[Stage L3] Étude de l'impact d'une représentation par blocs dans le compilateur vérifié CompCert

Odile Radet (ENS Rennes), sous la supervision de
David Pichardie et Jean-Christophe Léchenet (INRIA, équipe CELTIQUE)

Abstract

Dans la majorité des compilateurs modernes, le langage intermédiaire *RTL* est utilisé pour réaliser de nombreuses optimisations. Or, alors que la représentation canonique consiste à associer des blocs linéaires d'instruction à chaque nœud, le compilateur vérifié *CompCert* utilise une représentation atypique en associant une unique instruction à chaque nœud. Dans ce rapport, nous proposons une ébauche de représentation alternative avec des blocs d'instruction dans *CompCert*.

Introduction

Lorsqu'on écrit un programme, le langage de programmation utilisé comporte généralement un haut niveau d'abstraction, et n'est donc pas compréhensible par la machine. C'est pourquoi l'étape de compilation, qui consiste à transformer le code source en code objet et permet donc de générer un programme exécutable, est essentielle. Cependant, ce processus est susceptible d'introduire des erreurs dans le code compilé, rendant ainsi insuffisantes les vérifications effectuées sur le code source ; d'où l'intérêt des compilateurs *certifiés*. CompCert, un compilateur C écrit et vérifié à l'aide de l'assistant de preuves Coq, assure ainsi que le code exécutable se comporte de la même manière que le code source. L'utilisation de ce compilateur est donc d'autant plus pertinente lorsqu'il s'agit de systèmes critiques.

Le processus de compilation effectué par CompCert comporte 20 passes, utilisant 11 langages intermédiaires, parmi lesquels *register transfer language* (RTL) est l'un des plus importants (notamment en raison des nombreuses optimisations qui s'effectuent lors de cette étape). Dans ce langage, on représente un programme par un graphe de flot de contrôle dans lequel chaque nœud correspond à une instruction. Ceci est spécifique à CompCert : dans la plupart des compilateurs actuels, et en particulier gcc, un nœud représente un bloc d'instruction linéaire.

Ainsi, l'objectif de ce stage était de proposer une implémentation de cette représentation par des blocs d'instructions, afin d'en comparer les performances avec l'implémentation actuelle de RTL.

Nous commençons par présenter Coq, l'assistant de preuves utilisé, en section I, avant de préciser et formaliser la notion de compilateur vérifié en section II. Puis, nous détaillons la représentation initiale du langage RTL proposée par CompCert dans le paragraphe III-A, avant de détailler les modifications apportées afin d'obtenir un langage légèrement différent, mais dont la sémantique est équivalente, que nous appelons RTL_blocks (en paragraphe III-B). La section IV présente ensuite une implémentation d'une optimisation dans ce nouveau langage. Ce rapport s'achève par une discussion des résultats obtenus en section V.

I. Présentation de l'assistant de preuve Coq

Coq est un assistant de preuves interactif, fondé sur le calcul des constructions (CoC) et sur l'isomorphisme de Curry-Howard, grâce auquel on peut voir la preuve d'une proposition comme un programme dont le type est la proposition en question. Coq permet ainsi d'écrire des programmes et des preuves, ainsi que de vérifier formellement leur correction.

On peut trouver dans [2] la documentation officielle et dans [4] une introduction détaillée à l'utilisation de Coq, dont la théorie sous-jacente est détaillée dans [3], le premier ouvrage de référence concernant Coq.

En pratique, le langage utilisé est similaire à OCaml. Concernant les preuves, il s'agit d'appliquer une successions de « tactiques » prédéfinies (dans la bibliothèque standard ou manuellement), qui permettent de vérifier mécaniquement la preuve de la proposition. L'utilisation de l'environnement de développement dédié (CoqIDE), ou de Proof General (qui permet l'intégration de Coq dans l'éditeur de texte Emacs)

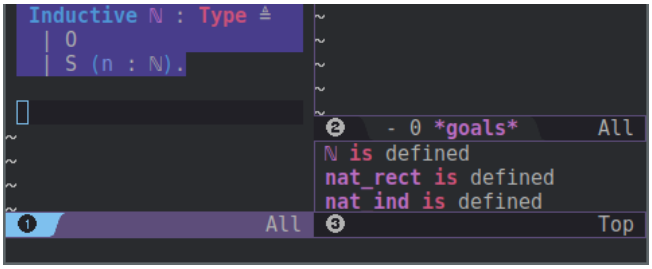

Fig. 1. Définition du type nat (l'ensemble des entiers)

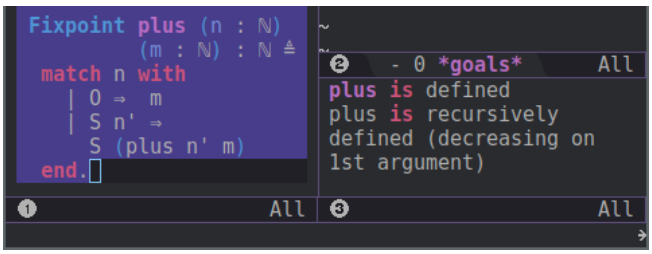

Fig. 2. Définition de l'addition sur les entiers

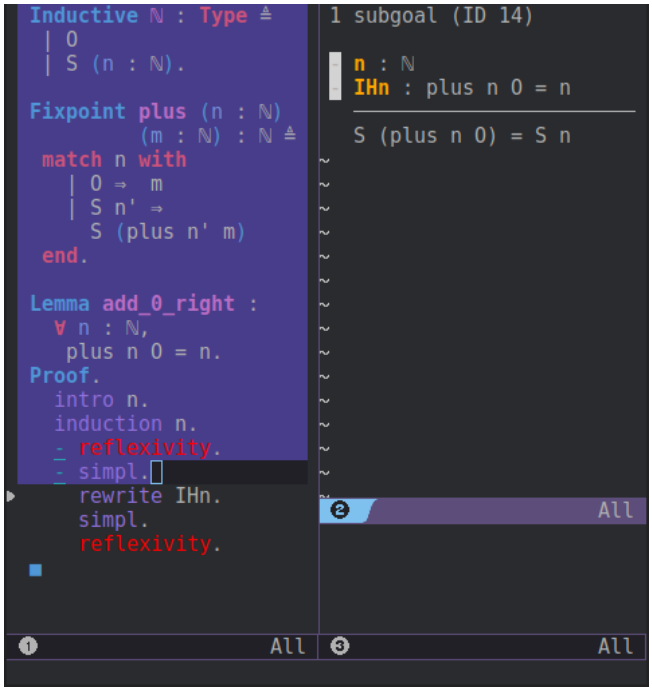

Fig. 3. Preuve de l'égalité $n+0=n$ pour n entier naturel

permet alors de vérifier pas à pas (d'où l'appellation d'assistant de preuve *interactif*) la preuve, en affichant à chaque étape les hypothèses courantes ainsi que le nom et le type des objets manipulés.

Par exemple, dans la figure 1, on définit le type `nat` ($\mathbb{N}$), de manière inductive.

Puis, dans la figure 2, on définit récursivement l'addition de deux entiers.

Enfin, on montre que $\forall n \in \mathbb{N}, n+0=n$ dans la figure 3. On y voit que la preuve est partiellement vérifiée (la partie vérifiée est recouverte en bleu) ; dans l'état actuel, après l'application de la tactique de simplification (`simpl`), Coq nous indique qu'on manipule un objet nommé n, de type « entier naturel », et qu'on dispose d'une hypothèse (l'hypothèse de récurrence fournie par la tactique `induction`), nommée `IHn`, d'après laquelle `plus n 0 = n`. Le but de la preuve est lui-aussi affiché : il s'agit de montrer que `S (plus n 0) = S n`.

Malgré la simplicité de ces exemples, Coq n'en reste pas moins un outil puissant, qui a permis, outre la finalisation de la preuve du « théorème des quatre couleurs » (voir [5]), la réalisation, par Xavier Leroy, d'un compilateur C écrit et vérifié en Coq, CompCert (présenté dans [7]).

II. Notion de compilateur vérifié

Expliquons à présent, à l'aide de l'article [6], ce que signifie le terme « vérifié » lorsque l'on qualifie CompCert de compilateur vérifié.

A. Notion de préservation sémantique

Considérons donc un programme source S, et notons C le code obtenu en compilant S. On s'intéresse alors à la *préservation sémantique*. Pour cela, en supposant qu'on se donne une sémantique sur le langage source et le langage cible, pouvant définir des comportements observables. Ainsi, lorsque S s'exécute avec le comportement observable B, on notera $S \Downarrow B$, où B peut signifier par exemple que le programme termine, ou diverge, ou rencontre une erreur.

La première idée de définition de la préservation sémantique lors de la compilation est la suivante :

$$\forall B, S \Downarrow B \Leftrightarrow C \Downarrow B.$$

Cependant, il ne s'agit pas d'une propriété que l'on souhaite imposer au compilateur. En effet, d'une part, le compilateur peut effectuer des choix lorsque le langage source est non-déterministe. D'autre part, éliminer du code menant à une erreur d'exécution (par exemple une

division par zéro), si celui-ci n'est pas utile dans la suite du code.

On obtient donc la définition suivante :

$$S \text{ safe} \Leftrightarrow (\forall B, C \Downarrow B \Rightarrow S \Downarrow B)$$

où S est dit *safe* lorsqu'aucun de ses comportements possibles ne mène à une erreur d'exécution.

Dans le cas de CompCert, les langages source et cibles, ainsi que les environnements d'exécution, sont *déterministes*. On peut donc remplacer la définition de la préservation sémantique ci-dessus par la définition équivalente suivante :

$$\forall B \notin \texttt{Wrong}, S \Downarrow B \Rightarrow C \Downarrow B$$

où `Wrong` désigne l'ensemble des comportements menant à une erreur.

On voit aisément que si cette propriété est vérifiée, alors les deux propriétés suivantes, souhaitées pour un compilateur, le sont également :

- la *correction du compilateur* :

  $$S \models \text{Spec} \Rightarrow C \models \text{Spec}$$

  où $S \models \text{Spec}$ signifie que S est *safe* et que tous les comportements de S satisfont la spéficication Spec ;
- *si S ne rencontre pas d'erreur, alors C non plus* :

  $$S \text{ safe} \Rightarrow C \text{ safe}.$$

B. Définition d'un compilateur vérifié

On peut à présent définir précisément ce qu'est un compilateur vérifié.

Tout d'abord, on modélise le compilateur par une fonction *Comp*, qui s'applique à un programme source S, et renvoie :

- soit un programme compilé C, noté $Comp(S) = \texttt{OK}(S)$;
- soit une erreur de compilation, notée $Comp(S) = \texttt{ERROR}$.

Les cas `ERROR` correspondent au cas où le compilateur affirme ne pas être capable de produire du code compilé, que ce soit en raison d'une erreur dans le programme source, ou parce que le programme excède ses capacités.

On dira qu'un compilateur *Comp* est *vérifié* s'il est accompagné d'une preuve formelle de la propriété suivante :

$$\forall S, C, \quad Comp(S) = \texttt{OK}(C) \Rightarrow S \approx C$$

où $\approx$ représente la préservation sémantique évoquée précédemment.

On remarque que le compilateur qui renvoie `ERROR` quel que soit le programme source, bien qu'inutile en pratique, vérifie la propriété. En fait, ce dont on cherche à s'assurer, c'est que le compilateur ne produit jamais du code « erroné » (c'est-à-dire ayant une sémantique différente du code source) silencieusement.

C. Composition des passes du compilateur et vérification

Puisque les compilateurs utilisent des langages intermédiaires et se décomposent donc en plusieurs étapes, appelées *passes*, on réutilise cette structure dans un compilateur vérifié. En effet, si on considère $Comp_1$ un compilateur du langage L_1 vers le langage L_2, et $Comp_2$ un compilateur de L_2 vers L_3, on peut définir:

```
Comp(S) = match Comp_1(S) with
    |ERROR → ERROR
    |OK(I)  → Comp_2(I)
```

et obtenir ainsi un compilateur de L_1 vers L_3. Il apparaît alors que si $Comp_1$ et $Comp_2$ sont vérifiés, alors *Comp* l'est également.

Ainsi, pour vérifier un compilateur, il suffit de vérifier chacune des passes du compilateur. Pour cela, on a le choix entre deux possibilités pour chaque passe :

- utiliser du code non vérifié, et fournir un validateur qui vérifiera a posteriori le code produit (le validateur devant à son tour être vérifié) ;
- prouver directement le code qui implémente la passe.

III. Représentations dans RTL et RTL_blocks

Intéressons nous à présent au langage intermédiaire RTL (dont l'implémentation commentée est disponible en [1]), utilisé dans CompCert. Notons au passage l'importance de ce langage : beaucoup d'optimisations sont effectuées à cette étape, comme représentées en figure 4.

A. La représentation RTL initiale

La représentation RTL consiste à représenter un programme par un graphe de flot de contrôle. Dans ce graphe, un nœud correspond à une instruction élémentaire. Plus précisément :

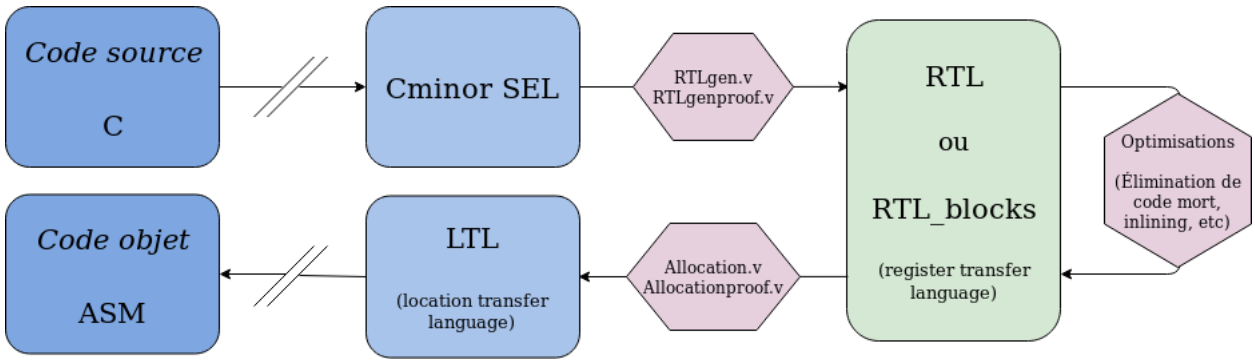

Fig. 4. Structure générale simplifiée de CompCert ; les hexagones représentent les étapes permettant de passer d'un langage intermédiaire à un autre

- les nœuds sont étiquetés par des entiers strictement positifs (le type `positive` en Coq) ;
- les instructions sont définies par le type inductif `instruction`, dans lequel chaque constructeur correspond à un type d'opération (effectuer une opération arithmétique, lire la valeur d'un registre, stocker la valeur d'un registre...), et prend en argument l'étiquette de l'instruction suivante à effectuer (ou des différentes instructions possibles lorsque le successeur dépend de l'évaluation d'une condition par exemple) ;
- un `code` est un `PTree.tree instruction`, c'est-à-dire une table d'association qui associe une `instruction` à chaque étiquette.

Ainsi, les instructions portent elles-même l'information de leur successeur, comme on peut le voir ci-dessous.

```
Inductive instruction : Type :=
  | Inop : node → instruction
  | Iop : operation → list reg → reg → node →
instruction
  | Iload : memory_chunk → addressing →
list reg → reg → node → instruction
  | Istore : memory_chunk → addressing →
list reg → reg → node → instruction
  | Icall : signature → reg + ident → list reg →
reg → node → instruction
  | Itailcall : signature → reg + ident →
list reg → instruction
  | Ibuiltin : external_function →
list (builtin_arg reg) → builtin_res reg →
node → instruction
  | Icond : condition → list reg → node →
node → instruction
  | Ijumptable : reg → list node → instruction
  | Ireturn : option reg → instruction.
```

Le type `instruction` permet ainsi de décrire dix opérations, dont certaines sont détaillées ci-dessous :

- `Inop succ` décrit l'absence d'opération et se branche simplement au successeur `succ` ;
- `Iop op args dest succ` décrit l'opération `op` sur les registres `args` (qui est une liste de registres, donc de type `list reg`), qui stocke le résultat dans le registre `dest` et se branche au successeur `succ` ;
- `Icall sig fn args dest succ` décrit l'opération qui appelle la fonction déterminée par `fn` (qui est soit un pointeur vers une fonction, soit un nom de fonction), de signature `sig`, l'applique sur les arguments `args`, stocke le résultat dans `dest` et se branche sur `succ` ;
- `Icond cond args ifso ifnot` décrit l'évaluation de la condition `cond` avec les valeurs des registres `args`, puis la transition vers `ifso` si la condition est vérifiée, et vers `ifnot` sinon ;
- `Ireturn` décrit la terminaison de l'exécution de la fonction courante, en renvoyant la valeur du registre donné en argument, où `Vundef` s'il n'y en a pas.

Enfin, une fonction est définie comme un type enregistrement, contenant la signature, la liste des registres liés aux valeurs des arguments, le nombre d'octets alloués dans la pile d'exécution, un graphe de flot de contrôle et un nœud d'entrée dans ce graphe.

D'un point de vue *sémantique*, on assimile l'exécution d'un programme à une séquence de transitions entre un état initial et un état final. Le type qui décrit les états a une définition complexe, dont les détails ne sont pas pertinents ici. On supposera donc qu'on dispose d'une sémantique pour chaque instruction, qu'on traitera donc telle quelle dans la suite. Cependant, la complexité de cette définition rend, par endroits, les preuves longues et techniques.

B. La nouvelle représentation : blocs d'instruction

Détaillons à présent les modifications apportées au langage RTL afin de créer un nouveau langage intermédiaire, RTL_blocks. Il s'agit ici de remplacer, dans le graphe de flot de contrôle, les instructions par des blocs d'instruction, et ce, dès que possible (c'est-à-dire dès lors qu'on a une succession linéaire d'instructions, sans conditions par exemple).

Notons que l'implémentation de ce nouveau langage est fortement inspirée de l'implémentation du langage LTL (le langage qui suit RTL dans l'implémentation de CompCert), qui manipule des blocs d'instructions, comme on peut le voir dans [1].

Dans ce langage, les `node` sont toujours étiquetés par des `positive` (on se contente de numéroter les nœuds).

En revanche, on implémente différement le type `instruction`. Dans cette représentation, une instruction ne porte plus l'information de son successeur. Ainsi, on supprime l'instruction `Inop`, et on ajoute une instruction `Ibranch`, qui prend en argument uniquement l'étiquette du nœud suivant. D'autre part, les instructions de branchement (`Icond`, `Ijumptable` et `Ireturn`) conservent en argument les étiquettes des nœuds correspondant aux successeurs. Ces instructions seront ainsi placées en fin de bloc.

On définit ensuite ce qu'est un « basic block » :

```
Definition bblock := list instruction.
```

Enfin, un `code` est un `PTree.tree bblock`, c'est-à-dire une table associant un bloc d'instruction à chaque étiquette.

Il suffit ensuite de réécrire la sémantique de ce nouveau langage, en s'inspirant de celle donnée pour RTL. En effet, étant donnée qu'on dispose d'une sémantique pour chaque instruction, il suffit de composer ces sémantiques pour obtenir la sémantique d'un bloc. Cela a posé de nombreux problèmes de typage dans les définitions, et rendu les preuves significativement plus complexes, puisqu'il fallait raisonner par récurrence.

IV. Un exemple d'optimisation dans le cas des blocs d'instructions : analyse de la durée de vie des variables

Comme mentionné plus haut, le langage RTL joue un rôle particulier dans CompCert, puisque de nombreuses optimisations sont réalisées à ce niveau (cela représente environ vingt passes). Nous avons souhaité en implémenter une avec ce nouveau langage RTL_blocks, et avons choisi l'analyse de la durée de vie des variables (implémentée dans le fichier `Liveness.v` dans CompCert). En pratique, il a d'abord fallu réaliser une version simplifiée de cette optimisation, en RTL, pour ensuite l'adapter à RTL_blocks, afin d'obtenir une passe relativement simple, et indépendante des autres passes.

A. Objectif de l'optimisation

Commençons par détailler quelques définitions utiles :

- une variable r est dite *activée* par une instruction i lorsque i lit une valeur dans la variable r (et on notera $r \in \mathbf{lues}(i)$) ;
- une variable r est dite *tuée* par une instruction i si i écrit une valeur dans la variable r (et on notera $r \in \mathbf{définies}(i)$) ;
- une variable r est *vivante immédiatement après* l'instruction i si et seulement si elle est *vivante immédiatement avant* l'une des instructions succédant à i ;
- si r n'est ni activée ni tuée par i, alors r est *vivante immédiatement avant* i si et seulement si elle est *vivante immédiatement après* i.

On note $\mathbf{vivant}_{\text{av}}(i)$ (respectivement, $\mathbf{vivant}_{\text{ap}}(i)$) l'ensemble des variables vivantes immédiatement avant (respectivement, après) l'instruction i.

On note également $\mathbf{succ}(i)$ l'ensemble des instructions succédant à i.

L'objectif est de déterminer la plus petite solution du système d'équations suivant :

$$\forall i, \begin{cases} \mathbf{vivant}_{\text{av}}(i) = \mathbf{lues}(i) \cup \left(\mathbf{vivant}_{\text{ap}}(i) \setminus \mathbf{définies}(i)\right) \\ \mathbf{vivant}_{\text{ap}}(i) = \cup_{j \in \mathbf{succ}(i)} \mathbf{vivant}_{\text{av}}(j) \end{cases}$$

Pour cela, on procède grâce à une analyse arrière, récursivement, en actualisant l'ensemble des variables vivantes au fur et à mesure. Cette analyse permet alors d'optimiser l'allocation des registres, ainsi que d'éliminer du code mort.

B. Implémentation

De la même manière que pour l'implémentation du langage RTL_blocks, l'adaptation des définitions (et notamment de la fonction de « transfert » qui met à jour la liste des variables vivantes) ne pose pas de problème majeur, en dehors de l'adaptation au traitement d'une

liste (qui nécessite donc des fonctions récursives et des fonctions intermédiaires).

Ainsi, on dispose d'une fonction `transfer_instr`, qui traite une unique instruction, d'une fonction `transfer_bblock` qui utilise la fonction `trnasfer_instr` pour traiter un bloc d'instruction, et enfin, d'une fonction `transfer` plus générale :

```
Definition transfer
           (f: function) (pc: node)
           (after: Regset.t) : Regset.t :=
  match f.(fn_code)!pc with
  | None ⇒
      Regset.empty
  | Some bb ⇒ transfer_bblock (bb) after
  end.
```

Les preuves de correction, en revanche, sont complexes et difficiles. En effet, elles nécessitent des raisonnements par récurrence, mais l'hypothèse de récurrence « naturelle » ne permet pas de conclure car pas assez générale. Il a donc fallu contourner ce problème en prouvant des énoncés plus forts.

V. Discussion

L'objectif initial était de comparer les performances des langages RTL et RTL_blocks en les testant sur une optimisation simple, l'analyse de durée de vie des variables, implémentée dans chacun des deux langages. En pratique, le manque de temps ne nous a pas permis d'explorer cet aspect. En effet, après avoir consacré du temps à l'apprentissage de Coq, la principale difficulté rencontrée a été d'explorer efficacement le code de CompCert. Il a également fallu renoncer à comprendre les détails d'implémentation de certains modules utilisés, et identifier précisément les parties de code qu'il était important de comprendre.

Afin de compléter ce travail, il faudrait tout d'abord intégrer RTL_blocks dans la chaîne des passes de CompCert (une ébauche de fichier permettant de générer du code RTL_blocks à partir de code RTL a été réalisée, mais la nécessité d'utiliser OCaml pour ceci n'a pas permis de finir l'intégration de ce fichier) afin de comparer les performances.

Enfin, il serait intéressant de prouver l'équivalence sémantique de RTL et RTL_blocks, mais cette preuve fait appel à des méthodes de bisimulation qu'il n'a pas été possible d'explorer pendant la durée du stage.

Conclusion

Finalement, le travail réalisé représente environ 1500 lignes de code, et propose une implémentation d'un nouveau langage intermédiaire, RTL_blocks, qui manipule des graphes de flot de contrôle dans lesquels les nœuds sont des blocs d'instruction. Une optimisation (une analyse de durée de vie des variables) a également été implémentée. Le code est disponible sur le dépot git suivant : https://gitlab.inria.fr/oradet/stage-l3-compcert.

En conclusion, si l'implémentation du langage reste relativement directe, la complexité des preuves se trouve accrue à cause de la présence des blocs d'instruction (représentés par des listes), d'où le choix initial d'éviter cette représentation, même si la réduction du nombre de nœuds laisse espérer un gain de performances lors de certaines optimisations.

References

[1] The compcert verified compiler. http://compcert.inria.fr/doc/.

[2] Reference manual of the coq proof assistant. https://coq.inria.fr/distrib/current/refman/.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag Berlin Heidelberg, 2004.

[4] Benjamin Pierce et al. Software foundations. https://softwarefoundations.cis.upenn.edu/, 2007.

[5] Georges Gonthier. A computer-checked proof of the four colour theorem. http://www2.tcs.ifi.lmu.de/~abel/lehre/WS07-08/CAFR/4colproof.pdf.

[6] Xavier Leroy. Formal verification of a realistic compiler. *Communications- ACM*, 52(7):107–115, July 2009.

[7] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert - a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.