Gestion mémoire multi-stratégie pour une plate-forme multi-languages

A. Lux[1] & B. Zoppis[2]

1: Laboratoire GRAVIR-IMAG,
INRIA Rhônes-Alpes
655 Avenue de l'Europe
38330 Montbonnot-Saint-Martin
`Augustin.Lux@imag.fr`
2: Laboratoire GRAVIR-IMAG,
`Bruno.Zoppis@imag.fr`

Résumé

Nous présentons un système développé au cours de ces dernières années pour les besoins de nos recherches en Vision par Ordinateur et Robotique Mobile. Il s'agit d'une plate-forme multi-language nommée RAVI (Robotique, Apprentissage, Vision, Intégration) permettant de combiner les avantages des langages C, C++, Scheme, Prolog et des systèmes de productions. Ce système intègre une machine virtuelle, un ramasse-miettes générique, et la possibilité d'extension dynamique du système par des modules écrits dans ces langages.

1. Introduction

Dans des domaines comme la Vision par Ordinateur et la Robotique, on trouve un ensemble important de travaux sur “l'intégration de systèmes” ; voir [CC94] pour un état de l'art significatif. Tous ces travaux partent d'un ensemble de programmes écrits en C, parfois en C++, et cherchent à ajouter une surcouche interactive et/ou interprétative, en vue d'aborder les problèmes de contrôle, ou pour faciliter l'expérimentation, et l'intégration de modules.

A notre avis, ajouter ainsi des nouvelles fonctionnalités à un système spécialisé conduit à une impasse, car on ne résout pas les problèmes dans leur généralité comme le font les concepteurs et implémenteurs de langages. Par contre, la disponibilité d'un système multi-langage général, et pour commencer d'un système “C/C++ – Lisp”, serait un avantage décisif pour ce type de travaux. En particulier, nous avons besoin d'un environnement dans lequel il est possible d'intégrer des programmes d'Intelligence Artificielle - générateur de plan, contrôleur d'exécution, resolveur de problèmes - avec des programmes de traitement d'images, de commande de robot, etc.

Le système RAVI tient son nom de ce contexte : il doit permettre l'Intégration et l'Apprentissage en Robotique et en Vision. Après avoir analysé quelques systèmes existants à l'époque (1993), entre autres Scm, Vscm, Elk, Scheme2c, nous avons décidé de construire un système "maison" pour réaliser les nombreux objectifs auxquels nous tenons : un système globalement conçu comme une boite à outils, dans laquelle chaque module est programmé au plus haut niveau possible ; un noyau en C++, une couche supérieur en Scheme et en Prolog, des modules d'analyse et de traduction de programmes - compilateur Scheme, analyseur C++, traducteur Scheme→C. Bien entendu, tout cela n'est pas terminé. Les points forts de la version opérationnelle à présent du système Ravi peuvent être résumés par les mots-clés suivants :

- système à base de C++
- interpréteur Scheme en Scheme
- bonne efficacité, comparée, par exemple, à Scm
- structure modulaire extensible
- chargement dynamique
- génération automatique d'interfaces
- portabilité Unix
- bibliothèques spécialisées vision
- interfaces xforms, pvm, Leda

Cet article décrit, dans une première partie, les fonctionnalités et l'organisation d'ensemble du système; puis dans une deuxième partie, un problème-clé de l'implémentation: les mécanismes de gestion mémoire. En effet, les questions de langage et de gestion mémoire sont couplées - traditionnellement, le premier impose le second: le tout-automatique dans les langages fonctionnels et logiques; le tout-manuel en C, C++, Java ou Pascal; un certain mélange en Ada. Un système multi-langage doit proposer une gestion à la fois générale, fiable, et commode.

Ce point, particulièrement important dans toute application qui assemble des modules, n'est pas traité avec la flexibilité qu'il nécessite dans les systèmes actuels.

2. Un système intégré C++-Scheme

2.1. Contexte

Avant d'être multi-lingue, un système doit être "bi-lingue". A cet égard, le couple C++ – Scheme est un choix optimal; ce sont deux langages très complémentaires, couvrant des zones bien différentes dans le spectre des langages de programmation. L'intérêt de les combiner a été reconnu à plusieurs reprises : on trouve d'excellents arguments en ce sens dans la présentation du système Talk de la société iLog [DPS94, Ilo94] et à propos de Elk qui est un "Extension Language Kit" [LB93].

Par ailleurs, vu le contexte de l'équipe, notre système doit répondre à des contraintes d'utilisation sévères. Pour être accepté par un chercheur qui travaille sur un problème pointu de vision (ou d'un autre domaine), l'aspect "multi-langage" ne doit pas introduire de limitations, ou de lourdeur. Nous avons opté pour une solution avec génération automatique d'interfaces, à l'instar de Talk et de Guile; par rapport à ces deux systèmes, nos objectifs vont cependant plus loin, et nous proposons des solutions techniques sensiblement différentes, plus générales en de nombreux points.

2.2. Objectifs

Nous cherchons à réaliser un système dynamique avec une intégration "forte" : un système interactif, à l'image d'une machine Lisp, dans lequel les entités définies dans un module écrit en C ou C++ se manipulent de la même façon que des entités définies en Scheme.

D'un point de vue strictement syntaxique, cela ne pose aucun problème: les deux langages partagent le concept abstrait de fonction, (qui inclut, vu le modèle de mémoire de Scheme, le concept d'objet). Par contre, placer une fonction écrite en C++, ou les méthodes d'une classe, dans un système Scheme pose des problèmes techniques épineux, même si ce système est lui-même écrit en C++. La raison fondamentale de ces difficultés peut s'exprimer de la façon suivante: les "fonctions Scheme" sont le sous-ensemble des "fonctions C++" dont les arguments et le résultat sont du type "objet-Scheme" qui est la racine de la hiérarchie de types manipulés en Scheme. Les objets de ce type sont soumis aux contraintes très fortes imposées par le gestionnaire de mémoire de Scheme; notamment, chaque objet porte l'information de son type (typage dynamique), il doit être accessible au ramasse-miettes, et le ramasse-miettes peut éventuellement le déplacer.

Cela est incompatible avec la majorité des types manipulés en C, où le typage statique permet d'éliminer l'information de type de la représentation effective d'un objet, et où aucune règle générale de gestion mémoire n'est imposée.

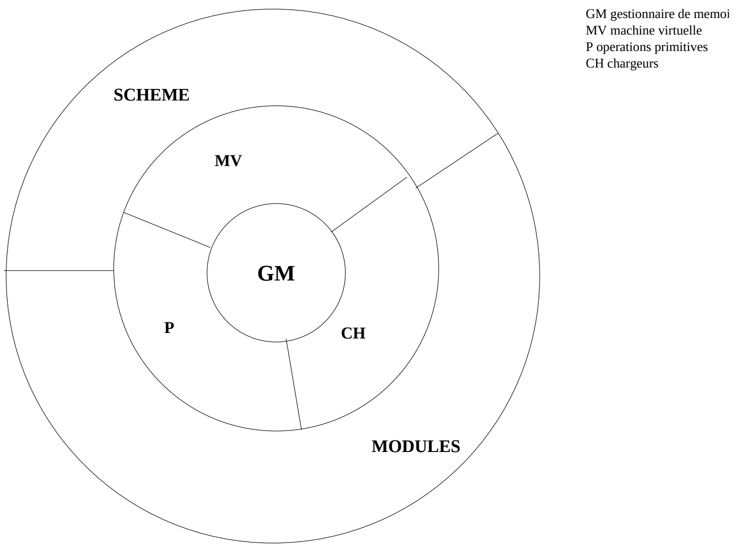
2.3. Approche choisie

La solution à ces problèmes comporte deux volets. Primo, nous ajoutons aux types de Scheme le type "objet-C" (nommé "typeC"). Un tel objet comporte deux champs : un champ "type", qui donne accès à toutes les informations sur son type C++, et un champ "valeur". Lors de l'appel d'une fonction C/C++, la première information sert à vérifier dynamiquement la légalité de l'appel, la deuxième de paramètre effectif. Secundo, il est nécessaire d'écrire un code d'interface pour chaque fonction C++. Ce code est relativement trivial pour des fonctions banales (par example une fonction `random`), mais son écriture devient franchement complexe quand il s'agit d'interfacer des sous-systèmes (voir le document très technique de Laumann [Lau93] pour s'en convaincre. Pour profiter de la puissance d'Elk il faut maitriser ces techniques!).

La génération automatique d'interfaces, d'abord introduite par Talk [1], est donc le deuxième volet de la solution. L'annexe montre un exemple qui illustre les techniques que nous avons choisies. A la différence de Talk, qui crée, pour chaque classe C++ et ses méthodes, une image dans un langage à objets en Lisp, nous créons systématiquement des fonctions, sans utiliser une extension objet de Scheme. Notre générateur d'interfaces est plus général que ceux de Talk et de Guile : il accepte à peu près toutes les constructions du langage C++ (et de C), y compris les template, surcharge, héritage (simple, pour l'instant), définitions d'opérateurs, et aussi quelques commandes du préprocesseur, fréquemment utillisées pour définir des constantes utiles. L'analyse se fait par modules, qui restent indépendants jusqu'au moment du chargement.

A l'aide de déclarations, on peut connecter les objets-C avec le ramasse-miettes de Scheme: la disparition d'un pointeur dans le monde Scheme peut déclencher une procédure spécifique au type; des procédures classiques peuvent être générées automatiquement (compteur de reférence, appel de destructeur). Une telle liaison est importante, car l'interface avec Scheme ajoute des pointeurs, dans les deux sens, entre le monde C et le monde Scheme, ce qui rend la destruction manuelle d'objets particulièrement délicate. Bien entendu, la gestion manuelle reste toujours possible.

1. Nous ne connaissions pas Talk au départ de notre projet, mais nous y avons trouvé des motivations très analogues aux nôtres.



Les 2 couches internes, écrites en C++, définissent un répertoire d'instructions. L'interpréteur ou shell du troisième niveau doit être écrit en Scheme, ou dans un autre langage compilable vers le noyau. Le noyau lui-même est ouvert : de nouvelles primitives sont ajoutées par chargement dynamique.

FIG. 1 – *Structure à niveaux du système Ravi.*

3. Une architecture à trois niveaux

Contrairement à d'autres implémentations de langages interprétés où l'interpréteur est directement écrit en C, l'architecture de Ravi est organisée autour d'une machine virtuelle dans une structure à plusieurs niveaux (cf. figures 2 et 1). Le langage pour le "bas niveau" est C++, et non C : nous avons estimé que le handicap pour l'efficacité serait largement compensé par les avantages de la programmation à objets.

3.1. Le noyau du système

Au centre de cette structure, ou au plus bas niveau, on trouve un gestionnaire de mémoire qui implémente les types natifs du système. La caractéristique essentielle de ce gestionnaire est une gestion entièrement dynamique et automatique, avec un ramasse-miettes ouvert et gnénérique, d'un ensemble de type varié et extensible ; notamment les types de base du langage Scheme, mais aussi des types pile, module de code, et typeC. Ce gestionnaire peut donc directement être utilisé pour l'implémentation d'autres langages, notamment ML et Prolog. Ce

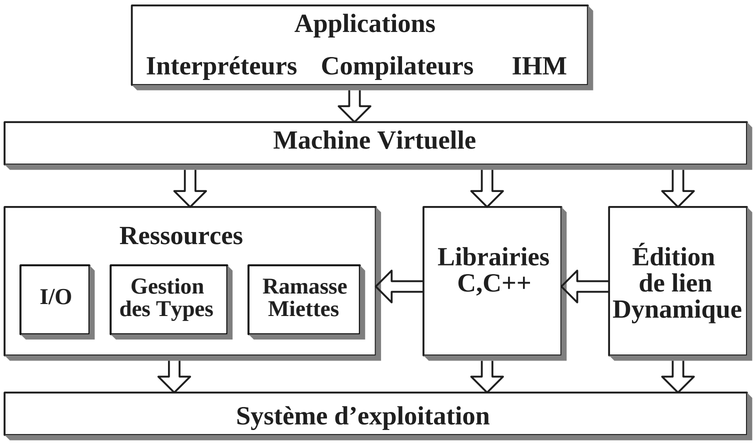


Fig. 2 – *Architecture du système Ravi*

gestionnaire de mémoire, qui se présente sous forme d'une bibliothèque de programmes en C++, fait l'objet du paragraphe 4 ci-après.

3.2. Le deuxième niveau : la machine virtuelle

Notre machine virtuelle dérive de celle utilisée par Vscm. Trois grandes fonctionnalités sont assurées à ce niveau : un mécanisme d'appel et une représentation pour les fonctions, des procédures de chargement dynamique de modules, et l'ensemble des fonctions primitives. Seul ce dernier point est spécifique au langage Scheme ; les deux autres ont un caractère beaucoup plus général.

Dans la mesure où plusieurs interpréteurs différents peuvent coexister à ce niveau, la machine virtuelle permet l'implémentation de plusieurs langages sans nécessiter de multiples couches d'interprétation.

3.3. Le troisième niveau : interpréteur et modules

Seul le troisième niveau est directement visible à l'utilisateur, notamment l'interpréteur (et le "shell" d'interaction). Il est mis en place lors du chargement initial, au lancement de la machine virtuelle, et évolue dynamiquement par la suite : tous les modules compilés (que le source soit en C++ ou en Scheme) s'ajoutent à ce niveau.

4. Gestion Mémoire

La gestion mémoire est une problématique relevant de facettes multiples : c'est à la fois une question fondamentale dans la conception d'une application, et une question "bassement technique" qui impose ses contraintes jusque dans les détail de codage des données, et des algorithmes.

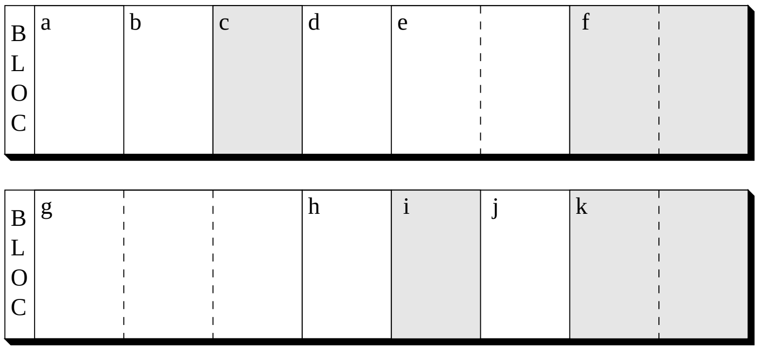
Dans une application complexe, il est précieux de pouvoir adapter la gestion-mémoire à chaque situation : on ne traitera pas de la même facon des éléments de liste, qui sont très dynamiques et occupent quelques octets seulement, et des images, plus "sédentaires", qui peuvent peser plusieurs MO. Cette situation se complique encore lorsqu'on intègre des modules logiciels existants, qui peuvent avoir chacun leurs propres méthodes de gestion-mémoire, et entre lesquels on va ajouter des liens non prévus.

4.1. Types

Dans un système à typage dynamique chaque donnée comporte une information sur son type. Cette connaissance doit également apparaître, d'une manière ou d'une autre, dans des applications à récupération implicite de la mémoire : champ "type", méthodes virtuelles de visite ou de destruction. Intégré au sein de chaque structure de donnée, cette information à un coût potentiellement élevé. Une cellule de liste comportant deux pointeurs voit sa taille augmenter de 50% si on lui ajoute un nouveau champ (en tenant compte des problèmes d'alignement). En définissant des zones mémoire où toutes les données sont du même type, cette information peut être factorisée et de simples tests de comparaison d'adresses permettent d'effectuer les contrôles de type. Cependant, l'utilisation de plages mémoire prédéfinies pour contenir des données pose deux problèmes :

- Il faut prévoir avant l'exécution du programme un espace suffisant pour chaque type de données.
- Si l'utilisation de la mémoire n'est pas homogène dans le temps, l'espace à prévoir initialement peut être beaucoup plus grand que l'espace réellement utilisé à chaque instant.

Pour éviter ces problèmes, nous avons décidé de découper la mémoire utilisée par le système en "pages" de tailles identiques dont chacune ne contient, à un instant donné, que des objets de même type (Figure 3). L'information de type est donnée par la page. Si un objet occupe plus d'espace que la taille d'une page, nous allouons plusieurs pages contiguës.



c,f,i,k : pages libres
a,b,d,j,j : pages de taille unitaires affectees chacune à un type spécifique
e,g : pages destinées à recevoir chacune un objet de dimension supérieure à la taille d'une page unitaire.

FIG. 3 – *Organisation de la mémoire*

Avec un système de récupération incluant un compactage, cette organisation nous permet d'une part d'assurer qu'un type donné peut utiliser autant de mémoire que disponible sans qu'il soit nécessaire de lui préallouer une zone particulière. Et d'autre part, si la consommation de mémoire évolue au cours du temps, celle-ci s'adapte automatiquement.

4.2. Ajout dynamique de types

Il y a deux façons d'ajouter dynamiquement un nouveau type de données au système : nouveau type natif, ou nouveau typeC. Premièrement, toute nouvelle bibliothèque peut définir des structures de données, de nouveaux types de pages ainsi que les fonctions associées pour intégrer le nouveau type au même niveaux que les types natifs prédéfinis. Cette possibilité est adaptée à des bibliothèques développées spécifiquement pour Ravi et pour des types qui sont utilisés en grande quantité par le système (justifiant l'utilisation d'au moins une page par type); par exemple un type *variable logique* ou *objet-Scheme*.

Pour des bibliothèques existantes et dont éventuellement seul le binaire est disponible, nous offrons le type polyvalent *typeC* permettant d'encapsuler tous types de données externes. L'interface de programmation de Ravi permet la déclaration dynamique de nouveaux objets typeC en spécifiant son nom et des fonctions spécifiques utilisées pour la lecture, l'impression ou la destruction de l'objet.

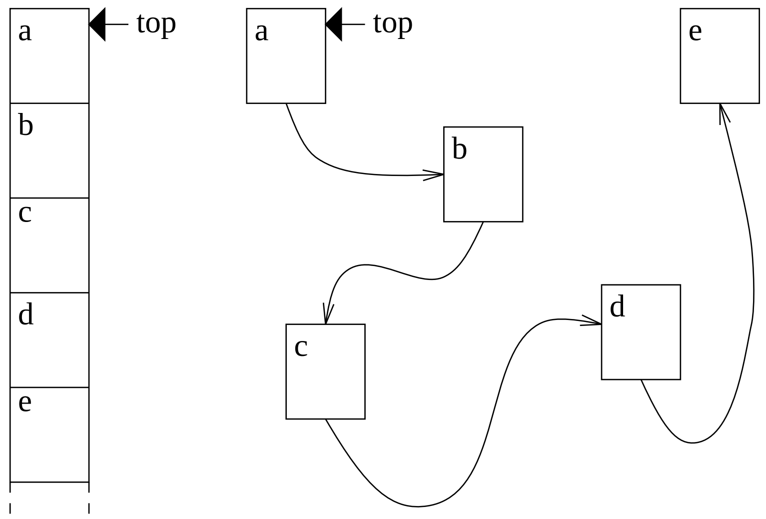


FIG. 4 – *Représentation de la pile*

4.3. Représentation de la pile

La pile pose également classiquement le problème de préallocation, elle doit être de taille suffisante pour le fonctionnement d'un programme sans occuper inutilement trop d'espace. Dans Ravi, nous utilisons une pile découpée en tranches dont chacune occupe une page (Figure 4). Ces pages de pile sont chaînées entre elles afin que la pile puisse croître de façon illimitée. Lorsque la pile décroit, les pages sont libérées, avec un retard - la dernière page libre reste disponible pour la pile afin de ne pas entrer dans des cycles d'allocation/libération de pages lorsque le sommet de pile est proche d'une frontière de page : entre deux allocations de page pour la pile, une page complète aura été remplie.

4.4. Ramasse-Miettes

Pour la gestion dynamique de la mémoire on peut distinguer trois grandes approches :

- Gestion ad-hoc, avec allocation/desallocation explicites, dispersées dans le programme, sous la responsabilité du programmeur.
- gestion semi-automatique avec compteurs de reférences.
- Gestion automatique avec ramasse-miettes.

Pour ce dernier, il existe de nombreuses techniques, (cf.[Wil92a]): marquage-balayage, arrêt-copie, générations, incrémental, parallèle ...

Partant du constat qu'il n'y a pas de solution universelle, nous cherchons un gestionnaire maximalement flexible utilisant la meilleure technique pour chaque structure, est ouvert à des extensions dynamiques.

Le ramasse-miettes de Ravi est générique en ce sens qu'il propose une structure de base, instanciée pour chaque type de données. Ce "squelette" est composé d'une file d'actions d'initialisation, d'une procédure "collecte", d'une procédure "visite", et d'une file d'actions de terminaison. Ce squelette permet d'implémenter efficacement le "stop and copy", la recopie-balayage, et d'autres techniques.

Pour la plupart des types natifs en Scheme on utilise le "stop and copy" [Che70]. Les données sont recopiées à partir des racines du système (la pile et quelques tables) dans un nouvel espace ; puis les anciennes pages sont libérées et deviennent disponibles. L'algorithme se décompose en phases de *collecte* des données et de *visite* des données recopiées pour y trouver d'autres pointeurs à suivre.

Cette technique offre des propriétés intéressantes en termes de compactage des données et de performances. Ce dernier point est surtout vrai si le coût de recopie des données est faible.
L'adaptation du squelette à chaque type de données permet de proposer un système de ramassage multi-stratégies. L'utilisation de méthodes virtuelles associées aux pages de mémoire assure un fonctionnement efficace de ce mécanisme.

- Pour les objets de grande taille (chaines de caractères ou vecteurs de plus d'une page), le ramasse-miettes effectue une *copie virtuelle* des données en marquant simplement des pages entières comme appartenant au nouvel espace.
- Pour les chaînes de caractères, nous avons implémentés un système de générations. Seules les chaînes récemment allouées sont systématiquement recopiées alors que cette recopie est moins fréquente pour les chaînes ayant survécu à plusieurs ramassages. L'information de génération est factorisée au niveau des pages.
  On peut noter que ce système de génération s'applique à des objets qui sont des feuilles dans le graphe des objets du systèmes. En terme d'efficacité, il n'impose aucun surcoût au système puisqu'il n'y a aucun risque de trouver un pointeur d'une génération ancienne vers une plus jeune.
- Les objets typeC utilisent un ramasse-miettes hybride. Après l'exécution d'un "stop and copy" classique, les pages comportant des objets de ce type sont parcourues et le ramasse-miettes appelle une fonction de destruction spécifique à chaque objet pour les

objets non ramassés. Ainsi, le ramasse-miette garantit l'appel des destructeurs des objets C++. De même, nous pouvons ainsi prendre en compte des systèmes de récupération de type comptage de référence en appelant la fonction de déréférence associée à l'objet.

5. Conclusion

Bilan technique - performances. Sur le plan technique, nos choix de conception de système s'avèrent solides. Les performances au niveau Scheme, malgré l'utilisation d'une machine virtuelle, sont très satisfaisantes (le code Scheme compilé s'exécute jusqu'à 2 fois plus vite qu'avec l'interpréteur Scm, le code interprété étant plus lent par un facteur 10). En contrepartie, l'existence de la machine virtuelle permet d'introduire un traitement d'interruptions et d'exceptions de haut niveau, très précieux pour l'intégration d'applications.

Modules interfacés. Les capacités de la génération d'interface facilitent l'expérimentation avec des bibliothèques de programmes, quelque soit leur taille. Nous avons notamment un module d'interface avec PVM (Parallel Virtual Machine), et avec Xforms (interface X). Pour le module XForms, on peut noter que l'interface de programmation de la bibliothèque XForms est particulièrement complexe (nombreuses fonctions, types) et pas toujours très bien écrite, ce qui n'empèche pas la génération automatique de l'essentiel de l'interface.

Utilisation du système Ravi. Le premier critère d'évaluation du système Ravi est l'avantage qu'il procure dans les travaux d'application dans notre équipe. Bien entendu, cela est difficile à chiffrer. Pour l'instant, le bilan nous semble très positif; comme on peut en juger aux publications récentes, par exemple [CC98], [MDC98], [MHC98], [Gua98], (voir le serveur de l'équipe pour de plus amples informations: `http://www-prima.imag.fr`) Ravi est effectivement utilisé pour des travaux très différents, qui profitent d'une symbiose symbolique-numérique; il faudra encore un peu de temps pour juger du coté "intégration".

Autres langages Outre scheme, C et C++, utilisés de façon intensive dans notre équipe, d'autres langages sont disponibles. Un interpréteur Prolog à été implémenté en scheme et utilisé dans le cadre d'un projet d'enseignement. Un moteur de règles en chainage avant de type CLIPS a servi pour des expérimentations en vision stéréoscopique [Zop97].

Nous avons préparé le terrain pour l'ajout de ML. Le module essentiel manquant étant l'inférenceur de types. L'avantage essentiel de ML dans notre contexte serait une traduction efficace de programmes de calcul mixte symbolique-numérique directement en C.

Nous travaillons actuellement un lien avec des langages “Temps Réel” dans le cadre d’un projet commun avec ORCCAD.

Références

[CC94] H. I. Christensen and J. L. Crowley, editors. *Experimental Environments for Computer Vision and Image Processing*, volume 11 of *Series in Machine Perception and Artificial Intelligence*. World Scientific, 1994.

[CC98] V. Colin de Verdière and J.L. Crowley. Reconnaissance d’Objets par Apparence Locale. In *11ème Congrès de Reconnaissance des Formes et d’Intelligence Artificielle, RFIA’98*, volume 2, pages 129–136, January 1998. In french.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, november 1970.

[DPS94] H. Davis, P. Parquier, and N. Séniak. Sweet harmony : The talk/c++ connection. In *LISP 94*, pages 121–127, june 1994.

[Gua98] Alvaro Guarda. *Apprentissage génétique de règles de reconnaissance visuelle*. PhD thesis, Institut National Polytechnique de Grenoble, October 1998.

[Ilo94] Ilog. Ilog talk : Technical overview, june 1994.

[Lau93] O. Laumann. Interfacing scheme to the ”real world” –a non-trivial example–, april 1993.

[LB93] O. Laumann and C. Bormann. Elk : The extension language kit, 1993.

[MDC98] J. Martin, V. Devin, and J.L. Crowley. Active hand tracking. In *IEEE Third International Conference on Automatic Face and Gesture Recognition (FG’98)*, Nara, Japan, April 1998.

[MHC98] J. Martin, D. Hall, and J.L. Crowley. Statistical recognition of parameter trajectories for hand gestures and face expressions. In *Workshop on Perception of Human Actions*, Freiburg, Germany, June 6th 1998.

[Wil92a] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In Springer Verlag, editor, *International Workshop on Memory Management*, september 1992.

[Zop97] B. Zoppis. *Outils pour l’Intégration et le Contrôle en Vision et Robotique Mobile*. PhD thesis, Institut National Polytechnique de Grenoble, juin 1997.

A. Example de génération d'interface

A.1. Fichiers source et utilisation finale

Le fichier source essentiel est le fichier de prototypes :

```
class Point{
public: Point (void);
        Point (int, int);
        Point (Point&);
        int GetX (void);
        int GetY (void);
        void SetX (int);
        void SetY (int);
        void Set (Point&);
        void Set (int, int);
        Point* operator-= (Point&);
        char * ObjectInfo (void);
private: int x_;
         int y_;};
```

Exemple d'utilisation en Scheme - c'est l'objectif final. Un peu de sucre syntaxique, avec les macros *new* et *send* évite l'emploi direct des fonctions d'interface aux noms compliqués :

```
(require 'modpoint)
(define p1 (new Point))
(send p1 GetY)
(send p1 SetX 13.2)
```

A.2. Extraits de fichier généré

Deux fichiers sont générés : un fichier C++, et un fichier Scheme annexe. Le premier contient essentiellement des fonctions, et des initialisations assurant l'édition des liens dans l'environnement Scheme.

Voici la fonction d'interface pour la méthode SetX qui est particulièrement simple ; on note des tests de type systématiques qui garantissent un appel correct des méthodes C++ sousjacentes. Le nombre d'arguments étant fixe, il a été vérifié par le mécanisme d'appel de Scheme.

```
void Sc_Point_SetX()
{
  ScVal * FB = & VS_ElemQ(- GetPar()); // FB FrameBase dans pile
  int num = 0;                         // FB[i] = arg no. i
  if(CheckTypeC(FB[0],TypeC_no_1))  // TypeC_no_1 généré ailleurs
    {
      Point * ceci = (Point *)GetCPtr(FB[0]);
      num++;
      if(IsFixNum(FB[1]))
```

```
        {
          int param_1 = GetFixVal(FB[1]);
          ceci -> SetX(param_1);
          SetResult(ScUnspecified);
        }
      else goto bad;
    }
  else goto bad;
  return;
  bad:
Errorf("bad arg [pos %d] for function %s",num,"SetX");
}
```

et la fonction d'interface pour le constructeur de la classe Point qui montre un exemple de surcharge qui produit une série de test imbriqués. On garantit toujours le bon typage des fonctions C++.

```
void Sc_Point_Sconstructor()
{
  ScVal * FB = & VS_ElemQ(- GetPar());
  int nbarg = ScMV::CI;
  if(nbarg == 0)
    SetResult(TypeC(TypeC_no_1,(void *)(new Point())));
  else if(IsFixNum(FB[0]))
    {
      int param_0 = GetFixVal(FB[0]);
      if(nbarg == 2)
        if(IsFixNum(FB[1]))
          {
            int param_1 = GetFixVal(FB[1]);
            SetResult (TypeC(TypeC_no_1,
                    (void *)(new Point(param_0,param_1))));
          }
        else goto bad;
      else goto bad;
    }
  else if(CheckTypeC(FB[0],TypeC_no_1))
    {
      Point * param_0 = (Point *)GetCPtr(FB[0]);
      if(nbarg == 1)
        SetResult
         (TypeC(TypeC_no_1,(void *)(new Point(* param_0))));
      else goto bad;
    }
  else goto bad;
  return;
  bad:
Errorf("bad args for function %s","Point::constructor");}
```