Littérature logicielle: des programmes informatiques agréables à lire, aptes à être publiés tels quels, et à servir de support à des cours exécutables.

Gérard Huet

Emeritus, Inria Paris-Rocquencourt

Séminaire Codes Sources
LIP6
22 Janvier 2015

Résumé

Donald Knuth a proposé la notion de “literate programming” il y a plus de 30 ans, afin de promouvoir l’écriture de programmes faciles à comprendre, à maintenir et à enseigner – pas juste des programmes jouets ou des algorithmes abstraits, mais les codes sources intégraux de logiciels de production, vus comme littérature exécutable. Nous présentons notre expérience du concept pour la publication et pour l’enseignement, expliquons le lien avec le “vernaculaire mathématique” proposé par de Bruijn, et donnons quelques raisons expliquant pourquoi ce concept n’a pas percé à ce jour autant qu’il l’aurait dû.

Préhistoire

- Il était une fois

Préhistoire

- Il était une fois
- Fortran

Préhistoire

- Il était une fois
- Fortran
- Cartes perforées

Préhistoire

- Il était une fois
- Fortran
- Cartes perforées
- Listings

Préhistoire

- Il était une fois
- Fortran
- Cartes perforées
- Listings
- Commentaires

Préhistoire

- Il était une fois
- Fortran
- Cartes perforées
- Listings
- Commentaires
- Flowcharts

Préhistoire

- Il était une fois
- Fortran
- Cartes perforées
- Listings
- Commentaires
- Flowcharts
- Langages de plus haut niveau

Knuth

Les fondements de l'informatique en tant que science ont été posés par Knuth, dans son ouvrage magistral "The Art of Computer Programming", qui mettait les *programmes*, et leurs abstractions mathématiques les *algorithmes*, avec tout l'arsenal mathématique de l'*analyse d'algorithmes*, au cœur de la discipline.
Science de nature mathématique, mais compromis d'ingénierie également (notamment le contrôle des ressources), et même notions d'esthétique. La programmation en tant que support littéraire des mathématiques constructives et à leur suite de toute l'ingénierie était mise au centre de la problématique.

Knuth

Knuth était très soucieux de la précision du langage technique, et faisait à Stanford un cours du soir sur le sujet. Notamment la discipline d'utilisation des formules mathématiques dans un discours argumenté en langue naturelle. L'utilisation d'identificateurs parlants, l'indentation, la déreliction des "goto"s, l'utilisation de *commentaires* adéquats, la modularité, étaient parmi ses préoccupations.
Il utilisait notamment toutes les ressources de la typographie la plus pointue pour éditer "The Art of Computer Programming". L'abandon de la typographie au plomb par Addison-Wesley l'amena à se préoccuper de typographie informatique. Il passa plusieurs mois à la bibliothèque de Stanford pour compulser les annales des meilleurs journaux mathématiques, et en tirer les primitives d'un langage typographique. Il y consacra l'essentiel de son énergie au début des années 80, avec la conception et la réalisation des logiciels TeX et MetaFont.

Le premier Web

Afin de construire des logiciels aptes à être fournis avec une documentation technique de qualité, Don Knuth inventa un langage modulaire de description d'algorithmes, appelé **Web**. Un compilateur **Tangle**, à partir d'une description Web d'algorithmes, pouvait produire deux documents. L'un était un programme Pascal bien formé, assemblé par recollage des graphes des modules algorithmiques, pouvant rejoindre une bibliothèque de programmes Pascal. L'autre était un code source TeX, dont la composition fournissait un livre ou un article technique très lisible documentant le logiciel.

TeX en Web

Le souci de bonne ingénierie poussa Knuth à utiliser lui-même ses outils. C'est ainsi qu'il devint typographe des volumes suivants de "The Art of Computer Programming" à travers l'utilisation de TeX, anticipant son utilisation progressive par les informaticiens à travers LaTeX, des mathématiciens avec AmsTeX, puis de l'ensemble des scientifiques. De même, il utilisa Web pour produire deux magnifiques volumes documentant les programmes Tex et MetaFont.

Literate Programming

Knuth théorisa le concept dans un article “Literate Programming” paru en 1984 dans le Computer Journal et disponible en
`http://www.literateprogramming.com/knuthweb.pdf`.
En 1992 Le CSLI édita un volume de même titre, anthologie d’articles de Knuth sur des sujets reliés.

Extraits de “Literate Programming”

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature.
Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

Literate Programming after the Web

Une version C de Web, Cweb, fut développée quand Pascal devint obsolète. Une version noweb indépendante des langages de programmation existe.
Un certain nombre d'outils visant à engendrer de programmes lisibles avec leur documentation ont été développés indépendamment de l'effort Web.
Ces outils sont parfois dénigrés par les promoteurs de l'effort "literate programming", qui est devenu au fil des ans plus le slogan d'une secte (voir par exemple le site `http://www.literateprogramming.com`) qu'une méthodologie de génie logiciel incontournable.

Ce que disent les ayatollas de Literate Programming

Ainsi, sur la page Wikipedia on peut lire:
`http://en.wikipedia.org/wiki/Literate_programming`
"Literate programming is very often misunderstood to refer only to formatted documentation produced from a common file with both source code and comments – which is properly called documentation generation – or to voluminous commentaries included with code. This is backwards: well-documented code or documentation extracted from code follows the structure of the code, with documentation embedded in the code; in literate programming code is embedded in documentation, with the code following the structure of the documentation."
Ce genre d'attitude est caractéristique d'aigris qu'un concept technologique élevé au rang de dogme n'ait pas pris, et qui se réfugient dans le déni sans analyser les raisons d'un échec patent. En effet, qui aujourd'hui peut évoquer le concept en utilisant le terme Web ?

Les raisons profondes de l'échec

Comprenons d'abord pourquoi la documentation intégrée au programme est difficile. Le commentaire n'a jamais été pris au sérieux par les concepteurs de langage de programmation et les implémenteurs de compilateurs. En Fortran, le commentaire était marqué par C en première colonne. Le programme proprement dit était la liste des "instructions". La principale innovation de LISP (McCarthy dixit) fut l'invention de l'expression conditionnelle. Le programme était conçu comme une formule mathématique. Mais le commentaire n'était toujours pas pris au sérieux, débutant par un point-virgule signalant au compilateur que le reste de la ligne était à jeter. Même dans un langage comme Caml, soucieux de pouvoir parenthéser du code à l'intérieur d'un commentaire, le commentaire n'était qu'un lexème à ignorer.

Syntaxe abstraite

Le commentaire étant structuré, puisque c'est un discours sur l'algorithme, il ne fait vraiment du sens que dans une méthodologie où les programmes eux-mêmes sont des données structurées (syntaxe abstraite), où les commentaires eux mêmes formalisés sont attachés précisément aux nœuds de l'arbre de syntaxe abstraite, et où les éditeurs de programmes permettent à l'utilisateur une liaison précise entre le commentaire vu comme assertion logique et la sous-expression documentée.
Ces idées furent testées notamment dans l'éditeur Interlisp de Xerox-PARC, facilitées par la nature structurée explicite d'un programme LISP comme S-expression (arbre binaire). Mais la liaison dynamique de LISP rendait très difficile les assertions sur les variables globales, dépourvues de scope.

Mentor/Centaure

Une tentative héroïque de conception d'un environnement de développement de programmes structurés fut engagée à l'IRIA avec Mentor (années 70) et à l'INRIA avec son successeur Centaur (années 80), avec l'espoir d'utiliser la sémantique dénotationnelle pour documenter les logiciels.
L'exemple Mentor-PASCAL pour l'élimination des récursions terminales fut très instructeur. Les langages de programmation existants étaient juste trop mal conçus pour permettre l'édition structurée correcte (l'infâme WITH).

Syntaxe abstraite: un concept ignoré

PARC dans les années 80, MESA.

Tout ça n'est que de la syntaxe

Même de nos jours, on voit bien que la syntaxe des langages de programmation n'est pas prise au sérieux :

```
match x with
   | A -> match y with
            | A -> "aa"
            | B -> "ab"
   | x -> "ac!"
```

La sémantique non plus, car 30 ans plus tard AUCUN langage de programmation n'a de définition formelle de sa sémantique (alors que c'était un prérequis de l'effort Stoneman d'ADA!)

Certification de programmes

Depuis Turing, à travers Floyd, Hoare, Diskstra et beaucoup d'autres, on cherche à vérifier la correction des programmes grâce à des assertions logiques insérées comme commentaires formels. Des représentants modernes de cette méthodologie sont les systèmes TLA+ et Why3. Ces systèmes représentent des compromis permettant dans une certaine mesure de présenter les programmes avec un appareillage logique permettant de vérifier qu'ils sont conformes à des spécifications formelles. Les assertions logiques permettent d'avoir une documentation formelle permettant d'assurer un certain niveau d'assurance d'absence de comportements aberrants, et l'utilisation de programmes ainsi vérifiés dans des applications où la sécurité joue un rôle critique. Il faut néanmoins souligner que, pour les langages de programmation impératifs, les assertions sont des formules logiques complexes dans des logiques modales pour tenir compte de l'état de la machine d'exécution, et que leur lisibilité n'est pas toujours évidente.

Programmes comme squelettes de preuve

Plus récemment, l'essor des langages de programmation applicatifs, et notamment des langages fonctionnels typés, a donné lieu à une nouvelle vision des programmes comme étant des objets mathématiques dénotant des preuves, à travers l'isomorphisme de Curry-Howard-de Bruijn.
Une nouvelle discipline, la Théorie des Types, poursuit cette voie en proposant des assistants de preuve permettant d'élaborer une preuve mathématique constructive de l'existence des objets calculés par des programmes, qui peuvent être vus comme le squelette de leurs preuves de correction. Le système Coq, et les extracteurs de programmes Ocaml ou Haskell associés, sont caractéristiques de cette méthodologie. Le coût inhérent du développement des preuves induites ne permet pas aujourd'hui de généraliser cette méthodologie à des programmes hors des applications sécuritaires. Néanmoins, cette vision futuriste permet de comprendre que les programmes sont bien au centre du dispositif.

Proof-directed vs program-directed development

La vision futuriste de programmes vus comme squelette de leur preuve de correction fait l'hypothèse que des environnements de preuve sophistiqués seront à la disposition d'ingénieurs de preuves pour développer des programmes certifiés par construction.
Toutefois, l'idée de contracter la preuve en un objet exécutable peut se renverser en le problème d'insérer dans le programme les invariants nécessaires, et il y a tout un compromis entre extraction de programmes de preuves et élaboration d'assertions permettant de certifier un programme donné. Ce compromis permet aux programmeurs de guider l'assistant de preuve à partir de leurs intuitions de programmeurs.
Exemple: crible d'Eratosthène et postulat de Bertrand.

Langages de programmation

Il y a progrès des langages de programmation:

- explicitation dans la syntaxe des primitives sémantiques
- typage, compositionnalité, modularité
- syntaxe abstraite manipulable (e.g. macro-processing Camlp*)
- assertions incorporées structurellement au source (e.g. WhyML)
- permettant leur vérification par déchargement d'obligations

Exemple: la famille ML

- syntaxe abstraite
- λ-calcul Algol
- let/where
- récursion Iswim
- typage ML
- fermetures
- monades Haskell
- types inductifs et programmation par motifs
- modules, foncteurs Ocaml

Publier ses programmes

Il ne faut pas attendre que les prototypes de recherche deviennent des produits largement utilisés pour publier ses programmes.
Il faut publier le source intégral dans un langage existant, pas un “pidgin”. Utiliser les technos ad-hoc (package listings de TeX, Ocamlweb) pour faire des documentations de qualité, diffusables et enseignables.
Plus, il faut publier les concepts algorithmiques dans des articles où le langage de programmation de leur implémentation est le formalisme définissant.
Assez de pseudo-mathématiques utilisées comme arguments d’autorité. Défendons l’exactitude d’algorithmes explicites.

Etre un écrivain fier de sa littérature

Nos programmes doivent être bonne littérature.
Nous devons être fiers de nos programmes en tant que poèmes mathématiques.
Le génie logiciel est l'ouvroir de la littérature algorithmique.

Quelques exemples personnels

- “Formal Structures for Computation and Deduction” 1986 mes notes de cours à CMU 1986
- “Constructive Computation Theory” 1992-2011 mon cours de lambda-calcul
- “The Constructive Engine” 1989 le premier Coq pour le CC
- “Böhm Theorem” 1993
- “Zipper” 1997
- “Zen” 2002
- “Sanskrit segmentation” 2005
- “Eilenberg machines” 2008
- “Sanskrit Heritage Platform” 2000-2015