Notes de Lectures: Thèse de Steven Varoumas, "High-level programming models for microcontrollers with scarce resources"

Hector Suzanne

Prélimianires

Où une étude de la structure matérielle des microcontrôleurs et de l'état de l'art de leur programmation justifie la création d'une machine virtuelle pour OCaml à faible ressource pour la programmation concurrente avec vérification formelle

Les microcontrôleurs

Les microcontrôleurs (mC) sont caractérisés par des une puissance de calcul et un prix inhabituellement faible dans le marché des années 2010-2020: de l'ordre de 10^7 instructions par secondes sur 8 ou 16 bits, quelques kilo-octets de RAM, et quelques dizaines de kilo-octets de mémoire flash, plus-ou-moins en lecture seule. L'entrée sortie se compose de quelques dizaines de signaux binaires ou analogues, les *broches*, et d'un système d'horloge et d'interruptions pour les prendre en compte. Ils sont typiquement utilisés dans un contexte embarqué, sans système d'exploitation (en *bare metal*). Les prix varient de 0.1€ à 10€.

On dénombre quatres grandes architectures et jeux d'instructions associés pour les microcontrôleurs: AVR, PIC, ARM et Intel 80xx. La plateforme AVR connaît un gain de popularité depuis quelques années grâce notamment à Arduino et assimilés, des plateformes de développement "tout-compris" apprécié des amateurs. PIC est principalement munie d'outils propriétaires avec support idoine, et ARM tient le haut du panier en terme de puissance et de capacité mémoire, et est donc plus chère.

La programmation de microcontrôleur se fait généralement avec des langages de bas-niveau: assembleur ou C (en vérité des sous-C), dotés parfois de macros augmentant la *quality of life* du programmeur. La connaissance du matériel et la maîtrise fine des ressources mémoires priment avant tout, en on en vient à apprécier les capacités "haut niveau" de C telles qu'elles avaient été appréciées par les programmeurs des années 80. La portabilité des programmes en souffrent car les API et jeux d'instructions sont rarement compatibles, et la stabilité

légendaire des programmes ASM et C est de mise, et peut causer des pannes et des destructions de matériel.

L'environnement de développement, quand il n'est pas fournit par le fabricant *via* une licence propriétaire, est souvent peu adapté à la création de projets complexe et de programmes sûrs: comme la dynamique du programme est étroitement liée à celle du système dans lequel il est embarqué, la simulation du programme demande de simuler l'électronique qui l'entoure. La connaissance des domaines experts du projet en cours et conseillée, et décourage plus d'un amateur. Le débogage sur machine est possible via un port série, et devient une tâche ingrate et chronophage, où l'on est la victime d'erreurs longtemps oubliées dans les contrées plus verdoyantes de la programmation haut niveau.

Gravir l'échelle de l'abstraction

Les programmeurs "normaux", eux, on depuis un certain temps abandonné l'assembleur, et l'on commence à observer la déréliction lente de C dans l'*userspace*. En s'extrayant de la correspondance entre calcul et machine, des outils comme le typage statique, la gestion automatique de mémoire, ou encore le débogage symbolique, ont permis un gain de productivité et de *quality of life* des programmeurs de systèmes complexes, tout en offrant de plus en plus de garanties formelles du bon fonctionnement des programmes. On souhaite donc, ne serait-ce que par charité, faire profiter aux programmeurs des microcontrôleurs de ces mêmes avancés. Certain c'y sont essayé, et l'on fait un tour de l'état de l'art de la programmation haut-niveau sur ces "basses-machines" qui sont notre sujet.

Compiler le code de haut-niveau, ça consiste à descendre l'échelle de l'abstraction pour finir sur le métal, en *code objet*. On lie ensuite les différents bouts de programme ensembles pour former un exécutable. La difficulté de la compilation vers microcontrôleur, c'est le manque d'expressivité du code-objet: le métal encore plus bas qu'à l'habitude. Des langages comme C++, qui ont pris soin de ne pas prendre leur lien avec le matériel, se compilent plutôt bien avec des outils propriétaires ou des compilateurs basés sur GCC. Mais en restant fidèle à son métal, C++ n'apporte pas les garanties que d'autres langages de haut-niveau offrent, en premiers la gestion mémoire automatique et le typage statique fort. La plateforme modulaire de compilation LLVM supporte ou à supporté des architectures de microcontrôleurs, permettant de compiler des sous-langages de *Rust* ou *Go* vers AVR ou PIC. Mais les progrès sont lents: le métal est *très* bas après tout.

Pour palier à ce problème, on peut utiliser une machine virtuelle. Cela consiste, si l'on pousser l'analogie, "faire remonter le métal". Le code source est compilé vers un *code-octet* (ou *bytecode*), compréhensible par un programme interprète qui sert donc de machine proprement virtuelle. Cet interprète est implémenté pour avec les langages de bas-niveau. Comme passer du métal au système complexe (coder en C) est trop complexe, et que passer du haut-niveau vers le

microcontrôleur l'est aussi, on les fait se rencontrer à mi-chemin. On a aussi l'avantage de pouvoir choisir le terrain d'entente: le code-octet n'a pas à obéir au lois du métal, ni à être élégant pour le programmeur. On bénéficie d'une vrai représentation intermédiaire. Les programmes en code-octet, plus la machine virtuelle, sont souvent plus petits que les équivalents en C, ce qui est un avantage indéniable quand on travaille sur peu-ou-prou sur 24ko de mémoire programme. Plusieurs machines virtuelles ont étés portées sur microcontrôleurs:

Java Parmi les efforts de création de machine virtuelle (VM) pour microcontrôleurs, on note en premier Java. Dès ses débuts, la JVM, développée *avec* et *pour* Java constitue un avantage du langage de Sun (puis Oracle): "*Write once, run everywhere*". Java dispose d'un vrai système de type avec sous-typage nominal et d'une riche bibliothèque de classes disponible n'importe où la JVM l'est. Le langage Java seul à été compilé via la JVM pour tourner sur 1ko de RAM et 16ko de mémoire flash. Mais si on inclut le ramasse-miette de la JVM, la fameuse bibliothèque standard et les threads, on passe à 24ko de RAM et 128ko de ROM minimum…

Python *MicroPython* permet de faire tourner du code Python 3 sur des microcontrôleurs ARM à grande capacité: on parle de 256ko de ROM et 16ko de RAM. Elle offre l'accessibilité de Python sur microcontrôleur et même un REPL distant, évaluant sur la machine hôte les commandes entrées depuis l'ordinateur du programmeur. Mais si on passe à des tailles de contrôleur plus modestes, le bât blesse. En dessous en 8ko de RAM, seuls des sous-parties de Python 2.5 ont étés portées, et ce sans bibliothèque standard. Python est connu pour faire un usage important des ressources machine pour fournir une programmation multi-paradigme hautement dynamique.

Scheme Le dérivé fonctionnel de Lisp possède un grand pouvoir expressif à l'aide des *macros* et du support de premier ordre des *continuations* comme valeurs. Le langage est défini par standard et de nombreuses implémentations existent, dont certaines, comme *Guile* du projet GNU, utilisent une VM. On peut ainsi exécuter sur PIC des programmes Scheme conformant au standard 4 du langage à partir de 8ko de RAM, et une sous-partie du standard 5 a été implémenté ne demandant que 1ko de RAM et 6ko de ROM.

OCaml OCaml fait intervenir sans son pipeline de compilation classique un bytecode pour la machine virtuelle ad-hoc *ZAM*, qui est au cœur de son interpréteur. Elle se compose uniquement de 148 instructions, qui permettent d'implémenter le styles de programmation objet, fonctionnel, modulaire et impératif. OCaml dispose aussi d'un typage statique fort et d'inférence de type. Le projet *OCaPIC* permet d'exécuter *l'intégralité* du langage dans seulement 4ko de RAM et 64ko de ROM sur le silicium PIC. Elle permet aussi d'optimiser le code OCaml pour les besoins spécifiques des microcontrôleurs.

Nous considérons que l'approche d'OCaPIC est la plus pertinente pour per-

mettre la programmation de haut niveau sur microcontrôleur. La légèreté de la VM d'OCaml permet une implémentation complète du langage, et OCaml dispose d'un typage statique fort et d'une gestion automatique de la mémoire par ramasse-miette. Afin d'élargir la compatibilité notre travail au delà d'une seule architecture, pour décidons d'implémenter une machine virtuelle générique à empreinte mémoire faible et bonne performances.

Programmation Synchrone

Les microcontrôleurs font souvent partie de systèmes en *temps réels*, parfois critiques. Leurs rôle y est d'orchestrer le comportement des différent composant électroniques du système sous des contraintes de latence fortes. Cette orchestration est inhéremment concurrente, et induit donc une notion *d'ordonnancement* des tâches dans le microcontrôleur au modèle d'exécution séquentiel. Si l'on peut borner statiquement le temps minimum entre deux événements que le microcontrôleur doit prendre en compte, alors l'ordonnancement peut être réalisé statiquement. Sinon, l'on fait appel à un ordonnanceur: un composant d'un *Real-Time Operatiing System (RTOS)* qui donne la main aux différentes tâches en fonctions des signaux extérieurs et des priorités internes relatives. Mais ces RTOS ne sont que peu voire pas compatibles avec les faibles ressources des microcontrôleurs considérés ici. De plus, des erreurs de concurrences peuvent encore avoir lieu en présence d'un bon ordonnancement, et peuvent bloquer des tâches prioritaires.

Mais un autre modèle de concurrence permet de sauver la mise: la *programmation synchrone*. Ce paradigme éprouvé dans le monde industriel se passe de machinerie ordonancielle en garantissant statiquement le dit ordonnancement de manière déterministe. Cette approche légère nous semble particulièrement adaptée à la création de logiciels pour microcontrôleur, sans OS et avec peu de ressources. Elle repose sur *l'hypothèse synchrone*:

> *Le temps de latence engendré par le calcul des sorties à partir des entrées est négligeable.*

En d'autres termes, le temps de synchronisation du microcontrôleur avec son environnement est nul. C'est le même genre d'hypothèse que font les concepteurs de circuits électriques (*le flux d'électron est instantané*), ou dans la gravitation newtonienne (*les interactions entre corps sont sans délais*). C'est une hypothèse bien sur: il ne faut jamais négliger sans bonne raisons (on y reviendra avant la fin du chapitre, promis). Sous l'hypothèse synchrone, la réaction du microcontrôleur se fait donc en un instant qui se répète régulièrement, et que l'on nomme *instant synchrone*. Le déroulement du programme suit alors une boucle: on lit les entrées, on calcule les sorties *en un instant*, on les écrits, et on recommence. Avec la programmation synchrone, on a simplifié la concurrence: plus besoin pour le programmeur de réfléchir manuellement à la *synchronicité* des composants entre eux.

La programmation synchrone est arrivée dans les laboratoires français dans les années 80, où ont été produit deux grandes familles de langages synchrones: ceux à *flux de contrôle* et ceux à *flux de données*. Dans les langages synchrones à flux de contrôle, un programme est composé de plusieurs modules possédants chacun leurs propres entrées/sortie et une boucle interne. Les événements en entrés débloques la boucle, qui produit des événements en sortie. Il y a ainsi un *flux de contrôle* passant de module en module. Les langages *Esterel* et *ReactiveML* sont membres de cette famille, utilisé dans les application musicales, les jeux vidéos, les simulation physiques, et plus récemment la programmation web.

En programmation synchrone à *flux de données*, tout est un flux temporel de valeurs de type constant. Les *noeuds* du programme lient déclarativement, par des équations, les flux en entrée et ceux en sortie. Durant un instant synchrone, tout les flux sont mis-à-jour, faisant de ces programmes des cas particuliers des *processus par réseaux de Kahn* pouvant communiquer *sans mémoire tampon* (c'est important). Les programmes synchrones utilisent des opérateurs spécifiques pour avoir accès aux valeurs d'instants passés, de pousser des valeurs dans le futur d'un flux, et de bloquer leur flux en l'attente d'une condition. Ces blocages induisent des *horloges* dans chaque flux, et les valeurs n'existent que quand les horloges le prescrive: toutes les valeurs ne sont pas calculés à tout les instants. Le programme synchrone se compile en une boucle simple, qui fait un tout par instant, parfois même sans allocation dynamique. Dans la famille synchrone à de donnée, on note *Lustre*, *Signal* et *Lucid*. On y ajoutera la notre, ayant choisi ce paradigme de programmation adapté au monde de l'électronique et peu gourmand en ressource.

Sûreté des programmes

Les applications de ces langages sont nombreuses dans l'industrie de haut-niveau. La plateforme industriel *SCADE* regroupe la programmation synchrone avec un éditeur de programme visuel et un générateur de code certifié. Cela permet de faire conformer les systèmes aux normes internationales draconiennes qui régulent l'industrie du transport ferroviaire et aérien, et celui de l'énergie.

En effet, toute défaillance d'un système embarqué dans ces application peu causer des accidents pouvant aller jusqu'à mort d'homme (pensez aux contrôleurs des centrales nucléaires). Dans ces *systèmes critiques*, la sûreté des systèmes, c'est à dire la garantie que des classes de comportement indésirables soient impossibles, est l'enjeu majeur. Cependant, la programmation de microcontrôleur se fait encore dans des langages offrant peu ou pas de garanties de sûreté. Notre intérêt dans cette thèse est donc de garantir des propriétés fortes de sûreté pour les programmes synchrones à flux de données sur microcontrôleurs, en se basant sur l'analyse statique et le typage, en spécifiant formellement nos outils et en développant une méta-théorie associée.

Le typage statique nous permet de garantir l'absence d'une grande classe d'erreurs classique, comme le déréférencement de pointeur nul, la réalisation

d'actions physiques indésirables, la destruction de composant. Par exemple, on utilisera les GADT d'OCaml pour garantir certaines interaction de bas niveaux correctes, et ce sans coût à l'exécution.

L'analyse statique nous permettra d'éclairer la validité de l'hypothèse synchrone (voila notre promesse tenue). On rappelle que pour qu'un programme synchrone fonctionne, il faut pouvoir négliger le temps de calcul de l'instant synchrone face au temps de réponse exigé par le système qui embarque le programme. Il faut garantir, que dans le pire des cas possible, le temps de calcul reste acceptable. Ce *Worst Case Execution Time (WCET)* est la plupart du temps estimé empiriquement comme la borne supérieure du temps d'exécution dans un banc de test se voulant représentatif. Mais on rappelle que la représentativité est le seul idéal auquel ses tests peuvent espérer: il est impossible de tester sur tout l'espace de recherche, même pour des microcontrôleurs ($2^{8kodeRAM} = 10^{2400}$ cas possibles). Il est donc nécessaire de faire appel à un raisonnement statique pour garantir le WCET, quitte à rejeter certains programmes subtilement corrects.

On peut borner le WCET d'un programme en générant son graphe flux de contrôle, puis en extrayant des contraintes linéaires du pire chemin dans ce graphe, que l'on maximise par programmation linéaire. Une autre approche basée sur la syntaxe consiste à borner co-inductivement les WCET de parties de plus en plus grandes du programme jusqu'à avoir une borne sur le programme complet. Dans tout les cas, ce calcul de WCET est rendu hautement complexe par l'usage dans les processeurs modernes de techniques d'optimisation de corrections douteuses, comme des pipelines de l'ordre de 100 instructions, des prédictions de branchement, ou des caches dans des caches dans des caches… Heureusement, la simplicité du matériel des microcontrôleurs nous sauvegarde de ce genre de problèmes. Le calcul de WCET en vrai temps physique y est concevable.

La formalisation de nos méthodes permettra de garantir la sûreté des programmes et leurs WCET. Elle se fait avec des outils comme *Coq* ou *Isabel-HOL*, qui permettent la génération de code. Ainsi, à partir de la spécification, il est possible de générer des outils *corrects par construction.* le compilateur C *CompCert*, prouvé correct par Coq, peux par exemple être couplé à un générateur de code C certifié pour former une chaîne de compilation sûre du code source, au code-objet, en passant par le bytecode.

Pour résumer, nous allons utiliser OCaml et la programmation synchrone pour permettre une programmation sûre sur microcontrôleur. La sûreté du programme sera garantie statiquement par la couplage des deux outils. Nous allons implémenter une VM à basse ressource et une extension synchrone d'OCaml, et développer une analyse automatique de WCET pour les programmes utilisant cette plateforme. La vérification formelle de cet outillage dépasse le cadre de cette thèse, mais des propriétés importante, notamment sur notre extention synchrone, sont prouvés avec Coq.