M2 MPRI - Rapport de stage

Un analyseur flots de données paramétré par un ordre d'itération pour le compilateur formellement vérifié CompCert

Roméo La Spina
Stage encadré par Sandrine Blazy et Delphine Demange
Equipe CELTIQUE/EPICURE (IRISA)

Le contexte général

Dans certains domaines, et notamment au sein des systèmes embarqués critiques, il est nécessaire de s'assurer que les programmes s'exécutent correctement. C'est pourquoi il est important d'utiliser un compilateur *vérifié* lorsqu'on souhaite compiler un programme dans ces domaines. Un compilateur vérifié est un compilateur accompagné d'une preuve qui montre que ce dernier n'introduit pas de bogue dans les programmes. Une référence du domaine est le compilateur CompCert du langage C [12, 13]. CompCert est entièrement écrit et prouvé à l'aide de l'assistant de preuve Coq, puis son code est extrait vers du code OCaml pouvant être exécuté en dehors de Coq. Ainsi, CompCert peut être exécuté comme n'importe quel autre compilateur. Les performances de CompCert en termes de temps d'exécution d'un programme compilé sont comparables à celles obtenues avec l'option `-O1` du compilateur de référence GCC (non vérifié). [12, 13]

Le problème étudié

Plusieurs optimisations de CompCert utilisent une analyse flot de données [13], qui résout un ensemble d'équations grâce à un algorithme dû à Kildall [10]. Ces équations sont à valeurs dans une structure mathématique remarquable appelée *treillis*. L'algorithme de Kildall a l'avantage d'être extrêmement simple, tant du côté de son implémentation que du côté de sa preuve de correction. Cependant, selon la taille du treillis utilisé pour faire l'analyse, la convergence de l'algorithme peut nécessiter un temps extrêmement long, voire ne pas être garantie. Un autre algorithme, développé par Bourdoncle [3], qui paramètre l'analyse par un ordre d'itération calculé au préalable, permet l'utilisation efficace d'opérateurs d'élargissement (*widening* en anglais). Les opérateurs d'élargissement sont une solution fréquemment utilisée en interprétation abstraite pour accélérer la convergence de solveurs d'équations, éventuellement au détriment de la précision de la solution calculée [7]. Bien que CompCert n'utilise pas pour l'instant de treillis suffisamment hauts pour qu'on puisse tirer un bénéfice de cette possibilité, certains compilateurs non vérifiés de l'état de l'art appliquent des optimisations qui reposent sur des analyses utilisant des treillis très hauts, par exemple l'analyse d'alias [9]. On pourrait intégrer dans le futur de telles analyses au compilateur CompCert afin d'améliorer les optimisations existantes. Bien que l'algorithme de Bourdoncle ait déjà fait l'objet d'études théoriques et qu'il fasse partie des algorithmes de référence dans le domaine de l'interprétation abstraite dès qu'on manipule des treillis très hauts ou infinis [11, 2], il serait intéressant de pouvoir l'utiliser dans le cadre d'un compilateur, qui plus est vérifié. L'étude de solveurs de points fixes formellement vérifiés a déjà été

faite dans un cadre général [8, 5], mais sur des exemples de solveurs *locaux* qui ont pour objectif de calculer de l'information en un point en particulier le plus précisément possible et en effectuant le moins de calculs possible. Or dans le cadre d'optimisations de compilation, on a besoin de calculer de l'information en tout point du graphe de flot de contrôle. De plus ces solveurs sont génériques et n'exploitent pas les spécificités d'un système d'équations obtenu dans le cadre d'une analyse flot de données.

La contribution proposée

Nous avons choisi d'implémenter l'algorithme de Bourdoncle dans CompCert et de comparer ses performances à celles de l'algorithme de Kildall, à la fois en termes d'efficacité et en termes de précision de l'analyse. Concernant le calcul de l'ordre d'itération, nous avons programmé et prouvé la correction d'une méthode naïve. Par la suite, dans le souci de pouvoir facilement modifier l'implémentation pour la rendre la plus efficace possible, nous nous sommes dans un premier temps tournés vers une approche de type validation *a posteriori*, c'est-à-dire que l'ordre est calculé par une fonction OCaml externe dont le résultat est ensuite vérifié par un validateur prouvé en Coq. On remarque que la validation réussit à chaque fois lorsqu'on compile des programmes de test, par conséquent on pourrait envisager dans le futur d'implémenter et prouver entièrement l'algorithme en Coq. Nous avons ensuite développé des scripts permettant de tester les performances de notre implémentation de manière automatisée.

Les arguments en faveur de sa validité

En compilant des programmes de test en C avec CompCert, on remarque que le temps de compilation n'augmente que légèrement lorsqu'on remplace l'algorithme de Kildall par celui de Bourdoncle, et le résultat de l'analyse est quasiment aussi précis. En conséquence, la contrepartie de la possibilité d'utiliser des opérateurs d'élargissement reste raisonnable. Le bénéfice apporté par Bourdoncle dépend des analyses flot de données considérées : telles qu'elles sont dans CompCert, une simple analyse à l'aide de l'algorithme de Kildall est suffisante. Toutefois dans des évolutions futures du compilateur, l'utilisation de l'algorithme de Bourdoncle pourrait être bénéfique voire nécessaire. De plus, notre implémentation est particulièrement robuste car bien que le calcul de l'ordre d'itération repose sur de la validation *a posteriori*, le calcul du point fixe avec la méthode de Bourdoncle est quant à lui formellement vérifié en Coq, de même que le reste du compilateur. Enfin, en tant qu'implémentation générale et non spécifique à une optimisation en particulier, notre contribution offre de nombreuses perspectives quant à l'évolution du compilateur CompCert.

Le bilan et les perspectives

Dans le futur, il faudrait pouvoir ajouter des opérateurs d'élargissement dans CompCert, puis ajouter de nouvelles optimisations nécessitant des treillis très hauts ou infinis de manière à tirer profit des avantages apportés par l'algorithme de Bourdoncle par rapport à celui de Kildall. On pourrait alors améliorer les passes d'optimisations existantes (par exemple, la propagation des constantes). On pourrait également peaufiner notre implémentation, notamment en prouvant en Coq la partie validée *a posteriori*. La prochaine question à se poser, après avoir ajouté ces améliorations, est : quelle est la meilleure stratégie à appliquer selon le programme et la passe d'optimisation considérée ? On pourrait ainsi jongler avec les différentes stratégies pour améliorer encore les performances du compilateur CompCert.

Table des matières

Remerciements

Je remercie vivement Sandrine Blazy et Delphine Demange, mes encadrantes de stage, qui ont toujours su me guider vers les bonnes intuitions au cours de mes recherches et lors de la rédaction de ce rapport. Nous avons notamment pu avoir des discussions très fructueuses et prometteuses pour une poursuite de ce travail. Merci également à Vincent Laporte, qui m'a encadré pendant mon stage de M1 et qui m'a donné de précieux conseils lors de sa venue à l'IRISA.

Je tiens enfin à remercier sans exception tous les membres de l'équipe Celtique/Epicure de l'IRISA pour leur accueil chaleureux tout au long de mon stage.

1 Introduction

1.1 Contexte du problème

1.1.1 CompCert

Pour s'assurer qu'un programme s'exécute correctement, on dispose déjà d'outils qui permettent de prouver la correction d'un programme à partir de son code, tels que Frama-C [1]. Cependant, comment s'assurer que le compilateur n'introduira pas de bogue dans le programme, c'est-à-dire que le programme cible généré par le compilateur aura toujours un comportement prévu lors de l'écriture du programme source ? En effet, un compilateur est un logiciel très complexe, qui peut bien sûr contenir des erreurs, notamment lors des passes d'optimisation du code. D'où l'intérêt de disposer d'un compilateur formellement vérifié dont la propriété fondamentale est de garantir que l'ensemble des comportements du programme cible est inclus dans l'ensemble des comportements du programme source. Ces comportements sont déterminés par la sémantique des langages source et cible. Une référence dans le domaine est le compilateur CompCert [12] qui supporte un très large sous-ensemble du langage C99, vérifié grâce à l'assistant de preuve Coq.

Lorsqu'on compile un programme avec CompCert, 18 transformations successives sont appliquées au programme, en utilisant 11 langages intermédiaires. CompCert utilise en particulier la représentation intermédiaire RTL, qui permet de représenter un programme sous forme d'un graphe appelé *graphe de flot de contrôle* (GFC) dont les nœuds sont les points du programme et les arêtes permettent de déterminer la structure du programme. Cette représentation intermédiaire est particulièrement adaptée pour pouvoir raisonner sur les propriétés du programme, c'est pourquoi les passes d'optimisation du code sont appliquées après transformation du code vers la représentation RTL. Un schéma très simplifié du fonctionnement de CompCert est disponible en figure 1.

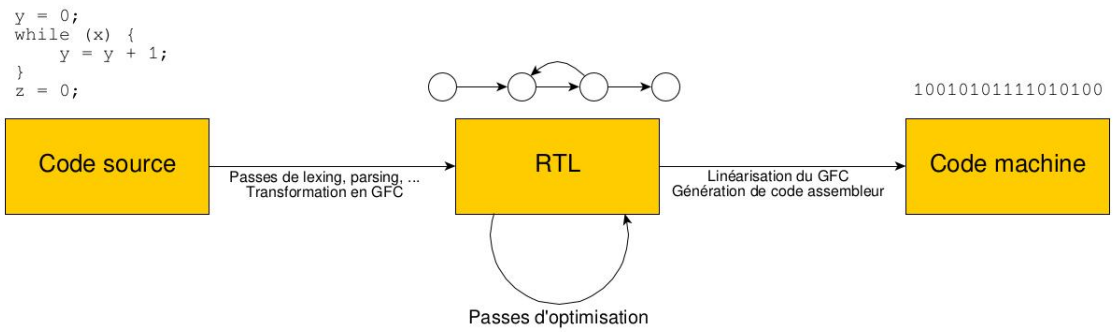


FIGURE 1 – Schéma de fonctionnement de CompCert

1.1.2 Analyse de flot de données

L'analyse de flot de données est une méthode d'analyse statique permettant de calculer de l'information à chaque point du programme (par exemple, l'ensemble des variables dont la valeur est égale à une constante). Une telle analyse peut permettre d'effectuer des optimisations, par exemple la propagation des constantes : on détermine le plus grand ensemble de variables égales à une constante possible à chaque point, puis on optimise le programme en remplaçant chaque référence à ces variables par la valeur constante associée.

Supposons que l'on dispose d'un GFC $G = (N, E)$, avec un nœud d'entrée n_e. On peut alors définir une fonction $\mathsf{succ} : N \to \mathcal{P}(N)$ qui à un nœud n associe l'ensemble des successeurs de n, c'est-à-dire telle que pour tout $n \in N$, $\mathsf{succ}(n) = \{s \in N \mid (n, s) \in E\}$.

L'information est calculée sous forme de valeurs abstraites. Ces valeurs appartiennent à une structure mathématique remarquable appelée *demi-treillis*.

Définition 1.1 (Demi-treillis)**.** Un *demi-treillis* est un ensemble L muni :

- d'un opérateur de borne supérieure $\sqcup$
- d'un plus petit élément $\bot$
- d'une relation d'ordre $\sqsubseteq$ définie de la manière suivante à l'aide de $\sqcup$:

$$\forall x, y \in L, x \sqsubseteq y \iff x \sqcup y = y$$

Dans la suite, on supposera qu'on dispose d'un demi-treillis $(L, \sqsubseteq, \sqcup, \bot)$.

On dispose également d'une fonction de transfert $T : N \times L \to L$ qui permet de calculer l'information à la sortie d'un nœud à partir de l'information à l'entrée d'un nœud. Notre objectif est de calculer un résultat *correct* de l'analyse de flot de données induite par T, c'est-à-dire un post-point fixe pour la fonction de transfert T. En d'autres termes, on doit calculer aval tel que pour tout $n \in N$ et $s \in \mathsf{succ}(n)$, la contrainte $T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s]$ soit satisfaite. Ainsi l'information en sortie d'un nœud sera toujours incluse à l'entrée des successeurs de ce nœud.

1.1.3 Algorithme de Kildall

Pour calculer une telle solution, CompCert utilise un algorithme dû à Kildall [10]. Soit W une liste de nœuds à traiter initialement vide et aval un tableau qui associe initialement $\bot$ à chaque nœud du graphe. Le principe de l'algorithme de Kildall est le suivant :

- Affecter une valeur d'entrée $\mathsf{aval}[n_e]$ au nœud d'entrée n_e
- Calculer l'application de T à $\mathsf{aval}[n_e]$
- Propager le résultat dans aval à tous les successeurs de n_e et ajouter tous les successeurs à la liste W
- Si W n'est pas vide, dépiler un nœud n dans W et répéter les deux dernières étapes en remplaçant n_e par n

Ainsi on parcourt le graphe en propageant l'information de successeur en successeur. Le pseudo-code est détaillé dans l'Algorithme 1.

Théorème 1.1. *La complexité de l'algorithme de Kildall est $O(h(L) \times |N| \times \deg(G))$, où $h(L)$ est la hauteur du demi-treillis L.*

Preuve. On fait au plus autant d'itérations de la boucle **tant que** que d'ajouts de nœuds dans W. Comme on ne peut ajouter un nœud à W qu'en augmentant strictement sa valeur abstraite, et qu'on ne peut augmenter cette valeur qu'au plus $h(L)$ fois, on fait $O(h(L) \times |N|)$ itérations. Comme le nombre de successeurs d'un noeud est borné par le degré de G, on exécute au plus $h \times |N| \times \deg(G)$ le contenu de la sous-boucle **pour chaque**. □

Remarque. Lorsqu'on considère des graphes de flot de contrôle où les nœuds sont des points de programme, le degré est en général au plus 2 : les nœuds aux instructions `if` et `while` ont deux successeurs, les autres nœuds ont au plus un successeur. On pourra donc en général considérer que la complexité de l'algorithme de Kildall est $O(h \times |N|)$.

L'avantage de cet algorithme est qu'il est simple de l'implémenter et de prouver sa correction. A titre d'indication, la preuve de correction de Kildall fournie dans CompCert tient en environ 300 lignes de développement Coq. Le principal inconvénient de l'algorithme de Kildall est que selon le demi-treillis L utilisé, le temps de convergence de l'algorithme (avant qu'une solution ne soit trouvée)

Algorithme 1 : Kildall

Entrées : Une fonction de transfert T, une fonction successeur succ, un nœud d'entrée n_e, une valeur d'entrée v_e, un demi-treillis $(L, \sqsubseteq, \sqcup, \bot)$

Output : Une solution aval sous forme d'un tableau associant une valeur de L à chaque nœud

```
1  W ← {n_e}
2  aval ← le mapping qui associe chaque nœud à ⊥
3  aval[n_e] ← v_e
4  tant que W ≠ ∅ faire
5  |  extraire un nœud n de W
6  |  out ← T(n, aval[n])
7  |  pour chaque s ∈ succ(n) faire
8  |  |  in ← aval[s] ⊔ out
9  |  |  si in ≠ aval[s] alors
10 |  |  |  aval[s] ← in
11 |  |  |  W ← W ∪ {s}
12 |  |  fin si
13 |  fin pour chaque
14 fin tq
```

peut être très long. Dans le cas où on utiliserait un demi-treillis de hauteur infinie, la convergence pourrait même ne pas être garantie. Or, pour pouvoir effectuer des optimisations plus évoluées comme on peut trouver dans les compilateurs non vérifiés de l'état de l'art, on a parfois besoin d'analyses flot de données utilisant de tels demi-treillis. L'analyse d'alias en est un exemple. Supposons qu'on veuille savoir sur quelles adresses mémoire peuvent pointer les variables d'un programme. Naturellement, si P est l'ensemble des variables et V l'ensemble des adresses, on prendra le demi-treillis $(\mathcal{P}(P \times V), \subseteq)$: pour chaque nœud, on cherche à calculer l'ensemble des couples (x, a) tels que le pointeur x peut pointer vers l'adresse a [9]. Ce demi-treillis a pour hauteur le nombre d'éléments de $P \times V$, c'est-à-dire qu'il est de taille proportionnelle au nombre d'adresses possibles. On pourrait imaginer que pour un nœud du graphe, on ajoute à chaque visite un couple (x, a) à l'ensemble calculé pour ce noeud. Même si x est toujours le même pointeur, on devrait alors faire autant d'itérations que d'adresses mémoire possibles ! Le demi-treillis est fini, mais il n'est pas envisageable d'utiliser notre solveur de point fixe pour faire une analyse basée sur ce demi-treillis.

Une solution classique en interprétation abstraite pour pallier à ce problème est d'utiliser des opérateurs d'*élargissement* [7]. Le principe d'un tel opérateur ∇ est de renvoyer une valeur supérieure à ce que renverrait la borne supérieure $\sqcup$ du treillis, ce qui a pour effet d'accélérer la convergence (le post-point fixe est plus vite atteint) au prix d'une certaine perte de précision.

La propriété principale d'un opérateur d'élargissement (la convergence) peut être définie comme suit :

Définition 1.2 (Chaîne strictement augmentante)**.** Une *chaîne strictement augmentante* d'un opérateur ∇ est une suite (x_n), paramétrée par une autre suite (y_n), telle que :

- $x_0 = y_0$, $x_1 = x_0 \nabla y_1$, …, $x_{i+1} = x_i \nabla y_{i+1}$
- $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_i \sqsubseteq \ldots$
- $x_0 \neq x_1 \neq \ldots \neq x_i \neq \ldots$

Définition 1.3 (Convergence)**.** Un opérateur ∇ *converge* si et seulement si toutes ses chaînes strictement augmentantes sont finies.

Toute la difficulté est de déterminer un ensemble de points d'élargissement (où on appliquera ∇ plutôt que $\sqcup$) qui soit :

- suffisamment petit pour limiter la perte de précision (si on applique ∇ à tous les nœuds, la solution calculée sera probablement très imprécise)
- suffisamment complet pour garantir la convergence de l'analyse.

L'algorithme de Kildall a le défaut de traiter les points du programme dans un ordre quelconque, déterminé uniquement par l'ordre dans lequel les noeuds sont extraits de W. Par conséquent il ne permet pas de distinguer un ensemble de points particuliers comme étant un bon ensemble de points d'élargissement. C'est un algorithme qui n'est donc pas adapté à l'utilisation de techniques d'élargissement.

1.2 Etat de l'art

Le problème du calcul d'un post-point fixe est central dans le domaine de l'analyse statique de programmes. Après Kildall, plusieurs auteurs se sont penchés sur la question. Un autre algorithme a été proposé par Bourdoncle [3]. Sa particularité est d'être paramétré par un ordre d'itération, calculé sur le flot de contrôle du graphe, tel qu'on puisse en déduire un ensemble relativement petit de points d'élargissement. On peut ainsi obtenir un solveur de post-point fixe qui converge rapidement quel que soit le treillis et qui soit relativement précis. L'algorithme de Bourdoncle est donc une très bonne base pour faire face à l'inconvénient majeur de l'algorithme de Kildall. Cette particularité lui vaut notamment d'être fréquemment utilisé pour calculer des post-points fixes dans des analyses statiques [2].

Bourdoncle donne en fait deux versions différentes de son algorithme, qui se basent sur des *stratégies d'itération* différentes : une stratégie dite *itérative* et une stratégie dite *récursive*.

Quant à la vérification d'algorithmes de calcul de points fixes, elle a été abordée dans un contexte bien plus général dans le cas de solveurs *locaux* [8, 5]. Un solveur local a pour but de calculer une information à un point de programme précis, tout en effectuant le moins de calculs possible, et non à faire une analyse sur tous les points de programme. De plus ces solveurs sont *génériques* c'est-à-dire qu'ils n'émettent aucune hypothèse sur les dépendences entre les points. Or, dans le cadre de notre problème, les dépendances entre les points de programmes peuvent être déterminées grâce au graphe de flot de contrôle. On pourrait donc avoir des algorithmes plus adaptés à l'analyse flot de données, qui utiliseraient cette information pour gagner en efficacité.

1.3 Solutions proposées

Pendant ce stage, nous avons remplacé l'implémentation de l'algorithme de Kildall existante dans CompCert par une implémentation comprenant

- la stratégie itérative de l'algorithme de Bourdoncle,
- le calcul préalable de l'ordre d'itération.

Bien que la stratégie itérative soit en général moins efficace d'après Bourdoncle [3], sa preuve de correction est beaucoup plus simple. Nous avons donc choisi de l'implémenter en premier afin d'avoir plus rapidement une implémentation fonctionnelle. Nous avons par la suite prouvé en Coq la correction de notre implémentation. Quant au calcul de l'ordre d'itération, nous avons dans un premier temps implémenté et prouvé en Coq une version naïve, puis nous avons adopté une approche de type validation *a posteriori* pour pouvoir modifier l'algorithme de calcul de manière simple et arriver à une version bien plus efficace.

Cette implémentation est générique vis-à-vis des différentes analyses considérées, et n'est pas spécifique à l'utilisation d'un demi-treillis en particulier. Dans CompCert, plusieurs analyses flot de données sont effectuées lors de la compilation, notamment :

- une analyse de vivacité (utilisée pour l'allocation de registres)

- une analyse de valeurs (utilisée pour la propagation des constantes, l'élimination de sous-expressions communes)
- la détection du code mort (pour l'élimination du code mort)

De plus, on peut effectuer des analyses en avant (comme dans le pseudo-code de l'algorithme 1) aussi bien qu'en arrière (c'est-à-dire qu'on propage l'information d'un nœud vers son prédécesseur, plutôt que l'inverse). Par exemple, l'analyse de vivacité est effectuée en arrière : on calcule l'ensemble des variables vivantes à un point du programme en fonction des variables qui sont vivantes aux successeurs. Pour effectuer une analyse en arrière, on fait en réalité une analyse en avant sur l'inverse du graphe de flot de contrôle.

Enfin, nous avons comparé ses performances avec celles de l'ancien algorithme en termes de précision et d'efficacité grâce à des expériences automatisées.

2 Algorithme de Bourdoncle

Contrairement à l'algorithme de Kildall, l'algorithme de Bourdoncle ne fonctionne pas avec une liste de nœuds à traiter. Il faut donc déterminer au préalable l'ordre dans lequel on va itérer sur les nœuds du graphe.

Cet ordre est construit à partir de la notion de *tri topologique faible* introduite par Bourdoncle [3], que nous allons définir dans la sous-section suivante. Informellement, un tri topologique faible sert à donner une structure à l'ensemble des nœuds du graphe, qui permet notamment de distinguer les boucles présentes dans le programme.

2.1 Tri topologique faible

Définition 2.1 (**Tri hiérarchique).** Un *tri hiérarchique* $\mathcal{H}$ d'un ensemble est une permutation des élements de cet ensemble. Pour deux élements u, v de l'ensemble, on notera $u \preceq_{\mathcal{H}} v$ (ou simplement $u \preceq v$ s'il n'y a pas d'ambiguité) la relation "u est avant v dans le tri hiérarchique, ou $u = v$". Cette relation définit un ordre total sur les élements de l'ensemble.

Remarque. Ici, nous utilisons une définition légèrement différente de celle de Bourdoncle [3]. En effet, cette dernière donne une permutation *bien parenthésée* de l'ensemble. Nous considèrerons par la suite de telles permutations, en introduisant la notion de parenthésage.

Définition 2.2 (**Parenthésage).** Un *parenthésage* d'un tri hiérarchique est un parenthésage bien formé sur les éléments de la permutation donné par le tri hiérarchique, tel qu'on n'a jamais deux parenthèses ouvrantes consécutives.

Définition 2.3 (**Composantes, têtes).** Une *composante* d'un parenthésage d'un tri hiérarchique correspond aux éléments contenus entre deux parenthèses correspondantes dans le parenthésage. Le premier élement de la composante est appelé la *tête* de la composante. Pour un élément u, on note $\omega(u)$ l'ensemble des têtes des composantes qui contiennent u.

Remarque. Si l'ensemble considéré est l'ensemble des nœuds du GFC, on pourrait construire un parenthésage en ouvrant une parenthèse au début de chaque boucle / sous-boucle du programme, puis en fermant la parenthèse à la fin de la boucle. Ainsi, chaque boucle forme une composante.

Remarque. Si h est la tête d'une composante d'un parenthésage de $\mathcal{H}$, alors on a $h \preceq_{\mathcal{H}} x$ pour tout x contenu dans la composante.

Exemple 2.1. *Soit* $S = \{1, 4, 5, 7, 10, 15\}$.

- $\mathcal{H}_S = [5, 7, 1, 15, 10, 4]$ *est un tri hiérarchique de* S

- *Un exemple de parenthésage de $\mathcal{H}_S$ est*

$$5\ (7\ 1\ (15\ 10)\ 4)$$

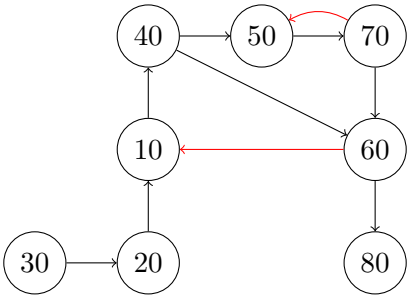
- *Ce parenthésage est formé de deux composantes, entre 7 et 4 et entre 15 et 10, dont les têtes sont respectivement 7 et 15.*

Définition 2.4 (**Arc arrière**). Etant donné un tri hiérarchique $\mathcal{H}$, un arc $(u, v) \in E$ du graphe de flot de contrôle est un *arc arrière* si $v \preceq u$.

Définition 2.5 (**Tri topologique faible**). Un *tri topologique faible* (t.t.f.) d'un graphe de flot de contrôle G est un parenthésage d'un tri hiérarchique de l'ensemble des nœuds de G, qui vérifie la propriété suivante : pour tout arc arrière $u \to v$ de G, on a $v \in \omega(u)$

Remarque. Intuitivement, un t.t.f. est un parenthésage comme décrit dans la remarque après la Définition 2.3. En effet, les arcs arrière pointent en général vers un nœud correspondant à une tête de boucle ; par conséquent, les têtes de boucles correspondent aux têtes des composantes du t.t.f..

Exemple 2.2. *Pour le graphe de flot de contrôle G_0 suivant,*



$w_0 = 30\ 20\ (10\ 40\ (50\ 70)\ 60)\ 80$ *est un t.t.f.. Les arcs arrières sont dessinés en rouge.*

Dans la suite, on supposera qu'on dispose d'un t.t.f. pour G. La question du calcul d'un t.t.f. sera traitée dans la Section 4. On représentera formellement un t.t.f. par une liste d'*éléments*, où un élément peut être défini inductivement de la manière suivante :

- $(V\ u)$ est un élément qui représente un nœud d'étiquette u
- $(C\ h\ l)$ est un élément qui représente la composante dont la tête est h, et qui contient (récursivement) les *éléments* de la liste l. La tête est une simple étiquette et non un *élément* : en effet, on ne peut pas avoir deux parenthèses ouvrantes consécutives dans le t.t.f., donc le premier élément d'une composante ne peut pas lui-même être une sous-composante.

On peut alors déduire d'un t.t.f. le tri hiérarchique utilisé (on rappelle qu'un t.t.f. est un parenthésage d'un tri hiérarchique), qu'on appellera *tri hiérarchique induit*, grâce à la définition suivante :

Définition 2.6 (**Tri hiérarchique induit**). Soit un t.t.f. w. On définit le *tri hiérarchique induit* $H(w)$ sous forme d'une liste de sommets en utilisant les fonctions mutuellement récursives H et H_{elt} suivantes :

- Pour un t.t.f. w, c'est-à-dire une liste d'éléments, $H(w)$ est défini comme la liste obtenue en concaténant les listes obtenues en appliquant H_{elt} sur les éléments de w
- $H_{elt}(V\ u) = [u]$
- $H_{elt}(C\ h\ l) = h :: H(l)$

Exemple 2.3. *La représentation du t.t.f. w_0 donné dans l'Exemple 2.2 est*

$$w_0 = [V\ 30, V\ 20, C\ 10\ [V\ 40, C\ 50\ [V\ 70]\ V\ 60]\ V\ 80]$$

et on aura $H(w_0) = [30, 20, 10, 40, 50, 70, 60, 80]$.

Avec notre représentation formelle, un t.t.f. doit vérifier les propriétés suivantes qui garantissent sa validité.

- (1) L'ensemble des éléments de $H(w)$ doit être clos par successeur : si $s \in \mathsf{succ}(n)$ et n est un élément de $H(w)$, alors s est aussi un élément de $H(w)$
- (2) Il ne doit pas y avoir de doublons dans $H(w)$
- (3) Le nœud d'entrée n_e du graphe doit être dans $H(w)$
- (4) On doit avoir la propriété caractéristique d'un t.t.f., donnée par la Définition 2.5. Ainsi, si $u \to v$ est un arc arrière dans $H(w)$, alors v est la tête d'une composante contenant u, c'est-à-dire qu'un élément $C\ v\ l$ tel que $u = v$ ou $u \in H(l)$ apparaît dans w.

2.2 Composantes externes

Pour appliquer la stratégie itérative de Bourdoncle, on doit raisonner sur les *composantes externes* d'un t.t.f.. Informellement, les composantes externes sont les composantes d'un t.t.f. qui ne sont contenues dans aucune autre composante du t.t.f.. Ce sont en pratique les boucles englobantes du programme. Cependant, la stratégie itérative ne considère pas les sous-composantes contenues à l'intérieur des composantes externes dans le t.t.f.. On a donc seulement besoin de connaître la liste des nœuds contenus dans chaque composante externe. Par conséquent on représentera les composantes externes par leur tri hiérarchique induit.

Définition 2.7 (Composantes externes). On définit $\mathcal{O}(w)$ comme étant la liste des *composantes externes* d'un t.t.f. w, c'est-à-dire la liste obtenue en appliquant H_{elt} sur les élements de w.

Remarque. Comme on a besoin de résoudre les équations sur tous les nœuds du graphe, on doit inclure les nœuds qui sont en dehors de toutes les composantes (par exemple, 20 dans w_0). Ainsi, nous avons une composante externe $[n]$ pour chaque nœud « isolé » n : en effet, $H_{elt}(V\ n) = [n]$.

Exemple 2.4. *Pour w_0, on a* $\mathcal{O}(w_0) = [[30], [20], [10, 40, 50, 70, 60], [80]]$

2.3 Stratégie itérative

Le principe de la stratégie itérative de Bourdoncle peut être décrit comme suit : pour chaque composante externe c,

- appliquer la fonction de transfert sur l'information calculée pour chaque nœud de c, et propager à chaque fois le résultat aux successeurs du nœud comme dans l'algorithme de Kildall,
- répéter jusqu'à atteindre un point fixe sur ces nœuds. On dit qu'on a *stabilisé* la composante.

Tout au long de l'algorithme, on maintient un tableau aval associant à chaque nœud une valeur abstraite de L. Ce tableau associatif représente la solution provisoire du système d'équations calculée jusqu'ici. Il contient l'information calculée à l'entrée de chaque nœud du graphe. L'objectif est de prouver que les équations sont toujours vraies sur les nœuds des composantes préalablement stabilisées ; par conséquent, on introduit une variable S qui contiendra l'ensemble des composantes déjà stabilisées. Cette variable n'intervient pas dans le calcul de la solution : elle est uniquement utile pour établir la preuve de correction. On doit alors maintenir l'invariant suivant :

$$\forall c \in S, \forall n \in c, \forall s \in \mathsf{succ}(n), T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s] \quad (*)$$

Remarque. Pour préserver cette propriété lors de l'exécution de l'algorithme, les composantes externes ne peuvent pas être stabilisées dans n'importe quel ordre. En effet, revenons à l'exemple de w_0. Si on stabilise [80] avant [20], alors lorsqu'on propagera le résultat de l'application de la fonction de transfert au nœud 20 vers le nœud 80 (cela se produira car 80 est un successeur de 20), les équations pourraient ne plus être vraies au nœud 80. Or [80] aura déjà été ajouté à S, donc $(*)$ ne sera plus vrai.

Ici, ce n'est heureusement pas un problème, car $\mathcal{O}(w)$ a été calculé en appliquant H_{elt} sur la liste des éléments de w. Par conséquent, les composantes externes sont dans le bon ordre : l'ordre donné par le tri hiérarchique induit par w est préservé. Plus précisément, si c est stabilisée avant c', alors tous les nœuds de c sont strictement avant tous les nœuds de c' dans le tri hiérarchique $\mathcal{H}$.

Nous pouvons finalement donner le pseudo-code de la stratégie itérative de Bourdoncle (voir Algorithme 2).

Algorithme 2 : Bourdoncle, stratégie itérative

```
   Entrées : Une fonction de transfert T, une fonction successeur succ, un nœud d'entrée n_e,
             une valeur d'entrée v_e et un t.t.f. valide w
   Output : Une solution aval
 1 S ← ∅
 2 aval ← le mapping qui associe chaque nœud à ⊥
 3 aval[n_e] ← v_e
 4 pour chaque c ∈ O(w) faire
 5 |  répéter
 6 |  |  m ← false
 7 |  |  pour n ∈ c faire
 8 |  |  |  out ← T(n, aval[n])
 9 |  |  |  pour chaque s ∈ succ(n) faire
10 |  |  |  |  in ← aval[s] ⊔ out
11 |  |  |  |  si in ≠ aval[s] alors
12 |  |  |  |  |  aval[s] ← in
13 |  |  |  |  |  m ← true
14 |  |  |  |  fin si
15 |  |  |  fin pour chaque
16 |  |  fin pour
17 |  jusqu'à m = false;
18 |  S ← S ∪ {c}
19 fin pour chaque
20 return aval
```

3 Preuve de correction

Dans cette section, nous donnons une preuve de la correction de l'Algorithme 2. Le but est de montrer que le tableau associatif aval fourni par l'algorithme est effectivement une solution du système d'équations.

On introduit d'abord quelques lemmes auxiliaires utiles, qui prouvent des bonnes propriétés sur les composantes externes du graphe.

Définition 3.1 (Composantes disjointes, inclusion). On dit que deux composantes c et c' sont *disjointes* s'il n'y a pas de nœud contenu à la fois dans c et c'. On dit que c est *incluse* dans c' si tous les nœuds de c sont dans c'.

Remarque. Par abus de notation, on pourra utiliser des notations ensemblistes pour les composantes : par exemple, $n \in c$ signifie « n est contenu dans c », $c \subseteq c'$ signifie « c est incluse dans c' », $c \cap c' = \emptyset$ signifie « c et c' sont disjointes ».

Lemme 3.1. *Soit w un t.t.f. valide. Alors les composantes de $\mathcal{O}(w)$ sont deux à deux disjointes.*

Preuve. Soit c, c' deux composantes de $\mathcal{O}(w)$ telles que $c \neq c'$. Alors il existe deux éléments distincts e, e' dans w tels que $H_{elt}(e) = c$ and $H_{elt}(e') = c'$. Supposons qu'il existe un nœud $n \in c \cap c'$. Alors comme $H(w)$ est la concaténation des listes obtenues en appliquant H_{elt} sur les éléments de w, il y aura au moins deux occurrences de n dans $H(w)$, ce qui contredit la propriété (2) d'un t.t.f. valide. □

Lemme 3.2. *Pour tout $n \in N$ accessible depuis n_e dans G, il existe une composante externe c qui contient n. En d'autres termes, tout nœud du graphe accessible depuis n_e est inclus dans une composante externe.*

Preuve. D'après les propriétés (1) et (3) d'un t.t.f. valide, $H(w)$ contient n_e et est clos par successeur. Par conséquent, si n est accessible depuis n_e, alors n est dans $H(w)$. Il existe donc un élément e de w tel que $H_{elt}(e)$ contient n. De plus, comme $\mathcal{O}(w)$ est la liste obtenue en appliquant H_{elt} sur les éléments de w, $H_{elt}(e)$ est une composante externe. En conclusion, on a bien trouvé une composante externe qui contient n. □

Lemme 3.3. *Soit c une composante de w. Alors elle est incluse dans une composante externe de w.*

Preuve. On prouve par induction que les nœuds qui apparaissent dans un élément e sont tous dans $H_{elt}(e)$. En effet,

- Seul n apparaît dans $V\ n$, et $H_{elt}(V\ n) = [n]$
- Les nœuds apparaissant dans $C\ h\ l$ sont h ainsi que tous les nœuds apparaissant dans un élément de l. Par hypothèse d'induction, tous les nœuds apparaissant dans un élément e de l sont dans $H_{elt}(e)$. Par conséquent, par définition de H, tous les sommets apparaissant dans un élément de l sont dans $H(l)$. Finalement, $H_{elt}(C\ h\ l) = h :: H(l)$, donc $H_{elt}(C\ h\ l)$ contient h ainsi que tous les nœuds apparaissant dans un élément de l.

En conclusion, si une composante c apparaît dans un élément e de w, alors tous ses sommets (qui apparaissent aussi dans e) sont dans $H_{elt}(e)$ qui est une composante externe de w. □

Lemme 3.4. *Soit c une composante externe qui contient n. Si $n \to s$ est un arc arrière de G, alors c contient s. En d'autres termes, tout arc arrière à partir d'un nœud d'une composante externe mène à un autre nœud de la même composante.*

Preuve. Par définition, les composantes externes ne contiennent que des éléments de $H(w)$ donc n est un élément de $H(w)$. Comme l'ensemble des éléments de $H(w)$ est clos par successeur, on peut en déduire que s est aussi un élément de $H(w)$.

D'après le Lemme 3.2, il existe donc une composante externe o_s qui contient s. Comme $n \to s$ est un arc arrière, d'après la propriété (4) d'un t.t.f. valide, s est aussi la tête d'une composante c_s de w qui contient n. De plus, d'après le Lemme 3.3, c_s est incluse dans une composante externe o'_s. Mais comme les composantes externes sont deux à deux disjointes d'après le Lemme 3.1 et que $s \in o_s \cap o'_s$, alors $o_s = o_{s'}$. Sachant que c_s est incluse dans o'_s, n est également dans $o'_s = o_s$. Finalement, comme les composantes externes sont deux à deux disjointes et que $n \in o_s \cap c$, alors $c = o_s$ donc c contient s. □

A présent, nous pouvons prouver des propriétés sur l'algorithme et ainsi aboutir au théorème final de correction. La preuve se base sur la préservation de l'invariant $(*)$ tout au long de l'exécution de l'algorithme, car à la fin de l'exécution toutes les composantes externes sont dans S et le Lemme 3.2 énonce que tous les nœuds sont dans une composante externe.

Lemme 3.5. *Soient S_0, aval_0 les valeurs de S et aval avant une itération de la boucle **répéter**. Soit aval_1 la valeur de aval après l'itération, quand la ligne 18 est atteinte.*

Alors pour tout $c \in S_0$ et $s \in c$, on a $\mathsf{aval}_1[s] = \mathsf{aval}_0[s]$.

Preuve. Soit c_{cur} la composante sur laquelle la boucle **pour** extérieure est en train d'itérer. Comme nous avons remarqué dans la section 2.3, tous les nœuds de c_{cur} sont strictement après tous les nœuds des composantes de S_0 dans le tri hiérarchique $\mathcal{H}$.

Soit $c \in S_0$ et $s \in c$, et supposons que $\mathsf{aval}_1[s] \neq \mathsf{aval}_0[s]$. Comme aval n'est modifié qu'à la ligne 12, s doit être un successeur d'un nœud $n \in c_{cur}$. De plus, $s \preceq_{\mathcal{H}} n$: l'arc $n \to s$ est donc un arc arrière.

Par conséquent, d'après le Lemme 3.4, on a $s \in c_{cur}$. En conclusion, on a $s \in c_{cur} \cap c$. Mais on a évidemment $s \not\prec_{\mathcal{H}} s$, ce qui contredit le fait que tous les nœuds de c_{cur} soient strictement avant tous les nœuds de c dans $\mathcal{H}$.

Par contradiction, on peut en déduire que $\mathsf{aval}_1[s] = \mathsf{aval}_0[s]$. □

Lemme 3.6. *Soient S_0, aval_0 les valeurs de S et aval avant une itération de la boucle **répéter** loop. Soit aval_1 la valeur de aval après l'itération, lorsque la ligne 18 est atteinte.*

*Soit $c_{cur} \in \mathcal{O}(w)$ la composante sur laquelle la boucle **pour** extérieure est en train d'itérer. Alors pour tout $n \in c_{cur}$ et $s \in \mathsf{succ}(n)$, on a $T(n, \mathsf{aval}_1)[n] \sqsubseteq \mathsf{aval}_1[s]$.*

Preuve. Lorsque la ligne 18 est atteinte, on est sorti de la boucle **répéter** avec $m = \mathsf{false}$. On doit donc prouver qu'à la fin d'une itération, si $m = \mathsf{false}$, alors pour tout $n \in c$ et $s \in \mathsf{succ}(n)$, on a $T(n, \mathsf{aval}_1)[n] \sqsubseteq \mathsf{aval}_1[s]$. En effet, si $m = \mathsf{false}$ à la fin de l'itération,

- m n'a jamais été mis à true pendant l'itération en cours,
- aval n'a jamais été modifié pendant l'itération en cours, car la condition à la ligne 11 ne peut jamais être vraie. Par conséquent, on a $\mathsf{aval} = \mathsf{aval}_1$ à tout moment pendant l'itération en cours.

De plus, comme la condition à la ligne 11 est toujours fausse, on a pour tout $n \in X$ et $s \in \mathsf{succ}(n)$,

$$\mathsf{aval}_1[s] \sqcup T(n, \mathsf{aval}_1[n]) = \mathsf{aval}_1[s]$$

D'après les propriétés de la borne supérieure,

$$T(n, \mathsf{aval}_1[n]) \sqsubseteq \mathsf{aval}_1[s] \sqcup T(n, \mathsf{aval}_1[n])$$

donc on a finalement $T(n, \mathsf{aval}_1)[n]) \sqsubseteq \mathsf{aval}_1[s]$. □

Lemme 3.7. *A la fin de chaque itération de la boucle **pour** extérieure, on a l'invariant $(*)$: si $c \in S$, $n \in c$, et $s \in \mathsf{succ}(n)$, alors $T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s]$.*

Preuve.

- Au début de l'exécution, on a $S = \emptyset$ donc $(*)$ est trivialement vrai
- Supposons que $(*)$ est vrai avant une itération donnée. Soit c_{cur} la composante sur laquelle on itère pendant cette itération. On note S_0 et aval_0 les valeurs de S et aval avant de commencer l'itération, et S_1 et aval_1 après l'iteration. Le seul point de la boucle où l'on modifie S est la ligne 18, par conséquent on a $S_1 = S_0 \cup \{c_{cur}\}$.

  Soient $c \in S_1$, $n \in c$ et $s \in \mathsf{succ}(n)$.

  - **Premier cas :** $c \in S_0$. Alors d'après le Lemme 3.5, et sachant qu'on ne modifie évidemment pas aval à la ligne 18, on a $\mathsf{aval}_1[n] = \mathsf{aval}_0[n]$. En conséquence, par hypothèse d'induction, $T(n, \mathsf{aval}_1[n]) = T(n, \mathsf{aval}_0[n]) \sqsubseteq \mathsf{aval}_0[s]$. Toutes les valeurs abstraites calculées dans aval croient pendant l'exécution de l'algorithme : lorsqu'on modifie une valeur, on prend une borne supérieure de la valeur courante. Ainsi, $\mathsf{aval}_0[s] \sqsubseteq \mathsf{aval}_1[s]$ et finalement, $T(n, \mathsf{aval}_1)[n] \sqsubseteq \mathsf{aval}_1[s]$.

- **Second cas :** $c = c_{cur}$. Alors d'après le Lemme 3.6, pour tout $n \in c = c_{cur}$ et $s \in \mathsf{succ}(n)$, on a $T(n, \mathsf{aval}_1)[n] \sqsubseteq \mathsf{aval}_1[s]$.

En conclusion, l'invariant $(*)$ est préservé lors d'une itération, il est donc finalement vrai après chaque itération.

□

Théorème 3.1 (Correction). *Soit* aval *le tableau associatif renvoyé à la fin de l'exécution de l'Algorithme 2. Pour tout* $n \in N$ *et* $s \in \mathsf{succ}(n)$*, on a* $T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s]$.

Preuve. A la fin de l'exécution, on a itéré sur toutes les composantes externes $c \in \mathcal{O}(w)$. On a donc $S = \mathcal{O}$ lorsqu'on sort de la boucle **pour** extérieure.

Ainsi, d'après le Lemme 3.7, après la dernière itération de la boucle **pour** extérieure, on a $T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s]$ pour tout $c \in \mathcal{O}$, $n \in c$, and $s \in \mathsf{succ}(n)$. De plus pour tout nœud $n \in N$, le lemme 3.2 nous permet de nous assurer qu'il existe bien une composante $c \in \mathcal{O}(w)$ telle que $n \in c$.

En conclusion, pour tout $n \in N$ et $s \in \mathsf{succ}(n)$, on a effectivement $T(n, \mathsf{aval}[n]) \sqsubseteq \mathsf{aval}[s]$. □

Remarque. Bourdoncle [3] a prouvé qu'il n'était nécessaire de vérifier si l'information a été modifiée (test à la ligne 11) uniquement au niveau des têtes des composantes. En ce qui concerne CompCert, les demi-treillis utilisés sont tels que les tests d'égalité sont en général peu coûteux. Cependant il pourrait être intéressant dans le futur d'exploiter cette propriété pour économiser des tests d'égalité potentiellement coûteux (par exemple, lorsque les valeurs abstraites sont des gros ensembles).

4 Calcul d'un tri topologique faible

4.1 Méthode naïve

Il existe une manière très simple de calculer un t.t.f.. En effet, il suffit de considérer chaque nœud du graphe, l'ajouter à la permutation et ouvrir une parenthèse si et seulement s'il existe un arc arrière dans le graphe qui mène à ce nœud. On ferme à la fin toutes les parenthèses.

Exemple 4.1. *Avec le graphe* G_0*, on obtiendrait* $W'_0 = 30\ 20\ (10\ 40\ (50\ 70\ 60\ 80))$

On obtient bien un t.t.f. : tout arc arrière $u \to v$ est bien tel que v soit une tête d'une composante qui contient u (car on a ouvert une parenthèse avant v, et toutes les parenthèses sont fermées à la fin : la composante dont la tête est v contient donc bien u qui est après v).

Dans un souci de simplicité de la preuve, nous avons dans un premier temps choisi cette méthode pour notre implémentation de l'algorithme de Bourdoncle. Nous avons écrit en Coq l'algorithme permettant de calculer un t.t.f. avec cette méthode, puis nous avons prouvé en Coq sa correction (i.e. qu'il renvoie bien un t.t.f. valide).

Cependant, utiliser un tel t.t.f. dans le cadre de l'algorithme de Bourdoncle rend le calcul de point fixe clairement inefficace. En effet, les composantes sont beaucoup plus larges que nécessaires. Or stabiliser une composante très large nécessite beaucoup de calculs, car dès qu'une itération modifie une valeur dans aval, on doit réitérer sur toute la composante. S'il y a un arc arrière qui pointe vers le premier nœud, il n'y aura in fine qu'une seule composante externe contenant tous les nœuds...

4.2 Méthode des sous-composantes fortement connexes

La méthode des sous-composantes fortement connexes (s.c.f.c.) permet de calculer un t.t.f. beaucoup plus précis. L'algorithme est donné en détail dans [4]. Cet algorithme se base sur une décomposition du graphe en composantes fortement connexes.

Afin de pouvoir modifier aisément l'implémentation de cet algorithme sans se préoccuper des preuves, nous avons dans un premier temps opté pour une approche de type validation *a posteriori*, c'est-à-dire que plutôt que d'écrire en Coq l'algorithme et de prouver sa correction par la suite, nous avons :

- implémenté l'algorithme en OCaml de manière non prouvée en nous inspirant de la bibliothèque `ocamlgraph` [6]
- écrit un validateur prouvé en Coq, qui appelle le code OCaml et n'accepte son résultat que lorsque le t.t.f. renvoyé est valide. Dans le cas contraire, on calcule un t.t.f. de manière naïve (cela ne devrait jamais arriver en théorie)

Ainsi on a finalement une preuve formelle que le t.t.f. donné en paramètre à l'algorithme de Bourdoncle est toujours un t.t.f. valide.

En effectuant des tests, on remarque que le validateur accepte le t.t.f. candidat fourni par le code OCaml à chaque fois : par conséquent, l'algorithme en OCaml semble être correct et nous pouvons espérer pouvoir le prouver en Coq dans le futur afin d'éliminer l'étape de validation *a posteriori* qui est en pratique très coûteuse.

5 Implémentation et preuve dans CompCert

Pendant le stage, nous avons implémenté et prouvé la stratégie itérative de Bourdoncle en Coq dans CompCert, en suivant les idées de la preuve présentée dans ce rapport.

Notre contribution en termes de développement Coq est constituée d'environ 2700 lignes de code. Plus précisément, approximativement 700 lignes sont consacrées à l'implémentation et sa spécification, et 2000 lignes sont consacrées aux preuves. A cette contribution s'ajoutent 500 lignes supplémentaires de code Coq servant au calcul d'un t.t.f. de manière naïve comme en Section 4.1, ainsi qu'à la preuve de correction de ce calcul.

5.1 Choix de représentation

Le choix de la représentation d'un t.t.f. est une question cruciale pour rendre l'implémentation et la preuve les plus directs et efficaces possibles. Dans un premier temps, nous avions décidé de représenter un t.t.f. w par

- la liste de ses composantes, chacune étant sous forme d'ensemble (`PositiveSet` en Coq)
- son tri hiérarchique induit $H(w)$

Cependant, il faut pouvoir caractériser les propriétés qui sont liées au fait que ce soit un t.t.f.. Nous avons alors rencontré des difficultés dans la mesure où tout t.t.f. devait être fourni avec un certain nombre de bonnes propriétés (sous forme de `Record` en Coq) ; par exemple, la tête d'une composante doit nécessairement être incluse dans la composante et avant tous les autres éléments dans $H(w)$. Cette représentation est devenue rédhibitoire lorsque l'on a voulu valider *a posteriori* le calcul du t.t.f. : en effet, on devait valider chaque propriété séparément, et la complexité du calcul pour certaines propriétés était très élevée (cubique). De plus l'implémentation de l'algorithme des s.c.f.c. fournie par `ocamlgraph` utilise une représentation différente ce qui nous aurait obligé à devoir transformer le t.t.f. obtenu avec un coût de calcul supplémentaire.

Par conséquent, nous avons adopté la représentation « récursive » donnée par `ocamlgraph` : un t.t.f. est alors une liste de termes appartenant au type `element` défini comme suit (notre définition en Coq puis la définition d'`ocamlgraph` en OCaml) :

```
Inductive element : Type :=
| Vertex : positive -> element
```

```
| Component : positive -> list element -> element.

type 'a element =
| Vertex of 'a
| Component of 'a * 'a t
```

Cette représentation est beaucoup plus naturelle et reflète directement l'idée de parenthésage associée à la structure de t.t.f.. Son seul défaut est qu'il faut définir manuellement un principe d'induction sur le type `element`. En effet, son constructeur `Component` est paramétré par un terme de type `list element`, et dans ce cas Coq ne génère pas automatiquement de principe d'induction utile (dans le principe d'induction automatiquement généré, il n'y a pas d'hypothèse d'induction sur les éléments de la liste en paramètre).

Les composantes externes, quant à elles, sont représentées par les listes des sommets qu'elles contiennent. Cependant, nous avons remarqué, en effectuant les tests de performance, que cette représentation est légèrement sous-optimale. En effet, comme remarqué à la suite de la Définition 2.7, chaque élément `Vertex` n est transformé en la composante externe $[n]$. En conséquence, lorsqu'on stabilisera cette composante, on itèrera deux fois l'équation au nœud n : la deuxième fois servira simplement à vérifier qu'on a bien atteint un point fixe. Or un tel sommet dans un t.t.f. n'est l'extrémité d'aucun arc arrière donc le point fixe est nécessairement atteint dès la première itération. Malheureusement nous n'avons remarqué cette possibilité d'amélioration qu'au moment d'effectuer les tests de performances, à la toute fin du stage. Nous l'avons donc appliquée au code extrait en OCaml, et la preuve en Coq de la correction de l'implémentation résultant de cette amélioration sera à réaliser dans le futur.

Nous avons donc décidé de représenter les composantes externes à l'aide du type suivant (ici en OCaml) :

```
type ocomponent =
| OVertex of positive
| OComponent of positive list
```

On n'itère alors qu'une seule fois sur les composantes de la forme `OVertex` n.

5.2 Preuves en Coq

Le code s'articule en deux parties : d'une part, un module `Wto.v` fournit les propriétés qui doivent être vérifiées par un t.t.f., et d'autre part, un module `Bourdoncle.v` contient l'implémentation de la stratégie itérative de Bourdoncle ainsi que des théorèmes finaux analogues à ceux fournis par le module `Kildall.v`, avec leur preuve. Par exemple, le théorème `fixpoint_solution` suivant énonce que le résultat calculé par l'algorithme est bien une solution du système d'équations :

```
Theorem fixpoint_solution:
  forall ev res n instr s,
  fixpoint ev = Some res ->
  code!n = Some instr ->
  In s (successors instr) ->
  (forall n x, L.eq x L.bot -> L.eq (transf n x) L.bot) ->
  L.ge res!!s (transf n res!!n).
```

Ce théorème correspond au Théorème 3.1, avec l'hypothèse supplémentaire $T(n, \bot) = \bot$ pour tout nœud n (qui est aussi présente dans le théorème de correction analogue de l'algorithme de Kildall).

Ces théorèmes sont ensuite appliqués lors de la preuve de correction des différentes passes d'optimisation de CompCert. Les théorèmes prouvés dans le module `Bourdoncle` étant les mêmes que ceux

prouvés dans le module `Kildall`, l'intégration du nouveau solveur de point fixe (basé sur l'algorithme de Bourdoncle) au sein des optimisations de CompCert est très facile. Nous avons ajouté une option `-Obourdoncle` à CompCert qui permet de choisir, lors de la compilation, d'appliquer l'algorithme de Bourdoncle pour faire les analyses flot de données.

6 Evaluation expérimentale

Les tests de performance ont été réalisés sur la base des programmes de test fournis avec les sources du compilateur CompCert (`https://github.com/AbsInt/CompCert/tree/master/test`). Ces programmes sont relativement représentatifs de programmes réalistes, leur taille variant de quelques dizaines à plusieurs milliers de lignes. On dispose par exemple du prouveur pour la logique du premier ordre `spass`.

6.1 Protocole expérimental

D'une part, il est pertinent de comparer les durées nécessaires pour effectuer l'analyse avec les deux algorithmes. Pour ce faire, nous avons modifié le code OCaml de notre nouvelle version de CompCert après extraction de manière à mesurer le temps d'exécution de l'analyse pour chaque passe puis afficher le résultat de ces mesures. En ce qui concerne l'affichage, nous nous sommes appuyés sur le fonctionnement de l'option `-timings` existante dans CompCert, qui permet d'afficher la durée de chaque étape de la compilation.

Remarque. A terme, l'objectif est de prouver que l'algorithme de calcul d'un t.t.f. avec la méthode des composantes fortement connexes est correct. Par conséquent, lorsque nous avons mesuré les performances obtenues avec l'algorithme de Bourdoncle, nous avons fait l'hypothèse que le t.t.f. calculé est toujours correct et nous avons retiré l'étape de validation *a posteriori* (qui est en pratique l'étape la plus coûteuse en temps dans notre implémentation) après extraction du code vers OCaml.

Afin d'avoir une comparaison plus fine, nous avons également décidé de comparer le nombre d'applications de la fonction de transfert et de l'opérateur $\sqcup$. C'est pourquoi nous avons ajouté une option `-stats`, qui fonctionne de façon analogue à `-timings`, qui permet d'afficher ces nombres à la compilation de chaque fichier.

Comparer la précision des deux algorithmes d'analyse est tout aussi crucial. Pour pouvoir faire cela, nous avons ajouté un module dans le code OCaml extrait. Ce module contient une fonction `fixpoint`, similaire aux fonctions `fixpoint` des modules `Kildall` et `Bourdoncle`, qui va en fait calculer le résultat d'analyse donné par Kildall, puis par Bourdoncle, et comparer les valeurs abstraites calculées à chaque nœud (si elles sont égales, incomparables, ou si l'une est strictement inférieure à l'autre). La fonction renvoie finalement le résultat donné par Kildall en tant que point fixe. Nous avons par la suite rajouté une option `-Ocompare` qui permet d'appeler cette fonction pour calculer un point fixe, au lieu d'appeler celle du module `Kildall`. En combinant cette option avec l'option `-stats`, on peut alors obtenir un bilan des comparaisons effectuées lors de chaque analyse de flot de données. On pourra ainsi observer une tendance au niveau d'une éventuelle différence de précision entre les deux algorithmes.

Nous avons réalisé ces expériences de manière automatisée, en écrivant un script qui :

- compile l'ensemble des programmes de test en mesurant le temps pris par chaque analyse de flot de données, avec Kildall ainsi qu'avec Bourdoncle. On répète 5 fois la compilation de manière à pouvoir faire une moyenne et minimiser les incertitudes.
- compile une nouvelle fois pour déterminer sur chaque programme d'une part le nombre de fois qu'on applique la fonction de transfert, et d'autre part le nombre de fois qu'on applique l'opérateur $\sqcup$.

Les résultats affichés par les options `-timings` et `-stats` sont ensuite convertis au format CSV, puis peuvent être affichés sous forme graphique grâce à un script en Python.

Les expériences ont été réalisées avec CompCert 3.10 sur une machine tournant sous le système d'exploitation Linux, disposant d'un processeur Intel i5-1035G1 (3,6GHz), 16 Go de RAM.

6.2 Précision de l'analyse

On remarque que dans l'immense majorité des cas, les deux algorithmes donnent la même valeur abstraite. Cependant, dans une minorité de cas, les valeurs sont différentes. Elles sont toujours comparables (i.e. l'une est strictement inférieure à l'autre), et c'est Kildall qui est le plus précis dans la plupart des cas (i.e. la valeur abstraite donnée par Kildall est strictement inférieure à celle donnée par Bourdoncle) lorsque cela arrive.

Ces différences n'arrivent pas dans toutes les analyses. Par exemple, dans le cas de l'analyse de vivacité, sur nos programmes de test, les valeurs abstraites calculées sont toujours les mêmes pour les deux algorithmes. Cependant, certaines valeurs calculées par l'analyse de valeurs sont différentes et en l'occurrence, les valeurs renvoyées par Kildall sont quasiment toujours inférieures à celles renvoyées par Bourdoncle. Dans tous les cas, en ce qui concerne les programmes de test considérés, le pourcentage de nœuds dont la valeur abstraite renvoyée diffère ne dépasse jamais 0.2%.

6.3 Efficacité

On constate de manière générale que Bourdoncle est légèrement moins performant que Kildall. Plus précisément, l'analyse avec Bourdoncle prend 1.6 à 2 fois plus de temps que l'analyse avec Kildall, nécessite environ 2 fois plus d'applications de $\sqcup$ et 1.4 fois plus d'applications de la fonction de transfert. Le fait que le rapport entre les temps d'analyse reste sensiblement le même peu importe la taille des programmes et la hauteur des treillis était attendu puisque la complexité des algorithmes de Kildall et Bourdoncle est linéaire aussi bien en la taille des treillis qu'en le nombre de nœuds dans le graphe. En ce qui concerne Kildall, ce résultat découle du Théorème 1.1 ainsi que de la remarque associée. Bourdoncle a quant à lui prouvé ce résultat de complexité pour la stratégie itérative [3].

Les figures 2,3,4 donnent respectivement la comparaison des deux algorithmes en termes de temps d'analyse, d'applications de $\sqcup$ et d'applications de T sur les analyses de vivacité et de valeurs, qui sont respectivement une analyse arrière et une analyse avant. Ces deux analyses sont les deux plus complexes parmi celles présentes dans CompCert.

Ce sont des nouvelles relativement bonnes. D'une part, si l'algorithme de Bourdoncle est effectivement plus lent, l'augmentation du temps d'analyse reste modérée (ce facteur pourrait être lié à des détails d'implémentation, par exemple on pourrait éviter certains tests d'égalité comme remarqué dans la Section 3) et n'affecte en contrepartie quasiment pas la précision de la solution calculée. D'autre part, dans le cas où on ajouterait des optimisations avec des treillis plus hauts et des opérateurs d'élargissement dans le futur, la tendance pourrait vraisemblablement s'inverser grâce aux bénéfices apportés par l'algorithme de Bourdoncle en termes de rapidité de convergence. En effet, alors que la complexité de l'algorithme de Kildall est linéaire en la taille du treillis quoi qu'il arrive, celle de la stratégie itérative de Bourdoncle est linéaire en la longueur maximale d'une chaîne strictement augmentante de l'opérateur d'élargissement considéré. Ainsi si on prend un opérateur d'élargissement suffisamment large, la complexité de notre nouvelle implémentation diminuera drastiquement. On pourrait même dans certains cas ne pas avoir le choix si les treillis sont infinis : l'algorithme de Kildall risquerait de ne pas terminer.

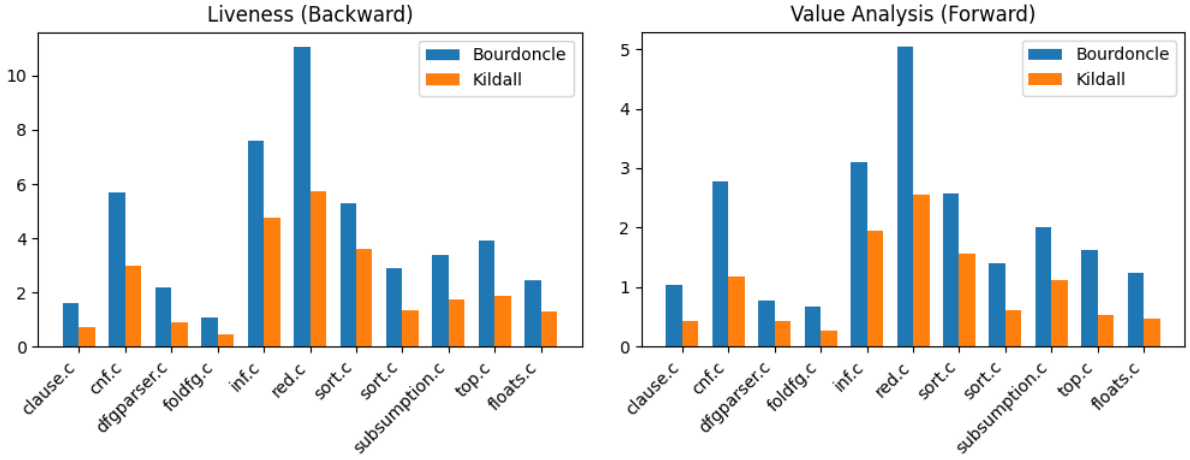


FIGURE 2 – Temps nécessaire aux analyses de vivacité (à gauche) et de valeurs (à droite) lors de 5 compilations des programmes de test

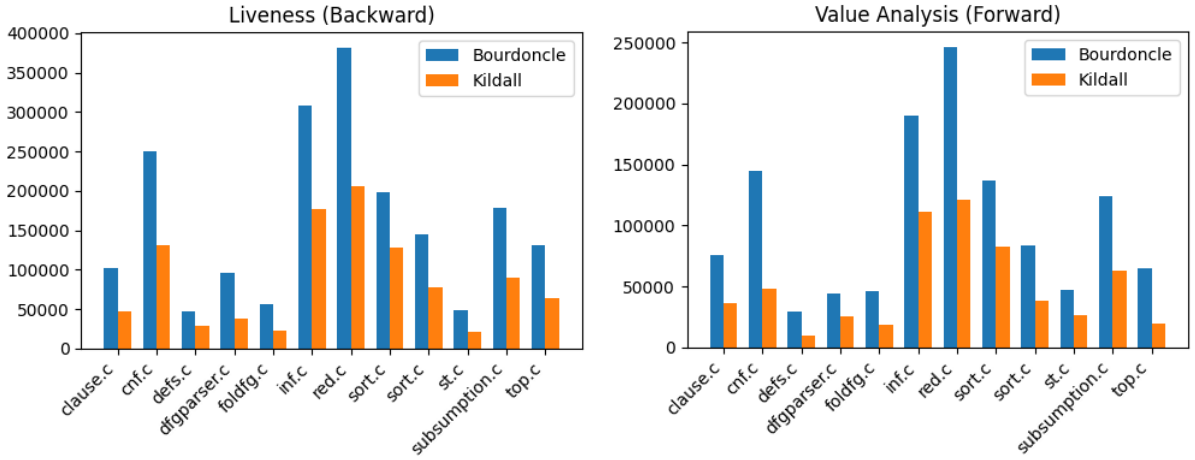


FIGURE 3 – Nombre d'applications de $\sqcup$ lors des analyses de vivacité (à gauche) et de valeurs (à droite) sur les programmes de test

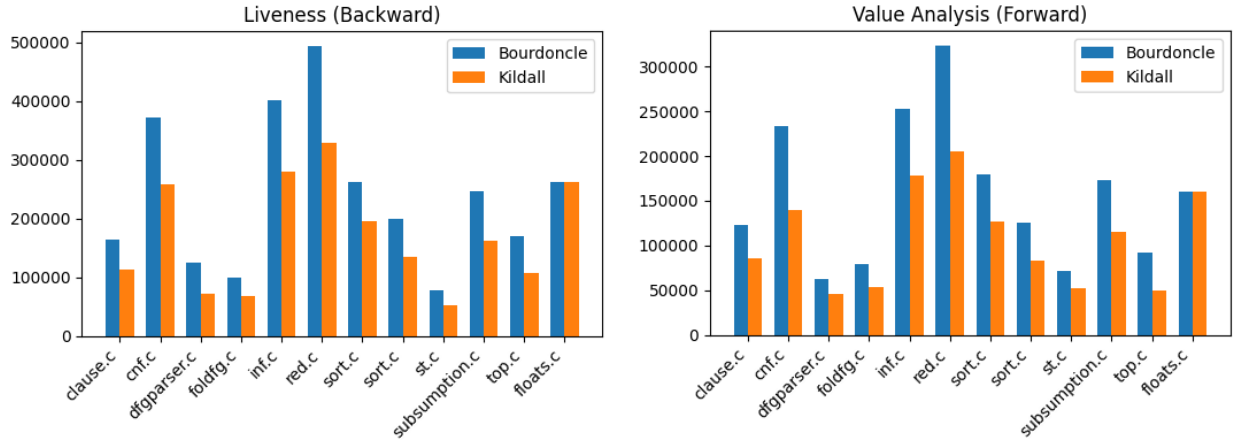


FIGURE 4 – Nombre d'applications de T lors des analyses de vivacité (à gauche) et de valeurs (à droite) sur les programmes de test

7 Conclusion et perspectives

Pendant ce stage, nous avons développé une approche prometteuse quant à l'évolution des performances du compilateur CompCert. En effet, les perspectives en termes de nouvelles optimisations étaient restreintes par le fait que l'algorithme de Kildall ne permet pas d'utiliser des opérateurs d'élargissement. Notre implémentation basée sur l'algorithme de Bourdoncle permettra de passer outre cet obstacle et ouvre la possibilité d'ajouter des analyses flot de données plus évoluées. Bien que dans la version courante de CompCert, son utilisation n'est pas nécessaire et n'apporte pour l'instant pas de gain en précision ou en efficacité, son utilisation est malgré tout réaliste dans la mesure où le temps nécessaire à l'analyse, même sans opérateur d'élargissement, reste raisonnable. La précision de l'analyse est également relativement peu affectée.

De plus, la perte de performances pourrait éventuellement être réduite en améliorant l'implémentation. Par exemple, l'ensemble S de l'Algorithme 2 est représenté sous forme de liste en Coq. Par conséquent, il est extrait dans le code OCaml, or il est complètement inutile pour le calcul (en dehors de la preuve). Pour pallier ce problème, il faudrait représenter S sous la forme d'un objet `stabilized` de type `component -> Prop` tel que `stabilized c` signifie $c \in S$. Ainsi, l'objet serait ignoré lors de l'extraction. De plus, comme remarqué dans la Section 3, il suffit de détecter la stabilisation des têtes des composantes du t.t.f. pour détecter la stabilisation d'une composante externe dans la stratégie itérative. On pourrait donc améliorer notre algorithme en exploitant cette propriété afin d'effectuer moins de comparaisons de valeurs abstraites.

Par la suite, afin d'exploiter au mieux les avantages de l'algorithme de Bourdoncle en termes de rapidité de convergence, il serait fortement souhaitable d'ajouter des opérateurs d'élargissement. Pour ce faire, il faudrait :

- déterminer à l'avance un ensemble de points d'élargissement W
- dans l'algorithme, tester à la ligne 3 si $s \in W$ et si oui, appliquer un opérateur d'élargissement plutôt que $\sqcup$.

D'après Bourdoncle [3], l'ensemble des têtes des composantes d'un t.t.f. est un ensemble de points d'élargissement qui est assez large pour garantir la convergence de la stratégie itérative. C'est en général un ensemble qui est relativement petit donc intéressant pour appliquer des techniques d'élargissement tout en conservant une solution assez précise. On pourra alors ensuite tirer profit de la présence d'opérateurs d'élargissement, par exemple, en s'inspirant de l'analyse d'alias développée dans [14] qui permet de savoir si deux pointeurs sont alias l'un de l'autre, et en améliorant sa rapidité de convergence grâce aux opérateurs d'élargissement.

La preuve en Coq de la correction du calcul d'un t.t.f. avec la méthode des s.c.f.c. sera quant à elle un passage nécessaire dans le futur, de manière à s'affranchir du coût de la validation *a posteriori* tout en conservant l'aspect formellement vérifié du compilateur.

Enfin, on pourrait se pencher sur le remplacement de la stratégie itérative par la stratégie récursive qui est en général plus rapide et plus précise. La stratégie récursive est similaire à la stratégie itérative, mais au lieu de stabiliser le contenu des composantes externes sans se soucier de si elles contiennent des sous-composantes, on stabilise récursivement les sous-composantes lorsqu'on les rencontre. L'implémentation de la stratégie récursive est assez intuitive étant donnée notre représentation d'un t.t.f.. Nous l'avons réalisée en OCaml de manière non prouvée à la fin du stage. Cependant, sa preuve, notamment de terminaison, reste plus complexe à réaliser. Cette preuve fait également partie de nos objectifs futurs.

A Références

[1] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs : The Frama-C software analysis platform. *Commun. ACM*, 64(8) :56–68, jul 2021.

[2] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In Manuel Fahndrich and Francesco Logozzo, editors, *SAS - 20th Static Analysis Symposium*, volume Lecture Notes in Computer Science of *7935*, pages 324–344, Seattle, United States, June 2013. Springer.

[3] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, pages 128–141, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[4] François Jérôme Bourdoncle. *Sémantiques des langages impératifs d'ordre superieur et interprétation abstraite*. PhD thesis, 1992. Thèse de doctorat dirigée par Cousot, Patrick Informatique Palaiseau, Ecole polytechnique 1992.

[5] Baudouin L Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical report, USA, 1992.

[6] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ML functors. 04 2007.

[7] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 269–295, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[8] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 340–355, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 265–282, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[11] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. Memory-efficient fixpoint computation. *CoRR*, abs/2009.05865, 2020.

[12] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, jul 2009.

[13] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[14] Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In Hawblitzel, Chris, Miller, and Dale, editors, *CPP 2012 - Second International Conference on Certified Programs and Proofs*,

volume 7679 of *Lecture Notes in Computer Science*, pages 11–26, Kyoto, Japan, December 2012. Springer.