le projet RefPerSys

Basile STARYNKEVITCH

mai 2025

(un moteur d'inférences, "intelligence artificielle" symbolique)

8 rue de la Faïencerie

92340 Bourg-la-Reine

les opinions me sont personnelles

basile@starynkevitch.net

Journées Du Logiciel Libre - Lyon - mai 2025

git 36644f42b276df71

Plan

`refpersys.org`

Logo par Gaëtan Tapon

motivations pour RefPerSys

- souveraineté numérique (moins dépendre de logiciels états-uniens, même libres)
- s'occuper
- expérimenter un logiciel
- populariser et affiner des idées méconnues (celles de *Jacques Pitrat*)
- **un projet en cours de dévelopement qui cherche des contributeurs**

Le code source est libre sous licence GPLv3+/CeCILLv2 **sans garantie** sur `github.com/RefPerSys/RefPerSys` et peut-être même difficilement utilisable en mai 2025.

REFlexive PERsistent SYStem

historique et systèmes inspirants pour RefPerSys

- **les moteurs d'inférences des systèmes experts** des années 1990 CLIPSRULES (NASA)
- **le système CAIA** ("chercheur artificiel en intelligence artificielle") de J.Pitrat en `github.com/bstarynk/caia-pitrat/` qui auto-génère son demi-million de lignes de C (1990-2018)
- **les outils d'aide à la démonstration** ou assistants de preuve ROCQ (ex COQ)
- La problématique de **l'amorçage des compilateurs**: tous les compilateurs se compilent eux-mêmes. Idéalement *RefPerSys devrait générer son propre code* mais en génère une petite partie en mai 2025.

En France, il existe une grosse culture mathématique de systèmes formels de preuve. Ma philosophie est plus pragmatique et plus ludique.

une propriété méconnue de Linux: les greffons à foison

Un programme sous Linux est (très souvent) utilisateur de nombreuses bibliothèques dynamiques ("shared objects"), les fichiers `lib*.so`. Ce mécanisme optimise l'utilisation de la mémoire RAM et il est utilisé pour les greffons ("plugins") chargés par la fonction **`dlopen(3)`**. Le programme obtient par **`dlsym(3)`**, durant l'exécution, les fonctions d'un greffon à partir de leur nom.

Un grand nombre de programmes sous Linux (dont le navigateur FIREFOX, l'éditeur programmable GNU EMACS, le compilateur GCC, les interprètes de commandes BASH ou ZSH ou LUA) acceptent des greffons, pour permettre d'en étendre le comportement. Ceux-ci peuvent être codés par d'autres développeurs.

En pratique **un programme qui s'exécute** (son processus) **peut charger un grand nombre de greffons** (dizaines ou centaines de milliers). Donc **un programme peut générer beaucoup de greffons temporaires**.

Cf `manydl.c` dans `github.com/bstarynk/misc-basile`.

Chargement et déchargement de code, une idée ancienne

Dans les années 1950 ou 1960, **les ordinateurs avaient peu de mémoire** (quelques kilo-octets ou un-demi méga-octet) **et** l'IBM 1130, les CAB500 et CAB2020 de la SEA, l'IBM 360 et d'autres machines[1] **chargeaient le code par bouts**. **Un segment de code devenu inutile était supprimé de la mémoire centrale**, et la gestion fine du chargement et déchargement de ces segments nécessitait la coopération des programmeurs humains, du système d'exploitation (le "moniteur") et était une source de bogues intéressants.

Sous Linux, il n'est guère indispensable de décharger un greffon[2] par `dlclose(3)`. La puissance des machines et la performance du noyau (sous-système de gestion de la mémoire virtuelle) permet de ne pas le faire[3].

[1] Les µ-ordinateurs des années 1970 ou 1980 aussi: PC/XT, TRS-80, Amiga 500.

[2] L'exception serait un code hypothétique de calcul qui tourne pendant des mois sur un *super-calculateur* du `top500.org` en chargeant des millions de greffons générés temporairement.

[3] On accepte alors une fuite mémoire sans dommage.

Génération de code, une idée ancienne

Les compilateurs traduisent un code source (compréhensible par le dévelopeur) en du code machine[4] et existaient déjà dans les années 1950 ou 1960 (Cobol, Fortran, PAF, IPL-V, Lisp).

De nos jours, **les développeurs utilisent des logiciels** spécialisés **générateurs de code** (en C par exemple, ou en Java ou Ocaml): GNU bison comme Carburetta ou Menhir génèrent tous du code pour faciliter l'analyse lexicale ou syntaxique. La théorie des languages formels est mûre (Chomsky).

[4] La notion de code objet ou intermédiaire est apparue progressivement dans les années 1960; certaines architectures -dont Harvard, qu'on retrouve encore dans les micro-contrôleurs à l'intérieur de nos claviers et de nos chaudières- séparent la mémoire du code et celles des données.

Génération de code, une idée moderne

Des bibliothèques logicielles pour générer du code existent: GNU lightning, AsmJit, et le compilateur GCC fournit sa libgccjit

Le Common Lisp libre SBCL génère du code machine à chaque interaction de son utilisateur. C'est un langage homoïconique: le code y est manipulable comme donnée.

La plupart des implémentations de Java (cf OpenJDK) ont une JVM qui traduit petit à petit le code-octet (java bytecode) en du code machine.

Évaluation partielle: une manière d'optimiser

Beaucoup de calculs longs[5] (durant des heures ou des semaines) dépendent de plusieurs paramètres.

Certains paramètres sont constants durant tout le calcul[6].

Dans certains cas, il peut être utile de particulariser le code pour ces paramètres constants.
Par exemple simpliste, une affectation pourrait être `u := a*x + b*y + c` et si on sait que `a` et `c` sont nuls elle se simplifie en `u := b*y` plus rapide à calculer.

[5] La météo nationale calcule le temps de demain en des heures de calcul sur super-calculateur; de même en astrophysique -simulation de collisions de galaxies-; ou en bioinformatique -simulation des mouvements moléculaires dans une cellule-.

[6] Pour la météo, les mesures physiques entrées des paramètres des capteurs, mais les résultats internes intermediaires varient durant ce calcul itératif.

Évaluation partielle: formalisation

Formellement:

- on connait le code source σ (millions de lignes)
- ce code calcule sur des paramètres fixes $\pi_1, \pi_2, \ldots \pi_n$
- ce code calcule sur des entrées variables $v_1, v_2, \ldots v_p$
- les résultats obtenus sont $r_1, r_2 \ldots r_q = \sigma(\pi_1, \pi_2, \ldots \pi_n, v_1, v_2 \ldots v_p)$
- par des algorithmes complexes on produit et génère un code dérivé δ qui dépend des paramètres π_i tel que $\delta(v_1, v_2, \ldots v_p) = \sigma(\pi_1, \pi_2, \ldots \pi_n, v_1, v_2 \ldots v_p)$

La dérivation du code δ à partir du code source σ et des valeurs connues des paramètres fixes π_i est compliquée (peut être indécidable).

Variété des connaissances

Un programme est écrit par un développeur[7] qui a besoin de connaître:

- des connaissances sur l'algorithmie (les tris, les automates, les arbres, ...)
- des connaissances sur les langages de programmation et les systèmes d'exploitation utilisés
- des connaissances sur le problème qu'il doit résoudre dans le logiciel qu'il développe.
- des connaissances sur les ordinateurs[8] sur lequel s'exécutera le logiciel.
- des connaissances sur les contraintes (délais, séverité des bogues) et habitudes du projet: le logiciel de pilotage d'un métro automatique a d'autres contraintes qu'un logiciel bureautique ou un compilateur.

[7] Sans présupposé sur son genre, étant constaté qu'il y a moins de femmes que d'hommes qui pratiquent ou étudient la programmation, et que le métier d'informaticien me semble hélàs plus masculin que dans les années 1980!

[8] Un super-calculateur est différent d'un portable, même si tous deux utilisent Linux!

Formaliser -ou pas- les connaissances.

Certaines connaissances ne sont pas ou ne peuvent pas être formalisées. Les legislations[9], les habitudes, les intérêts individuels, la science et la technique informatiques imposent des limitations.

Le dévelopeur explicite certaines connaissances dans le code source et dans sa documentation:

- choix du (ou des) langages (formels) utilisés et de la forme des entrées et sorties.
- importance des noms choisis
- importance des commentaires
- séparer l'interface logicielle de son implantation
- importance de l'historique du code source et du projet logiciel et des "normes" ou "standards"
- cohérence relative et imparfaite entre le code source et sa documentation[10]

[9] Cf *RGPD* ou *EU-AI act*

[10] La programmation lettrée est difficile, et souvent coûteuse, peu utile ou inpraticable.

méta-connaissances

Ce sont les connaissances sur les connaissances.

- pour formaliser des connaissances: par exemple grammaire formelle de C ou de la syntaxe BNF des grammaires formelles.
- pour utiliser des connaissances: par exemple: un compilateur C[11] peut supposer que chaque fichier C a un nombre raisonnable de définition de fonctions de taille raisonnable.

Un développeur aguerri a des méta-connaissances sur la programmation.

Le même formalisme peut être utilisé pour représenter des méta-connaissances qui peuvent s'appliquer à elles-mêmes.

[11] Les compilateurs C n'arrivent guère à compiler en l'optimisant une fonction C de cent milles lignes de code dans un fichier C généré par un logiciel.

connaissances déclaratives ou procédurales

Une connaissance déclarative est un énoncé qui peut être utilisé de plusieurs façons. Par exemple: (grammaire française) *"l'adjectif s'accorde en genre et en nombre avec le nom auquel il se rapporte"*. Elle s'étend facilement (grammaire latine: *"... et en cas"*). Elle peut être formalisée: $\forall x \in \mathbb{R}^*, \quad 0x = 0 \quad \wedge \quad x^0 = 1 \quad \wedge \quad 0 + x = x$

Une connaissance procédurale n'est utilisable que d'une seule façon. Le code binaire[12] est procédural. En C `while (x>0) printf("%d\n",x--);`

En fait **le distinguo est graduel** et ces définitions varient selon les époques et les lieux [13].

[12] La complexité et le coût des microprocesseurs s'explique aussi par le fait que le silicium réordonne dynamiquement l'exécution des instructions machines ("out of order superscalar processors").

[13] Aux USA "declarative programming" a un sens différent de "programmation déclarative" en France!

Avantages de connaissances plus déclaratives

Fournir des connaissances formalisées et plus déclaratives augmente la productivité des développeurs.

Le génie logiciel (“software engineering”) tend à privilégier cette productivité[14].

Les outils logiciels pour les développeurs doivent favoriser la déclarativité des formalismes.

L'ordre dans lequel sont formalisées les connaissances déclaratives n'a pas d'importance!

[14] Au détriment d'autres facteurs, y compris énergétiques: par exemple l'utilisation de PHP ou Python dans des serveurs Web consomme plus de courant et de temps que le même service codé en C et compilé. Le numérique consommerait 5 à 10% de la production électrique européenne.

Traitement en machine des connaissances plus déclaratives

Le formalisme déclaratif est représenté pour son utilisateur par du texte (ou des images) formel[15], et en machine par des structures de données compliquées. C'est mathématiquement un graphe ou un hypergraphe. On suppose qu'il tient en mémoire centrale.
Un formalisme plus ou moins déclaratif peut être "interprété" ou "compilé" par l'outil logiciel l'acceptant.

- **interprétation**: l'outil logiciel parcourt directement et itérativement, pour traiter ce formalisme, l'hypergraphe de sa représentation et modifie les données traitées.
- **compilation**: l'outil logiciel transforme cette representation interne en d'autres représentations plus procédurales, in fine interprétées ou transformées en du code exécutable (ou plus efficace mais moins déclaratif).

[15] La présentation de ce texte - couleurs et polices de caractères - en facilite la lisibilité et sa compréhension par des humains.

une définition de l' “Intelligence Artificielle”

Enseigné par Jacques Pitrat:

définition de l'IA

Un logiciel d'“intelligence artificielle” est un programme qui peut surprendre agréablement ses utilisateurs.

Tout gros logiciel est bogué (et son nombre de bogues reste non-nul) **donc surprend désagréablement ses utilisateurs.**

Un logiciel d'IA peut être partiellement introspectif ou réflexif: en particulier en inspectant son code, ses données et ses piles d'exécution.

> *Loi de Hofstadter Il faut toujours plus de temps que prévu, même en tenant compte de la loi de Hofstadter.*

Elle s'applique au dévelopement de RefPerSys et à l'écriture de ces transparents.

Règles d'inférences et bases de règles

Une **règle d'inférences** comporte:

- un ensemble (*non ordonné*, fini) de conditions (déclanchant la règle)
- une séquence d'actions (pouvant s'exécuter quand ou si les conditions sont déclanchées)
- des méta-informations supplémentaires

Exemple: Si x est le père de y et si z est la fille de y alors on peut en déduire que x est le grand-père de z.

Une **base de règles** contient un ensemble (*non ordonné*, fini) de règles d'inférences, et des méta-informations supplémentaires Une **base de faits** contient un ensemble (*non ordonné*, fini) de faits (expressions supposées vraies, entités, ...)

Un **système expert** comporte une ou des bases de règles et des bases de faits

Persistance

Un logiciel persistant sauvegarde son état sur disque et peut redémarrer (le lendemain?) avec cet état.

RefPerSys persiste son état dans des fichiers textuels (JSON et code C++ généré) **portables**.

Comment gérer la co-évolution du logiciel et des données persistentes et du code?

RefPerSys en mai 2025

- dévelopé sous Linux (Debian ou Ubuntu ou Mint, parfois Raspbian)
- c'est du logiciel libre sous licence GPL/CeCILL
- historique accessible par `github.com/RefPerSys/RefPerSys/`
- codé en C++17 et compilé par un GCC récent (v14 ou v15)
- nombreuses dépendances (de logiciels libres) dont FLTK et libbacktrace et Carburetta
- quelques contributeurs bénévoles (Suède, Inde, Egypte, France, USA?)

Pour le compiler: définir la variable d'environment `REFPERSYS_TOPDIR` au répertoire source (obtenu par `git clone`) puis `make config` (logiciel interactif) puis `make -j4`
Votre `$HOME/tmp/` doit exister (votre répertoire temporaire)

tester RefPerSys

Quelques tests:

- `make test-load` et d'autres (essayez `make showtests`).
- `make redump` ou `make altredump` pour la persistance et regénération (doit être suivi de `make clean` puis `make -j4`)
- `make testcarb1` (à améliorer)

Données manipulées par RefPerSys

- le pointeur `nullptr` a un status spécial (une absence de valeur).
- les valeurs (un mot de 64 bits souvent pointeur, en C++ `Rps_Value`) peuvent être:
  1. entiers directs (62 bits)
  2. des valeurs scalaires immuables (chaînes, flottants, ...)
  3. des valeurs composites immuables (ensembles, tuples, fermetures, instances)
  4. des reférences (en C++ `Rps_ObjectRef`) à des objets lourds et mutables
- les valeurs composites et les objets peuvent être temporaires ou persistents
- chaque valeur a sa classe (modèle ObjVLisp) et son type (en C++ `enum class Rps_Type`)
- quelques objets sont des racines préexistantes (fichier C++ généré `generated/rps-roots.hh`)

Cadres d'appel explicites

Le code C++ représente un cadre d'appel réifié par `Rps_CallFrame`. On déclare en C++ ses valeurs locales par une macro `RPS_LOCALFRAME`, e.g.

```
RPS_LOCALFRAME(RPS_ROOT_OB(_1aGtWm38Vw701jDhZn)[16], //the_agenda,
               RPS_NULL_CALL_FRAME, // no caller frame
               Rps_ObjectRef obtasklet;
               Rps_InstanceValue descrval;
               Rps_ClosureValue clostodo;
);
```

En C++ le cadre d'appel courant est `_` et (dans l'exemple ci-dessus) la référence locale à son objet `obtasklet` se note `_f.obtasklet`, etc.
Ces cadres d'appels RefPerSys sont parcourus par le ramasse-miettes et les routines d'introspection.
On peut en C++ ajouter des collections C++ de valeurs au cadre d'appel courant en explicitant le code C++ qui les parcourt.

[16] Cet objet décrit le cadre d'appel pour en faciliter la réflexion et l'inspection.

Les valeurs scalaires dans RefPerSys

En plus des entiers directs (“tagged integers”) elles peuvent être une zone mémoire copiable par le ramasse-miettes pour (au choix):

- les chaînes de caractères emboîtées UTF-8
- les flottants emboîtés
- les valeurs JSON

On devrait pouvoir rajouter d’autres types de valeurs (images, vecteurs de nombres complexes, ...) **et en générer le code C++ les gérant**.

Les objets dans RefPerSys

Ils ont chacun un identifiant "mondialement" unique et aléatoire et persistant, leur **objid**. En mémoire 128 bits, dans les fichiers textuels (JSON) ou le C++ une séquence telle que `_1Io89yIORqn02SXx4p` à partir duquel on forme des noms de fonctions C ou C++ tels que `rpsapply_61pgHb5KRq600RLnKD`.

De plus chaque objet contient en propre:

- son verrou C++ `std::recursive_mtx`
- son espace de persistance (une référence à un objet ou `nullptr` pour les objets temporaires)
- sa classe, qui est un `Rps_ObjectRef` RefPerSys persistant.
- une association d'attributs (en C++ un `std::map`). Chaque attribut est un objet, sa valeur quelconque (sauf `nullptr`)
- une séquence de composants (en C++ un `std::vector`) de valeurs (parfois `nullptr`)
- une charge utile optionnelle (sous classe de `Rps_Payload`) qui connait son objet propriétaire.
- quelques pointeurs de fonctions C++ qui sont `dlsym`-ables et `dladdr`-ables

Les valeurs composites dans RefPerSys

Elles sont **immuables** et *copiables* par le ramasse-miettes. Chacune peut être temporaire ou persistante.

- **ensemble** fini d'objets ordonné par leur *objid*. Le test d'appartenance est rapide (par dichotomie).
- **tuple** (ou n-uplet) de réferences d'objets; une composante de tuple peut-être absente (`nullptr`).
- **fermeture**: sa routine est donnée par un objet qui décrit cette routine. Ses valeurs closes sont immuables. De plus elles peuvent contenir un méta-descripteur (référence d'objet) et un méta-rang (nombre descriptif).
- **instance**: organisée comme une fermeture, elles contient une connective (objet), des fils (valeurs fixes), et un méta-descripteur et un méta-rang.

Attention à notre terminologie: **une instance n'est pas un objet** (car immuable) mais toutes les valeurs composites ont leur classe et leur type. Les instances pourraient être utiles pour représenter des nœuds d'une syntaxe ou des faits, etc...

application d'une fermeture

Chaque fermeture connait son code (par un objet, sa "connective") C++ (une fonction C++ dont le nom commence par `rpsapply`).

Appliquer une fermeture à des arguments (tous des valeurs), c'est appeler cette fonction C++ (avec la valeur fermeture).

En C++ cette application se fait par du code tel que `_f.clostodo.apply1(&_, _f.obtasklet)`

Cette application renvoie deux valeurs (dans un couple en C++ `Rps_TwoValues`) qui par convention ne doivent pas être toutes deux `nullptr`. Elle devrait être presque aussi rapide qu'un appel de méthode `virtual` en C++.

envoi de message

Inspiré par Smalltalk ou Common Lisp, il est plus souple (mais plus coûteux en temps) qu'un appel de méthode `virtual` de C++.

On envoie à une valeur cible un message donné par un sélecteur et des arguments. En C++ par exemple `Rps_TwoValues two = _f.refpersysv.send3(&_, _f.gencodselob, _f.dumpdirnamev, _f.tempsuffixv, _f.genstoreob);`

Le selecteur est un objet

L'envoi de message est implémenté par application d'une fermeture. Chaque classe RefPerSys est un objet dont la charge utile contient une superclasse et un dictionnaire de méthodes asssociant à un selecteur la fermeture réalisant l'envoi. Ce dictionnaire peut évoluer avec le temps.

Ce modèle ressemble à celui de JavaScript, mais s'appuie sur les classes (pas de prototypes).

Le ramasse-miettes dans RefPerSys

Il ("garbage collector" ou glaneur de cellules) est conçu pour devenir plus efficace (en C++ `class Rps_GarbageCollector`) donc parfois copieur générationnel:

- les routines du R-M devraient être facilement générables.
- les valeurs scalaires ou composites (toutes immuables) doivent être copiables par le R-M
- les objets lourds sont fixes en mémoire (contenant le verrou `std::recursive_mutex` de chaque objet) et le R-M les marquent.
- parcours de la pile d'exécution et de ses cadres d'appels (`Rps_CallFrame`)

La persistance dans RefPerSys

Le code C++ de persistance ressemble par sa forme au R-M. Les deux devraient devenir générés. Au démarrage:

- à froid préallocation statiques des zones mémoire des objets racines
- chargement des objets vides de chaque espace
- remplissage des objets et initialisation des pointeurs de fonction conventionnels par `dlsym`

Le tas persistant ("persistent heap") **est donc chargé à chaque démarrage à partir de fichiers textuels.**

À la fin normale du processus (et peut-être tous les quarts d'heure) **on écrit sur disque le tas persistant mis à jour et le code généré en C++**. Le code C++ utilise `dladdr` pour retrouver le nom des pointeurs de fonctions.

Écriture du tas persistant

Le tas persistant est écrit sur disque, avec le code C++ généré qui va avec.

Chaque espace de persitance correspond à un fichier JSON. On peut donc distinguer le système (les objets nécessaires à la génération de code C++) et plusieurs “applications”.

On pourrait à l'avenir **utiliser d'autres représentations textuelles** (HJSON?, Yaml?) pour les données persistées. **Il faut veiller à leur portabilité**: un tas écrit sur x86-64 devrait être utilisable et chargeable sur RISCV.

agenda et millitâches

L'agenda est un objet racine persistant `_1aGtWm38Vw701jDhZn` nommé `the_agenda`, unique de sa classe `_3s7ztCCoJsj04puTdQ` $agenda \in class$.

Cet agenda contient des millitâches ("tasklets"), qui sont des objets (persistants ou temporaires). **L'exécution de chaque millitâche doit durer quelques millisecondes**.
Il existe plusieurs[17] **filaments d'exécution** (POSIX threads).

[17] Leur nombre est figé au démarrage de RefPerSys (entre 3 et 24, selon la puissance du processeur et son nombre de cœurs).

exécution parallèle des millitâches de l'agenda

Chacun des filaments d'exécution de l'agenda **boucle** sur:

1. choix[18] et retrait d'une millitâche à exécuter dans l'agenda
2. exécution de cette millitâche par envoi d'un message
3. cette exécution est rapide (millisecondes) et peut inspecter et modifier l'agenda (peut-être s'y ajouter à nouveau)

Cette boucle (interrompue par le ramasse-miettes) s'arrête quand RefPerSys termine normalement.

Des événements extérieurs [19] peuvent créer et ajouter (ou ôter) des millitâches dans l'agenda.

---

[18] En mai 2025 l'agenda contient quelques files d'attente de priorité haute, normale, ou basse de millitâches en attente d'exécution.

[19] interaction graphique ou Web avec l'utilisateur, ou JSONRPC, terminaison d'un processus -compilation d'un greffon généré en C++?-, interruption temporelle toutes les secondes?

A faire avec votre aide

- définir une syntaxe (règles, métarègles, ...) avec des exemples (cf carbrepl_rps.cbrt)
- accroître le tas persistent pour décrire déclarativement dans des objets RefPerSys le code C++ à générer (en reprenant des idées de GCC MELT)
- coder une interface graphique (esquissée avec FLTK ou autre et son code C++ doit en partie être généré à partir d'objets RefPerSys).
- coder les générateurs de codes C++ (y compris l'inférence) et machines[20]
- générer du code C++ qui remplace le code C++ écrit à la main (qu'on `git mv` dans `attic/`

[20] Le code machine généré n'est pas persisté car non-portable; il pourrait être généré rapidement par du code C++ généré qui utilise *GNU lightning* ou autre!

co-développement itératif de RefPerSys

Ce développement de RefPerSys devrait être itératif, collaboratif, et communautaire. Le tas persistant de RefPerSys doit générer de plus en plus de code (qui remplacerait celui existant codé[21] à la main) à partir de formalismes plus déclaratifs donc plus concis et plus compréhensible.

Le formalisme de RefPerSys est appelé à évoluer avec le temps en des representations de plus en plus déclaratives qui requièrent votre collaboration.

Un système d' "intelligence artificielle" doit expliquer son comportement aux utilisateurs pour être acceptable.

[21] J'ai pris soin de nommer les identifiants C++ de manière à espérer les générer

applications possibles de RefPerSys

- jeux
- optimisation et pilotage de codes de calculs sous Linux
- aide à la décision (jardinage, médical?, pilotage de projets? CAO?)
- et toute autre chose
- ...

En respectant le cadre légal et l'éthique du libre.