := g ‖ h := k0 - 1
(p_2) j2 := j1 ‖ f := g0 * h
(p_3) f0 := f ‖ j3 := j2 ‖ e := mem[j2+8]
(p_4) e0 := e ‖ f1 := f0 ‖ m :=mem[j3+16]
(p_5) e1 := e0 ‖ m0 := m ‖ b :=mem[f1]
(p_6) b0 := b ‖ m1 := m0 ‖ c := e1 + 8
(p_7) b1 := b0 ‖ m2 := m1 ‖ d := c
(p_8) b2 := b1 ‖ d0 := d ‖ k1 := m2 + 4
(p_9) d1 := d0 ‖ k2 := k1 ‖ j4 := b2
(p_{10})
Sortie : d1 j4 k2

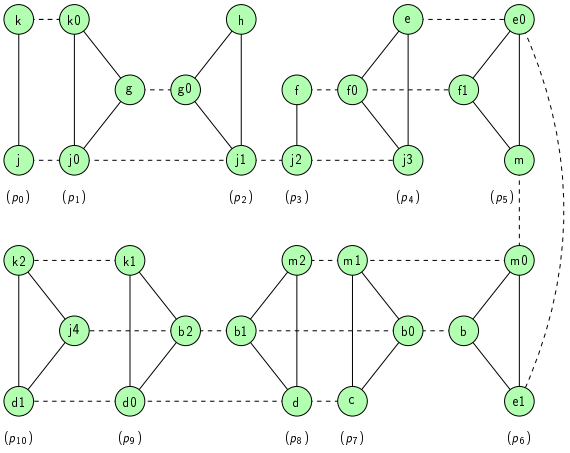


FIG. 7.1 – Le programme de référence, son graphe d'interférence-affinité et son graphe éclaté. Le graphe éclaté donne une bonne représentation du flot de données du programme.

7.1.1 Cliques d'instruction

Outre ces intérêts conceptuels, les graphes éclatés disposent d'une structure propice à l'étude des points de programme : chaque clique d'interférences correspond à une instruction du programme.

Théorème 30 (Cliques d'instruction et points de programme)**.** *Soit P un programme. Chaque instruction de P correspond à une composante connexe d'interférence de P. De plus, une telle composante est toujours une clique, que nous appelons clique d'instruction.*

Démonstration. Soit P un programme, i une instruction de P et p_1 et p_2 les points de programme précédant et suivant i. Deux cas doivent être distingués, comme pour la construction des graphes éclatés.
Supposons dans un premier temps qu'aucune nouvelle variable n'est définie et que i est de la forme $v_0^1 = v_0^1 \; op \; v_1^1$. Par définition des graphes éclatés, i est remplacée par la série d'instruction suivante :

$$\bullet p_1$$

$$v_0^2 \leftarrow v_0^1 \parallel v_1^2 \leftarrow v_1^1 \parallel \ldots \parallel v_n^2 \leftarrow v_n^1$$

$$v_0^2 := v_0^2 \; op \; v_1^2$$

$$\bullet p_2$$

Dans ce cas, les variables v_j^1 sont toutes copiées en parallèle et les variables v_j^2 interfèrent toutes ensemble. Les interférences engendrées par l'instruction i forment donc une clique dont les sommets sont les représentants des v_j^2. L'instruction $v_0^2 := v_0^2 \; op \; v_1^2$ n'introduit en effet pas de nouvelle interférence. Cette clique est de plus une composante d'interférences maximale puisque toutes les variables vivantes en p_2 seront ensuite copiées.

Dans le second cas, une nouvelle variable v est définie. Par définition des graphes éclatés, l'instruction i est remplacée comme suit :

$$\bullet p_1$$

$$v_0^2 \leftarrow v_0^1 \parallel v_1^2 \leftarrow v_1^1 \parallel \ldots \parallel v_n^2 \leftarrow v_n^1 \parallel v := v_0^1 \; op \; v_1^1$$

$$\bullet p_2$$

La preuve est similaire à celle du premier cas, les variables v_j^2 forment une clique et sont seules à interférer. □

7.1.2 Couplage asservissant

Les cliques d'instruction sont également liées d'une façon très caractéristique des graphes éclatés, voire plus généralement des copies parallèles, que nous détaillons dans cette partie.

Définition 51 (Couplage d'affinités)**.** *Un couplage d'un graphe est un ensemble d'arêtes n'ayant aucune extrémité en commun. Un couplage est dit maximal s'il n'est strictement inclus dans aucun autre couplage et un couplage est dit maximum s'il est de cardinal maximal.*
Un couplage d'affinités est un couplage ne contenant que des affinités.

Par exemple, l'ensemble $\{(b0, b)(m0, m1)\}$ de la figure 7.1 est un couplage d'affinités. De plus, il constitue un couplage d'affinités maximal dans le graphe induit par les sommets $\{b, b0, m0, m1\}$.

Ce sont ces couplages maximaux qui permettent de détecter certains points de découpage qui peuvent être supprimés. En effet, un tel couplage représente un ensemble de copies parallèles qui peuvent toutes être fusionnées. Cependant, nous ne souhaitons fusionner les affinités d'un couplage que s'il existe une fusion optimale telle que ces affinités sont fusionnées. C'est pourquoi nous introduisons le concept de couplage asservissant de deux cliques, qui modélise le fait qu'une clique d'instruction impose sa coloration à une autre clique d'instruction.

Définition 52 (Cliques asservissantes et asservies, couplage d'affinité asservissant et cliques d'instructions parallèles)**.** *Soient C_1 et C_2 deux cliques d'instructions. C_1 asservit C_2 s'il existe un couplage d'affinités M tel que :*

1. *M ne contient que des affinités dont une extrémité appartient à C_1 et une extrémité appartient à C_2 ;*
2. *chaque sommet de C_2 est saturé par M, i.e. chaque sommet de C_2 est extrémité d'une arête de M ;*
3. *aucune affinité de M n'a deux extrémités précolorées de couleurs différentes ;*
4. *pour tout v de C_1 et tout v' de C_2 précolorés de la même couleur, l'affinité (v, v') appartient à M ;*
5. *pour tout sommet v de C_2, le poids de l'arête de M incidente à v est supérieur ou égal au poids total des autres affinités incidentes à v.*

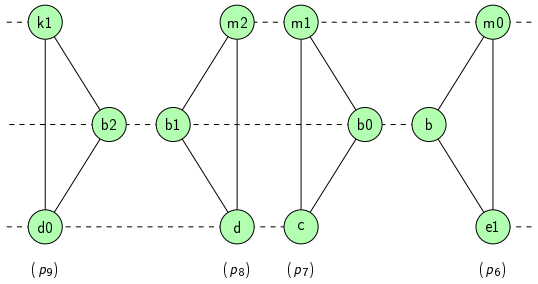
Dans ce cas, on dit également que C_1 et C_2 sont parallèles et que M est un couplage asservissant de C_1 et C_2.

Les quatre premières conditions semblent peu restrictives et correspondent à une configuration classique à un point de découpage. La seule originalité est que C_1 et C_2 sont ici des cliques. En revanche, la dernière condition semble très forte, alors que non. En effet, au sein d'un bloc de base, *i.e.* d'une partie linéaire du code, le poids d'une affinité est souvent la moitié de la somme des poids des affinités de chacune de ses extrémités puisque chaque sommet représentant une variable v est le plus souvent incident à seulement deux affinités : une résultant de la copie qui définit v et une de la copie de v. De plus, au sein d'un bloc de base, les deux copies vont usuellement être réalisées un même nombre de fois, d'où les poids identiques des deux affinités incidentes à un même sommet.

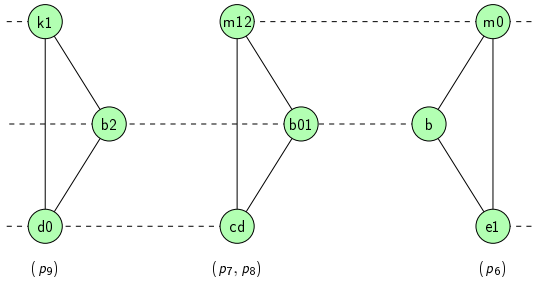
La figure 7.2 illustre un sous-graphe du graphe éclaté présenté figure 7.1. Ce sous-graphe, (en haut de la figure), contient quatre cliques d'instructions. Les cliques induites par les ensembles $\{b0, c, m1\}$ et $\{b1, d, m2\}$ sont respectivement dénotées $\mathcal{C}$ et $\mathcal{C}'$ par la suite. Dans ce graphe, $\mathcal{C}$ asservit $\mathcal{C}'$ (et ici, réciproquement, $\mathcal{C}'$ asservit $\mathcal{C}$). Nous définissons la recomposition des durées de vie (ou recomposition), par antagonisme au découpage des durées de vie, comme la suppression d'un point de découpage. Celle-ci consiste à déterminer un couplage asservissant $\mathcal{M}$ de $\mathcal{C}$ et $\mathcal{C}'$ et à fusionner les affinités de $\mathcal{M}$, supprimant de ce fait le point de découpage séparant l'instruction de $\mathcal{C}$ et l'instruction de $\mathcal{C}'$. Appliquée à l'exemple de la figure 7.2, la recomposition consiste à trouver le couplage asservissant $\mathcal{M} = \{(b0, b1), (c, d), (m1, m2)\}$ et à fusionner toutes ses affinités. Le graphe résultant est donné en bas de la figure. Ce graphe correspond exactement au graphe éclaté du programme après suppression du point de découpage p_8. En effet, dans le code correspondant, les copies de $b0$ et $m1$ ont été retirées et la séquence d'instructions `c :=e1+8 ; d :=c` a été simplifiée en l'instruction `d :=e1+8`.

Afin de pouvoir itérer le principe de recomposition des durées de vie, nous généralisons le concept de clique d'instructions en clique de bloc d'instructions.

Définition 53 (Bloc d'instructions)**.** *Un bloc d'instructions est une suite d'instructions qui ne sont séparées par aucun point de splitting.*



(p_5) e1 := e0 ‖ m0 := m ‖ b :=mem[f1]
(p_6) b0 := b ‖ m1 := m0 ‖ **c := e1 + 8**
(p_7) b1 := b0 ‖ m2 := m1 ‖ **d := c**
(p_8) b2 := b1 ‖ d0 := d ‖ k1 := m2 + 4



(p_5) e1 := e0 ‖ m0 := m ‖ b :=mem[f1]
(p_6, p_7) b1 := b ‖ m2 := m0 ‖ **d := e1 + 8**
(p_8) b2 := b1 ‖ d0 := d ‖ k1 := m2 + 4

FIG. 7.2 – En haut, deux cliques parallèles et le morceau de programme correspondant. En bas, le même sous-graphe et son code correspondant après une étape de recomposition.

Les blocs d'instructions sont initialement réduits à des singletons, puis grossissent au fur et à mesure des suppressions des points de découpage.

Définition 54 (Cliques de bloc d'instructions)**.** *Les cliques de bloc d'instructions d'un graphe éclaté sont les cliques d'interférences maximales de ce graphe.*

Ces cliques de bloc d'instructions correspondent chacune à un bloc d'instructions. Elles sont donc initialement réduites aux cliques d'instructions. L'exemple de la figure 7.2 crée un bloc d'instructions de deux instructions en supprimant le point de découpage p_8. La clique d'instructions correspondant à ce bloc est la clique induite par les sommets $b01$, $m12$ et cd.

La recomposition permet donc de fusionner deux blocs d'instructions et non seulement deux instructions. Mieux encore, celle-ci préserve l'optimalité. Autrement dit, si s est une solution optimale du problème après recomposition, alors s est une solution optimale avant recomposition.

Théorème 31 (Optimalité de la recomposition)**.** *Si C_1 et C_2 sont deux cliques parallèles telles que C_1 asservit C_2 et M un couplage asservissant de C_1 et C_2, alors il existe une fusion optimale telle que toutes les affinités de M sont satisfaites.*

Démonstration. Soit C_1 et C_2 deux cliques parallèles telles que C_1 asservit $C2$, M un couplage asservissant de C_1 et C_2, et Col une coloration optimale du graphe.
Soit Col' la coloration obtenue à partir de Col en affectant aux sommets de C_2 la couleur affectée au sommet de C_1 auxquels ils sont liés par une arête de M. Chaque sommet de C_2 est bien lié à un sommet de C_1 par définition des couplages asservissants. Nous devons

montrer que Col' est une coloration du graphe de valeur supérieure à celle de Col.
D'une part, Col' est bien une coloration du graphe. Seuls les sommets de C_2 n'ont pas la même couleur dans Col et Col'. Il suffit donc de montrer que Col' ne viole aucune interférence de C_2, ce qui découle immédiatement de la définition de Col' et du fait que C_1 est une clique d'interférence-affinités.
D'autre part, la valeur de Col' est effectivement supérieure à celle de Col. Il suffit pour cela de remarquer que les affinités non-incidentes aux sommets de C_2 sont satisfaites dans Col' si et seulement si elles le sont dans Col et que la somme des poids des affinités incidentes à chaque sommet de C_2 est plus grande dans Col' que dans Col, de par la définition des couplages asservissants. □

7.2 Algorithme de recomposition

Implanter le concept de recomposition nécessite deux fonctions principales. Une fonction permettant de calculer un couplage asservissant et, par conséquent, de déterminer deux cliques parallèles, et une fonction itérant les étapes de recomposition.

7.2.1 Détection des cliques parallèles

La détection des cliques parallèles est, bien heureusement, un problème polynomial. Etant données deux cliques d'instructions C_1 et C_2, déterminer si elles sont parallèles peut être fait en complexité $O(K \cdot m_A)$, où m_A désigne le nombre d'affinités incidentes aux sommets de C_2.

Les lignes 1 à 17 de l'algorithme 1 calculent l'ensemble E des affinités du graphe éclaté susceptibles d'appartenir à un couplage asservissant. Initialement, E est l'ensemble des affinités liant un sommet de C_1 à un sommet de C_2. La ligne 2 supprime de E les affinités qui ne respectent pas la précoloration, *i.e.* dont les extrémités sont précolorées avec des couleurs différentes. Les lignes 3 à 7 suppriment de E les affinités incidentes à un sommet précoloré si la couleur de ce sommet est affectée à la fois à un sommet de C_1 et à un sommet de C_2, sauf l'éventuelle affinité liant ces deux sommets. Ensuite, les lignes 8 à 17 suppriment les affinités dont le poids n'est pas assez grand pour qu'elles appartiennent à un couplage asservissant. Plus précisément, une affinité peut être supprimée de E si son poids n'est pas plus grand que la moitié du poids total des affinités incidentes à son extrémité appartenant à C_2. Les affinités ayant les deux extrémités de la même couleur ne peuvent être supprimées de cette façon. Il ne reste alors qu'à examiner si les arêtes de E contiennent un couplage tel que chaque sommet de C_2 possède une arête incidente dans ce couplage. Pour ce faire, nous calculons un couplage maximum inclus dans E. Ceci est possible en temps polynomial car le graphe induit par E est biparti [Wes00]. Si le cardinal de ce couplage est égal au nombre de sommets de C_2, alors le couplage est asservissant, sinon, il n'en existe pas.

Algorithm 1 `couplage_asservissant` (C_1,C_2)

Require: Deux cliques d'instructions C_1 et C_2
Ensure: Un couplage asservissant $\lfloor M \rfloor$ si C_1 asservit C_2, $\emptyset$ sinon

1: E := {affinités ayant une extrémité dans C_1 et une dans C_2}
2: supprimer de E les affinités d'extrémités précolorées de couleurs différentes
3: **for all** couleur c **do**
4: **if** $\exists v_1 \in C_1$ et $\exists v_2 \in C_2$ précolorés avec c **then**
5: supprimer de E toute affinité incidente à v_1 ou v_2 autre que (v_1, v_2)
6: **end if**
7: **end for**
8: **for all** $v \in C_2$ **do**
9: $Aff\_weight(v) := \sum_{x \in N_a(v)} weight_{(v,x)}$
10: **end for**
11: **for all** $v_2 \in C_2$ **do**
12: **for all** v_1 such that $(v_1, v_2) \in E$ **do**
13: **if** $weight_{(v_1,v_2)} < \frac{1}{2} \cdot Aff\_weight(v_2)$ et v_1 et v_2 ne sont pas de la même couleur **then**
14: supprimer (v_1, v_2) de E
15: **end if**
16: **end for**
17: **end for**
18: M := couplage maximum du graphe induit par les arêtes de E
19: **if** cardinal(M) = cardinal$(V(C_2))$ **then**
20: **return** $\lfloor M \rfloor$
21: **else**
22: **return** $\emptyset$
23: **end if**

7.2.2 Algorithme général

Fusionner deux cliques de bloc d'instructions est équivalent à supprimer le point de découpage qui les sépare. Par conséquent, la fusion de deux cliques de bloc d'instructions parallèles mène à une nouvelle clique de bloc d'instructions. Ainsi, les caractéristiques des graphes éclatés sur lesquelles reposent la recomposition sont préservées. La réduction peut donc être itérée tant que la taille du graphe décroît. De plus, cette fusion conduit à un graphe dans lequel de nouvelles cliques parallèles ou des sommets simplifiables (de bas degré, sans affinités et non précolorés) peuvent être apparus. Ces sommets peuvent être simplifiés comme dans les autres algorithmes [CAC+81, GA96], créant ainsi de nouvelles cliques parallèles.

L'algorithme 2 applique la recomposition tant que c'est possible. L'algorithme commence par calculer toutes les cliques d'instructions (ligne 1), ainsi que les cliques d'instructions liées par une affinité (lignes 2 à 4). Puis, la recomposition et la suppression des sommets de bas degré sont itérées tant qu'elles réduisent la taille du graphe (lignes 5 à 23). Afin d'accélérer le processus, nous calculons pour chaque clique de bloc d'instructions

i l'ensemble N_i des cliques de bloc d'instructions liées par (au moins) une affinité à un sommet de i.

Algorithm 2 `recomposition` (G)

Require: Un graphe éclaté G
Ensure: Un graphe éclaté G'

```
1: calculer les cliques d'instructions
2: for all clique d'instructions i do
3:     N_i := {cliques d'instructions liées à i par une affinité}
4: end for
5: red := 1
6: while red ≠ 0 do
7:     red := 0
8:     for all clique de bloc d'instruction i do
9:         for all j ∈ N_i do
10:            M := couplage_asservissant(i, j)
11:            if M ≠ NULL then
12:                fusionner chaque affinité de M et mettre à jour les poids
13:                red := red+1
14:                N_ij := N_i ∪ N_j − {i, j}
15:                for all k ∈ N_i ∪ N_j do
16:                    N_k := N_k ∪ {ij} − {i, j}
17:                end for
18:            end if
19:        end for
20:    end for
21:    red := red + |{sommets trivialement colorables}|
22:    simplifier les sommets trivialement colorables
23: end while
```

7.2.3 Exemple d'application

La figure 7.3 présente la succession des cliques parallèles rencontrées lors de l'application de la recomposition au graphe éclaté de la figure 7.1, pour un nombre de registres fixé à 3. Chaque détection et fusion de deux cliques parallèles est suivie d'une phase de simplification des sommets de bas degré. Cette phase de simplification n'est illustrée que pour sa première exécution, afin d'alléger quelque peu la figure.

Au commencement de l'algorithme, quatre paires de cliques parallèles sont détectées (graphe (a)) et fusionnées. En tout, dix affinités sont fusionnées : $(k, k0)$, $(j, j0)$, $(f, f0)$, $(j2, j3)$, $(m1, m2)$, $(b0, b1)$, (c, d), $(k1, k3)$, $(b2, b5)$ et $(d0, d1)$. Le graphe (b) est le graphe résultant de ces fusions. Toujours dans un souci d'allègement de la figure, un sommet résultant d'une fusion n'y est identifié que par l'un des deux sommets dont il est le fruit. La phase de simplification a alors lieu, supprimant les sommets h, $k0$ et $k1$. Nous arrivons alors au graphe (c). Une nouvelle phase de détection des cliques parallèles est opérée,

révélant deux nouvelles paires de cliques parallèles. L'algorithme continue ainsi jusqu'à aboutir à un graphe irréductible, ici le graphe vide.

Le fait que le graphe irréductible auquel aboutit l'algorithme est vide signifie que l'algorithme de réduction permet de résoudre de façon optimale cette instance en temps polynomial. Il ne reste plus qu'à dépiler les sommets et donner aux sommets fusionnés la même couleur. Il est intéressant de remarquer que la solution ainsi calculée ne requiert que trois couleurs alors que n'importe quel algorithme de fusion appliqué sur le graphe d'interférence-affinité, tel que défini par Chaitin, n'aurait pu satisfaire les affinités du graphe en n'utilisant que trois couleurs. Ce n'est d'ailleurs pas la faute de l'algorithme, puisqu'une telle coloration n'existe tout simplement pas, mais du modèle, trop restrictif. En effet, si l'affinité (j, b) de ce graphe est satisfaite alors, après fusion de celle-ci, les sommets $\{e, f, m, jb\}$ forment une 4-clique, ce qui implique que le graphe n'est pas 3-colorable.

Il est simple de voir pourquoi nous avons pu trouver une 3-coloration du graphe éclaté : k et $k2$ ne sont pas colorés de la même couleur, tout comme j et $j5$. Il en était d'ailleurs de même pour le graphe d'interférence-affinité de ce programme sous forme SSA.

7.3 Stratégie de fusion optimale

7.3.1 Le processus

La recomposition est la première étape de notre stratégie de fusion optimale. Cette démarche est composée de quatre étapes et préserve l'optimalité de la fusion. Le processus complet de cette réduction est détaillé figure 7.4.

La deuxième étape consiste en une méthode de décomposition classique utilisant les cliques de bloc d'instructions comme séparateurs du graphe. La troisième étape est la résolution de la fusion dans chacune des composantes de la décomposition. Cette phase peut être traitée par n'importe quel algorithme de fusion. Cependant, une résolution optimale est privilégiée puisqu'elle permet d'atteindre ensuite une solution optimale sur le graphe entier. La dernière étape est la construction de la solution complète à partir de chacune des solutions obtenues à l'étape précédente.

7.3.2 Décomposition via les cliques de séparation

La décomposition par cliques de séparation est une approche classique de décomposition de problèmes valable en particulier pour les problèmes de coloration ou d'allocation de registres [TY84, GSO94]. La décomposition que nous proposons ici s'appuie sur la structure des graphes et sur les cliques d'instructions pour trouver les cliques de séparation en temps polynomial.

Définition 55 (Ensemble et clique de séparation)**.** *Un ensemble de séparation est un ensemble de sommets du graphe dont la suppression fait augmenter le nombre de composantes connexes du graphe. Une clique de séparation est un ensemble de séparation et qui induit une clique.*

Diviser pour mieux régner. Les ensembles de séparation permettent généralement de

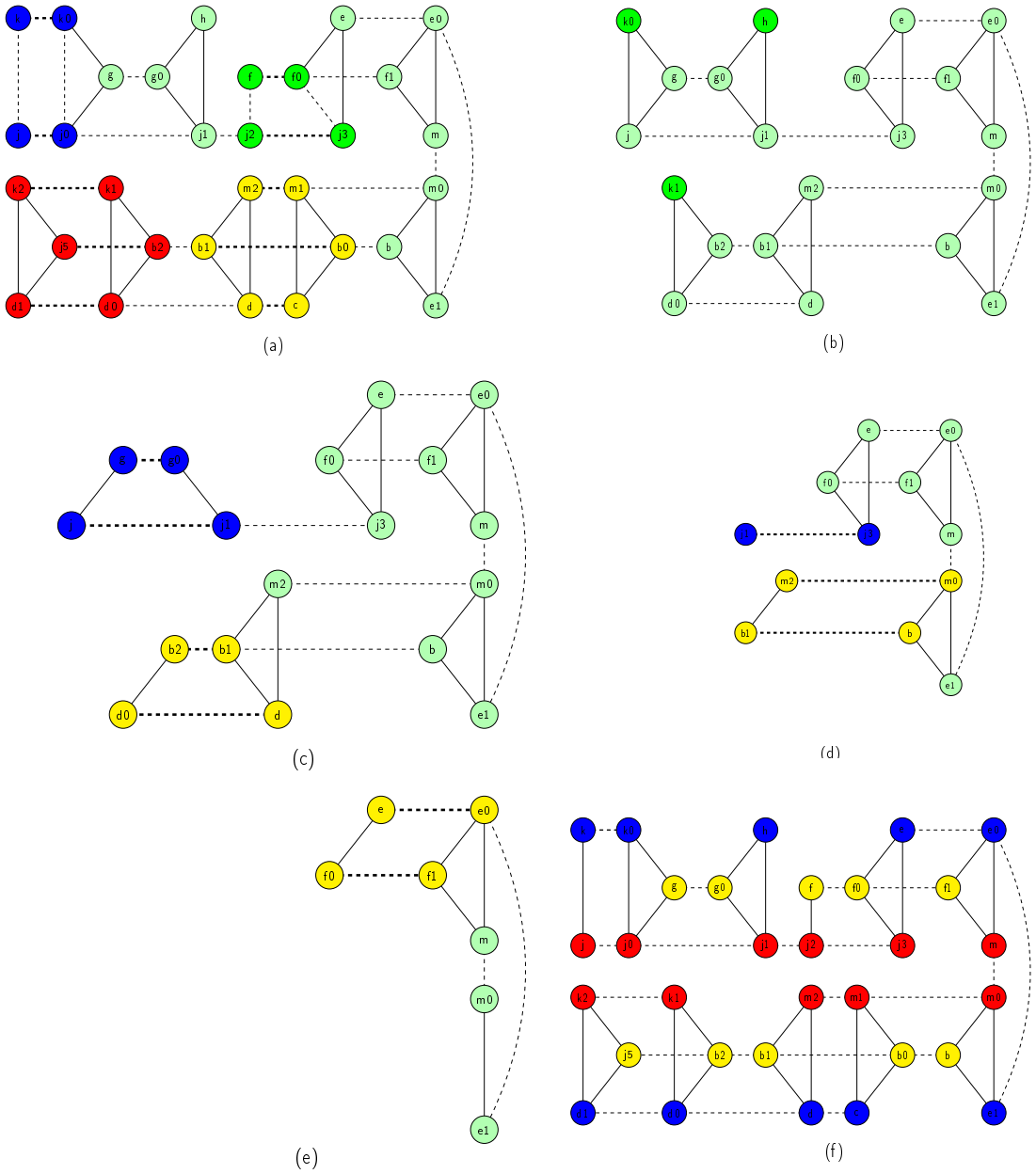


FIG. 7.3 – Application de la recomposition à l'exemple. Les couplages asservissant sont en gras à chaque itération.

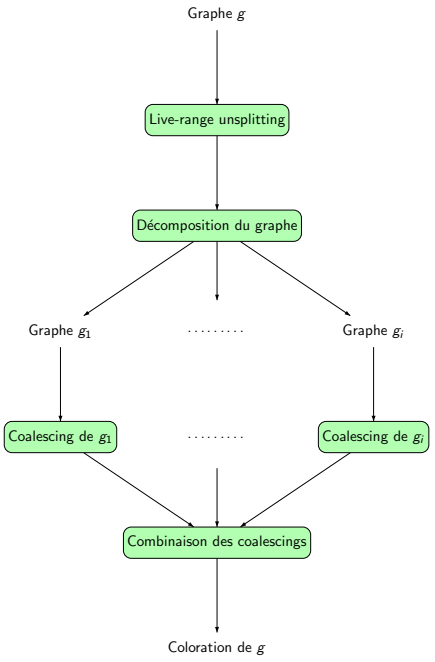


FIG. 7.4 – Le processus complet de fusion.

résoudre des problèmes sur chacune des composantes connexes résultant de leur suppression puis de reconstruire une solution globale. Encore faut-il que les ensembles de séparation soient convenablement choisis : il faut être sûr de pouvoir construire la solution du problème initial à partir des morceaux de solution. Les cliques de séparation conviennent pour le problème de fusion, comme le montre le théorème suivant.

Théorème 32 (Correction de la décomposition)**.** *Soit s une clique d'interférences d'un graphe G qui sépare ce graphe en les composantes $c_1, \ldots, c_j$. Soient $col_1, \ldots, col_j$ des colorations des graphes induits par $c_1 \cup s, \ldots, c_j \cup s$ et col_s une coloration de s.*
Soit Π_i, pour $1 \leq i \leq j$, une permutation de $\{1, \ldots K\}$ telle que pour chaque sommet v de s, $\Pi_i(col_i(v)) = col_s(v)$. La coloration col définie telle que $\forall x \in c_i, col(x) = \Pi_i(col_i(x))$ est une coloration de G.

Démonstration. col est une coloration de $c_i \cup s$, pour tout i de $\{1, \ldots, j\}$. De plus, l'intersection des c_i est vide et les permutations sont construites afin de faire coïncider les couleurs affectées aux sommets de s, ce qui implique que col est bien une coloration de G. □

En d'autres termes cette propriété implique qu'il est possible, après l'éventuelle application de permutations, de coller les colorations des différentes composantes issues de la décomposition pour obtenir une coloration du graphe initial.

De plus, cette permutation préserve l'accession à des solutions optimales pour le graphe entier. En effet, si l'on sait calculer les solutions optimales pour chacune des composantes de

la décomposition, leur appliquer des permutations et les combiner conduit à une coloration optimale : les permutations ne modifient pas la valeur de la solution, et l'optimalité dans chaque composante induit l'optimalité globale.

Théorème 33 (Préservation de l'optimalité par décomposition)**.** *Soit s une clique d'interférences d'un graphe G qui sépare G en les composantes $c_1, \ldots, c_j$. Soient respectivement $col_1, \ldots, col_j$ des colorations optimales des graphes induits par $c_1 \cup s, \ldots, c_j \cup s$ et col_s une coloration de s.*
Soit Π_i, pour $1 \leq i \leq j$, une permutation de $\{1, \ldots K\}$ telle que pour chaque sommet v de s, $\Pi_i(col_i(v)) = col_s(v)$. La coloration col telle que $\forall x \in c_i, col(x) = \Pi_i(col_i(x))$ est une coloration optimale de G.

Démonstration. D'après le théorème précédent, *col* est bien une coloration de G. Il suffit de remarquer que la valeur d'affinité de *col* est la somme des valeurs d'affinité des col_i pour $1 \leq i \leq j$. Ainsi, *col* est optimale puisque toutes les col_i sont optimales. □

Contrairement aux graphes quelconques, cette décomposition peut être réalisée en temps linéaire dans les graphes éclatés. En effet, la tâche la plus difficile est en principe de trouver les cliques de séparation. Généralement cette recherche se focalise sur les plus petits séparateurs, mais cela nous est ici inutile ; s'en tenir aux cliques maximales d'interférence est suffisant, toute clique de séparation étant incluse dans l'une de celles-ci. Or, dans les graphes éclatés ce problème est polynomial.

Un simple algorithme permet de calculer toutes les cliques d'interférences maximales qui sont des cliques de séparation. Il consiste dans un premier temps à construire un graphe G tel que tout sommet de G correspond à une clique d'interférences maximale du graphe éclaté. Deux sommets de G sont liés par une arête si des sommets des cliques qu'ils représentent sont liés par une arête dans le graphe éclaté. Les séparateurs de G sont alors exactement les cliques d'interférence maximales que nous recherchons. La recherche des sommets de séparation de G est confiée à un algorithme classique de parcours de graphe [Wes00].

Rechercher les cliques de séparation est d'autant plus légitime que la recomposition tend à faire des cliques de bloc d'instructions des cliques de séparation. En effet, si l'union de deux cliques parallèles est un ensemble de séparation alors la clique résultant de la fusion des deux premières est une clique de séparation. Mais la force de cette décomposition réside surtout dans sa faculté à casser les permutations du problème, tout en réduisant la taille de chacun des sous-problèmes qu'il faut résoudre. C'est d'autant plus important pour un problème de coloration, dont on sait que la résolution est corsée par le grand nombre de permutations. De plus, dans l'optique d'une résolution optimale reposant sur l'algorithme de Grund et Hack [GH07], supprimer des permutations peut grandement améliorer les performances des solveurs de programmation linéaire, lesquels sont très sensibles aux permutations.

Nombre initial de sommets	Nombre d'instances	Ratio du nombre de sommets après/avant	Ratio du nombre d'arêtes après/avant
0-499	292	18%	33%
500-999	97	14%	27%
1000-2999	63	13%	27%
plus de 3000	22	13%	8%

FIG. 7.5 – Réduction de la taille des instances de l'OCC. Le ratio présenté est celui entre le nombre de sommets (respectivement arêtes) appartenant à la plus grande composante de la décomposition et le nombre de sommets (respectivement arêtes) appartenant au graphe initial.

7.4 Résultats expérimentaux

Comme mentionné précédemment, nous utilisons les 474 graphes éclatés de l'OCC comme jeu de test. Notre démarche de fusion appliquée sur les graphes de l'OCC génère des graphes plus petits qui sont donnés en entrée du programme linéaire défini dans [GH07]. Nous utilisons le solveur CPLEX 9.0, accompagné du modeleur AMPL, sur un PENTIUM 4 cadencé à 2.26Ghz. La première partie des résultats expérimentaux se focalise sur les statistiques des graphes générés par notre réduction tandis que les deux dernières parties concernent respectivement les résultats portant sur les résolutions optimales et sub-optimales du problème de fusion.

7.4.1 Réduction et décomposition

La première mesure importante est le ratio entre la taille d'un graphe de l'OCC et la taille de la plus grande composante résultant de notre décomposition appliquée à ce graphe. Nous ne considérons que la plus grande composante restante car la résolution du problème sur celle-ci requiert généralement la majeure partie du temps passé à résoudre toutes les instances. Les résultats sont détaillés dans figure 7.5.

La réduction moyenne est significative puisque les nombres de sommets et d'arêtes sont respectivement divisés par sept et quatre par notre algorithme. Le ratio est d'ailleurs meilleur pour les plus grands graphes, ce qui peut mécaniquement s'expliquer par le fait que les sommets précolorés (au nombre de six pour les instances de l'OCC) ne sont jamais supprimés du graphe et représentent un pourcentage de sommets parfois non négligeable pour les plus petits graphes. Au sein même de cette réduction, 90% de la réduction de taille est due uniquement à la recomposition. Cette réduction importante ne nécessite que six secondes de calcul pour l'ensemble complet des instances de l'OCC, alors que le gain de combinatoire engendré est immense.

7.4.2 Solutions optimales

Nous calculons les solutions optimales pour chacune des composantes issues de la décomposition, à l'aide de l'algorithme de coupes de Grund et Hack [GH07]. Pour chaque

interférence, nous ne générons qu'une coupe de chaîne, correspondant à une plus courte chaîne d'affinité joignant ses extrémités, grâce à l'algorithme de Dijkstra [Wes00]. La figure 7.6 illustre une chute des temps de calcul des solutions optimales après usage de notre méthode de réduction. Par exemple, l'algorithme de coupes n'est capable de calculer que 430 solutions optimales en moins de cinq minutes (par instance) tandis qu'après réduction l'algorithme de coupes permet de déterminer 436 solutions en moins d'une seconde (par instance). En moyenne, l'algorithme de coupes permet d'accéder aux solutions optimales 300 fois plus rapidement lorsqu'il est combiné avec notre réduction, comme le montre la figure 7.7. De plus, seules six instances requièrent plus d'une minute pour être résolues et seulement trois nécessitent plus de 150 secondes.

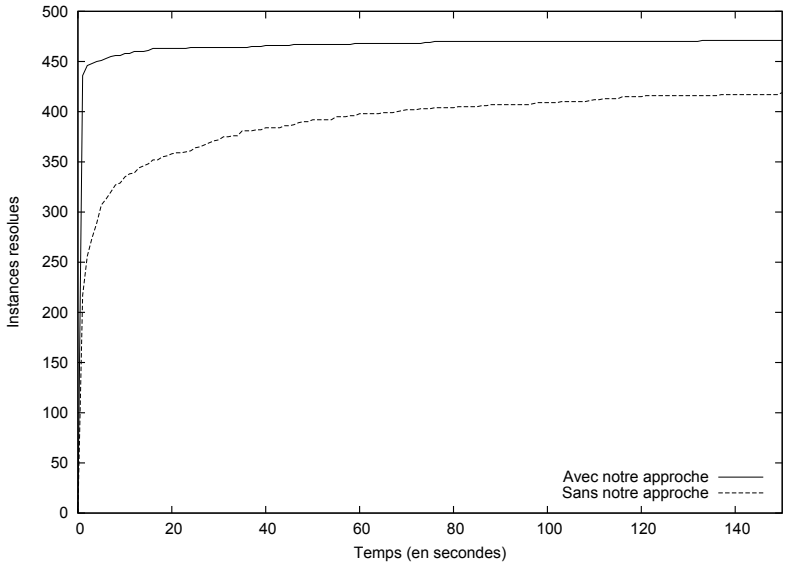


FIG. 7.6 – Nombre d'instances de l'OCC résolues en moins de 150 secondes par l'algorithme de coupes avec et sans réduction préalable.

Enfin, toutes les instances de l'OCC sont résolues de façon optimale pour la première fois. En effet, trois de ces instances restaient ouvertes suite à l'échec de l'application directe de l'algorithme de coupes. Nous avons trouvé une solution optimale pour chacune de ces trois instances. Pour deux d'entre elles, la solution optimale est de même valeur qu'une solution admissible trouvée par l'algorithme de coupes dont l'optimalité n'était pas avérée. La dernière solution est strictement meilleure que celle connue jusqu'alors.

7.4.3 Solutions sub-optimales

La plupart des instances de l'OCC sont résolues en moins de quelques secondes. Néanmoins, certains problèmes n'échappent pas à l'explosion combinatoire. Afin de mieux évaluer les raisons de cette explosion, nous paramétrons le solveur pour les six instances prenant plus d'une minute pour être résolues. Les résultats numériques observés en fonction des paramètres modifiés sont donnés tableau 7.8.

Nombre d'instances résolues	Temps avec réduction (s)	Temps sans réduction (s)	Ratio de temps sans/avec
436	1	298	298
446	2	636	318
448	3	732	244
450	4	1198	300
451	5	1228	245
453	6	3153	525
455	7	3321	474
456	8	3574	447
458	10	4022	402
460	12	5630	469
461	15	7214	480
463	16	8709	544
464	24	10397	433
465	37	11757	317
466	40	14941	373
467	59	24916	422
468	74	25490	344
469	76	25663	337
470	133	69956	525
471	11026	71208	7

FIG. 7.7 – Comparaison des ratio de temps de calcul des solutions optimales avec et sans notre réduction sur les 471 instances résolues optimalement sans réduction.

Une première façon de paramétrer le solveur est de fixer une limite de temps de calcul maximum. En effet, trouver une solution optimale (ou quasi optimale) ne prend souvent que 10% du temps de résolution total, le reste étant dévolu à vérifier qu'aucune meilleure solution n'existe. Il faut préciser que la limite est fixée pour chaque composante, ce qui peut conduire à un temps de résolution supérieur à la limite pour une instance initiale. Cependant, nous avons constaté qu'empiriquement le problème n'est difficile à résoudre que sur une composante ; la résolution sur les autres étant quasi instantanée. Cette méthode possède néanmoins un défaut majeur : elle peut échouer si aucune solution n'est trouvée dans le délai imparti.

Une façon d'éviter cet obstacle est de paramétrer le solveur afin d'obtenir une solution de qualité imposée, dont la valeur est par exemple à moins de 10% de la valeur optimale. En effet, le solveur dispose à tout moment d'une borne de la valeur de la meilleure solution accessible. Si l'écart entre cette borne et la meilleure solution connue est inférieur au seuil, il en découle que la meilleure solution connue est à moins du seuil de l'optimum. La philosophie cachée derrière cette approche est en quelque sorte l'inverse de la première : le temps imparti pour la résolution était fixé et la qualité de la solution était évaluée, la qualité de la solution est maintenant fixée et le temps de calcul est évalué.

Instance	144	304	371	387	390	400
Optimum	129332	6109	1087	3450	339	1263
20s	129333	none	none	none	417	1388
30s	129333	none	1285	none	365	1263
10% gap	132040	6448	1094	3550	339	1263
5% gap	129342	6273	1094	3450	339	1263

Instance	144 s.	304 s.	371 s.	387 s.	390 s.	400 s.
Optimum	11026 s.	1058 s.	132 s.	29543 s.	102 s.	75 s.
10% gap	15 s.	36 s.	62 s.	115 s.	86 s.	21 s.
5% gap	17 s.	64 s.	62 s.	1187 s.	92 s.	21 s.

FIG. 7.8 – Comparaison entre les différentes approches pour résoudre les instances les plus difficiles de l'OCC. Le tableau du haut contient les valeurs des solutions, celui du bas le temps de calcul en secondes. *none* signifie qu'aucune solution n'est calculée dans le délai imparti. Les temps sont exprimés en secondes.

Les résultats des tableaux 7.8 donnent un aperçu de la qualité de la fusion sur les instances les plus difficiles de l'OCC : les nombres indiqués sur la première ligne correspondent aux numéros d'instances dont la résolution prend plus d'une minute. Ce tableau permet tout d'abord d'observer que la limite de temps de 20 secondes échoue ou permet d'obtenir une solution de bonne qualité, selon les cas. Ici, le temps imparti est insuffisant pour trouver une solution admissible pour trois des instances. Intuitivement, cela signifie que trouver une solution admissible est difficile, mais que dès lors qu'une est trouvée, il est généralement peu difficile d'accéder à une solution de bonne qualité. Faire augmenter la limite de temps semble donc intéressant.

Nous avons donc dans un second temps fixé la limite de temps à 30 secondes. Dans ce cas, une solution admissible est trouvée pour une quatrième instance et toutes les solutions admissibles sont à moins de 20% de l'optimum. Ce comportement confirme donc bien la convergence rapide vers la solution optimale. Limiter le temps permet donc de mieux cerner la difficulté d'une instance. Pour l'OCC, deux instances seulement semblent vraiment nécessiter un long temps de calcul. Les temps de calcul à l'optimum confirment d'ailleurs cette conjecture.

La convergence vers l'optimum semblant rapide, calculer une solution sub-optimale de qualité garantie apparait judicieux. Pour les six instances considérées, fixer une limite maximale à 10% de l'optimum conduit à des solutions de très bonne qualité (en fait à moins de 5% de l'optimum) en moins de deux minutes par instance. Fixer un seuil de qualité plus restrictif, tel que 5% fait par contre apparaître le phénomène d'explosion combinatoire puisque les temps de calcul peuvent grandement croître, comme pour l'instance 387. Notons d'ailleurs que pour cette instance, 1187 secondes suffisent à trouver une solution optimale et à assurer que celle-ci est au pire à 5% de l'optimum. Les 28000 secondes de calcul (soit environ 8h) supplémentaires ne servent qu'à s'assurer de l'optimalité de la solution.

7.5 Travaux connexes

Il nous faut mentionner qu'en parallèle des travaux présentés dans ce chapitre et réalisés à la fin de l'année 2007, Bouchez, Darte et Rastello ont observé des règles de réduction très proches des nôtres, ce qui n'est guère surprenant au vu de la configuration très particulière des graphes de l'OCC [Bou08]. Les résultats diffèrent cependant légèrement sur deux points. Nous autorisons les sommets simplifiables à être supprimés pour faire apparaître de nouveaux couplages asservissants et donc d'appliquer la réduction en boucle, la rendant de ce fait plus efficace, ce qu'ils ne font pas. En contrepartie, leur résultat théorique est plus général puisqu'ils permettent de fusionner non pas seulement deux cliques parallèles, mais un sous-graphe G_1 et un graphe isomorphe à un sous-graphe de G_1. Notre restriction aux cliques n'est pas gênante ici, puisque nous avons pu prouver que seules des cliques apparaissent dans les graphes éclatés, mais leur résultat peut espérer être étendu à d'autres classes de graphes, et en particulier aux graphes élémentaires.

7.6 Perspectives

7.6.1 Vers une stratégie de découpage idéale

L'algorithme que nous venons de présenter pallie partiellement le défaut principal du découpage extrême, *i.e.* l'ajout de copies inutiles. Un enjeu majeur pour l'allocation de registres par coloration de graphe est de savoir définir clairement une stratégie idéale de découpage, *i.e.* qui soit aussi précise que le découpage extrême (en ajoutant toutes les copies susceptibles d'être effectivement utilisées pour améliorer la qualité de l'allocation de registres) et se limite à un nombre bien plus restreint de copies. Cette question, soulevée lors de discussions avec des membres de l'équipe Compsys, n'a à l'heure actuelle toujours aucune réponse définitive.

Si l'on sait qu'une stratégie idéale se situe entre la forme SSA et le découpage extrême, celle-ci s'avère être très difficile à identifier. Pour lors, la recomposition ne nous sert qu'à supprimer des points de découpage préalablement construits et non à déterminer si un point de découpage doit être ajouté avant la construction du graphe d'interférence-affinité. Néanmoins, nous avons pu remarquer que les blocs de base ne contenant pas de sommets précolorés ont tendance à être contractés en un seul bloc d'instructions. Les points de découpage conservés ont donc tendance à se situer avant les branchements ou après les jonctions. Nous retrouvons donc des similitudes frappantes avec la forme SSA. Les sommets précolorés jouent cependant un rôle crucial et interdisent nombre de recompositions. Mieux cerner les restrictions qu'ils engendrent semble donc primordial pour définir une stratégie de découpage idéale.

7.6.2 Comparaison des graphes élémentaires et éclatés

Afin de mieux comprendre les rouages de la recomposition, et éventuellement de la généraliser, nous avons décidé, après discussion avec des membres de l'équipe Compsys, de réfléchir à la recomposition sur les graphes élémentaires. Il en ressort que ce principe très local, au sens où il suffit de considérer une toute petite partie du graphe éclaté pour

décider d'une recomposition, n'est pas applicable aux graphes élémentaires. En effet, nous ne sommes pas parvenus à définir une propriété analogue aux cliques asservissantes dans ce type de graphes. Cette conclusion nous a poussé à étudier les différences entre ces deux modèles. Si l'on considère le graphe élémentaire donné figure 7.9, on peut rapidement remarquer que copier une variable avant sa dernière utilisation est inutile si cette dernière utilisation est une copie. Par exemple, copier c en $c0$ (ou encore $b2$ en $b3$) est inutile. En effet, copier c en $c0$ puis $c0$ en d peut se simplifier à copier c en d. Il est donc possible de supprimer les sommets $c0$ et $b3$ et d'ajouter les affinités (c, d) et $(b2, j5)$.

Une fois cette opération faite sur le graphe élémentaire, nous pouvons remarquer que le graphe éclaté de ce programme est un sous-graphe de son graphe élémentaire tel que les seuls sommets qui appartiennent au graphe élémentaire et non au graphe éclaté représentent, semble-t-il, des variables copiées au point de programme précédant leur dernière utilisation. Sur l'exemple, retirer les sommets $k1$, $g1$, $h0$, $j4$, $f2$, $e2$ et $m3$ du graphe élémentaire conduit au graphe éclaté. Ce résultat n'est pour lors qu'au stade de conjecture. Le prouver est une perspective intéressante pour établir le lien entre ces deux modèles.

Changer une variable de registre au moment de sa dernière utilisation est *a priori* peu utile. Toutefois, la présence de sommets précolorés peut à première vue induire ce besoin. Observer expérimentalement le nombre de copies effectuées juste avant la dernière utilisation d'une variable apparaît comme une solution envisageable pour déterminer si de telles copies sont profitables. Si tel est le cas, raisonner sur les graphes élémentaires pourra s'avérer plus précis. Autrement, les graphes éclatés pourront leur être privilégiés.

7.7 Bilan

Notre motivation première était de travailler sur un modèle plus précis, sur lequel une résolution optimale est souhaitable, et résoudre l'allocation de registres par programmation linéaire. Plutôt que de travailler sur le programme linéaire, nous avons voulu exploiter les propriétés méconnues des graphes éclatés pour réduire la taille de ces graphes. Cette approche est d'autant plus légitime que résoudre un programme linéaire est une opération exponentielle en la taille de celui-ci.

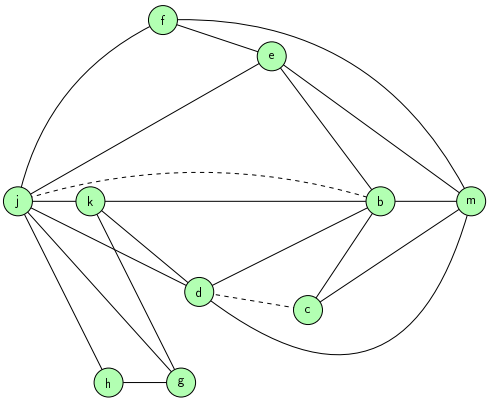
Plus précisément, nous nous sommes focalisés sur le principal défaut du découpage extrême : l'ajout de copies inutiles. Nous avons défini une méthode de reconnaissance de certaines de ces copies fondée sur des propriétés locales du graphe et en avons déduit un algorithme de suppression de ces copies. Nous décomposons ensuite le graphe afin de résoudre la fusion sur de plus petites composantes puis recomposons le graphe pour obtenir la solution finale. N'importe quel algorithme de fusion peut être appliqué sur les composantes, mais un algorithme optimal est privilégié puisqu'une résolution optimale sur chaque composante mène à une solution finale optimale. La combinaison de cette stratégie de réduction avec l'algorithme de coupes de Grund et Hack se révèle efficace, puisqu'elle permet de résoudre pour la première fois toutes les instances de l'OCC.

Ce travail constitue un premier pas vers la définition d'une stratégie de découpage plus efficace, *i.e.* qui n'introduirait qu'un petit sur-ensemble des découpages nécessaires à la résolution optimale de l'allocation de registres. La définition d'une telle stratégie de découpage, intermédiaire entre la forme SSA et la forme élémentaire, est une perspective

Entrée : k j

(p_0) g := mem[j+12]
(p_1) h := k - 1
(p_2) f := g * h
(p_3) e := mem[j+8]
(p_4) m := mem[j+16]
(p_5) b := mem[f]
(p_6) c := e + 8
(p_7) d := c
(p_8) k := m + 4
(p_9) j := b
(p_{10})

Sortie : d j k



Entrée : k j

(p_0) k0 := k ‖ j0 := j
g := mem[j0+12]
(p_1) j1 := j0 ‖ g0 := g ‖ k1 := k0
h := k1 - 1
(p_2) j2 := j1 ‖ g1 := g0 ‖ h0 := h
f := g1 * h0
(p_3) f0 := f ‖ j3 := j2
e := mem[j3+8]
(p_4) e0 := e ‖ f1 := f0 ‖ j4 :=j3
m :=mem[j4+16]
(p_5) e1 := e0 ‖ m0 := m ‖ f2 := f1
b :=mem[f2]
(p_6) b0 := b ‖ m1 := m0 ‖ e2 := e1
c := e2 + 8
(p_7) b1 := b0 ‖ m2 := m1 ‖ c0 := c
d := c0
(p_8) b2 := b1 ‖ d0 := d ‖ m3 :=m2
k2 := m3 + 4
(p_9) d1 := d0 ‖ k3 := k2 ‖ b3 :=b2
j5 := b3
(p_{10})

Sortie : d1 j5 k3

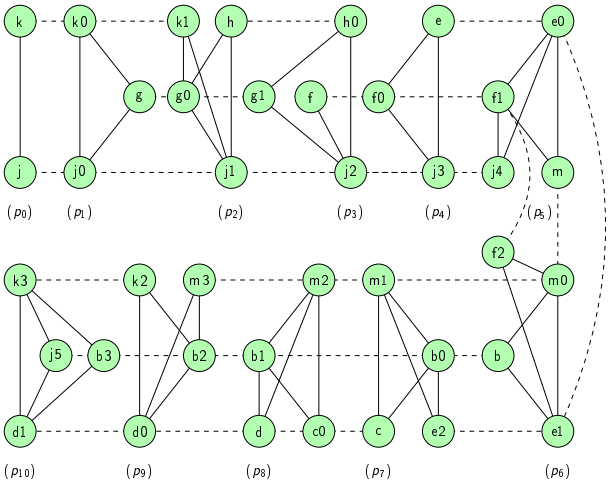


FIG. 7.9 – Le programme de référence, son graphe d'interférence-affinité, sa forme élémentaire telle que définie par Palsberg et Pereira et son graphe éclaté.

peu considérée dans la littérature mais dont l'impact sur l'allocation de registres pourrait s'avérer important.

Chapitre 8

Conclusions et perspectives

L'allocation de registres est une passe de compilation cruciale. La coloration de graphe d'interférence-affinité permet de modéliser de façon simple et intuitive ce problème, et de le résoudre. Cependant, il est bien connu que ce modèle sur-approxime les contraintes réelles pesant sur l'allocation de registres. Dans ce manuscrit, nous avons soutenu que les méthodes d'optimisation appliquées pour résoudre l'allocation de registres, et par conséquent les approches de vérification formelle employées, dépendent essentiellement de la représentation du programme interne au compilateur. Nous avons ainsi abordé trois problèmes très différents répondant à des questions essentielles pour trois formes de programme connues : les programmes quelconques, les programmes sous forme SSA et les programmes sous forme élémentaire. Le point commun de ces approches est l'utilisation du modèle d'allocation de registres par coloration de graphe.

8.1 Contributions

8.1.1 Graphes de Chaitin et vérification formelle

L'évolution des graphes de Chaitin pour intégrer les affinités a mené au modèle de graphes d'interférence-affinité les plus généraux, applicable à tout programme. C'est le modèle d'allocation de registres tel qu'il est posé depuis les années 80. Ce modèle souffre d'un léger manque de précision, cependant parfaitement acceptable, et le problème correspondant est NP-difficile , ce qui conduit à résoudre ce problème de manière heuristique.

L'IRC est une heuristique classique et efficace d'allocation de registres par coloration de graphe. Nous avons spécifié, développé et prouvé correcte une implantation purement fonctionnelle de cette heuristique dans le système Coq . Celle-ci peut être considérée comme une implantation de référence de cette heuristique initialement publiée avec des bogues et souvent implantée de façon erronée. Nous avons ensuite pu intégrer cet algorithme à un prototype du compilateur CompCert . Ce travail a notamment nécessité de façonner une bibliothèque extractible de graphes d'interférence-affinité. Cette dernière est composée d'une interface fondée sur une représentation très expressive des graphes d'interférence-affinité et d'une implantation efficace de cette interface. L'expérience de la formalisation de l'IRC nous a permis de généraliser cette approche afin de simplifier l'accès à la vérifica-

tion formelle d'algorithmes d'allocation de registres par coloration de graphe, pourvu que l'algorithme en question n'utilise que les quatre fonctions classiques utilisées pour colorer les graphes d'interférence-affinité, *i.e.* la simplification, la fusion, le gel et le vidage.

Cette expérience nous pousse à penser que la preuve de programme est adaptée à ce type d'algorithme. Pouvoir assurer la correction d'un algorithme optimisant d'allocation de registres est une force pour un compilateur. Même un compilateur désireux d'implanter un algorithme plus complexe d'allocation de registres et de le valider *a posteriori* peut avoir besoin d'un algorithme correct par construction afin de le suppléer en cas de déni du validateur. Ainsi, l'allocation de registres ne peut plus être une cause potentielle d'échec de la compilation.

8.1.2 Graphes SSA et approche fondée sur la largeur d'arbre

La forme SSA est aujourd'hui la représentation intermédiaire de programme au cœur de toutes les attentions en termes d'allocation de registres. Celle-ci conduit à un modèle d'allocation de registres relativement précis, si bien que les résolutions heuristiques ou optimales sont toutes deux légitimes. De plus, les récentes découvertes, en particulier le caractère triangulé des graphes d'interférence des programmes sous forme SSA, ont permis des avancées qualitatives importantes pour le vidage, mais restent peu exploitées pour la fusion.

Nous avons défini une nouvelle approche de résolution de la fusion par programmation linéaire dans les graphes SSA s'appuyant sur la largeur d'arbre du graphe d'interférence-affinité. La façon dont est construit le programme montre bien que la fusion doit se fonder sur des propriétés globales du graphe et non sur des critères de fusion très locaux comme c'était le cas jusqu'à présent pour accéder à des résolutions optimales.

En termes de vérification formelle, nous préconisons la validation *a posteriori*, à la fois en raison de la complexité de la preuve de cet algorithme et des structures de données qu'elle manipule, et de l'utilisation d'un programme externe, le solveur de programmation linéaire.

8.1.3 Graphes éclatés et recomposition

Le découpage extrême conduit au modèle par coloration de graphe le plus précis puisqu'il devient possible de décider de l'emplacement dans lequel est placée chaque variable du programme à tout instant de son exécution. En effet, chaque sommet du graphe représente une variable en un point de programme. Cette construction du graphe d'interférence-affinité permet de résoudre le vidage de façon optimale, comme l'ont décrit Appel et George [AG01]. Elle induit en outre deux conséquences importantes pour la résolution de la fusion. Tout d'abord, certains points de découpage sont naturellement inutiles pour la fusion. Ensuite, la taille du graphe est gigantesque. Or, il est inutile de construire un modèle si précis si ce n'est pour résoudre le problème de coloration de façon optimale, ce qui nécessite un nombre exponentiel d'opérations puisque celui-ci est NP-difficile . Afin de rendre la résolution de la fusion dans un tel contexte envisageable, nous avons défini une stratégie dite de recomposition et un algorithme de recherche de points de découpage inutiles permettant de réduire amplement la taille des graphes d'interférence éclatés et ainsi

de résoudre plus efficacement le problème de fusion dans ces graphes. Cette fois encore, la seule forme de vérification formelle envisagée est la validation *a posteriori*.

8.1.4 Trois problèmes, trois solutions

Pour conclure sur cette étude, nous n'avons pas abordé un problème, celui de l'allocation de registres, mais trois, selon la représentation du programme interne au compilateur. Les solutions proposées diffèrent d'ailleurs très largement car il est primordial que l'algorithme de résolution soit en adéquation avec le modèle en termes de précision. La vérification formelle de ces algorithmes est d'autant plus ardue que leur efficacité algorithmique est importante. La complémentarité entre preuve et validation permet de vérifier formellement tous ces algorithmes, sans risque d'erreur de compilation, puisque la version correcte par construction de l'IRC est toujours en mesure de prendre le relais du validateur en cas de d'échec de la validation d'une solution. En ce sens, ces travaux ont contribué à optimiser et rendre plus sûre l'une des phases les plus complexes de compilation, l'allocation de registres.

8.2 Perspectives

8.2.1 Algorithme prouvé pour architectures à peu de registres

L'IRC a été choisi comme algorithme d'allocation de registres pour le compilateur CompCert en raison de son adéquation avec les architectures `PPC` et `ARM` qui disposent de nombreux registres. En revanche, pour des architectures disposant de peu de registres et soumis à des contraintes plus fortes, comme l'architecture x86, cette heuristique atteint ses limites. C'est d'autant plus vrai que garder deux des huit registres allouables sur cette architecture uniquement pour insérer le spill code constitue une hypothèse trop pessimiste et restrictive.

A l'heure actuelle, la solution la plus adaptée pour résoudre et vérifier formellement l'allocation de registres sur de telles architectures est la validation *a posteriori*. En effet, le validateur de CompCert développé récemment permet de ne pas monopoliser deux des registres pour insérer le spill code puisqu'il ne procède pas par coloration de graphe mais par vérification d'une correspondance entre les graphes de flot de contrôle du programme avant et après insertion du spill code. Il est de plus capable de gérer les paires de registres dont les valeurs sont dépendantes, comme il en existe sur ce type d'architectures. Cependant, les contraintes de ce type d'architectures sur les registres sont si fortes qu'il est d'autant plus délicat d'implanter un allocateur de registres parfaitement exempt de bogues.

Nous souhaitons donc poursuivre dans la voie de la preuve. A court terme, la perspective la plus logique est sans nul doute d'intégrer la reconstruction à notre implantation de l'IRC. Libérer les registres monopolisés par l'insertion de spill code permettrait ainsi d'aller vers des allocations de registres plus fines et correctes par construction. Cette reconstruction doit donc être étudiée, pour en prouver à la fois la terminaison, éventuellement via la mesure que nous avons conjecturée chapitre 4, et la correction. Cette correction passe par une preuve de préservation sémantique suite à l'introduction du spill code. Cette preuve est déjà présente dans CompCert et il est probable que celle-ci soit suffisante pour montrer

la correction de la reconstruction. Toutefois, vérifier formellement des algorithmes apprend à ne pas vendre la peau de l'ours avant de l'avoir tué, et seule une implantation de la reconstruction sera à même de confirmer cette supposition.

A plus long terme, intégrer un nouvel algorithme d'allocation de registres prouvé correct et dédié aux architectures CISC pourrait être envisageable. Cependant, il est difficile d'évaluer la quantité de travail nécessaire à cet objectif, puisque les mécanismes à considérer, comme la superposition des registres en particulier, pourraient conduire à de nouvelles problématiques de formalisation et de preuve.

8.2.2 Fusion guidée par la largeur d'arbre

L'approche de fusion guidée par la largeur d'arbre du graphe d'interférence-affinité permet de séparer les phases de fusion et d'affectation des registres, et ainsi de découpler les complexités des deux problèmes sans perte théorique de qualité de l'allocation de registres. De plus, la largeur d'arbre du graphe permet de guider la phase de fusion. En effet, une stratégie de fusion est conservative si elle fait décroître la largeur d'arbre du graphe en dessous du nombre K de registres allouables. Nous avons pu modéliser par la programmation linéaire la recherche d'un ensemble d'affinités dont la fusion fait décroître la largeur d'arbre en dessous de K dans les graphes SSA et ainsi résoudre d'une façon nouvelle le problème de fusion conservative dans les graphes SSA. Ce travail ouvre la voie vers deux perspectives que nous comptons explorer à court terme.

Tout d'abord, nous souhaitons utiliser ces résultats pour définir une nouvelle heuristique d'allocation de registres pour les graphes SSA fondée sur la largeur d'arbre du graphe d'interférence-affinité. L'idée serait d'utiliser une stratégie de fusion conservative sur les zones du graphe qui font grandir la largeur d'arbre au-dessus de K, puis d'appliquer une stratégie de fusion agressive. Le problème réside dans le fait que la fusion conservative peut ne jamais mener à un graphe de largeur d'arbre inférieure à K. Il faut donc déterminer s'il est possible de définir une heuristique qui assure cette décroissance nécessaire à la définition de cet algorithme.

Ensuite, si les résultats obtenus sur les graphes SSA sont satisfaisants, nous aimerions étendre ce principe de largeur d'arbre aux graphes de Chaitin. La plupart de ces graphes s'avèrent être triangulés, et il se pourrait qu'ils soient parfaitement triangulés ou possèdent une largeur d'arbre proche de K après le vidage. Des expérimentations doivent être menées pour déterminer si cette démarche a un sens.

8.2.3 Vérification formelle de compilateurs et forme SSA

La vérification formelle d'un compilateur incorporant une représentation sous forme SSA du programme ainsi que les optimisations dédiées à la forme SSA est une perspective d'ampleur. En effet, la forme SSA s'impose peu à peu dans les compilateurs en raison de ses caractéristiques propices à l'optimisation. En revanche, les aspects sémantiques liés à cette représentation restent assez mal décrits, et encore moins formalisés. Ce travail conduirait à de nouveaux défis de vérification formelle d'algorithmes pour la compilation. A la modeste échelle de la vérification formelle d'algorithmes d'allocation de registres pour

les programmes sous forme SSA, ce problème est déjà éminemment complexe, et le travail à fournir, énorme.

Non seulement le compilateur CompCert ne contient pas de forme SSA, mais même si c'était le cas, il n'est absolument pas certain que les propriétés de la forme SSA puissent être facilement exploitées. En effet, il y a un grand pas entre mettre les programmes sous forme SSA et montrer que le graphe d'interférence de tout programme sous forme SSA est triangulé. Il faut, entre autres, définir les graphes triangulés et les décomposition arborescentes, prouver qu'un graphe est triangulé si et seulement s'il admet une décomposition arborescente, et montrer que les durées de vie de chaque variable forme un sous-arbre de l'arbre de dominance.

Prouver la caractérisation des graphes triangulés pourrait être intéressant, aussi bien afin de développer une bibliothèque de graphes triangulés qu'afin d'évaluer la quantité de travail à fournir pour profiter des bienfaits de la forme SSA pour l'allocation de registres. Ce travail constitue donc une perspective potentielle pour aller vers la formalisation d'optimisations SSA. Une autre voie envisageable à plus court terme consiste à étudier les propriétés d'un compilateur d'un langage purement fonctionnel vérifié formellement, dont les représentations intermédiaires sont parfois très proches de la forme SSA, afin d'évaluer la faisabilité d'un compilateur SSA vérifié formellement.

8.2.4 Stratégie de découpage idéale

Le découpage extrême permet de définir un modèle précis d'allocation de registres par coloration de graphe. Cependant, sur des programmes conséquents, la taille des graphes peut vite devenir très grande et il devient alors difficile de les manipuler dans un compilateur. La stratégie de recomposition permet de réduire le nombre de points de découpage après avoir construit le graphe d'interférence-affinité. L'enjeu est désormais de définir comment savoir si un point de découpage est nécessaire ou non avant la construction du graphe et de se diriger ainsi vers un modèle précis mais de taille modérée, de connaître les caractéristiques des graphes ainsi construits, et de ne se concentrer que sur la résolution du problème sur le graphe, sans pâtir des défauts de sur-approximation que peuvent engendrer le modèle par coloration de graphe. Nous pensons qu'étudier les graphes élémentaires permettra d'obtenir de nouveaux indices sur la voie à suivre pour définir une stratégie de découpage idéale.

8.2.5 Vérification formelle et optimisation combinatoire

Le développement d'une bibliothèque de graphes extractibles au sein du système Coq est un élément qui pourrait servir de point de départ à la formalisation d'autres algorithmes de théorie des graphes. La formalisation présentée dans ce manuscrit spécifie une petite partie des opérations usuelles de théorie des graphes mais peut d'ores et déjà permettre d'ouvrir des discussions sur la façon de les formaliser dans Coq , dans le but d'intégrer ensuite ces travaux à la bibliothèque standard. Nous espérons ainsi que ce fragment de départ pourra être étendu aux besoins que peut classiquement ressentir un utilisateur.

A sa modeste échelle, cette thèse a contribué à vérifier des algorithmes de théorie des graphes et à simplifier l'intégration de certains algorithmes dans le système Coq . Rendre ce

système accessible à la communauté d'optimisation combinatoire est un objectif que nous espérons voir doucement se concrétiser. Le développement de la bibliothèque de graphes est un premier pas. La preuve de correction de modèles de programmation linéaire vis-à-vis de leur spécification et la validation *a posteriori* de la résolution de ces modèles par des solveurs externes serait un nouveau pas en termes de confiance accordée à de tels modèles, parfois très complexes.

Bibliographie

[AC76] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3) :137–147, 1976.

[AG00] Andrew W. Appel and Lal George. Optimal coalescing challenge, `http://www.cs.princeton.edu/~appel/coalesce`, 2000.

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 243–253, 2001.

[AH76] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. *Bulletin of the American Mathematical Society*, 82 :711–712, 1976.

[And03] Christian Andersson. Register allocation by optimal graph coloring. In *Compiler Construction (CC)*, pages 33–45, 2003.

[App98a] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[App98b] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4) :17–20, 1998.

[AVL62] G M Adelson-Velskii and E M Landis. An algorithm for the organization of information. *Soviet Math. Doklady*, (3) :1259–1263, 1962.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88 : Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art : The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions On Programming Languages and Systems (TOPLAS)*, 16(3) :428 – 455, 1994.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *PLDI '97 : Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 287–295, New York, NY, USA, 1997. ACM.

[BDGR05] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under ssa. Technical report, ENS Lyon, Research report n° 2005-33, 2005.

[BDGR06] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation : What does the np-completeness proof of chaitin et al. really prove ? or revisiting register allocation : why and how ? In *LCPC '06 : Proceedings of the 19th international workshop on languages and compilers for parallel computing*, volume 4382, pages 283–298. Springer Berlin / Heidelberg, nov 2006.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Formal Methods (FM) 2006*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, 2006.

[BDR07a] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *Int. symposium on code generation and optimization (CGO'07)*, San Jose, USA, mar 2007. IEEE Computer Society Press. Best paper award.

[BDR07b] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *Acm sigplan/sigbed conference on languages, compilers, and tools for embedded systems (LCTES'07)*, San Diego, USA, jun 2007.

[BDR08] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES '08 : Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 147–156, New York, NY, USA, 2008. ACM.

[BFPR06] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions : A practical tool for the Coq proof assistant. In *International Symposium on Functional and Logic Programming (FLOPS)*, pages 114–129, 2006.

[BGG+07] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC'06 : Proceedings of the 19th international conference on Languages and compilers for parallel computing*, pages 267–282, Berlin, Heidelberg, 2007. Springer-Verlag.

[BN02] Gertrud Bauer and Tobias Nipkow. The 5 colour theorem in Isabelle/Isar. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2410 of *LNCS*, pages 67–82, 2002.

[Bou08] Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, Université de Lyon-Ecole Normale Supérieure de Lyon, December 2008.

[BPM05] Macbeth J. Brisk P., Dabiri F. and Sarrafzadeh M. Polynomial time graph coloring register allocation. *14th Internat. Workshop on Logic and Synthesis*, 2005.

[BR09] Sandrine Blazy and Benoit Robillard. Live-range unsplitting for faster optimal coalescing. *SIGPLAN Not.*, 44(7) :70–79, 2009.

[BRA10] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010.

[Bri92] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.

[BRS08] Sandrine Blazy, Benoît Robillard, and Eric Soutif. Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe. *JFLA 2008, Étretat, France*, 2008.

[BVI07] Philip Brisk, Ajay Kumar Verma, and Paolo Ienne. An optimistic and conservative register assignment heuristic for chordal graphs. In *CASES '07 : Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 209–217, New York, NY, USA, 2007. ACM.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6 :47–57, 1981.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *symposium on Principles Of Programming Languages (POPL)*, pages 238–252, 1977.

[CDE05] Keith Cooper, Anshuman Dasgupta, and Jason Eckhardt. Revisiting graph coloring register allocation : A study of the chaitin-briggs and callahan-koblenz algorithms. In *IN PROC. OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC¿05*, 2005.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, 1991.

[CH84] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *SIGPLAN '84 : Proceedings of the 1984 SIGPLAN symposium on Compiler construction (CC)*, pages 222–232, New York, NY, USA, 1984. ACM.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[Cha82] G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction (CC)*, 17(6) :98 – 105, 1982.

[Che03] Jing-Chao Chen. Dijkstra's shortest path algorithm. *Journal of Formalized Mathematics*, 15, 2003.

[Cho94] Ching-Tsun Chou. A formal theory of undirected graphs in higher-order logic. In *Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 144–157, 1994.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Not.*, 26(6) :192–203, 1991.

[Cla08] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[Com] Projet CompCert. `http://compcert.inria.fr/`.

[Coq] Coq. The coq proof assistant, `http://coq.inria.fr`.

[CS98] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98 : Proc. of the 7th Int. Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[Dar09] Zaynah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Université Paris Diderot, July 2009.

[Dav03] Maulik A. Dave. Compiler verification : a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6) :2–2, 2003.

[Del00] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 85–95, Reunion Island (France), November 2000. Springer.

[FCL00] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *J. Algorithms*, 37(1) :37–65, 2000.

[FG69] Delbert R. Fulkerson and O. A. Gross. Incidence matrices, interval graphs, and seriation in archaeology. *Pacific Journal of Mathematics*, 28 :565–570, 1969. `http://projecteuclid.org/Dienst/UI/1.0/Summarize/euclid.pjm/1102995572`.

[Fil] Jean-Christophe Filliâtre. Why : A multi-language multi-prover verification tool.

[Fle90] E. Fleury. Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions. Rapport de Stage, jul 1990.

[FW02] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35 : Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions On Programming Languages and Systems (TOPLAS)*, 18(3) :300–324, 1996.

[Gav72] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 :180–187, 1972.

[GGP03] Jia Guo, Maria Jesus Garzaran, and David Padua. The power of belady's algorithm in register allocation for long basic blocks. In *The 16th International Workshop on Languages and Compilers for Parallel Computing*, 2003.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction, CC'07, Braga, Portugal*, volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[Gon05] G Gonthier. A computer-checked proof of the four colour theorem. *Microsoft Research*, (57), 2005.

[Gon08] Georges Gonthier. Formal proof – the four-color theorem. *Notices of the American Mathematical Society*, 55(11) :1382–1393, December 2008.

[Gro06] Common Criteria Working Groups. Common criteria for information technology security evaluation, `http://www.commoncriteriaportal.org/`, 2006.

[GSO94] Rajiv Gupta, Mary Lou Soffa, and Denise Ombres. Efficient register allocation via coloring using clique separators. *ACM Transactions on Programming Languages and Systems*, 16(3) :370–386, 1994.

[GT04] Loukas Georgiadis and Robert E. Tarjan. Finding dominators revisited : extended abstract. In *SODA '04 : Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 869–878, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8) :929–965, 1996.

[Hac05] Sebastian Hack. Interference Graphs of Programs in SSA Form. Technical Report 2005-15, Universität Karlsruhe, June 2005.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA form.* PhD thesis, Universität Karlsruhe, 2007.

[Hai05] Max Hailperin. Comparing conservative coalescing criteria. *ACM Transactions on Programming Languages and Systems*, 27(3) :571–582, 2005.

[HCS06] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *In Static Analysis, 13th Int. Symp., SAS 2006*, pages 281–300. Springer, 2006.

[HG06] Sebastian Hack and Gerhard Goos. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, 98(4) :150–155, 2006.

[HG08] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. *SIGPLAN Not.*, 43(6) :227–237, 2008.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *In Compiler Construction (CC) 2006, volume 3923 of LNCS*. Springer Verlag, 2006.

[Hob08] Aquinas Hobor. Oracle semantics for concurrent separation logic. In *In Proc. European Symp. on Programming (ESOP 2008*, 2008.

[JRA03] Dominique Méry Jean-Raymond Abrial, Dominique Cansell. Formal derivation of spanning tree algorithms. In *ZB 2003*, volume 2651 of *LNCS*, pages 627–628, 2003.

[Kem79] A. B. Kempe. On the geographical problem of the four colors. *American Journal of Mathematics*, 2 :193–200, 1879.

[KG05] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO '05 : Proceedings of the international symposium on Code generation and optimization*, pages 269–280, Washington, DC, USA, 2005. IEEE Computer Society.

[KG06] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI '06 : Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 204–215, New York, NY, USA, 2006. ACM.

[KG09] David Koes and Seth Copen Goldstein. Register allocation deconstructed. In *SCOPES '09 : Proceedings of the 12th international workshop on Software & compilers for embedded systems*, 2009.

[KH93] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *PLDI'93*, pages 268–277, 1993.

[KMM00] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided reasoning : ACL2 case studies.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, 2009.

[Let04] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, July 2004.

[LG97] Guei-Yuan Lueh and Thomas Gross. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22 :2000, 1997.

[LG07] Allen Leung and Lal George. A New MLRISC Register Allocator. 2007.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1) :121–141, 1979.

[Muz93] Michal Muzalewski. An outline of pc mizar, 1993.

[MZ05] J. Strother Moore and Qiang Zhang. Proof pearl : Dijkstra's shortest path algorithm verified with ACL2. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 373–384, 2005.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5) :83–94, 2000.

[NPP07] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis, 14th Int. Symp., SAS 2007, August, 2007, Proc.*, volume 4634 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2007.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Oho04] Atsushi Ohori. Register allocation by proof transformation. *Science Computer Programming*, 50(1-3) :161–187, 2004.

[ONI+10] Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Coloring-based coalescing for graph coloring register allocation. In *CGO '10 : Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 160–169, New York, NY, USA, 2010. ACM.

[Par95] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. PhD thesis, Ecole Normale Supérieure de Lyon, jan 1995.

[PEK97] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc : A system for fast, flexible, and high-level dynamic code generation. In *In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121. ACM Press, 1997.

[PM89] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA*, pages 328–345, 1993.

[PM98] J. Park and S.-M. Moon. Optimistic register coalescing. In *PACT '98 : Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 196, Washington, DC, USA, 1998. IEEE Computer Society.

[PM04] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Transactions On Programming Languages and Systems (TOPLAS)*, 26(4) :735–765, 2004.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. *3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, 3780 :315–329, 2005.

[PP10] Fernando Magno Quintão Pereira and Jens Palsberg. Punctual coalescing. In *Compiler Construction (CC)*, pages 165–184, 2010.

[PS99a] Frank Pfenning and Carsten Schurmann. System description : Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16*, pages 202–206. Springer-Verlag LNAI, 1999.

[PS99b] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5) :895–913, 1999.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS '98*, volume 1384 of *LNCS*, pages 151–166, 1998.

[QP08] Fernando Magno Quint ao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI '08 : Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.

[QS08] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of conurrent systems in cesar. In *25 Years of Model Checking*, pages 216–230, 2008.

[RL10] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *Compiler Construction (CC)*, pages 224–243, 2010.

[Rob07] Benoit Robillard. Vérification formelle d'un algorithme d'allocation de registres. Mémoire de fin d'étude ENSIIE/Master MOCS (CNAM), 2007.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3) :309–322, 1986.

[RTC09] RTCA. La norme do178b, 2009.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88 : Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.

[Sam75] H. Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code*. PhD thesis, Stanford University, 1975.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended linear scan : an alternate foundation for global register allocation. In *CC'07 : Proceedings of the 16th international conference on Compiler construction*, pages 141–155, Berlin, Heidelberg, 2007. Springer-Verlag.

[SE02] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. *SIGPLAN Not.*, 37(7) :139–148, 2002.

[Soz08] M. Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris-Sud, December 2008.

[SRH04] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04 : Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI '98 : Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 142–151, New York, NY, USA, 1998. ACM.

[Tri09] J.B. Tristan. *Formal Verification of Translation Validators*. PhD thesis, Université Paris Diderot, November 2009.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, 1984.

[Ty10] M. Ty. An analysis of iterated register coalescing and its implementation in a formally verified compiler. Internship report, jul 2010.

[Wes00] Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.

[Wie06] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[WM05] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05 : Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM.

[YG87] Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2) :133–137, 1987.

[YNHT95] Mitsuharu Yamamoto, Shin-ya Nishizaki, Masami Hagiya, and Yozo Toda. Formalization of planar graphs. In *Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 369–384, 1995.

[YTH+98] Mitsuharu Yamamoto, Koichi Takahashi, Masami Hagiya, Shin-ya Nishizaki, and Tetsuo Tamai. Formalization of graph search algorithms and its applications. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, pages 479–496, 1998.

Annexe A

Le système Coq

Le système Coq est un assistant à la preuve interactif qui permet de décrire à la fois des programmes et des propriétés mathématiques. Cette double fonction est en fait le reflet de l'isomorphisme de Curry-Howard qui établit le lien existant entre logique intuitionniste et calcul.

Ce système peut-être utilisé à deux fins, souvent confondues mais pourtant très différentes : soit en tant qu'environnement de preuve de théorème, soit en tant qu'environnement de développement. Dans le premier cas, le but est de prouver un théorème. Dans le second cas, un programme est spécifié, une implantation de celui-ci est écrite et la preuve de correction de l'implantation vis-à-vis de la spécification est démontrée. C'est cette dernière utilisation que nous faisons du système Coq . Dans cette partie, nous décrivons les principaux rouages de ce système qui nous sont nécessaires pour la formalisation de nos algorithmes d'allocation de registres.

A.1 Le lambda-calcul

Le lambda-calcul, introduit en 1930 par Church, est aujourd'hui le fondement de la programmation fonctionnelle. Les éléments du lambda-calcul, appelés lambda-termes ou, plus simplement, termes, peuvent être construits de trois façons, comme le suggère la définition suivante des lambda-termes :

$$\Gamma, t, u := \ x \mid \lambda x.t \mid t\ u$$

Un lambda-terme est donc soit une variable x prise parmi un ensemble infini, soit l'abstraction $\lambda x.t$, assimilable à l'association $x \mapsto t$, soit enfin l'application de la fonction t à son argument u, notée $t\ u$.

La fonction f qui à x associe $x+2$ (où $f : x \mapsto x+2$) s'exprime dans le lambda-calcul sous la forme $\lambda x.x+2$. L'application de cette fonction à l'entier 1 se note $(\lambda x.x+2)\ 1$. Cette dernière forme peut être réécrite en 3 grâce à la règle de calcul unique du lambda-calcul, la β-réduction :

$$(\lambda x.t)\ u \rightarrow_\beta t[u/x]$$

où $t[u/x]$ désigne la substitution de toutes les occurrences de la variable x dans t par u.

Ainsi, $(\lambda x.x + 2)\ 1$ se β-réduit en $1 + 2$.

Le lambda-calcul est donc un langage de programmation permettant, entre autres, d'encoder les fonctions récursives. Cependant, celui-ci permet d'écrire des termes qui se β-réduisent en eux-mêmes, ce qui implique que le calcul ne termine pas. Afin de pallier ce désagrément, Church introduit le typage : une fonction n'est appliquée à un terme que si leurs types sont compatibles, *i.e.* que si cette application a un sens. Les types sont définis via deux règles : les types de base et le type flèche $\tau_1 \rightarrow \tau_2$ où τ_1 et τ_2 sont des types. Les règles de construction des termes sont alors les suivantes :

- la création de variables. Si $x : \alpha \in \Gamma$ alors $\Gamma \vdash x : \alpha$
- la règle d'abstraction. Si $\Gamma, x : \alpha \vdash t \in \beta$ alors $\Gamma \vdash \lambda x, t : \alpha \rightarrow \beta$
- la règle d'application. Si $\Gamma \vdash t : \alpha \rightarrow \beta$ et $\Gamma \vdash u : \alpha$ alors $\Gamma \vdash tu : \beta$

A.2 Programmes et preuves

L'isomorphisme de Curry-Howard établit la correspondance entre les termes du lambda-calcul typé et les preuves en déduction naturelle dans la logique intuitionniste. Par exemple, une preuve de $A \Rightarrow A$ revient à écrire une fonction de type $\lambda x : A.x$ qui a pour type $A \rightarrow A$.

Intuitivement, toute preuve doit être accompagnée d'un algorithme permettant de construire cette preuve. Ceci implique en particulier l'absence de la propriété $A \vee \neg A$, bien connue sous le non de tiers exclu, en logique intuitionniste. En effet, prouver A ne permet pas de construire une preuve de $\neg A$, et réciproquement. Par exemple, dans les langages de programmation les plus couramment utilisés, l'égalité de deux éléments d'un même type est considérée comme étant toujours décidable. Ainsi il est fréquent de tester des égalités du type **if** `i` = `j`. Dans Coq , l'égalité est un prédicat binaire et non une fonction. Pour pouvoir tester l'égalité il faut construire un programme capable de décider si deux éléments sont égaux ou non, qui renvoie une preuve de l'égalité ou une preuve de la diségalité.

Le système Coq fût bâti sur le calcul des constructions [CH88], un lambda-calcul typé d'ordre supérieur. D'un point de vue logique, le calcul des constructions étend l'isomorphisme de Curry-Howard aux preuves dans le calcul des prédicats, ce qui permet de disposer d'une logique des prédicats d'ordre supérieur. Le calcul des constructions fût par la suite enrichi pour devenir le calcul des constructions inductives [PM93] sur lequel repose aujourd'hui le système Coq . Le noyau de Coq est simplement un typeur pour le calcul des constructions inductives puisque l'isomorphisme de Curry-Howard assure qu'une preuve est correcte si le terme correspondant est bien typé.

A.3 La construction Inductive

Le système Coq permet de facilement définir des types algébriques, comme le type `option` qui permet d'encoder les fonctions partielles. Un élément de type `option A` est soit

`Some a` où `a` est de type `A`, soit `None`.

```
Inductive option (A : Type) : Set :=
|None : option A
|Some : A → option A.
```

`None` et `Some` sont appelés constructeurs du type `option`.

Il est également possible de définir des types inductifs, comme les listes polymorphes. La bibliothèque standard de Coq spécifie qu'une liste polymorphe est soit la liste vide `nil`, soit une liste `a :: l` obtenue par ajout d'un élément `a` de type `A` en tête de la liste `l` de type `list A`.

```
Inductive list (A : Type) : Type :=
|nil : list A
|cons : A → list A → list A.
```

La construction **Inductive** permet aussi de définir des propriétés, de façon parfaitement analogue aux structures de données. Cette fois le type de ce qui est construit n'appartient plus à **Set**, que l'on peut considérer comme l'espace de calcul, mais à **Prop**, que l'on peut considérer comme l'espace des propositions. Voici par exemple le prédicat qui statue qu'un élément appartient à une liste.

```
Inductive In (A : Type) : Prop :=
|in_head : ∀ (a : A), In a (a :: l)
|in_tail : ∀ (a b : A) (l : list A) , In a l ⇒ In a (b :: l)
```

Le mot-clé **Inductive** assure que le type construit est le plus petit point fixe de la relation définie par les deux constructeurs. Ainsi, tout élément de type `list A` est construit à partir de l'un de ses deux constructeurs. Cette propriété autorise le raisonnement par induction, pièce maîtresse de la preuve au sein du système Coq . Un principe d'induction, permettant de raisonner sur un type inductif, est automatiquement généré dès la définition de ce type. Par exemple, le lemme d'induction sur les listes polymorphes indique que si une propriété `P` est prouvable à la fois pour la liste vide `nil` et pour toute liste `a :: l` avec `a` un élément de type `A` et `l` une liste de type `list A` alors `P` est vraie pour toute liste de type `list A`.

```
∀ (A : Type) (P : list A ⇒ Prop),
       P nil ⇒
       (∀ (a : A) (l : list A), P l ⇒ P (a :: l)) ⇒
       ∀ l : list A, P l
```

A.4 Les constructions Definition et Fixpoint

Le langage purement fonctionnel de programmation du système Coq permet de manipuler aisément ces données simples. La première façon possible de définir des fonctions est la construction **Definition**. Voici par exemple la fonction qui à une liste associe sa queue, *i.e.* la liste privée de sa tête. Si la liste est vide, la fonction échoue et renvoie `None`.

```
Definition tail (A : Type) (l : list A) : option (list A) :=
match l with
| nil → None
| _ :: q → Some q
end.
```

Il n'est pas surprenant d'observer l'utilisation du filtrage par motif puisque la définition du type `nat` est inductive. Notons que la construction **Definition** est loin de se limiter à cette utilisation : elle permet plus généralement d'associer à un terme un nom.

Naturellement, les fonctions récursives, comme par exemple une fonction de construction de liste contenant les n premiers entiers naturels triés par ordre décroissant, peuvent facilement être exprimées dans ce langage.

```
Fixpoint firstn (n : nat) {struct n} : list nat :=
match n with
| 0 → nil
| S p → n :: (firstn p)
end.
```

Comme le suggère son nom, la construction **Fixpoint** repose sur un calcul de plus petit point fixe. Celle-ci peut être vue comme le pendant calculatoire de la construction **Inductive** qui, elle, ne l'est pas. L'en tête de la fonction `firstn` contient, en outre de l'entier naturel `n`, l'indication {**struct** `n`}. La construction **Fixpoint** requiert en effet que l'un de ses arguments, ici `l`, décroisse structurellement[1] à chacun des appels récursifs afin d'assurer la terminaison de la fonction.

A.5 Récursivité non structurelle

La majeure partie des fonctions récursives implantées dans le système Coq décroissent structurellement. Néanmoins, il arrive que cette propriété ne soit pas respectée soit à cause de la structure de données trop complexe, soit à cause de la complexité de l'algorithme.

Une autre solution pour prouver qu'une fonction récursive f termine consiste à prouver qu'il existe un ordre bien fondé $<_\sigma$ sur les arguments de la fonction qui décroît à chaque appel récursif. Autrement dit, il suffit que si $f(a_1, \ldots, a_n)$ fait appel à $f(a'_1, \ldots, a'_n)$, alors $(a'_1, \ldots, a'_n) <_\sigma (a_1, \ldots, a_n)$. Une application courante de cette propriété est la preuve de terminaison fondée sur la décroissance d'une mesure. Une mesure est une fonction m qui aux arguments de f associe un entier naturel. Prouver que $m(a'_1, \ldots, a'_n) < m(a_1, \ldots, a_n)$ suffit à établir la terminaison de f.

Des études récentes ont permis d'ajouter des constructions, comme **Function** [BFPR06] ou **Program** [Soz08] permettant de gérer plus aisément ces principes de terminaison. Nous utiliserons uniquement la construction **Function** dans ce manuscrit.

L'algorithme `quicksort` de tri de liste peut ainsi être implanté de la façon suivante.

```
Function quicksort (l : list A) {measure length l}: list A :=
```

[1] Un argument a d'une fonction f décroît structurellement si tout appel récursif de fa fait appel à fb, où b est un sous-terme strict de a.

```
match l with
          | nil → nil
          | h::t → let (less,more) := partition h t in
                       quicksort(less) ++ h::quicksort(more)
end.
```

où la `partition h t` est la partition de la liste `t` en deux listes, la première contenant les éléments de `t` plus petits que `h` et la seconde contenant les éléments de `t` plus grands que `h`.

Comme pour la construction **Fixpoint**, le critère de terminaison est précisé entre accolades. Il s'agit ici d'une mesure, la longueur de la liste. La définition d'une fonction par la construction **Function** n'est complète qu'à partir du moment où la décroissance de la mesure lors de chaque appel récursif est prouvée.

Par ailleurs, la construction **Function** donne accès à un principe d'induction, nommé **functional induction**, pour raisonner en fonction des branchements empruntés lors du filtrage. Ce principe d'induction facilite le raisonnement sur les fonctions définies par cette construction, si bien qu'elle est parfois utilisée en lieu et place de la construction **Fixpoint** dans le seul but de disposer de ce principe d'induction.

A.6 Données complexes et types dépendants

Certaines données sont plus complexes que celles décrites jusqu'ici. Un graphe, par exemple, est une structure de données englobant à la fois des sommets, des arêtes et des propriétés. Ces données complexes peuvent être décrites comme des enregistrements via la construction **Record**. Les listes d'entiers supérieurs à 7 peuvent ainsi être définies comme suit.

```
Record seven_lists := Make_seven_lists {
  l : list nat ;
  parite : ∀ (x : nat), In x l ⇒ x > 7
}
```

Pour construire un **Record** il faut non seulement fournir la liste `l`, mais aussi la preuve que tout élément de `l` est bien supérieur à 7. Construire cette preuve n'est pas toujours simple, comme nous le verrons au cours de la présentation de la vérification formelle de l'IRC, car le type de `parite` dépend de la valeur de la liste `l`. Une structure dont le type dépend d'une valeur, comme c'est ici le cas, est dite de type dépendant. Ces types dépendants s'avèrent plus naturels pour raisonner sur des structures de données complexes, telles que les graphes par exemple, pour lesquelles la définition de propriétés est nécessaire, mais ont tendance à entraîner des difficultés supplémentaires. Par exemple, si l'on veut ajouter un élément à une liste de type `seven_lists` il faut prouver que la nouvelle liste ne contient que des éléments de valeur supérieure à 7. Il faut donc raisonner sur `l` pour créer `parite`.

A.7 Preuves

Une fois les programmes spécifiés et implantés, reste à lier ces deux acteurs par la preuve. Dans le système Coq , les preuves sont facilitées par un langage de tactiques [Del00]. Ces tactiques génèrent les termes de preuve correspondant aux opérations logiques qu'elles opèrent. A la fin d'une preuve, les termes ainsi générés sont combinés afin de former le terme qui doit être soumis au typeur pour vérifier la validité de la preuve.

Présentons un exemple afin d'illustrer comment fonctionne ce langage de tactiques. Le théorème que nous allons prouver est le suivant :

```
Theorem lower_in_first : ∀ n p,
                         p < n ⇒
                         In p (firstn n).
```

Nous montrons ce théorème par induction. Aussi, les premières lignes du script de preuve sont les suivantes.

```
Proof.
induction n; intros.
```

Le mot-clé **Proof** marque le début de la preuve. La tactique **induction** `n` indique que le résultat va être prouvé par induction sur `n`. Les entiers naturels étant représentés à la Peano, deux sous-buts sont générés, l'un où `n` est égal à 0, l'autre ou il est le successeur d'un autre entier. La tactique **intros** introduit simplement les variables dans le contexte.

Le premier des deux sous-buts est le suivant :

```
p : nat
H : p < 0
============================ (1/2)
In p (firstn 0)
```

L'hypothèse `H` est contradictoire. Nous appliquons une tactique dite d'inversion qui va créer un sous-but par constructeur possible de l'hypothèse `H`. Ici, il n'existe aucun constructeur capable de dériver `H` puisque `H` est absurde. Ainsi, le but est résolu.

```
inversion H.
```

Reste le second sous-but à traiter :

```
n : nat
IHn : ∀ p : nat, p < n ⇒ In p (firstn n)
p : nat
H : p < S n
============================ (1/1)
In p (firstn (S n))
```

Après simplification, le but devient :

```
S n = p ∨ In p (firstn n)
```

L'hypothèse `H` indique que l'égalité `S n = p` est clairement fausse, ainsi nous devons prouver `In p (firstn n)`. Nous allons une fois de plus utiliser la tactique d'inversion sur l'hypothèse `H` afin de créer deux sous-buts.

```
inversion H.
```

L'hypothèse `H` est renforcée en `p = n` dans le premier but, et en `S p ≤ n` dans le second.

```
n : nat
IHn : ∀ p : nat, p < n ⇒ In p (firstn n)
p : nat
H : p < S n
H1 : p = n
═══════════════════════════════════ (1/2)
In n (firstn n)
```

Ce résultat est facile à prouver. Il faut néanmoins considérer deux sous-buts en fonction de l'entier `n`. Nous utilisons une fois encore l'induction même si l'hypothèse d'induction ne nous est pas ici utile.

```
induction n; simpl; auto.
```

Reste le dernier sous-but :

```
n : nat
IHn : ∀ p : nat, p < n ⇒ In p (firstn n)
p : nat
H : p < S n
m : nat
H1 : S p ≤ n
H0 : m = n
═══════════════════════════════════ (1/1)
In p (firstn n)
```

L'hypothèse d'induction peut ici être appliquée. Il reste alors à prouver `p < n` ce qui découle directement des hypothèses. La fin du script de preuve est donc :

```
apply IHn; auto.
Qed.
```

Le mot-clé **Qed** clôt la preuve et appelle la vérification du terme de preuve par le typeur. Le terme correspondant à cette preuve tient en environ une page alors que le script de tactiques tient en seulement huit lignes.

A.8 Architecture des développements

Nous utilisons dans cette thèse deux des structures Coq permettant d'architecturer les développements : les sections et les modules. Ceux-ci répondent à des besoins différents que nous décrivons rapidement ici.

A.8.1 Sections

Les sections **Section** permettent de définir du code paramétrable par de nombreuses variables, qui peuvent aller du type de données aux preuves en passant par les fonctions. Ces paramètres sont définis avec le mot-clé **Variable** pour les variables et avec le mot clé **Hypothesis** pour les preuves. Par exemple, il est possible de définir un algorithme de tri polymorphe comme suit.

```
Variable A : Type.
Variable A_eq_dec : ∀ a b : A, {a = b}+{¬a = b}.
Variable Order : A → A → Prop.

Hypothesis order_antisym :
                ∀ a b : A, Order a b ⇒ Order b a ⇒ a = b.
Hypothesis order_trans :
                ∀ a b c : A, Order a b ⇒ Order b c ⇒ Order a c.
Hypothesis order_dec : ∀ x y : A, {Order x y}+{¬Order x y}.
Hypothesis order_total : ∀ a b : A, {Order a b}+{Order b a}.

Fixpoint insert (a : A) (l : list A) :=
match l with
| nil → [a]
| h :: t → match order_dec a h with
        | left p → h :: (insert a t)
        | right p → a :: l
        end
end.

Fixpoint sort (l : list A) :=
match l with
| nil → nil
| h :: t → insert h (sort t)
end.
```

L'algorithme `sort` peut être prouvé correct rien qu'avec les hypothèses décrites dans l'implantation. On peut ainsi définir un algorithme de tri sur mesure en définissant `A`, `A_eq_dec` et `Order`. Les paramètres définis avec le mot-clé **Hypothesis** ne seront demandés que si des lemmes utilisant ces propriétés, typiquement la preuve de correction de l'algorithme de tri, sont utilisés. On retrouve ici l'importance de la décidabilité de propriétés. `order_dec`, par exemple, n'est pas un prédicat, mais une fonction qui renvoie une preuve de `Order x y` ou une preuve de `¬Order x y`. Ces réponses sont encapsulées dans des constructeurs **left** et **right**. Ainsi, si le filtrage détermine que `order_dec a h` vaut **left** `p`, alors `p` est une preuve de la propriété `Order x y`.

A.8.2 Modules

Les modules ont un rôle analogue à celui que l'on peut rencontrer dans n'importe quel langage de programmation, à ceci près qu'un module peut contenir des preuves et non seulement des fonctions.

Le système Coq incorpore également une construction `Module` **Type** permettant de définir des types de modules. Il est par exemple possible de définir le type de modules des types ordonnés `OrderedType` (cf. `http://coq.inria.fr`. Les modules ainsi typés peuvent être utilisés comme paramètres d'autres modules. La bibliothèque sur les ensembles ordonnés `FSets` de la bibliothèque standard prend par exemple un module représentant un ensemble ordonné en paramètre. Ces types de modules représentent uniquement les interfaces des modules. Les variables y sont définis via le mot clé **Parameter** et les preuves via le mot-clé **Axiom**. Un petit exemple de type de module est celui du type des types à égalité décidable.

```
Module Type DecidableType.

  Parameter t : Type.

  Parameter eq : t ⇒ t ⇒ Prop.

  Axiom eq_refl : ∀ x : t, eq x x.
  Axiom eq_sym : ∀ x y : t, eq x y ⇒ eq y x.
  Axiom eq_trans : ∀ x y z : t, eq x y ⇒ eq y z ⇒ eq x z.

  Parameter eq_dec : ∀ x y : t, { eq x y } + { ¬ eq x y }.

End DecidableType.
```

Un module qui implante l'interface de module de type décidable est par exemple le module des entiers à la Peano.

```
Module Nat_as_DT <: DecidableType.

  Definition t := nat.

  Definition eq := @eq nat.
  Definition eq_refl := @refl_equal t.
  Definition eq_sym := @sym_eq t.
  Definition eq_trans := @trans_eq t.

  Definition eq_dec := eq_nat_dec.

End Nat_as_DT.
```

L'utilisation de modules introduit une notation pointée afin de désigner les éléments de ce module. Par exemple, si l'on veut utiliser le lemme d'égalité décidable `eq_dec` du module `Nat_as_DT`, il faut y référer en tant que `Nat_as_DT.eq_dec`.

Index