L'air de rien N° 3

L'aléastriel du **L**aboratoire de **R**echerche et de **D**éveloppement de l'**E**PITA[1] Numéro 3, Avril 2006

Edito

par *Akim Demaille*

Qu'est-ce qui peut pousser dix permanents et vingt-cinq étudiants à s'enfermer dans les sous-sols de Paritalie, alors que le beau temps est enfin de retour ? Est-ce l'argent ? Que non, c'est de la tâche elle-même que vient la motivation. Parfois très théorique, elle est d'autres fois la conséquence d'une problématique bien concrète, comme l'enseignement du C++. Enfin, elle peut être romantique, un peu comme l'est une chasse au trésor : beaucoup de chercheurs, peu d'inventeurs... mais le prix en vaut la chandelle, même s'il est en dollars.

Je sers la science et c'est ma joie...

par *Sylvain Peyronnet*

Phrase connue tirée d'un imaginaire amusant qui laisse à penser que le scientifique est quelqu'un un peu en dehors du monde, qui passe son temps à inventer des trucs pas forcément utiles et à martyriser un disciple un peu paresseux.

Qu'est-ce qu'être chercheur ? Est-ce que ça rapporte ? À quoi reconnaît-on un bon chercheur ? Tant de questions que les gens se posent, auxquelles je vais tenter de répondre.

Tout d'abord, le chercheur a une première mission : la Recherche. Il doit en effet travailler à la découverte de nouvelles connaissances. Pour cela, le chercheur fait de longues (et fastidieuses) recherches bibliographiques pour éviter de réinventer la roue à chaque nouvelle étape de son travail, puis il doit se montrer créatif et toujours se mettre un peu en danger en explorant des pistes de travail sans savoir si elles aboutiront à quelque chose (parfois après plusieurs années). Lorsqu'il réussit à former des hypothèses *a priori* cohérentes, il doit les tester (si c'est un expérimentateur) ou bien les prouver (si c'est un théoricien). En cas de réussite totale, le chercheur a alors fait avancer la science, dans le cas contraire il doit revoir ses hypothèses (la Recherche est un processus récursif !) ou bien changer complètement d'idée. Je voudrais insister sur le fait que le métier du chercheur n'est absolument pas l'innovation technologique. Certains en font, par envie ou par esprit de lucre, mais ce métier d'innovation est plutôt tourné vers l'ingénierie et est souvent l'apanage de ceux que l'on appelle les ingénieurs de recherche (une « caste » à mi-chemin entre les chercheurs et les ingénieurs). Enfin, la plupart du temps le chercheur a plusieurs casquettes : il est enseignant (comme nous au LRDE par exemple) et il effectue un temps important en tâches administratives (rédaction de projets de recherche pour trouver des financements, gestion de filières d'enseignement, *etc.*).

Est-ce que ce métier rapporte ? Pour faire court, la plupart du temps la réponse est « non », mais la motivation du chercheur est rarement de ce côté-là, et certains s'en tirent bien (comme Sergey Brin et Larry Page par exemple).

À quoi reconnaît-on un bon chercheur ? En fait, personne ne sait vraiment donner des critères objectifs. La plupart du temps les chercheurs d'un même domaine (une « communauté de recherche » comme on dit) sont capables de se jauger entre eux et de créer une hiérarchie de « niveau ». Ensuite, et c'est ce qui est pratiqué dans les instances de recherche (comme le CNRS), on peut compter... Compter quoi ? Ce que je n'ai pas écrit tout à l'heure est que le chercheur écrit des articles qui sont publiés dans des conférences et des journaux scientifiques internatio-

[1] L'air de rien, `http://publis.lrde.epita.fr/LrdeBulletin`.

naux. Ces conférences et journaux ne publient pas n'importe quoi, en effet pour « passer » un article, il faut que celui-ci soit accepté par un comité de programme, qui est en fait une assemblée d'experts d'un domaine. Ce sont ces experts qui décident si les papiers sont méritants ou non. Je disais donc qu'on pouvait compter, et ainsi classer, les chercheurs selon leur nombre de publications. Ce critère est toutefois un pis-aller car certains chercheurs publient beaucoup d'articles d'un niveau normal, alors que d'autres en écrivent peu, mais d'un niveau fantastique, le critère quantitatif n'est donc pas forcément le meilleur (mais il a le mérite d'être objectif et de montrer l'activité existante).

Bref, vous l'aurez peut-être compris, le métier de chercheur n'est pas forcément ce que l'on croit de prime abord, et si vous voulez en savoir plus, n'hésitez pas à venir discuter avec nous. À part ça, si vous me trouvez un disciple, je prends si je peux le martyriser tranquillement...

Modern Compiler Implementation in C++

par *Akim Demaille, Roland Levillain*

En 1999 s'engageait une réflexion entre quelques jeunes nouveaux profs de l'EPITA : comment faire comprendre aux étudiants les contraintes des projets de grande échelle que sont les tests, la correction et la rénovation permanente, la documentation ? Comment les sensibiliser aux techniques modernes de la programmation orientée objet, et tout particulièrement les patrons de conception (*design patterns*) ? Comment les amener à lire et écrire de l'anglais ?

C'est d'une séance de brainstorming qu'est née l'idée du projet Tiger[2] : l'implémentation en C++ d'un compilateur pour un langage petit, mais costaud, en suivant un tout nouveau livre dédié à la construction des compilateurs. Les étudiants, travaillant en groupe, auraient à rendre à intervalles réguliers leur compilateur équipé de modules supplémentaires, et... de nouvelles corrections des composants plus anciens. Sa conception serait le support du cours de modélisation objet de Thierry Géraud. Comme l'écriture *ex nihilo* est une tâche considérable — spécialement dans un langage impératif — nous (Théo et Akim) nous engagions à fournir le code de logistique.

Nous n'avions pas compris l'ampleur de la tâche... On pensait pouvoir se contenter de n'engager qu'un doigt, mais Tiger nous a bouffé tout le bras. Que j'apprenne le cours de compilation quelques heures avant les étudiants passe encore, mais quelle incroyable galère que d'essayer d'implémenter le compilo en avance pour le donner aux étudiants. Comme, bien sûr, nous ne sommes pas prescients, il nous a souvent fallu modifier du code que nous avions déjà donné aux étudiants — ou comment se faire maudire par 350 jeunes gens en pleine santé. C'est avec une certaine émotion que l'on se remémore les souvenirs de nuits Tiger pendant lesquelles rien ne marchait : entre G++ et NetBSD, le courant ne passait vraiment pas. C'est avec nostalgie, mais sans aucun regret, qu'Akim se souvient faire passer seul deux promotions entières en soutenance 5 fois par an — avant l'invention des Yakas.

Le temps a bien passé depuis, et aujourd'hui quelques chiffres permettront de mesurer l'ampleur qu'a pris le projet Tiger : 7 promotions épitéennes, 250 pages de sujet Tiger, 2 000 étudiants, 2 500 révisions, 5 000 compilateurs rendus et testés, 25 000 lignes de ChangeLog, 42 000 lignes de C++ et... 3 200 lignes de Tiger (les batteries de tests).

Originalités

Comparé à son modèle, décrit par Andrew Appel, ou bien encore aux implémentations proposées par les livres consacrés aux compilateurs écrits en C++, notre conception offre un certain nombre d'originalités et d'innovations, tant d'un point de vue technique que d'un point de vue pédagogique. Nous en présentons certaines ici.

Techniques

L'architecture est globalement celle d'A. Appel, voir la figure 1, à l'exception notable de l'introduction de trois visiteurs (`Bind`, `Type`, `Translate`) là où il recommande une seule passe. Cette séparation rend plus claire l'existence de trois tâches distinctes, permet une écriture beaucoup plus intelligible, et est plus ouverte sur des extensions du langage telles que le support de la surcharge de fonctions. Dans le détail, on pourra noter également que notre syntaxe abstraite est plus précise et plus riche ; elle exploite entre autre l'héritage multiple pour installer des annotations sur les AST.

Une utilisation intensive des visiteurs nous a permis d'affiner l'implémentation de ceux-ci dans le C++ par rapport à l'état de l'art : généricité pour factoriser les visiteurs lecture ou lecture/écriture, surcharge pour simplifier l'écriture, `operator()` pour s'intégrer au cadre des objets fonctions, héritage et polymorphisme pour la factorisation de parcours commun

[2] projet Tiger, `tiger.lrde.epita.fr`.

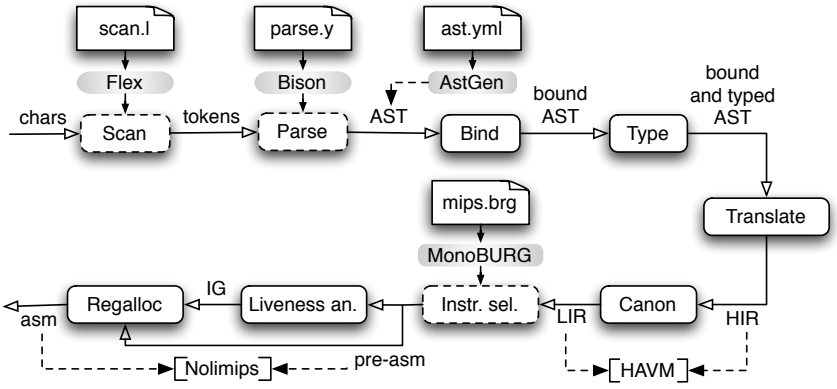

FIG. 1 – La structure du compilateur Tiger (par Olivier Gournet)

Le « *scanner* » segmente le flot de caractères d'entrée en un flot de « lexèmes » (des mots, ou *tokens*) à partir desquels le « *parser* » construit un arbre de syntaxe abstraite (AST). Les utilisations d'identificateurs sont liées à leur définition correspondante en tenant compte des règles de portée. Après la vérification des types, deux étapes effectuent des traductions simples vers un langage intermédiaire à partir duquel l'on peut faire de la génération de code pré-assembleur. La seule différence entre ce dernier et le véritable assembleur est qu'il s'autorise autant de registres qu'il le souhaite. L'analyse des vies des variables permettra la production d'un graphe d'exclusion mutuelle à partir duquel l'allocation des registres pourra se faire, livrant l'assembleur final.

(e.g., à vide ou en recopie), et divers sucres en fonction des situations.

Autre innovation, la « syntaxe concrète du pauvre ». Passées les étapes d'analyse, un compilateur passe le plus clair de son temps à transformer des arbres — on parle de réécriture — et par conséquent son code contient beaucoup d'instructions pour reconnaître des motifs d'arbres puis les transformer. Ces instructions sont toujours indigestes et horribles à déboguer : manipuler indirectement la syntaxe d'un langage est lourd. Au contraire, manipuler directement la syntaxe d'un langage (on parle dans ce cas de « syntaxe concrète » par opposition à la syntaxe abstraite) est extrêmement flexible, puissant, mais… exige des technologies *ad hoc* ; c'est le cas de l'environnement de transformation de programmes Stratego/XT. Notre proposition permet d'avoir une partie des bénéfices de la syntaxe concrète dans un environnement C++. À titre d'exemple, la transformation d'une comparaison de chaîne de caractères $\alpha \triangleleft \beta$ (où $\triangleleft$ désigne <, =, *etc.*) en un appel à la routine de comparaison de chaîne de caractères s'écrit simplement `parse ("strcmp (" << α << ", " << β << ")" << ◁ << "0")`.

Outils

De façon à offrir aux étudiants les meilleures conditions pour implémenter leur compilateur, de nombreux projets ont été étendus, voire créés.

GNU Bison Les plus anciens s'en souviennent peut-être encore, mais autrefois nous utilisions des parsers générés en C ! Aujourd'hui, non seulement ils sont en C++, mais le GLR — technologie qui permet de traiter n'importe quelle grammaire hors contexte — est également utilisable. Les capacités de mise au point, de traçage, de gestion de mémoire ont également été améliorées à l'intention des épitéens, mais aussi au bénéfice de tous les utilisateurs de Bison.

AstGen Chaque année l'AST subit des modifications, ce qui a également un impact considérable sur les visiteurs. Un outil, `astgen` nous permet la génération automatique de tout l'AST, et d'une grande partie des visiteurs. En bonus, il gère le concept de code caché, que l'on montre ou non aux étudiants.

Havm Quelle galère que de mettre au point une traduction vers un langage qu'on ne peut pas exécuter ! C'est bien pire encore pour le correcteur qui doit évaluer la correction de 75 compilateurs… Et pourtant, la représentation intermédiaire d'un compilateur (comme les RTL de GCC) disposent rarement d'un modèle d'exécution. *Enters* HAVM, écrite par Robert Anisko, et qui permet depuis quelques années d'exécuter le résultat des passes `Translate` et `Canon` (voir la figure 1).

MonoBURG Une étape de traduction, c'est fondamentalement un parcours d'arbre associé à la production d'un autre. La génération du code pré-assembleur à partir de la représentation intermédiaire ne fait exception. Le seul soucis : c'est très long et extraordinairement pénible à écrire en C++. C'est ce qui a décidé Clément Vasseur, Benoît Perrot et Michaël Cadilhac à reprendre le générateur de générateur (si si) de code du projet Mono : MonoBURG. Depuis les étudiants peuvent effectuer des améliorations sur le code produit par leur compilateur, ou bien même viser une autre architecture telle que IA-32.

Nolimips Pourquoi diable recommandons-nous de compiler vers l'architecture MIPS, puisqu'elle a disparu des machines de bureaux ? D'une part parce que c'est une architecture bien plus moderne que les x86 et encore très présente en particulier dans l'embarqué, mais aussi parce qu'il existe de nombreux simulateurs ; ainsi, que ce soit sur Mac OS X ou sur IA-32, pas d'inégalité ! De façon à fournir un service pédagogique amélioré, Benoît Perrot a conçu Nolimips, *yet another* MIPS *emulator*, mais qui, entre autres, permet de « régler » le nombre des différents registres. Ceci permet d'une part d'exécuter du pré-

assembleur (avant l'allocation des registres !), mais aussi d'imaginer des micro-processeurs MIPS « dégradés » de façon à rendre l'allocation des registres plus difficile.

tc-check Tout vrai projet inclut des tests, et Tiger dispose d'un outil dédié à cette tâche, `tc-check`. Cette « moulinette », qui teste les différentes étapes de TC, sert à la fois à vérifier le compilateur de référence et à évaluer les rendus des étudiants. Elle est jalousement gardée par les responsables du projet, car la mise au point de tests (outils et scénarios) fait partie intégrante de l'exercice.

Tiger Compiler 1.0

Qu'est-ce qu'une version 1.0 ? Il y a toujours quelque chose à faire, et c'est pour cela que nous en étions il y a peu à la version 0.92 alors que le compilateur est complet depuis plusieurs années. C'est le jour où Andrew Appel est venu nous rendre visite qui a donné la symbolique nécessaire au passage à 1.0.

Mais l'histoire n'est pas finie, comme peuvent vous le dire les 2008. Quant à nous, on songe déjà aux changements pour l'année prochaine !

$\mathcal{P}$ et $\mathcal{NP}$ *in a nutshell*

par *Michaël Cadilhac*

Des problèmes à un million de dollars de l'Institut de Mathématiques Clay[3], le plus populaire est sans doute celui cherchant à établir une relation entre les classes de complexité $\mathcal{P}$ et $\mathcal{NP}$. Pour rappel, un problème est dans $\mathcal{P}$ si l'on peut lui trouver *facilement*, comprendre en temps polynomial, une solution ; il est dans $\mathcal{NP}$ si l'on peut vérifier *facilement* qu'une séquence donnée est effectivement une solution.

Il vient immédiatement de ces définitions la relation $\mathcal{P} \subset \mathcal{NP}$. Tout le jeu consiste à qualifier la véracité de l'inclusion inverse : « vraie », « fausse », ou bien le très rassurant « indécidable ». Bien que la dernière possibilité soit peu discutée, des débats des plus vivants animent les deux premières réponses, et pour cause !

Si $\mathcal{P} = \mathcal{NP}$, résoudre des problèmes telle que la factorisation en nombres premiers serait *facile*, ce qui ne serait pas bien grave si une immense partie de la cryptographie actuelle ne partait du prédicat contraire. Les papiers qui « prouvent » l'égalité passent par des chemins variés : montrer qu'un des problèmes difficiles de référence (on les dit $\mathcal{NP}$-complets) a une solution algorithmique qui s'exécute en temps polynomial[4], transporter un problème dans une autre logique ou représentation d'où le résultat découle[5], ou encore tenter des parallèles entre l'algorithmique et les bulles de savon[6].

Dans le cas contraire, si $\mathcal{P} \neq \mathcal{NP}$, il existe une classe intermédiaire entre nos deux sujets. Cette classe, que certains appellent d'ores et déjà $\mathcal{NP}$-*I*, pour $\mathcal{NP}$-Intermediate, contiendrait des problèmes tels que l'isomorphisme de graphes. Quelques preuves de la non-égalité passent par l'exhibition d'un problème qui serait dans $\mathcal{NP}$ mais pas dans $\mathcal{P}$, matérialisant $\mathcal{NP}$-*I*, mais ces preuves par l'absurde tiennent généralement plus du sophisme que de l'argumentation[7]. D'autres prennent des tournures plus théoriques, mais ne sont hélas pas traitées par des spécialistes du domaine abordé[8].

De nos jours, un nombre impressionnant de preuves fleurissent de par le Net. Souvent trop naïves ou élémentaires, elles tombent dans des abus rapidement identifiables du fait de leurs fréquences. Le bon niveau d'étude pour la question de l'égalité de $\mathcal{P}$ et $\mathcal{NP}$ semble donc se dessiner : ce ne serait ni l'algorithmique, ni la programmation, ni même l'informatique, mais la logique, la théorie des ensembles et la théorie des modèles.

En bref

Les publications (disponibles sur publis.lrde.epita.fr)

– Image Restoration with Discrete Constrained Total Variation, Part I : Fast and Exact Optimization. Part II : Levelable Functions, Convex Priors and Non-Convex Cases, par Jérôme Darbon et Marc Sigelle, à paraître dans *Journal on Mathematical Imaging and Vision*, 2006.

Les séminaires du LRDE Les étudiants des promos 2007 et 2008 vont présenter leurs travaux les mercredis après-midi suivants : 17 et 24 mai 2006, 7 et 14 juin 2006.

[3] Institut de Mathématiques Clay, `http://www.claymath.org`.
[4] Exemple : `http://rattler.cameron.edu/swjpam/Vol1_1996/plotnikov.ps.gz`
[5] Exemple : `http://www.tarusa.ru/~mit/ENG/sigma01_e.zip`
[6] les bulles de savon, `http://arxiv.org/abs/cs/0406056`.
[7] Exemple : `http://www.math.usf.edu/~eclark/NPvsP.pdf`
[8] Exemple : `http://users.i.com.ua/~zkup/pvsnp_en_002.pdf`