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 s'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.

OMicroB

Où l'on développe une machine virtuelle générique et configurable pour l'exécution du byte-code OCaml sur plusieurs familles de microcontrôleurs à faibles ressources, et l'on déploie des efforts d'optimisation de taille de code et d'usage mémoire

Le langage OCaml

On passe très rapidement sur cette partie.

OCaml est un langage supportant la programmation fonctionnelle et modulaire, ainsi que objet et impérative. Les constructions du langage sont:

- Les valeurs fonctionnelles (fermetures)
- Les tuples & les projections
- Les (G)ADT & le filtrage par motif
- Les enregistrements & les variantes polymorphes
- Les références et la mutabilité
- Les objets & les classes
- Les exceptions
- Les modules & *foncteurs* (c.à.d. fonctions entre modules)

Ocaml supporte un typage statique avancé, muni de

- polymorphisme structurel
- inférence de type
- type principal (c.à.d. le plus général possible)

Et un *runtime* avec gestion automatique de la mémoire par ramasse miette.

La machine virtuelle ZAM

La machine virtuelle ZAM (ou juste "la ZAM") est développée depuis 1996 à l'INRIA, et sert d'environnement d'exécution standard à OCaml. C'est une implémentation de la *machine de Krivine*, à appel par valeur (c.à.d. une sémantique stricte du λ-calcul). Les programmes OCaml, compilés en bytecode, sont soit exécutés sur la ZAM soient compilés d'avantage vers du code objet natif pour une variété de plateformes.

Le bytecode ZAM se compose de 148 instructions (pour OCaml 4.06), pas toutes orthogonale (p.ex., `PUSHACC1` = `PUSH; ACC1`). On peut consulter à loisir le projet *Cadmium* pour l'intégralité de la sémantique. Un fichier de bytecode ZAM contient six sections:

- `CODE` pour les instructions bytecode elles-même
- `DATA` pour les constantes
- `PRIM` qui liste les primitives utilisées est leurs "nom" dans le programme
- `DLLS`, la liste des bibliothèques externes à lier dynamiquement avec le programme

- `DLPT`, la liste des bilbiothèques OCaml utilisés
- `DBUG` (optionelle) pour les informations de débogage.

Les fichiers bytecode contiennent en plus du code propre du programme compilé, un prélude initialisant l'environnement du programme, et le code du module `Pervasives`, qui est chargé avec chaque programme OCaml.

La ZAM est une machine virtuelle à pile, munis des registres classiques `sp`, `pc`, `global_data` et `acc`, mais aussi `trapSp` pointant vers le récupérateur d'exception courant, `extra_args` qui dénote le nombre d'argument saturant l'appel curryfié courant, et `env` qui pointe vers la fermeture courante.

Les données manipulées par la ZAM ont une représentation uniforme, afin de supporter le polymorphisme d'OCaml. Comme tout ce ressemble au runtime, pas besoin de prendre des pincette avec les arguments de type quantifiés. Chaque valeur prend un mot en mémoire (32 ou 64 bits), et le bit de poids faible indique si cette valeur est immédiate, ou un pointeur sur le tas. Comme les pointeurs doivent être alignés sur un mot dans les plateformes usuels, ce bit est déjà nul sur tout les pointeurs valides. Les valeurs allouées sur le tas (on dit *enboité* en bon OCaml), possèdent un en-tête stockant des méta-données de taille, de couleurs pour le GC, et un tag distinguant les types dynamiques (les fermetures ont le tag 247 par exemple). Pour des raisons d'efficacité, les pointeurs vers les valeurs *enboités* pointent vers le premier mot *après* l'en-tête, qui est le premier mot utile de la valeur pour le programme.

La ZAM possède une riche bibliothèque d'exécution, implémentée majoritairement en OCaml puis compilée en bytecode, et complémentée avec du C. La mémoire allouée dynamiquement par la ZAM est libérée par une ramasse-miette hybride. Les valeurs sont allouées dans un tas mineur qui est nettoyé par un algorithme *Stop-and-Copy*. Les valeurs survivantes sont copiés dans le tas majeur, qui est traité par un *Mark-and-Sweep* incrémental, servant aussi à défragmenter le tas.

Compilation et exécution avec OMicroB

OMicroB implémente une machine virtuelle pour le bytecode ZAM en C, profitant ainsi de ubiquité des compilateurs C sur les plateformes de microcontrôleurs. La chaînes de compilation est la suivante: Le code OCaml est compilé en bytecode par le compilateur `ocamlc` standard, puis nettoyé par `ocamlclean`. Il est ensuite empaqueté dans un tableau C statique, puis envoyé avec le code source de la VM au compilateur C du microcontrôleur cible.

Le nettoyage du bytecode OCaml est une étape nécessaire pour garantir une taille minimale de l'exécutable final. En effet, `ocamlc` inclut dans le bytecode toutes les valeurs et fermetures définies dans chaque module compilé, afin de respecter les interfaces que ces modules déclarent, et donc de permettre le chargement dynamique de module. Ainsi, le programme OCaml vide `let _ = ()` produit 2279 instructions ! L'outil `ocamlclean`, développé pour le projet `OCaPIC`,

permet de supprimer ce code mort du bytecode des programmes compilés pour microcontrôleurs, où l'on ne fait pas usage du chargement dynamique. Le programme vide passe alors à 81 instructions.

Le bytecode nettoyé est ensuite envoyé à `bc2c`, qui produit un fichier C valide stockant les différentes sections du fichier bytecode dans des structures C valides, surtout des tableaux d'énumérations. Il définit aussi la pile et le tas comme des tableaux de taille fixe. La représentation des instruction est plus compacte que sur la machine ZAM, qui aligne ses instructions sur 32 bits. `bc2c` utilise une représentation à longueur variable, avec un octet par entrée dans le tableau d'instruction. Le code résultant est en moyenne 3.5x plus petit. Les compilateurs en aval pouvant représenter ce type de *mémoire programme* dans le stockage flash, `bc2c` utilise cette fonctionnalité dès que possible.

L'interprète d'OmicroB reprend la sémantique de celui de la ZAM, avec les mêmes registres. Une différence majeure est que le registre `pc` pointe désormais sur la mémoire flash. Aussi, le branchement dynamique basé sur le hashage, qui utilise 31 bits de hash dans la version standard, est remplacé par un hashage sur 15 bits, afin de tenir sur 2 octets dans la RAM. Cela permet le support complet des objets et des variantes polymorphes.

Représentation des valeurs

On peut paramétrer OmicroB pour imposer une représentation uniforme des données sur 16, 32, ou 64 bits. Contrairement à la ZAM, les flottants ne sont pas alloués sur le tas. Pour supporter la primitive `compare : 'a -> 'a -> bool`, il est nécessaire d'adapter la représentation des flottants pour la rendre compatible avec la relation d'ordre sur les entiers machines. Les flottants négatifs sont donc représentés en complément à deux. Une opération XOR booléenne doit être effectuée pour permettre les opérations arithmétiques standard sur les flottants.

Contrairement à la machine ZAM où la distinction valeur immédiate/valeur enboité est faite sur le bit de poids faible, les flottants immédiats en complément à deux impose de choisir une autre distinction. Autrement, le GC ne pourra pas inspecter les valeurs et suivre les pointeurs alloués par le bytecode non-typé. Du fait du faible espace mémoire des microcontrôleurs, on peux profiter du *NaN-boxing* pour encoder les pointeurs jusqu'à 2Mo dans la mantisse des flottants NaN en configuration 32 bits, et »1Go dans la configuration 64bits. Les valeurs NaN des flottants sont comprimées dans un unique flottant NaN, où seul le 21ème bit de la mantisse est égal à 1. Dans la configuration à 16 bits, on impose que le dernier bit de la mantisse soit nul (passage de 10 à 9 bits de précision).

Runtime et bibliothèque standard

On garde la plupart des modules de la bibliothèque standard OCaml tels quels, ceux n'ayant aucun sens (`Unix`) ont été retirés, et d'autres (`Avr`, `LiquidCrystal`) ont été créés spécialement pour l'usage avec les microcontrôleurs. Les GADT

permettent d'encoder quels bits de quels registres spéciaux peuvent être configurés à quel usage:

```
type 'a register =
  | SPCR : spcr_bit register
  | SPSR : spsr_bit register
  ...
val set_bit : 'a register -> 'a -> unit
```

On peut ainsi éviter des erreurs subtiles, que la programmation par macro de C ne permet pas de détecter.

Le ramasse-miette implémente un algorithme $Stop\text{-}and_{Copy}$, simple mais rapide. Il existe un tas *vivant* et un tas de *copie*. Lors du ramassage, les valeurs sont copiées du premier vers le second. Quand le ramassage est terminé, on inverse les tas. Cet algorithme rapide souffre d'un doublage de la taille du tas, mais offre un gain de temps d'exécution. Nous avons aussi implémenté un ramasse-miette *Mark-and-Sweep*, plus lent mais fonctionnant sur un seul tas.

La dernière étape de la compilation est la génération de code natif depuis le C. Utiliser C comme assembleur portable permet de compiler pour plusieurs familles de microcontrôleurs, et de compiler avec un `gcc` natif pour débogage sur la machine de l'utilisateur. De plus, le portage de la VM d'OmicroB ne nécessite que des modification *a minima*, pour prendre en compte les différences entre les différentes plateformes.

Enfin, OmicroB, à l'image de son ancêtre OCaPIC, dispose d'un simulateur de circuit, permettant de décrire le microcontrôleur et son environnement électrique, et de simuler, par exemple, des boutons et des écrans OLED.

Optimisations

La réduction de la taille des opcodes de la ZAM de 32 à 8 bits n'est qu'une des optimisations que nous avons apportées à notre VM pour l'adapter aux environnements à faible mémoire que sont les microcontrôleurs.

Premièrement, nous avons implémenté une forme d'évaluation partielle des programmes bytecode OCaml. Notre composant `bc2c` peux en effet exécuter la phase d'initialisation du programme lors de la compilation, avant de sérialiser le tas et la pile tels qu'ils seront avant la première entrée/sortie dans le fichier C.

Aussi, les 148 instructions de la ZAM (encore plus si l'on compte nos spécialisations à différentes tailles d'arguments) ne sont pas toute nécessaire à exécution d'un programme particulier. Il n'est donc pas nécéssaire d'inclure de code C les implémentant. l'outil `bc2c` se dispense donc d'inclure le code C des opcodes non présents dans le programme à compiler. Les techniques plus avancées, d'inlining notamment, permettraient de gagner encore plus de temps, mais nous avons jugé qu'elles ont des coût prohibitifs dans le cadre des microcontrôleurs.

OCaLustre

Où l'on définit une extension synchrone d'OCaml pour la programmtion synchrone – inspirée de Lustre, libérant ainsi l'espace mental des programmeurs de OmicroB des menus problèmes de concurrence, et présente son système de typage d'horloge assurant le cadencement correct des programme

Syntaxe

Dans la pratique le code OCaLustre vit dans les fichiers OCaml normaux, et les noeuds sont déclarés avec la directive `let%node`, de la manière suivante:

```
let%node plus_moins (x,y) ~return:(p,m) =
  p = x + y;
  m = x - y
```

OCaLustre supporte:

- Les entiers, flottants et booléens;
- Avec les opérations idoines;
- Le retard initialisé `k0 >>> k`, qui implémente l'initialisation et l'opérateur de retard classique `pre`;
- Les flux locaux (définitions internes aux noeuds);
- L'application d'un noeuds à des flux dans un autre noeud;
- L'appel de fonctions OCaml avec `call`;
- Le sous-échantillonnage, appliqué à une équation de flux ou à une expression;
- Et `merge`, qui opère la fusion de deux flux sous-échantillonnés complémentaires;

On note que certains opérateurs exigent de se voir fournir des constantes d'initialisation pour bien typer, bien qu'elles ne soient pas strictement utiles (*c'est le prix de la sûreté…*) Aussi, l'application de noeud est techniquement l'application de *L'instance* d'un noeud. L'appel de fonctions externes se comporte comme un foncteur de d'OCaml vers OCaLustre, des valeurs ponctuelles vers les flux. Mais il faut faire attention à ne pas briser les hypothèses de concurrence. La mutabilité illimité d'OCaml est par exemple à proscrire, par l'ordonnancement des corps des noeuds entre eux n'est pas défini statiquement. De même, les fonctions partielles font toujours planter les programmes, pas de miracle ici.

Typage & Cadencement

OCaLustre admet un typage classique à la ML d'un intérêt théorique… faible. Ce qui vaut le détour c'est le typage des horloges. Chaque noeud possède une horloge interne implicite, mais le programme compilé fonctionne avec une boucle explicite qui tourne chaque instant global. Le typage impose que les horloges

internes forment un treillis, dont le minimum peut être alors calé sur l'horloge globale.

Les constantes sont de type polymorphe, par exemple, `2` : $\forall\bullet, int$, et on a une surcharge des opérations sur `int`, `float` et `bool`, qui peuvent être évaluées sur des flux avec la même syntaxe que pour les valeurs OCaml. Les noeuds ont eux un *schéma de type* quantifiant leur horloge interne. En général, le type d'un noeud est $\forall\bullet, (x_1 : ck_1 \times ... \times x_n : ck_n) \rightarrow (y_1 : ck'_1 \times ... \times y_n : ck'_n)$, avec les ck_i et ck'_i calés sur l'horloge interne $\bullet$. L'élaboration des types des instances des noeuds se fait par substitution de l'horloge $\bullet$, puis par celle des noms de ces arguments formels (mais pas de *leurs horloges*, on ne substitue que $\bullet$). Ce choix simplifie la compilation séparée des noeuds, sans pour autant causer une trop grande perte d'expressivité.

OCaLustre supporte *l'application conditionnelle*: en fonction de la position dans l'équation de l'échantionnage par `when`, on peut provoquer *ou* pas l'exécution des noeuds qu'elle appelle: soit le noeud n'est pas mis à jour, soit il l'est est son résultat est ignoré. Bien faire la différence avec la mise à jour conditionnelle des flux.

Le système de type d'horloge garantie la cohérence du modèle synchrone d'OCaLustre par rapport à sa traduction en OCaml: aucune valeur non-présente à un instant t pour cause de sous-échantillonnage n'est lue pour calculer l'instant $t + 1$. Les règles importantes du système sont: la fusion de flux complémentaires, calés sur ck **on** x et ck **onnot** x respectivement pour former un noeud calé sur ck, et l'instantiation de schéma de type d'horloge des noeuds aux sites d'applications des dits noeuds.

On note que le système de type ne garantie pas que seules les valeurs utiles sont calculées. Il existe alors plusieurs solutions opérationnelles à la sémantique d'un programme OCaLustre valide. L'ambiguïté que cela cause est résolue statiquement par un ordonnancement et un mécanisme de causalité, au coeur de la compilation d'OCaLustre vers OCaml.

Compilation

OCaLustre est compilé par un pré-processeur qui se greffe au compilateur OCaml standard via l'interface PPX, et est activé par la construction syntaxique `let%node`. IL renvoie un AST OCaml qui peut être rendu tel quel dans un fichier de syntaxe concrète ou compilé vers du bytecode ou du code natif, indépendamment de tout usage d'OmicroB. Contrairement à l'approche suivi par Lustre, on l'inline pas le corps des noeuds, afin de limiter la duplication de code, et donc l'empreinte mémoire du bytecode résultant.

La compilation procède en quatre étapes:

- La **normalisation** réduit la syntaxe du langage à sa syntaxe dite *normale*;
- Puis l'**ordronnancement** trie au sein de chaque noeud les diverses équations et détecte les boucles de causalité au sein d'un même instant syn-

chrone;
- Ensuite, l'**inférence des types d'horloge** rend le cadencement explicite au sein des équations conformément aux règles définies plus haut;
- Enfin, une étape de **traduction** vers OCaml termine le pipeline de compilation.

La normalisation a pour but de placer dans des équations séparées les expressions à effets de bord, afin de pouvoir implémenter la conditionnelle d'OCaLustre en la traduisant en celle d'OCaml. Si les sous-expressions peuvent causer des effets de bord, les deux sémantiques dévient pour cause de cadencement. Les opérations à effet de bord sont:

- `>>>` est les constructions associées. Il est nécéssaire de séparer les arguments droits dans leurs propres équations afin de pouvoir conserver systématiquement les anciennes valeurs des noeuds, même si l'opérateur `>>>` est gardé par un sous-échantillonnage.
- Idem pour le corps des noeuds, que l'on peut évaluer pour leurs effets. Donc, la construction `f a b c …` est extraite vers sa propre équation.
- Et enfin les appels à OCaml `call f a b c …`, encore pour les mêmes raisons.

La normalisation procède en deux temps: d'abord, elle "explose" les tuples à gauche des équations en des équations sur leurs composantes. Puis la normalisation à proprement parler à lieu, en produisant pour chaque noeuds des équations supplémentaires.

L'ordonnancement sert à déterminer dans quel ordre causal les équations d'un noeud sont évaluées dans la boucle principale *au cours d'un même isntant*. On procède en collectant pour chaque flot les identifieurs desquels il dépend au cours d'un instant. On construit alors un graphe orienté de dépendance, qui s'il est acyclique, admet un tri topologique qui induit l'ordonnancement dans le noeud. Dans le cas contraire, on abandonne la compilation avec un message d'erreur. À cette étape, certains programmes valides peuvent être rejetés à cause de notre approche de compilation séparée: Il peut exister des cycles de dépendances artificiels, par exemple où une dépendance est gardée par `>>>`, et donc sans implication pour l'ordonnancement, mais cachée dans un autre noeud. La détection de ce non-cycle exige un inlining, que l'on se refuse de faire. Cette limitation parait acceptable pour éviter de recopier le corps d'un noeud.

L'inférence de noeud est inspiré de l'inférence à la Hindley-Milner et par les travaux de Colaço & Pouzet, mais on restreint le polymorphisme en ne quantifiant que sur l'horloge de base d'un noeud. Les horloges inférés pour les constantes et les équations sont explicitées dans l'AST.

Enfin, le code explicitement cadencé est traduit en OCaml. Les valeurs n'existant pas à un instant données sont à ce point garanties de ne pas être lues, mais le typage OCaml peut exiger que l'on les produisent. On utilise alors la construction "dangereuse" `Obj.Magic : unit -> 'a` pour produire la valeur inutile. Les expressions sont traduites en éliminant les `when` et `whennot` et en

compilant `merge x a b` en `if x then a else b`, ce que l'on peut faire car les effets de `a` et `b` sont systématiquement évalués. C'est là que la normalisation à sa justification.

Les instances des noeuds sont traduits en allouant des références pour chaque usage de `>>>`. Ces registres mutables sont mis à jour lors de la boucle principale, et forment la mémoire du code synchrone hors de l'instant présent. De même, chaque appel à un noeud crée un instance de ce noeud étiqueté afin de dés-ambigüer l'usage des différentes instances au sein d'un même noeud. Chaque équation est traduite par un `let y = guard ck ... in ...`. La fonction `guard` empèche l'exécution d'une équation si son horloge n'est pas active, et produit un n-uplet de `Obj.magic ()` (possiblement réduit à la seule magie si `n = 1`). Les fonctions `on` et `onnot` sur les horloges sont implémentées par les opérateurs booléens afférent: `ck on x` est en fait `ck && c`, etc. Enfin, la boucle principale de chaque noeud mets à jour les registres mémoire induit par `>>>` avant de terminer.

La boucle principale d'un noeud du type $x_1, ..., x_n \to y_1, ...y_m$ est traduite par une fermeture du type OCaml `unit -> (x1 * … * xn) -> (y1 * … * yn)`. Le compilateur OCaLustre va, enfin, produire un squelette de l'interface du système de noeud que constitue le programme, afin de permettre à l'utilisateur d'implémenter les entrées-sorties en OCaml.

Formalisation

> *où l'on donne une sémantique formelle à un pseudo-ML dans lequel on peut traduire OCaLustre sans perte d'information de typage, pouvant ainsi qu'un typeur spécifique à ce dernier langage est inutile; et l'on introduit un "vérificateur d'horloge" issu de la spécification formelle du bon-cadencement des noeuds OCaLustre*

Typage algorithmique formel

La formalisation en Coq de la traduction OCaLustre vers OCaml permet de prouver l'équivalence entre le bon-typage dans ces deux langages. La traduction ne perd pas d'information de typage et ne peux pas introduire d'erreurs. On peut alors se dispenser d'un typeur spécifique à OCaLustre: un programme snychrone est bien-typé ssi sa traduction l'est aussi.

La preuve de cette équivalence procède en passant par une représentation à la ML permettant des définitions mutuellement récursives de variables, nommément `let … with … with … … in`. En OCaml, les définitions mutuellement récursives ne peuvent porter que sur les valeurs fonctionnelles. De plus, notre représentation intermédiaire distingue les identifieurs introduits par la traduction des identifieurs originaux du code OCaLustre. La traduction du code synchrone suit celle introduite à la section précédente, à la différence que les listes d'équations sont traduites par la définition simultanée `let … with … in`

…. L'équivalence du typage entre ces deux formes et d'abord prouvée pour les noeuds exposants tout leurs flux, puis étendue dans un second temps à ceux qui utilisent des flux internes.

Il reste ensuite à définir l'équivalence du typage entre le code avec définitions simultanées et le code ML classique avec `let … in …` imbriqués. Cette équivalence n'existe que pour les noeuds bien ordonnancés. Cela s'explique par le fait que l'imbrication des `let … in …` reproduit la chaîne causale entre les différents flux d'un noeud. L'équivalence est donc établie sous hypothèse de bon-ordonnancement, ce qui permet de conclure. La typage dans OCaLustre et OCaml sont équivalent, et l'on a donc pas besoin de typeur dédié. On notera que la preuve ne porte que sur les programme à un seul noeud, il faudra considéré l'appel de noeud pour conclure sur tout le langage.

Typage formel d'horloge et extraction d'un vérificateur

Le système d'horloge permet d'associer à des flux des conditions de présence, et donc de s'assurer que le programme synchrone n'accède pas à des valeurs non-existantes. La plupart des opérateurs du langages opèrent sur des flots cadencés sur la même horloge. Nous avons certifié avec Coq que notre système de type d'horloge garantie cette propriété, et en avons extrait un *vérificateur d'horloge* algorithmique inclut dans notre compilateur. La vérification à lieu après l'inférence (non-vérifiée) des horloges, et rends notre compilateur *certifiant*. On note que des noeuds bien cadencés peuvent quand même ne pas émettre des valeurs, mais que cela n'impacte justement pas le bon déroulement du programme.

La vérification d'horloge procède en commençant par les expressions: les constantes, constructeurs et opérateurs de sous-échantionnage sous annotés avec les horloges inférés, qui sont vérifiées. Le passage des expressions aux équations demande de gérer l'appel de noeud: l'instanciation des variables d'horloges, implicites (comme $\bullet$) ou explicite (comme `on` y), est vérifié par la présence d'une substitution de support adéquat qui transforme le schéma de type d'horloge du noeud appelé en type effectif au site d'appel. Les schémas de types des noeuds vérifiés sont séquentiellement disponibles pour la vérification des noeuds suivants.

L'extraction des règles de typages décrites avec l'outil *Ott* vers Coq créent des types inductifs décrivant les dérivations valides dans notre système d'horloge. Nous avons implémenté en Coq des version algorithmiques de ces règles en forme de procédure de décision, et avons prouvé l'équivalence des systèmes déclaratifs et algorithmiques. Il nous a suffit ensuite d'extraire la version algorithmique de Coq vers OCaml et de l'intégrer comme phase à notre compilateur. On pourrait étendre cette preuve à l'adéquation du système d'horloge à la sémantique formelle d'OCaLustre décrite section 3, et achever de vérifier formellement la validité du sytème.

Calcul du WCET

Où l'on définit un bytecode idéal et non-déterministe sur lequel porte une analyse de pire temps d'exécution par interprétation exhaustive, et l'on porte cette analyse sur OmicroB avec un nouveau mode de compilation "non-allouant".

WCET idéal en Coq

L'approche VM de OmicroB justifie une approche *byte-crawler* du calcul de WCET, paramétrée par un coût des instructions de bytecode. On introduit dans Coq un bytecode ideal avec assignement à des identifieurs, déclaration, arithmétique, branchement conditionnel et arrêt. L'état de la machine virtuelle idéale est $\sigma = (P, pc, M)$, c.à.d. un programme P, un pointeur de programme pc et une mémoire M des identifieurs vers les entiers. Muni d'une sémantique opérationnelle, on définit les *traces finies* comme les traces ayant l'arrêt comme dernière instruction. Avec une métrique de coût par instruction, on obtient un coût par trace en sommant les $P[pc]$ pour chaque σ de la trace.

Pour modéliser les entrées/sorties, on ajoute pour signifier les valeurs inconnues en mémoire. Un *effacement de $ $* est un état où des valeurs mémoires sont inconnues, et autrement identique à σ. On étend cette définition aux effacements de traces. Le *maximum de coût d'un état* est alors le maximum de coûts des traces que σ peut engendrer (on considère alors les branchements aux résultats non-deterministes). On prouve dans Coq que pour tout effacement d'un σ sans inconnu, la trace de σ est dans l'ensemble des traces de cet effacement. Cela justifie que, dès qu'un programme à des traces *pouvablement* finies, on peut borner son coût en calculant les coûts pour tout les effacements possibles.

WCET réel des programmes OCaLustre

Hors usage d'appel à OCaml, les programmes OCaLustre se compilent tous en fonctions de calcul d'instant de traces nécessairement finies. Notre recherche de borne fonctionne donc pour OCaLustre. Comme les microcontrôleurs ont des modèles mémoire simple, sans caches ou prédiction de branche par exemple, on peut en effet sommer un coût par instruction pour avoir un coût par trace correct.

Nous avons obtenu grâce à l'analyseur statique pour C *Bound-T* des coûts pour les primitives de la VM d'OmicroB (implémentation des instructions + entrée-sortie). Une fois ce pré-calcul effectué, on peut utiliser la méthode décrite plus haut pour avoir un WCET.

Le problème majeur de cette méthode est que nous ne considérons pas le GC d'OmicroB. Hors bornage très pessimiste du coût par instruction d'un lancement du GC, on doit adapter notre approche. On a implémenté un autre pipeline de compilation dit *non-allouant*, qui évite pour OCaLustre les allocations sur le tas. On associe à chaque noeud un ADT mutable pour son état interne, allouable

sur la pile et ré-écrit en place à chaque instant synchrone. On note que la partie algorithmique en OCaml n'est pas prise en compte ici.

L'approche donnée ici, qui consiste à factoriser l'analyse statique du code OCaLustre pour différentes plateformes, peux être étendue à d'autres mesures, comme la taille de la pile ou le nombre de cellules allouées par exemple.

Performances, Applications, Ouvertures

Performances d'OmicroB

les performances de la VM d'OmicroB peuvent être configurées par les options suivantes:

- la longueur des mots sur 16, 32, 64 bits
- Le GC avec un algorithme *Stop-and-Copy* induisant un doublage de la taille du tas, ou *Mark-and-Compact*, plus lent
- La pré-exécution du bytecode jusqu'à la première entrée-sortie
- La taille de la pile
- La taille du tas, qui influe sur la fréquence d'activation du GC
- Le nettoyage du code mort de la VM (celui des instructions non-utilisées).

En comparant OmicroB et la ZAM, on voit que notre VM est 2-3x plus lente que l'implémentation standard d'OCaml. Causes: le non-alignement des instructions sur la tailles des mots sur architecture x86(-64), et l'activation plus courante du GC, le tas ne faisant que quelques kilo-octets. L'augmentation de la taille du tas permet de réduire le temps d'exécution de 20%-33%.

Sur microcontrôleurs *Arduino Uno*, on obtient 200K instructions OCaml par secondes, et 30K si utilisation massive des flottants, avec doublement du temps d'exécution si l'on double la taille des mots (de 16 à 32bits sur une architecture 8bits). En comparaison avec OCaPIC, qui compile le bytecode OCaml en assembleur PIC, OmicroB est 3-4x plus lente. Mais c'est une extrapolation des tests précédents, et de plus OCaPIC bénéficie d'une meilleure optimisation du code au coeur de l'interpréteur. La comparaison avec MicroPython (encore en extrapolant) montre un avantage de l'ordre de 1.5 à 2 pour OmicroB.

L'occupation de RAM des programmes est relativement prévisible, le tas et la pile de interpréteurs étant de taille fixes et alloués statiquement. La différence entre les consommations de RAM vient des différent besoin mémoire des instructions des différents programmes de tests. Cette différence influe d'autant plus sur la consommation de mémoire flash: le chargement des instructions spécifiques à la gestions des objets ou au déclenchement du GC provoque l'ajout du code idoine de la VM. Le code des modules cachés derrière les appels de foncteurs est de même systématiquement inclus, sans nettoyage par `ocamlclean`. On note que chaque programme utilise un sous-ensemble réduit des instructions de la VM, mais variable au cas-par-cas.

Performances d'OCaLustre

Un étudie les performances d'OCaLustre sur un additioneur bit-à-bit avec retenue à base de booléens et de booléens. Il est écrit dans un noeud OCaLustre, qui est combiné avec une copie de lui-même pour former un additioneur sur 2bits, puis 4bits et enfin 8bits. On fait calculer 100000 fois l'instant synchrone additionant 255 et 255. Cette exécution demande sur PC une pile 80 mots, un tas de 400, et 1096o de RAM. L'usage de ressource d'OCaml est donc faible. Le temps d'exécution se stabilise autour de 0.23s quand la taille du tas alloué dépasse effectue le "mur de GC" vers 1200 mots. Avec un tas de 400 mots, on 5 appel de GC par instant synchrone. La version *no-alloc* admet des performances comparable à celle avec "beaucoup" de tas, soit 383M d'instruction bytecode par secondes.

La comparaison avec Lustre Synchrone, projet abandonné mais encore fonctionnel, montre bien l'efficacité d'OCalustre dans les situations où la mémoire est faible: Lustre est 4x plus lent sur PC. L'usage par ce dernier de variantes polymorphes et de nombreuses références vers des n-uplets rempli rapidement le tas. Comme Lustre génère aussi de l'OCaml, il est compilable avec OmicroB, et nous avons étendu la comparaison sur microcontrôleurs. OCaLustre sans allocation est 6x plus rapide dans un contexte ou le tas est maximisé à ~1.6ko.

Des tests sur PC de programmes synchrones exécutés sur 1 millions d'instants synchrones valident nos résultats de faible surcoût en vitesse et faible empreinte mémoire. OCaLustre est bien compatible avec des environnement d'exécution de 2ko de RAM.