Comment un chameau peut-il écrire un journal ?

Julien Signoles[1]

1: CEA, LIST, Laboratoire de Sûreté des Logiciels
91191 Gif-sur-Yvette Cedex
Julien.Signoles@cea.fr

Résumé

Dans *Frama-C*, plate-forme d'analyse de code *C* développée en *OCaml*, un journal est un script *OCaml* généré automatiquement et permettant de reproduire les actions utilisateurs, notamment effectuées *via* l'interface utilisateur. Outre la reproductibilité des résultats qui est nécessaire dans un contexte industriel soumis à des exigences de certification fortes comme la norme avionique DO-178C, un journal permet d'automatiser le pilotage de l'outil dans un contexte d'utilisation particulier. Cet article présente comment le mécanisme de génération du journal de *Frama-C*, appelé *journalisation* et requérant intrinsèquement de l'introspection, a été développé en *OCaml*, en combinant typage statique et dynamique.

1. Introduction

Frama-C [4] est une plate-forme d'analyse de code *C* développée en *OCaml* ayant vocation à être – et étant déjà – utilisée pour obtenir des garanties sur des programmes embarqués critiques dans un contexte industriel soumis à des contraintes de certification fortes, comme la norme avionique DO-178C [5]. Dans ce contexte, les outils d'analyse de code doivent fournir un moyen de reproduire les résultats obtenus lors d'une session utilisateur. Cette reproductibilité permet notamment de rejouer facilement les suites de tests et de tracer plus facilement la provenance des résultats obtenus (en particulier en cas de mauvais résultats, par exemple s'ils sont trop imprécis ou, pire, incorrects).

Pour satisfaire cette exigence, *Caveat* [2], l'ancêtre de *Frama-C* notamment utilisé lors du développement de l'A380, possède une notion de *journal*. Un journal est un script, ici en *OCaml*, généré automatiquement et permettant de reproduire toutes les actions utilisateurs, notamment effectuées au travers de l'interface utilisateur (GUI). Outre la reproductibilité requise par les processus de certification, un journal possède également l'avantage de permettre à l'utilisateur de créer facilement des scripts pour paramétrer l'outil. Automatiser ce pilotage de l'outil permet de gagner un temps important et, donc, de réduire les coûts. Grâce au journal, il est possible de réaliser ce processus une première fois manuellement sur un exemple précis, de récupérer le journal automatiquement généré, de le modifier éventuellement légèrement (pour, par exemple, le généraliser un peu), avant de l'exécuter autant de fois que nécessaire sur différentes applications. Le fait que le code *OCaml* du journal soit généré automatiquement permet, en outre, de réduire l'expertise requise à son écriture, tout particulièrement en *OCaml* et en connaissance des interfaces programmatiques (APIs) de l'outil. Ceci nécessite néanmoins que le code généré soit humainement lisible et raisonnablement détaillé.

Ces avantages de reproductibilité et d'automatisation demeurent en dehors de toute utilisation industrielle de l'outil. Par exemple, la reproductibilité permet d'obtenir de meilleurs rapports de bogues en provenance des utilisateurs car, en son absence, il est parfois difficile à ces derniers de décrire comment reproduire une erreur observée dans la GUI, à la suite d'un enchaînement d'événements consécutifs imprévus et survenus un peu par hasard. L'automatisation permet, quant à elle, de combiner facilement plusieurs analyses dans la GUI, pour en obtenir une nouvelle, plus puissante,

à faible coût. C'est tout particulièrement vrai pour *Frama-C*, fondé sur une architecture collaborative dans laquelle chaque analyseur est un greffon particulier pouvant notamment exploiter les résultats des autres, le noyau de *Frama-C* étant chargé de consolider les résultats globaux [3].

Pour ces différentes raisons, il est justifié de développer une notion de journal similaire à celle de *Caveat* au sein de *Frama-C*. Néanmoins, alors que *Frama-C* est développé en *OCaml*, le mécanisme de génération du journal dans *Caveat* – appelé *journalisation* – est programmé en *C++* et brise la sûreté du typage apportée par ce langage en abusant du transtypage (autrement dit, des *casts*). Le but de cet article est de montrer comment il a été possible d'obtenir une fonctionnalité similaire dans un langage à typage statique fort comme *OCaml*, en combinant typage statique et dynamique. L'implantation utilise en particulier, certes à faibles doses, types fantômes, types localement abstraits, types algébriques généralisés (GADTs) et récursion polymorphe. Par ailleurs, nous ne connaissons pas d'autres outils, en *OCaml* ou non, présentant un système similaire.

La section 2 offre une vue générale de la journalisation dans *Frama-C*. La section 3 présente les possibilités de typage dynamique de *Frama-C* utiles à notre étude. Enfin, la section 4 détaille comment ce mécanisme est implémenté.

2. Vue générale de la journalisation

Cette section explique les spécifications attendues de la journalisation (section 2.1), puis leurs déclinaisons dans le contexte de *Frama-C* (section 2.2), avant de présenter les principes de ce mécanisme au sein de cette plateforme (section 2.3) et de détailler un exemple (section 2.4).

2.1. Spécification informelle

Lors de l'exécution d'un logiciel, en particulier s'il bénéficie d'une GUI, un certain nombre d'actions sont entreprises par l'utilisateur, chacune d'elles déclenchant une série d'opérations internes engendrant un certains nombres d'effets dont certains sont présentés à l'utilisateur. Le but de la journalisation est d'instrumenter cette exécution, de manière à pouvoir générer un programme, appelé journal, reproduisant cette série d'opérations internes et, donc, engendrant les mêmes résultats.

Cette spécification générale de la journalisation permet d'en comprendre l'objectif global. Elle est néanmoins informelle et succincte et laisse ainsi la place à une liberté d'interprétation qu'il convient de préciser au cas par cas : qu'entend-on en particulier par *« opérations internes »*, *« effets »* et *« résultats observés »* ? La liberté laissée est nécessaire pour atteindre l'objectif initial qui est, rappelons-le, de pouvoir reproduire les résultats observés : l'exécution du journal doit aboutir à un résultat observationnellement équivalent à l'exécution dont il est issu, mais cette notion d'observabilité (*i.e.* d'équivalence observationnelle) dépend du contexte. Il est en effet peu probable que ce soit la même pour un programme encodant un protocole réseau échangeant des messages, pour un éditeur de texte ou pour un analyseur de code *C*.

Ce cadre général est néanmoins suffisant pour exprimer trois prérequis à remplir pour que l'exécution d'un journal J soit bien observationnellement équivalente à une exécution donnée d'un programme P dans un environnement d'utilisation particulier. D'abord, les effets produits par P doivent pouvoir l'être par J *(propriété P1)*, le moyen le plus simple pour cela étant que J ait accès à l'API de P. Ensuite, les résultats observés ne doivent dépendre que des effets : « mêmes effets, mêmes résultats » pourrait-on dire *(P2)*. Pour y parvenir, il faut que P et J soient déterministes, ce qui implique par exemple de fixer une même graine stable au générateur de nombres aléatoires ou de ne pas avoir plusieurs ordonnancements de processus possibles (à moins qu'un seul processus concerné n'ait un impact sur les résultats). Enfin, il faut disposer d'un état initial de J et d'un état initial de P équivalents avant d'entreprendre la première opération à écrire dans le journal *(P3)*, cette notion d'équivalence d'états dépendant de l'équivalence observationnelle préalablement mentionnée.

2.2. Contexte : *Frama-C*

Pour rendre le discours plus concret, plaçons nous désormais dans le cadre de *Frama-C*. Cet outil est une plate-forme extensible d'analyse de code *C* développée en *OCaml* (et uniquement en *OCaml* : l'emploi de *Camlp4* ou *Camlp5* est notamment prohibé) [5]. L'extensibilité est obtenue grâce à une architecture modulaire fondée sur des greffons pouvant communiquer entre eux *via* un noyau central. Elle est schématisée figure 1. Le noyau fournit également, d'une part, l'arbre de syntaxe abstraite du code *C*, annoté en *ACSL* [1], qui doit être analysé et, d'autre part, un certain nombre de services de base, sous forme d'autant de bibliothèques dédiées, dont le journal fait parti.

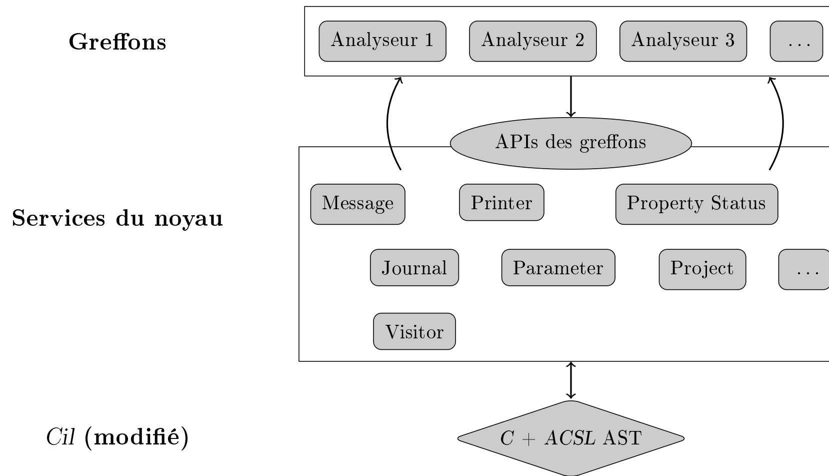

FIGURE 1 – Vue fonctionnelle de *Frama-C*.

Les greffons sont des programmes *OCaml*, chargés le plus souvent dynamiquement, ayant accès aux APIs du noyau et à celles des autres greffons [9]. Ce sont en général des analyseurs de code (analyse de valeurs par interprétation abstraite, preuve de programmes par calcul de plus faible précondition, transformations de programmes dédiées à la vérification de propriétés temporelles ou à la vérification d'annotations à l'exécution, ...) [4], mais rien n'empêche par exemple de programmer un serveur web comme un greffon *Frama-C*[1]. Il est également possible de développer des greffons légers, appelés *scripts*, qui requièrent un peu moins de développement (pas de `Makefile`, pas de pré-compilation nécessaire, ...) au prix de fonctionnalités légèrement bridées (un seul fichier *OCaml*, ...). Ces scripts sont parfaits pour développer de petites extensions de la plateforme, pour conduire des expérimentations ou piloter rapidement et automatiquement des analyses.

Dans ce contexte, il est apparu naturel de proposer à l'utilisateur le journal comme un script *Frama-C*. En effet, les trois prérequis *P1*, *P2* et *P3* mentionnés préalablement sont alors remplis : le script a accès aux APIs de *Frama-C (P1)*, *Frama-C* garantit que les mêmes effets conduisent aux mêmes résultats *(P2)* et sa phase d'initialisation, quoique complexe, assure que le chargement du script interviendra au moment adéquat pour être dans le bon état initial *(P3)*. Nous reviendrons néanmoins plus précisément sur ces points ci-après.

Nous pouvons maintenant clarifier pour *Frama-C* et son langage support *OCaml* ce que nous entendons par *« opérations internes »*, *« effets »* et *« résultats observés »* dont nous avons discutés en section 2.1. Les opérations internes sont des fonctions *OCaml* exportées par l'API de *Frama-C*, de manière à pouvoir être appelées par le journal et à garantir *P1*. Les résultats observés sont ceux affichés sur la sortie standard lorsque *Frama-C* est lancé en mode console (ils sont aussi présents

1. D'ailleurs, cela a déjà été réalisé pour expérimenter une GUI de *Frama-C* alternative dans un navigateur.

dans un des panneaux de la GUI). Ils comprennent notamment les résultats de toutes les analyses effectuées. Par ailleurs, *Frama-C* possède un état global constitué de valeurs mutables bien identifiées, dont une bibliothèque du noyau possède une vue complète [7]. Les résultats fournis par la plateforme ne dépendent que de cet état global et des éventuelles entrées (options de ligne de commande, contenu des fichiers lus, de certaines variables d'environnement, ...). Dès lors, les effets sont les modifications apportées à une des valeurs mutables constituant l'état global et, couplé au fait que *Frama-C* est déterministe, cela garantit *P2* et même plus (notamment, tous les composants de la GUI dépendants de l'état global de *Frama-C* – ce qui inclut toutes les cases à cocher, zones textuelles et autres composants graphiques permettant de paramétrer l'outil – sont automatiquement positionnés dans le bon état après lecture du journal). En outre, on peut considérer que le journal est dynamiquement chargé par *Frama-C* suffisamment tôt pour qu'aucune modification de l'état global n'ait pu survenir (voir [9], section 4.14), ce qui garantit *P3* [2].

2.3. Principe de la journalisation dans *Frama-C*

La génération d'un journal requiert, d'une manière ou d'une autre, d'instrumenter le programme pour *intercepter suffisamment* d'opérations afin de capturer l'ensemble des effets ayant un impact sur les résultats à observer. Ensuite, il faut *modifier* l'exécution de chacune de façon à provoquer aussi son écriture dans le journal. Dans *Frama-C*, il faut donc *intercepter* et *modifier* le comportement de *suffisamment* de fonctions de son API pour capturer toute modification de son état global.

Dès lors, trois questions se posent. D'abord, comment intercepter les opérations ? Ensuite, comment les modifier ? Enfin, comment faire pour qu'il y en ait suffisamment afin de ne rater aucun effet ?

La bibliothèque présentée ici apporte une réponse simple : il est de la responsabilité du développeur de choisir les fonctions à journaliser, c'est-à-dire à écrire dans le journal, *via* un appel à `Journal.register`, qui prend en argument une fonction à journaliser, et l'enregistre comme telle, en effectuant au passage les modifications nécessaires. Celles-ci constituent le cœur de la journalisation et sont décrites en section 4. Nous expliquons aussi en fin de section comment *Frama-C* ne laisse pas, malgré tout, l'entière responsabilité du choix des fonctions à journaliser au développeur. La réponse à la dernière question est plus compliquée, en particulier dans le cas d'un logiciel de la taille de *Frama-C* (près de 150 *kloc* dans sa version libre, avec un état global constitué de près de 1000 valeurs mutables). En considérant le graphe d'appel complet de *Frama-C*, la condition à remplir est la suivante :

> si l'état global de *Frama-C* est modifié pendant l'appel à une fonction f, il doit y avoir au moins une fonction journalisée sur tout chemin entre le point d'entrée de *Frama-C* et f *(P4)*.

Notons que *P4* impose d'avoir *au moins* une fonction journalisée par chemin d'exécution, mais ne requiert pas d'en avoir *exactement* une. Le cas où plusieurs appels de fonctions journalisées sont imbriqués est détaillé en section 4.4.

Deux cas extrêmes peuvent facilement être considérés pour garantir *P4* : soit journaliser le point d'entrée de *Frama-C* exécuté à l'issue de la phase d'initialisation, soit journaliser les fonctions feuilles produisant directement les effets sur l'état global. Malheureusement, aucune de ces deux solutions n'est satisfaisante. Dans le premier cas, le journal ne contiendrait alors qu'un seul appel de fonction, celui du point d'entrée, ce qui rend caduque son utilisation comme support à l'écriture de script par un utilisateur et, surtout, ne permet pas de rendre compte des actions utilisateurs dans la GUI. Dans le

2. En réalité, ce n'est pas totalement exact : certains effets très particuliers ont lieu très tôt lors de la phase d'initialisation, comme par exemple le fait de passer en mode débogage, et peuvent avoir une influence sur cette dernière en terme d'affichages produits, violant ainsi *P3* : même si ces effets sont enregistrés dans le journal, celui-ci n'étant pas encore chargé en mémoire, ces affichages ne peuvent pas avoir lieu lors de son exécution. Néanmoins, aucun de ces affichages n'est critique. En outre, la première action effectuée par le journal sera d'exécuter ces effets, rétablissant ainsi *P3* en prenant comme état initial de P celui obtenu à la fin de la phase d'initialisation et comme état initial de J celui obtenu à la fin de cette action préliminaire.

second cas, il devient nécessaire d'exporter dans l'API de *Frama-C* l'ensemble des fonctions feuilles de bas niveau produisant des effets, ce qui pose un grave problème en terme de préservation d'invariants internes et de maintenabilité de l'outil.

Il convient donc d'apporter une solution pragmatique intermédiaire, qui ne laisse pas la responsabilité de la correction du journal aux seuls contributeurs, notamment extérieurs (dont le nombre croît et est non borné du fait du caractère ouvert de la plateforme). L'architecture de *Frama-C* nous offre ici une solution simple. Comme les greffons doivent enregistrer leur API dans le noyau, il suffit principalement de journaliser ces fonctions ou, en tout cas, celles d'entre elles ayant un effet sur l'état global. En pratique, les fonctions configurant les greffons sont ainsi automatiquement journalisées, tandis que la journalisation des autres consiste juste à positionner un drapeau booléen au moment de l'enregistrement de chacune d'elles dans le noyau. Le choix de journalisation une fonction ou non est en outre facile à effectuer selon qu'elle effectue des calculs utiles à un analyseur ou qu'elle se contente d'exploiter un résultat calculé par ailleurs. Pour être tout à fait complet, il convient aussi de journaliser quelques fonctions du noyau accessibles directement depuis la GUI en dehors de toute analyse, mais leur nombre demeure limité.

2.4. Exemple

Voici à présent un exemple permettant de rendre le journal plus concret. Nous commençons par lancer la GUI de *Frama-C* sur les fichiers `.c` du répertoire courant avec la ligne de commande suivante.

```
$ frama-c-gui -journal-enable *.c
```

L'option `-journal-enable` permet de déclencher la génération du journal. Sans cette option, il n'est généré que lorsque l'utilisation de la GUI provoque une erreur interne. Une fois la GUI lancée, nous exécutons l'analyse de valeurs par interprétation abstraite, puis nous sélectionnons une instruction s et formulons une requête de slicing demandant à générer un programme C préservant la sémantique de s : un nouveau programme C est alors généré dans un nouveau projet *Frama-C*, tandis que les instructions conservées sont mises en évidence à l'écran. Nous nous déplaçons à présent dans le nouveau projet pour inspecter le code généré, puis quittons *Frama-C*. À cet instant, le journal est créé dans le fichier `frama_c_journal.ml` dont la fonction principale est présentée figure 2.

Cette fonction reproduit les effets des actions utilisateurs sur l'état global de *Frama-C* en appelant des fonctions de son API. On peut notamment constater que le code est mis en forme et raisonnablement lisible, que des structures de données complexes peuvent être générées (listes, tables d'association, types de données internes à *Frama-C*), que des variables locales sont créées si nécessaire, puis utilisées et que les étiquettes et les arguments optionnels sont correctement gérés. Pour rejouer la session utilisateur, il suffit maintenant d'exécuter la commande suivante.

```
$ frama-c-gui -load-script frama_c_journal.ml
```

3. Typage dynamique dans *Frama-C*

Le mécanisme de journalisation détaillé dans cet article s'appuie sur la bibliothèque de typage dynamique de *Frama-C*. Nous nous contentons ici d'en rappeler les caractéristiques utiles à notre étude. Le lecteur curieux peut se référer à [8] pour plus de détails, notamment concernant son implantation.

Pour chaque type τ, cette bibliothèque permet de construire une unique *valeur de type* de type τ `Type.t` représentant à l'exécution le type τ. Elle fournit des valeurs de type pour les types primitifs,

3. Ici, les commentaires triplement étoilés `(*** ... ***)` ont été ajoutés manuellement *a posteriori* afin d'expliquer le code généré, mais il pourrait l'être automatiquement. Par ailleurs, le journal complet, incluant le code ne variant pas d'une génération à l'autre, est en annexe A.

```
(* Run the user commands *)
let run () =
  (*** positionnement des fichiers .c à analyser ***)
  Dynamic.Parameter.StringList.set ""
    [ "CruiseControl.c"; "CruiseControl_const.c" ];
  (*** construction de l'AST à partir des fichiers .c donnés ***)
  File.init_from_cmdline ();
  (*** exécution de l'analyse de valeurs ***)
  !Db.Value.compute ();
  (*** construction d'une requête de slicing ***)
  let cil_datatype__Varinfo__t_map =
    !Db.Slicing.Select.select_stmt
      Db.Slicing.Select.empty_selects
      ~spare:false
      (fst (Kernel_function.find_from_sid 306))
      (Globals.Functions.find_by_name "CruiseControl")
  in
  (*** construction d'un projet de slicing ***)
  let sl_project_Slicing1 = !Db.Slicing.Project.mk_project "Slicing1" in
  (*** ajout de la requête précédente au projet de slicing ***)
  !Db.Slicing.Request.add_persistent_selection
    sl_project_Slicing1
    cil_datatype__Varinfo__t_map;
  (*** application de la requête de slicing ***)
  !Db.Slicing.Request.apply_all_internal sl_project_Slicing1;
  (*** suppression des fonctions non appelées dans le code généré ***)
  !Db.Slicing.Slice.remove_uncalled sl_project_Slicing1;
  (*** création du nouveau projet contenant le code généré ***)
  let p_Slicing1__export =
    !Db.Slicing.Project.extract "Slicing1 export" sl_project_Slicing1
  in
  (*** changement de projet courant ***)
  Project.set_current p_Slicing1__export;
  ()
```

FIGURE 2 – Exemple de code généré dans le journal. [3]

tout en permettant au développeur d'en créer de nouvelles. Ainsi, la valeur `Datatype.int`, de type `int Type.t`, représente le type `int`, tandis que la valeur de type `Datatype.func (Datatype.list Datatype.string) Datatype.unit`, de type `(string list → unit) Type.t`, représente le type `string list → unit`. Cette bibliothèque permet de représenter n'importe quel type, y compris les types abstraits en préservant l'abstraction, à l'exception des types polymorphes pour lesquels des générateurs de valeurs de type pour leurs instances monomorphes sont fournis. Les étiquettes et les arguments optionnels sont également pris en charge pour les valeurs de type fonctionnelles.

Grâce à ces valeurs de type, la bibliothèque propose des fonctions génériques bien typées. On trouve ainsi la fonction `Datatype.pretty_code` de type α `Type.t` → `Format.formatter` → α → `unit` qui permet d'afficher une valeur sous la forme d'un code *OCaml* correct. Le type dynamique permet de sélectionner la fonction d'affichage adéquate en fonction de la valeur de type v fournie, le typeur d'*OCaml* garantissant que v coïncide avec le type de l'argument à afficher. Le développeur peut définir cette fonction pour chacun de ses propres types de données. Dans le cas des types abstraits, le

code généré ne doit dépendre que des fonctions visibles dans l'API. Par exemple, la fonction d'affichage des `kernel_function` de *Frama-C*, représentant les fonctions *C* annotées, génère un code appelant `Globals.Functions.find_by_name` qui recherche une `kernel_function` à partir de son nom :

```
let pretty_code fmt kf =
  Format.fprintf
    fmt "Globals.Functions.find_by_name \"%s\"" (Kernel_function.get_name kf)
```

Différentes tables hétérogènes sont également proposées. Par exemple, l'interface des tables hétérogènes sur les clés et polymorphiquement homogènes sur les valeurs associées est la suivante [4].

```
module Obj_tbl: sig
  type α t                                        (* α = type des valeurs associées *)
  val create: unit → α t
  val add: α t → β Type.t → β → α → unit
  val find: α t → β Type.t → β → α
end
```

Ici, la valeur de type v fournie en argument aux fonctions `Obj_tbl.add` et `Obj_tbl.find` permet de garantir statiquement que le type de la clé coïncide avec v au moment de l'application de la fonction.

Par ailleurs, la bibliothèque fournit quelques possibilités supplémentaires. Ainsi, la fonction `Type.Function.get_gadt_instance` de type α `Type.t` $\rightarrow$ α `Type.Function.gadt_instance` permet d'obtenir, pour une valeur de type fonctionnelle représentant un type $\tau_1 \rightarrow \tau_2$, les valeurs de type correspondant à τ_1 et τ_2, ainsi que les éventuels étiquette et valeur par défaut associés à l'argument. Le type α `Type.Function.gadt_instance` est ici un type algébrique gardé (GADT) [10, 6] de manière à établir un lien entre le type fonctionnel et les types de son argument et de sa valeur de retour : [5]

```
type _ gadt_instance =
  | No_instance
  | Instance: α Type.t × β Type.t → (α → β) gadt_instance
```

Ce GADT permet de garantir statiquement que la fonction `Type.Function.get_gadt_instance` ne peut renvoyer le constructeur `Instance` que si elle est appelée avec, comme argument, une valeur d'un type $(\tau_1 \rightarrow \tau_2)$ `Type.t`. Il garantit alors que ses première et deuxième composantes sont respectivement des valeurs de type τ_1 `Type.t` et τ_2 `Type.t`, correspondant aux valeurs de type de l'argument et du résultat, respectivement. Dans ce cas particulier, l'implémentation (non montrée) de la fonction `Type.Function.get_gadt_instance` garantit aussi la réciproque : si elle renvoie `No_instance`, alors son argument n'est pas une valeur de type fonctionnelle.

4. Journaliser des fonctions *OCaml*

La modification des fonctions permettant l'écriture de leur appel dans le journal est effectuée au moment de leur enregistrement dans la bibliothèque de journalisation *via* la fonction `Journal.register`. Cette fonction est un *wrapper* prenant en argument une fonction quelconque et renvoyant une fonction observationnellement équivalente *modulo* l'écriture supplémentaire dans le journal. Idéalement, son type attendu serait donc le suivant.

4. Seules les fonctions principales sont présentées ici. Plus de détails sur l'implantation des tables hétérogènes sont fournis dans l'article associé [8].

5. La version actuelle de *Frama-C* (Fluorine-20130601) doit être compatible avec la version 3.12.1 d'*OCaml*. Elle ne peut donc pas utiliser de GADTs, seulement introduits dans la version 4 du compilateur. Ainsi, elle utilise actuellement une version sans GADT, mais comprenant quelques occurrences de `Obj.magic` pour contourner des problèmes de typage.

```
val register: (α → β) → α → β
```

Il est en réalité légèrement différent, comme nous allons l'expliquer. Plus généralement, plutôt qu'introduire l'algorithme de journalisation *in extenso*, nous allons partir du cas le plus simple (les fonctions de type `unit → unit`) pour introduire progressivement les cas plus généraux et complexes.

4.1. Fonctions de type `unit → unit`

La première difficulté consiste à imprimer le nom de chaque fonction f à journaliser : ce n'est pas possible en *OCaml*. Pour la contourner, nous demandons au développeur de fournir le nom de f tel qu'il apparaît dans l'API du logiciel. Le type de `Journal.register` devient alors le suivant.

```
val register: string → (unit → unit) → unit → unit
```

Dès lors, coder cette fonction est en fait très simple, en supposant l'existence d'une valeur `fmt` de type `Format.t` correspondant au canal d'écriture considéré (le journal) [6].

```
(* fonction d'écriture dans le journal *)
let print_sentence name fmt = Format.fprintf fmt "%s ();" name

let register name f () =
  (* écriture dans le journal *)
  print_sentence name fmt;
  (* exécution de la fonction journalisée *)
  f ()
```

Prenons l'exemple de la fonction du noyau de *Frama-C* `File.init_from_cmdline: unit → unit`, utilisée dans la figure 2, et qui génère un AST à partir des fichiers *C* fournis préalablement. Journaliser cette fonction revient à exécuter le code suivant.

```
(* file.ml *)
let init_from_cmdline () = ...
let init_from_cmdline = Journal.register "File.init_from_cmdline" init_from_cmdline
```

Le fait de n'appliquer que partiellement la fonction `Journal.register` est fondamental : la fermeture renvoyée correspond à la fonction journalisée modifiée qui, à chaque application, effectue son effet et l'écrit dans le journal.

En pratique, plutôt que d'écrire directement dans le canal `fmt`, nous utilisons une file enregistrant les commandes d'affichage à exécuter, pour ensuite générer le journal à la fin de l'exécution. Le code modifié de `Journal.register` devient alors le suivant.

```
(* générateur d'instructions dans le journal *)
module Sentences: sig
  val add: (Format.formatter → unit) → unit          (* ajout d'une instruction *)
  val write: Format.formatter → unit       (* écriture de toutes les instructions *)
end = struct
  let sentences: (Format.formatter → unit) Queue.t = Queue.create ()
  let add pp = Queue.add pp sentences
  let write fmt = Queue.iter (fun pp → pp fmt) sentences; Format.fprintf fmt "()"
```

6. Par soucis de clarté, ce code ne prend pas en compte le fait que l'option `-journal-enable` ait été ou non positionnée : on suppose que c'est toujours le cas.

```
end

let register name f () =
  Sentences.add (print_sentence name);
  f ()
```

4.2. Arguments d'un type quelconque

Lorsque l'argument de la fonction à journaliser n'est pas `()`, son impression dépend de son type et de sa valeur à l'exécution. Pour effectuer cette impression, on utilise la fonction `Datatype.pretty_code` de la bibliothèque de typage dynamique présentée en section 3 pour afficher des arguments quelconques. Cela requiert néanmoins de prendre un type dynamique en argument.

```
let print_sentence name ty_x x fmt =
  Format.fprintf fmt "%s %a;" name (Datatype.pretty_code ty_x) x

let register name ty_x f x =
  Sentences.add (print_sentence name ty_x x);
  f x
```

Le type de la fonction `Journal.register` est donc le suivant.

```
val register: string → α Type.t → (α → unit) → α → unit
```

Ainsi, la fonction du noyau de *Frama-C* `Project.set_current: Project.t → unit` qui change le projet courant est-elle journalisée de la manière suivante.

```
(* project.ml *)
type t = ...                                          (* type des projets *)
let ty: t Type.t = ...                       (* valeur de type des projets *)
let set_current prj = ...
let set_current = Journal.register "Project.set_current" ty set_current
```

Arguments fonctionnels Les valeurs de certains types de données ne peuvent néanmoins pas être affichés sous la forme de code *OCaml* sans indication externe. C'est tout particulièrement le cas des fonctions, pour lesquelles il faudrait être en mesure de générer du code pour les fermetures. Ici, nous utilisons une table hétérogène globale, fournie par la bibliothèque de typage dynamique, de manière à associer à chaque valeur non affichable une chaîne de caractères la représentant : cette dernière est utilisée pour l'écriture dans le journal. Par exemple, pour les fonctions, cette chaîne est celle apparaissant dans l'API du logiciel. À ce stade, cette table ne contient d'ailleurs que des fonctions, mais d'autres valeurs non fonctionnelles y seront ajoutées en section 4.3. Même si elle grossit de manière monotone, sa taille demeure petite en pratique.

```
(* associe une chaîne à une valeur *)
module Binding: sig
  val add: α Type.t  → α → string  → unit
  val find: α Type.t  → α → string
end = struct
  let bindings: string Type.Obj_tbl.t = Type.Obj_tbl.create ()
  let add ty v var = Type.Obj_tbl.add bindings ty v var
  let find ty v = Type.Obj_tbl.find bindings ty v
end
```

Les fonctions journalisées sont automatiquement ajoutées dans cette table. Les autres éventuelles doivent l'être manuellement par le développeur, mais cela demeure exceptionnel dans *Frama-C*[7]. La fonction d'affichage des appels journalisés est maintenant la suivante.

```
let pp ty fmt x =
  try
    let name = Binding.find ty x in
    (* utiliser la chaîne pré-enregistrée si elle existe *)
    Format.fprintf fmt "%s" name
  with Not_found →
    (* utiliser l'afficheur par défaut sinon *)
    Datatype.pretty_code ty fmt x;

(* utiliser pp plutôt que l'afficheur générique *)
let print_sentence name ty_x x fmt = Format.fprintf fmt "%s %a;" name (pp ty_x) x
```

En outre, la fonction d'enregistrement associe aussi le nom à la fermeture *via* le module `Binding`.

```
let register name ty_x f x =
  (* auto-enregistrement de la fonction journalisée *)
  Binding.add (Datatype.func ty_x Datatype.unit) f name;
  Sentences.add (print_sentence name ty_x x);
  f x
```

Polymorphisme Il s'agit de la principale limitation de la journalisation : la bibliothèque de typage dynamique ne supportant pas les types polymorphes (mais uniquement leurs instances monomorphes), il n'est pas possible de journaliser de fonctions polymorphes. Néanmoins, de telles fonctions ne peuvent de toute façon pas être dans les APIs des greffons [8] qui définissent la (quasi) totalité des fonctions à journaliser. En outre, les greffons représentent des analyseurs : ils n'ont pas spécialement vocation à être générique. De ce fait, l'absence de polymorphisme n'est pas un problème pratique dans *Frama-C*.

4.3. Valeurs de retour

Si une fonction journalisée f renvoie une valeur v, il est possible qu'une autre fonction journalisée g la prenne en argument. Dans ce cas, créer une nouvelle valeur v' observationnellement équivalente à v n'est pas nécessairement correct, en particulier si v est représentée à l'exécution par un pointeur. La seule représentation correcte consiste à lier le résultat de f à l'argument de g à l'aide d'une variable locale pour préserver l'égalité physique ==.

Pour ce faire, nous allons utiliser à nouveau le module `Binding`, introduit à la section 4.2 : lorsque le type de retour de la fonction n'est pas `unit`, nous générons une nouvelle variable locale pouvant être utilisée dans la suite du journal. En supposant l'existence d'un générateur de noms de variables *OCaml* `gen_binding: unit → string`, la fonction `print_sentence` est modifiée comme suit.

```
let print_sentence name ty_x x fmt =
  (* ancienne version de la fonction *)
  let pp_sentence fmt = Format.fprintf fmt "%s %a" name (pp ty_x) x in
  if Type.equal ty_x Datatype.unit then Format.fprintf fmt "%t;" pp_sentence
```

7. Dans la version actuelle de *Frama-C*, on ne trouve qu'un unique ajout non automatique. Il s'agit d'une fonction non journalisée pouvant être donnée en argument d'une journalisée d'ordre supérieur. Le cas est rare car les fonctions journalisées sont les fonctions de haut niveau, qui sont, d'une part, rarement d'ordre supérieur et, d'autre part et de par l'architecture de *Frama-C*, rarement appelées avec pour argument une fonction de (plus) bas-niveau.

```
  else
    (* ajoute une liaison locale *)
    let v = gen_binding () in
    Binding.add ty_x x v;
    Format.fprintf fmt "let %s = %t in " v pp_sentence
```

Le corps de la liaison locale est généré par la suite de l'exécution (dans le cas de la fin d'exécution, un `()` terminal est ajouté par la fonction `Sentences.write` de la section 4.1). Aussi le test effectué ici n'est pas strictement indispensable, mais il le deviendra dans la section 4.6. Il permet en outre de générer un code plus lisible qu'un `let () = ... in ...`.

4.4. Appels de fonctions elles-mêmes journalisées

Dans la section 2.3, nous avons expliqué qu'il fallait journaliser suffisamment de fonctions pour ne rater aucun effet produit par le logiciel. En contrepartie, le risque est d'avoir *trop* d'appels à des fonctions journalisées écrites dans le journal. En effet, considérons le cas d'une fonction f journalisée appelant une autre fonction g elle-même journalisée : lors d'un appel à f avec un argument x, $f\ x$ est écrit dans le journal, puis l'appel est effectivement exécuté engendrant l'appel de la fonction g à un argument y, générant à son tour l'écriture dans le journal de $g\ y$ et l'appel correspondant. *In fine*, le journal généré contient la séquence `f x; g y` et son exécution provoque l'exécution de f, incluant celle de g, puis une nouvelle exécution de g, ce qui n'est pas équivalent au programme initial : la fonction g a été exécutée une fois de trop.

Pour résoudre ce problème, nous suspendons l'écriture dans le journal des fonctions journalisées lorsqu'une est déjà en cours d'exécution, grâce à une référence additionnelle.

```
(* !started vaut true ssi une fonction journalisée est en cours d'appel *)
let started = ref false
let register name ty_x f x =
  Binding.add Datatype.func ty_x Datatype.unit f name;
  if !started then f x
  else begin
    (* n'écrire que s'il n'y a pas déjà d'écritures en cours *)
    Sentences.add (print_sentence name ty_x x);
    (* maintenant, la journalisation est en cours *)
    started := true;
    f x;
    (* restauration de la valeur initiale *)
    started := false
  end
```

Notons que l'utilisation de cette référence ne permet pas d'utiliser le journal dans un contexte concurrent. Néanmoins, nous nous sommes déjà placés dans un cadre déterministe (cf. section 2.1).

4.5. Exceptions

Une des premières utilisations du journal est de reproduire les erreurs survenues au cours de l'exécution. En *OCaml*, ces erreurs correspondent à des levées d'exceptions. La journalisation doit les prendre doublement en compte : d'une part, la levée d'une exception ne doit pas entraver l'écriture dans le journal et, d'autre part, le code généré doit prendre en compte proprement cette exception dont on sait à l'avance qu'elle surviendra lors de son exécution.

Pour continuer à écrire dans le journal en présence de comportements exceptionnels, il faut aussi effectuer le post-traitement dans ces cas-là. Aussi, pour prendre en compte les exceptions, lorsqu'une est levée, nous devons la rattraper dans le code produit de manière à générer la suite de l'exécution dans le cas exceptionnel, tandis que la suite du cas nominal n'est jamais exécutée : on peut donc y générer un `assert false`. Le code écrit dans le journal est donc différent selon que la fonction journalisée lève ou non une exception. Le code est donc maintenant généré *après* l'exécution de la fonction, selon deux schémas différents, et non plus avant. On obtient donc le code suivant.

```
let register name ty_x f x =
  Binding.add Datatype.func ty_x Datatype.unit f name;
  if !started then f x
  else begin
    started := true;
    try
      f x;
      (* cas nominal: aucune exception levée, comme précédemment *)
      Sentences.add (print_sentence name ty_x x);
      started := false
     with exn →
       (* cas exceptionnel: gérer l'exception dans le journal *)
       Sentences.add
         (fun fmt →
            Format.fprintf fmt "try %t assert false with exn (* %s *) → "
              (print_sentence name ty_x x)
              (Printexc.to_string exn));
       started := false;
       (* re-lever l'exception pour préserver le comportement de la fonction *)
       raise exn
  end
```

À présent, les exceptions sont gérées correctement par le journal. Par exemple, considérons le schéma de code suivant dans lequel les fonctions `f` et `g` sont journalisées mais par `h`.

```
let f () = ... raise E ...
let g () = ...
let h () = try ... f () ... with E → g ()
```

Si l'exception `E` est effectivement levée lors de l'appel à `f` effectué *via* `h`, alors le journal généré contiendra le code suivant.

```
let run () =
  ...
  try f (); assert false with exn (* E *) → g (); ...
```

Néanmoins, si l'exception `E` n'était jamais rattrapée (et notamment pas dans `h`), le logiciel serait arrêtée sur la levée de cette exception et l'exécution du journal reproduirait ce comportement. C'est le comportement initialement souhaité : le journal a le même comportement que le programme dont il est issu. Cependant, d'une part il devient difficile pour l'utilisateur de distinguer le cas où le journal reproduit l'exception initiale et le cas éventuel où l'exécution du journal lève une exception car un problème quelconque est survenu (même si ce n'est pas censé survenir) et, d'autre part, arrêter l'exécution du logiciel sur une exception alors qu'on a obtenu le comportement attendu (le journal a reproduit le comportement initial) n'est pas souhaitable.

Pour cette raison, si le programme initial lève une exception `E` jamais rattrapée, le code généré dans le journal la lève à son tour, mais encapsulée dans une autre – l'exception `Exception E` – de façon à pouvoir la distinguer des autres (par exemple, celles levées par une exécution du journal qui ferait échouer le logiciel pour une raison inconnue). Ainsi, la fonction principale du journal [8], qui appelle la fonction `run`, peut rattraper cette exception particulière pour afficher un message à l'utilisateur (indiquant que le programme initial avait levé telle exception), avant d'arrêter proprement le logiciel. Pour générer la levée de cette exception spéciale, le module `Sentences` est modifié de la façon suivante.

```
module Sentences: sig
  val add: (Format.formatter → unit) → bool (* exception levée ? *) → unit
  val write: Format.formatter → unit
end = struct
  type t = { pp: Format.formatter → unit; raise_exn: bool }
  let sentences: t Queue.t = Queue.create ()
  let add pp exn = Queue.add pp { pp = pp; raise_exn = exn } sentences
  let write fmt =
    (* la dernière instruction a levé une exception ? *)
    let finally_raise = ref false in
    Queue.iter (fun s → s.pp fmt; finally_raise := s.raise_exn) sentences;
    (* si le dernier appel journalisé a levé une exception non rattrapée,
       l'encapsuler dans l'exception propre au journal. *)
    Format.fprintf fmt "%s"
      (if !finally_raise then "raise (Exception (Printexc.to_string exn))"
       else "()")
end
```

4.6. Arguments multiples

En *OCaml*, les fonctions à plusieurs arguments de type $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$ sont isomorphes aux fonctions prenant un unique argument de type τ_1 et renvoyant une fermeture de type $\tau_2 \rightarrow \ldots \rightarrow \tau_n$. L'algorithme précédent pourrait donc fonctionner. Néanmoins, générer une nouvelle liaison locale pour chaque application partielle rend le journal illisible. Nous préférons donc modifier l'algorithme pour retarder l'écriture dans le journal d'un appel de fonction jusqu'à son application totale. Ceci suppose néanmoins que les différentes applications partielles n'engendrent pas d'effets de bord, ce qui est le cas en pratique dans *Frama-C*. Ici, le principe est d'itérer récursivement sur le type de la fonction pour construire dans une continuation l'instruction finale à générer, en y ajoutant au fur et à mesure les arguments appliqués. Étendre la continuation de type `Format.formatter → unit` est simple.

```
(* écrit un argument supplémentaire arg dans la continuation f_acc *)
let extend_continuation f_acc pp_arg arg fmt =
  Format.fprintf fmt "%t %a" f_acc pp_arg arg
```

Le cas de base de la récurrence correspond à journaliser un type non fonctionnel, auquel cas la continuation est directement appliquée. Pour ce faire, on modifie la fonction `print_sentence` :

```
let print_sentence f_acc ty fmt =
  if Type.equal ty Datatype.unit then f_acc fmt;
  else
    let v = gen_binding () in
    Binding.add ty x v;
    Format.fprintf fmt "let %s = %t in " v f_acc;
```

8. Présentée en annexe A.

Dans le cas récursif, dans lequel la valeur à journaliser est une fermeture f, il nous faut avoir accès à son argument. Pour cela, nous déconstruisons f, effectuons le traitement attendu (appliquer la fonction à son argument, étendre la continuation et journaliser le résultat), et reconstruisons finalement la nouvelle fermeture. L'algorithme devient alors le suivant (la gestion des exceptions est ici omise).

```
let rec journalize: τ. (Format.formatter → unit) → τ Type.t → τ → τ =
  fun (type t) f_acc (ty: t Type.t) (x:t) →
    match Type.Function.gadt_instance ty with
    | Type.Function.No_instance →
      (* x est une valeur non fonctionnelle:
         l'application (éventuellement 0-aire) en cours est totalement effectuée *)
      (* écriture de la continuation dans le journal *)
      if not !started then Sentences.add (print_sentence f_acc ty);
      (* renvoie du résultat *)
      x
    | Type.Function.Instance(ty_y, ty_res) →
      (* x est une fonction: on reconstruit la fermeture *)
      fun y →
        if !started then x y
        else begin
          started := true;
          (* application de la fonction à son argument *)
          let res = x y in
          started := false
          (* extension de la continuation *)
          let f_acc = extend_continuation f_acc (pp ty_y) y in
          (* journalisation du résultat de l'appel *)
          journalize f_acc ty_res res
        end
```

On peut noter que la fonction ci-dessus est récursivement polymorphe, ce qui explique son annotation de type explicite. Désormais, `Journal.register` se contente d'appeler cette fonction avec la bonne continuation initiale affichant la valeur x à journaliser (*i.e.* le nom de la fonction si x en est une).

```
let register s ty x =
  Binding.add ty x s;
  (* initialisation de la continuation *)
  let f_acc fmt = pp ty fmt x in
  journalize f_acc ty x
```

Contrairement à précédemment, cet enregistrement ne fonctionne plus uniquement sur des fermetures. Son type peut donc être généralisé.

```
val register: name → α Type.t → α → α
```

5. Conclusion

Cet article a présenté le mécanisme de journalisation de *Frama-C*. Il permet de générer automatiquement un script *OCaml* permettant de reproduire automatiquement les actions utilisateurs, notamment effectuées dans la GUI. Le surcoût engendré, aussi bien en temps qu'en mémoire, est totalement négligeable dans le contexte de *Frama-C*.

En pratique, l'algorithme de journalisation possède quelques fonctionnalités qui n'ont pas été présentées dans cet article. Ainsi, les étiquettes et les arguments optionnels sont correctement gérés et affichés le cas échéant dans le code généré. Un effort particulier a été effectué pour générer du code lisible, correctement indenté et minimalement parenthésé. Le code généré peut aussi contenir des commentaires explicatifs.

Par ailleurs, pour être bien typé, cet algorithme combine typage statique et dynamique, *via* un type fantôme, et utilise, certes à faibles doses, plusieurs fonctionalités récemment introduites dans *OCaml* : types localement abstraits, récursion polymorphe et GADTs. En outre, même si elle n'avait jamais été présentée jusqu'à aujourd'hui, la journalisation est présente dans *Frama-C* depuis 2008. L'algorithme n'a pas été modifié depuis, sauf pour intégrer les facilités de typage offertes depuis *OCaml* 3.12. Plus que tout autre argument déjà avancé, cette stabilité est un gage de confiance envers sa correction et envers le fait qu'elle remplit la fonctionnalité souhaitée.

Remerciements Je tiens à remercier chaleureusement François Bobot de m'avoir soufflé l'idée des GADTs et de m'avoir fait part de ses remarques éclairées. Je remercie également Zaynah Dargaye et les rapporteurs anonymes pour leurs commentaires avisés m'ayant permis d'améliorer cet article. Par ailleurs, si Benjamin Monate ne m'avait pas, en 2008, expliqué la journalisation de *Caveat* et soumis le challenge d'implémenter une fonctionnalité similaire en *OCaml* pour *Frama-C*, cet article et l'implémentation qui s'y rapporte n'auraient sans doute jamais été écrits. Enfin, ces travaux ont été soutenus par le projet européen FP7 Stance.

Références

[1] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language, v1.7*, April 2013. `http://frama-c.com/acsl.html`.

[2] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat : a tool for software validation. In *International Conference on Dependable Systems and Networks (DSN'02)*, pages 537+, 2002.

[3] L. Correnson and J. Signoles. Combining Analyses for C Program Verification. In M. Stoelinga and R. Pinger, editors, *Formal Methods for Industrial Case Studies (FMICS'12)*, volume 7437 of *Lecture Notes in Computer Science*, pages 108–130. Springer, August 2012.

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a software analysis perspective. In *International Conference on Software Engineering and Formal Methods (SEFM'12)*, pages 233–247. Springer, October 2012.

[5] P. Cuoq and J. Signoles. Experience report : Ocaml for an industrial-strength static analysis framework. In G. Hutton and A. P. Tolmach, editors, *International Conference of Functional Programming (ICFP'09)*, pages 281–286. ACM, September 2009.

[6] J. Garrigue and J. Le Normand. Adding GADTs to OCaml : a direct approach. In *Workshop on ML (ML'11)*. ACM, September 2011.

[7] J. Signoles. Foncteurs impératifs et composés : la notion de projet dans Frama-C. In A. Schmitt, editor, *Journées Francophones des Langages Applicatifs (JFLA'09)*, volume 7.2 of *Studia Informatica Universalis*, pages 245–280, September 2009. In French.

[8] J. Signoles. Une bibliothèque de typage dynamique en OCaml. In *Journées Francophones des Langages Applicatifs (JFLA'11)*, Studia Informatica Universalis, 2011.

[9] J. Signoles, L. Correnson, and V. Prevosto. *Frama-C Plug-in Development Guide*, April 2013.

[10] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In A. Aiken and G. Morrisett, editors, *Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM, January 2003.

A. Code complet d'un journal

Cette annexe présente le contenu du fichier journal `frama_c_journal.ml` généré par *Frama-C* (version `Fluorine-20130601`) pour l'exemple de la section 2.4. Le code de la figure 2 est la fonction `run` de ce fichier.

```
(* Frama-C journal generated at 10:53 the 04/10/2013 *)

exception Unreachable
exception Exception of string

(* Run the user commands *)
let run () =
  Dynamic.Parameter.StringList.set ""
    [ "CruiseControl.c"; "CruiseControl_const.c" ];
  File.init_from_cmdline ();
  !Db.Value.compute ();
  let cil_datatype__Varinfo__t_map =
    !Db.Slicing.Select.select_stmt
      Db.Slicing.Select.empty_selects
      ~spare:false
      (fst (Kernel_function.find_from_sid 306))
      (Globals.Functions.find_by_name "CruiseControl")
  in
  let sl_project_Slicing1 = !Db.Slicing.Project.mk_project "Slicing1" in
  !Db.Slicing.Request.add_persistent_selection
    sl_project_Slicing1
    cil_datatype__Varinfo__t_map;
  !Db.Slicing.Request.apply_all_internal sl_project_Slicing1;
  !Db.Slicing.Slice.remove_uncalled sl_project_Slicing1;
  let p_Slicing1__export =
    !Db.Slicing.Project.extract "Slicing1 export" sl_project_Slicing1
  in
  Project.set_current p_Slicing1__export;
  ()

(* Main *)
let main () =
  Journal.keep_file "frama_c_journal.ml";
  try run ()
  with
  | Unreachable → Kernel.fatal "Journal reaches an assumed dead code"
  | Exception s → Kernel.log "Journal re-raised the exception %S" s
  | exn →
    Kernel.fatal
      "Journal raised an unexpected exception: %s"
      (Printexc.to_string exn)

(* Registering *)
let main : unit → unit =
  Dynamic.register
```

```
    ~plugin:"Frama_c_journal"
    "main"
    (Datatype.func Datatype.unit Datatype.unit)
    ~journalize:false
    main

(* Hooking *)
let () = Cmdline.run_after_loading_stage main; Cmdline.is_going_to_load ()
```