Aspects de la programmation d'applications Win32 avec un langage fonctionnel

Riccardo Pucella[1]
Erik Meijer[2]
Dino Oliva[1]

[1] *Bell Laboratories, Lucent Technologies*
Murray Hill, USA
{riccardo,oliva}@research.bell-labs.com

[2] *Department of Computer Science, University of Utrecht*
Utrecht, The Netherlands
erik@cs.ruu.nl

Résumé

Un langage de programmation qui se veut utile doit être capable d'exprimer des programmes qui profitent des services et des mécanismes de communication supportés par le système d'exploitation. Nous examinons dans cet article le problème de la programmation d'applications Win32 dites "natives" sous le système d'exploitation Windows avec le langage fonctionnel Standard ML. Nous introduisons une infrastructure basée sur le langage d'interfaces IDL et sur une interface aux fonctions étrangères minimaliste pour explorer le support de l'API Win32 et de la technologie COM dans le contexte de Standard ML.

1. Introduction

L'utilité d'un langage de programmation moderne ne se juge pas uniquement à la lumière de la qualité intrinsèque du langage. Comme l'indique Wadler [13], un langage doit de plus posséder un environnement de programmation capable d'exprimer des programmes qui coexistent avec un modèle d'application supporté par le système d'exploitation. Ce modèle d'application peut inclure des services fournis par le système d'exploitation, comme l'interface-usager et la gestion d'événements, et aussi inclure des mécanismes de communication et des "contrats" entre différentes applications sous un même système et par-delà un réseau.

Dans cet article, nous examinons un tel problème en détail. Spécifiquement, nous étudions la programmation d'applications dites *natives Win32* sous un système Windows avec le langage fonctionnel Standard ML (SML) [7]. Nous nous concentrons sur la communication avec le système d'exploitation via l'API Win32, et la communication avec d'autres applications via l'interface COM (*Components Object Model*).

Plusieurs buts guident notre entreprise. Tout d'abord, nous voulons unifier les différentes visions du support du système d'exploitation Windows, en fournissant une infrastructure pour exprimer ces différentes visions. Nous englobons le travail effectué par l'équipe du langage de programmation Haskell [9, 2], c'est-à-dire le *scripting* de composants COM à partir d'un langage fonctionnel, et le complétons en fournissant accès à l'API Win32, nécessaire pour le développement de programmes Windows. L'utilisation du langage SML apporte des avantages notables comparé à l'utilisation de Haskell dans ce contexte. Principalement, le système de modules de SML permet de capturer de façon abstraite la notion d'interface avec le système Win32, tout en permettant des implantations différentes pour différents compilateurs. Notre second but, plus pédagogique, est la présentation informelle de la méthodologie moderne d'utilisation de composants telle qu'implantée dans l'environnement Windows. Cet article donne un survol de notre project, en montrant qu'avec les outils appropriés, il est simple de fournir une interface utilisable vers l'API Win32, et montre comment le reste de l'infrastructure Windows peut suivre. Un prototype du système existe, et est implanté avec le compilateur Standard ML of New Jersey [1].

Nous imposons la contrainte majeure de garder la génération d'applications Win32 entièrement en SML. Nous voulons éviter de devoir programmer des aspects de l'application en C[1], et nous irons même jusqu'à désirer éviter de requérir un compilateur C. Ceci nous permet d'obtenir un système relativement fermé, simplifiant l'utilisation et permettant une interactivité toujours possible avec les environnements SML courants.

Il est utile de remarquer que la plupart des remarques que nous allons effectuer sur la communication avec le système d'exploitation et l'utilisation des mécanismes de composants peuvent être généralisées, notablement aux composants CORBA disponibles sur plusieurs systèmes. Nous n'allons pas nous attarder sur de telles généralisations ici. De même, ce travail se généralise immédiatement à d'autres langages, notablement OCAML [5].

Le plan général de l'article est le suivant. Nous allons tout d'abord décrire la structure générale de l'API Win32 et de la technologie COM, ainsi que les diverses dérivations telles que OLE et ActiveX. Nous décrivons ensuite une interface aux fonctions étrangères minimaliste, qui nous permettra de communiquer avec le système d'exploitation. Ensuite, nous parlerons d'IDL,

1. Il peut être désirable de programmer des parties de l'application en C, mais ce ne devrait jamais être nécessaire.

le langage utilisé pour décrire les interfaces de l'API Win32 et d'objets COM, et de l'outil qui permet de générer du code SML à partir d'une description IDL. Nous montrons comment interfacer l'API Win32 dans ce contexte, et comment utiliser et créer des objets COM. Finalement, nous allons discuter certaines généralisations que nous allons explorer dans le futur.

2. La programmation Win32

L'API Win32 est l'ensemble des fonctions systèmes fournies par les différents systèmes d'exploitation 32 bits de Microsoft[2]. Ces fonctions sont distribuées dans des librairies dynamiques (*dynamic link libraries* ou *dll*) qui forment le coeur des systèmes d'exploitation Windows.

La fonctionalité de l'API peut se décomposer en sept grandes classes:

- gestion des fenêtres;
- interface graphique;
- contrôles communs;
- accès au *shell*;
- services du système;
- services de réseau;
- internationalisation;

Il n'est pas utile de fournir l'accès à toutes ces fonctions. Par exemple, la gestion de la mémoire est automatique en SML; la gestion des entrées/sorties est fournie par la *Basis Library*, ainsi que les services de réseaux et certains services du *shell*. Nous nous concentrons ici sur les fonctions de gestion des fenêtres, de l'interface graphique et certaines fonctions de service. Les librairies qui nous intéressent sont principalement `user32.dll`, `gdi32.dll` et `kernel32.dll`.

L'API Win32 est intrinsèquement lié à une structure particulière de programmes, que nous revisons ici. Un programme Win32 se compose d'un point d'entrée appelé `WinMain`. Le rôle de cette fonction est d'initialiser l'application, typiquement en enregistrant différentes classes de fenêtres utilisées. L'enregistrement d'une fenêtre nécessite la définition d'une fonction *callback* qui va traiter les messages addressés à la fenêtre. Cette fonction de traitement de messages est la fonction principale de toute application Win32. Il peut exister plusieurs fonctions de traitement de messages si plusieurs fenêtres sont exposées par l'application — une fonction par fenêtre est la règle.

Plutôt que d'accumuler ici des détails inintéressants, nous référons le lecteur à un ouvrage tel que [8] pour un vision en profondeur de la programmation Win32.

2. En fait, Windows NT et Windows 95 implantent des sous-ensembles un tant soit peu différents de l'API Win32. Ces différences ne nous concernerons pas.

2.1. COM

La technologie moderne des systèmes d'exploitation Windows pour la réutilisation de code et la distribution de librairies converge vers la technologie COM [6, 12]. COM est une interface binaire, et repose sur un minimum de principes simples, permettant une expressivité de concept impressionante.

L'idée de base sous-jacente à COM est la notion d'une *interface* vers un objet. Une interface est une vue d'un composant, et correspond simplement à un tableau de pointeurs de fonctions (méthodes). Étant donné une interface et un objet COM, il est possible de questionner l'objet pour savoir s'il supporte l'interface considérée. Si l'interface est supportée, un pointeur vers l'interface est retourné, et à travers ce pointeur il est possible d'invoquer les méthodes implantées par l'interface.

L'identification d'objets COM et d'interfaces s'effectue à l'aide d'identificateurs uniques (GUID): un identificateur de classe (CLSID) est utilisé pour parler d'un objet COM en particulier, et un identificateur d'interface (IID) pour parler d'une interface particulière. Il est important de noter que le CLSID d'un objet COM fait partie de sa description formelle. Installer un objet COM sur un système ajoute son CLSID à la liste maintenue dans le *Registry*, et de même pour les interfaces implantées par l'objet. Utiliser un objet COM requiert le CLSID de l'objet en question, ainsi qu'un IID correspondant à l'interface initiale que l'on voudrait obtenir. Par exemple, la fonction Win32 `CoCreateInstance` prend un CLSID et un IID en arguments, crée une instance de l'objet en question et retourne un pointeur vers l'interface correspondante, si elle est supportée par l'objet.

Une interface spéciale, l'interface `IUnknown`, est l'interface de laquelle toutes les interfaces héritent, et doit être supportée par tout objet COM. Cette interface est définie ainsi en IDL, un langage pour spécifier les interfaces (voir Section 4):

```
interface IUnknown {
  HRESULT QueryInterface ([in] const IID& iid,
                          [out,iid_is (iid)] void **ppv);
  unsigned long AddRef ();
  unsigned long Release ();
}
```

La fonction la plus intéressante est la première. Les deux autres fonctions ont trait à la gestion de la mémoire de l'objet. La fonction `QueryInterface` nous permet, donné une interface, d'accéder à une autre interface via son IID. Puisque toutes les interfaces héritent de `IUnknown`, toutes les interfaces fournissent la fonction `QueryInterface`.

Nous avons pratiquement terminé la description de la technologie COM, tout au moins du point de vue de l'utilisateur. Il y a quand même beaucoup de détails à considérer, notablement pour ce qui a trait à la gestion de la mémoire et de la durée de vie des objets COM, ainsi que des conventions à respecter, tels que l'idée qu'effectuer un `QueryInterface` sur une interface quelconque d'un objet donné pour l'interface `IUnknown` doit retourner le *même* pointeur, permettant l'utilisation de l'interface `IUnknown` comme identificateur utile pour comparer l'identité d'objets COM. Le lecteur intéressé par de tels détails se devra de lire un manuel dans la veine de [12].

Les technologies dites avancées de Microsoft sont toutes bâties sur les fondations fournies par les objets COM et la notion d'interface. Ces technologies sont simplement des spécifications d'interfaces que des composants doivent implanter pour interagir avec le système. Nous revisons ici de manière informelle les plus importantes, c'est-à-dire DCOM, Automation, OLE et ActiveX.

2.2. DCOM

DCOM est une extension de COM qui permet aux serveurs d'objets COM de vivre sur une autre machine du réseau. Une communication de type RPC (*Remote Procedure Call*) permet d'interagir avec les objets ainsi crées.

2.3. Automation

La technologie Automation permet de communiquer de manière interprétée et dynamique avec un objet COM. Un objet COM supporte Automation s'il implante l'interface `IDispatch`:

```
interface IDispatch : IUnknown {
  HRESULT GetTypeInfoCount ([out] UINT* pctinfo);
  HRESULT GetTypeInfo ([in] UINT iTInfo,
                       [in] LCID lcid,
                       [out] ITypeInfo **ppTInfo);
  HRESULT GetIDsofNames ([in] const IID& riid,
                         [in,size_is (cNames)] LPOLESTR* rgszNames,
                         [in] UINT cNames,
                         [in] LCID lcid,
                         [out, size_is (cNames)] DISPID* rgDispId);
  HRESULT Invoke ([in] DISPID dispIdMember,
                  [in] const IID& riid,
                  [in] LCID lcid,
                  [in] WORD wFlags,
                  [in,out] DISPPARAMS* pDispParams,
                  [out] VARIANT* pVarResult,
                  [out] EXCEPINFO* pExcepInfo,
                  [out] UINT* puArgErr);
}
```

Ces méthodes permettent d'appeler certaines fonctions en fournissant leur nom sous forme de chaîne de caractères. La méthode `GetIDsofNames` transforme cette chaîne de caractères en DISPID, un entier correspondant à la fonction, si elle existe. Ce DISPID est passé à la méthode `Invoke` avec les paramètres à passer à la fonction.

Les fonctions que `Invoke` peut appeler (la *dispinterface* en langue COM), sont arbitraires et dépendent de l'implantation de `Invoke` pour une interface donnée. Un des aspects le plus intéressant d'Automation concerne l'implantation d'interfaces dites *dual*, c'est-à-dire d'interfaces COM dont les méthodes peuvent également être invoquées de manière dynamique. Une interface *dual* dérive non pas de `IUnknown`, mais de `IDispatch`. L'implantation de `Invoke` pour cette interface est telle que sa dispinterface correspond aux méthodes de l'interface. Ceci permet d'accéder aux méthodes d'une interface de manière directe via les mécanismes de COM ou interprétée via les mécanismes d'Automation.

Une des restrictions imposées par Automation concerne le type des paramètres passés par l'interface de `Invoke`. Les paramètres doivent être de type `VARIANT`, une union des types de bases connus du système. Ceci implique que les fonctions invoquées par `Invoke` doivent vérifier dynamiquement que leurs arguments ont un type correct. Il n'existe pas de moyen automatique pour vérifier que tout est correct lors de l'appel de la fonction. Les librairies de types (*type libraries*) allègent ce fardeau, mais nous ne nous y attarderons pas dans cet article.

Notons enfin que Automation est le mécanisme principal pour la distribution de composants vers des langages tels que Visual Basic.

2.4. OLE

OLE est un ensemble d'interfaces COM standardisées, ayant principalement rapport à des notions d'interfaces-usager. Celles-ci incluent les notions suivantes: documents composés, activation en place, *drag and drop*. OLE définit une interface `IDataObject` qui permet de transférer des informations entre applications, par exemple par le biais du *Clipboard*, ou de *drag and drop*.

Une application qui veut implanter le *drag and drop* doit fournir au système d'exploitation un *Drop Target Object* avec lequel le système va communiquer lorsque l'usager traîne un icône sur l'application. Cet objet est un objet COM qui implante l'interface OLE `IDropTarget`, et qui s'enregistre auprès du système via la fonction `RegisterDragDrop` de l'API Win32. Lorsque un objet est traîné sur l'application, une méthode de `IDropTarget` de l'application est appelée, qui se voit passer un objet `IDataObject` représentant l'information traînée.

2.5. ActiveX

Le terme "ActiveX" est le terme le plus sur-employé de la littérature, après le terme "visuel". *ActiveX* est une extension de la technologie OLE, permettant à des applications OLE d'interagir avec et par-delà l'Internet. D'autre part, *ActiveX Controls* est le nouveau nom de ce qui était auparavant appelé *OLE Controls* , derivés des contrôles *VBX* de Visual Basic. Ils permettent l'implantation de contrôles, qui sont une généralization des fenêtres-enfants fournies par Win32, comme par exemple le contrôle d'édition (*Edit Control*).

Finalement, *ActiveX Scripting* dénote la technologie qui permet à une application d'exécuter du code interprété en appelant un évaluateur externe, tel qu'un interpréteur JavaScript ou Visual Basic.

3. Une interface aux fonctions étrangères

Interagir avec des fonctions fournies par le système d'exploitation nécessite une interface aux fonctions étrangères (*foreign function interface* ou FFI) qui nous permet de communiquer avec le système. Pour avoir une utilité et une aise d'utilisation maximale, la FFI devrait pouvoir communiquer de manière entièrement dynamique avec le sysème, c'est-à-dire sans nécessiter la génération de code C ou de code compilé séparément. De plus, pour nos besoins particuliers, (écrire des callbacks et générer des objets COM) nous avons aussi besoin de pouvoir transformer des fonctions SML en des pointeurs de fonctions qui peuvent être appelés de C.

Nous présentons ici l'interface minimaliste qui est utilisée pour un prototype de notre système, avec l'idée qu'elle est assez simple pour être portée sans trop de problèmes, puisqu'elle ne dépend que très peu sur la représentation internes de données dans le compilateur SML.

```
structure Wffi : sig
  type addr = Word32.word
  type word = Word32.word
  type library

  val alloc : int -> addr
  val free : addr -> unit
  val offset : addr * int -> addr
  val store : addr * word list -> unit
  val read : addr * int -> word list

  val addrToFun : addr -> (word list -> word)
  val funToAddr : (word list -> word) -> addr

  val library : string -> library
  val function : library -> string -> addr
end
```

L'idée de base de l'interface est celle de représenter l'information traversant l'interface sous une forme primitive: une représentation en mots. Nous aurions pu aussi tout ramener sous forme d'octets, mais étant donné que les arguments des fonctions de l'API qui nous concernent ont tous des grandeurs multiples de mots, notre choix nous simplifie la vie. Une fonction étrangère est une fonction de type `word list -> word`, prenant comme argument une liste de mots représentant les arguments de la fonction. Cette représentation accomode le passage d'arguments de grandeur de plus d'un mot (des structs, par exemple).

Le coeur de l'interface est la paire de fonctions `funToAddr` et `addrToFun`, qui respectivement transforment une fonction de type `word list -> word` (une *fermeture*) en pointeur de fonction (une addresse), et transforment un pointeur de fonction en fonction de type `word list -> word`. En général, ce type d'implantation nécessite une méthode de génération de code-objet à l'exécution (*runtime code generation*), que nous empruntons à l'une des FFI existantes de Standard ML of New Jersey [4].

Les fonctions `library` et `function` permettent de charger une librairie dynamique, et d'en extraire des pointeurs de fonction. Par exemple, pour accéder à la fonction `ShowWindow` de la librairie `user32.dll`:

```
val showWindow = let
  val l = Wffi.library ("user32.dll")
  val f = Wffi.function (l,"ShowWindow")
in
  Wffi.addrToFun (f)
end
```

Il y a plusieurs points importants que nous n'abordons pas ici. Tout d'abord, le fait qu'il existe plusieurs conventions d'appel (*calling conventions*) possibles, qui déterminent si l'appelant ou l'appelé est en charge de dépiler les arguments lorsqu'un appel de fonction se termine. Nous ne ferons que noter que (presque) toutes les fonctions de l'API Win32 et COM sont implantés avec la convention dites *Pascal*, qui dit que l'appelé dépile les arguments. Un autre point que nous n'allons pas discuter concerne l'ordre dans lequel les éléments de structures doivent être empilées. Ces détails, aussi importants soient-ils, ne feraient que compliquer inutilement la présentation.

4. Un langage de définition d'interfaces

Notre but premier étant de fournir à l'utilisateur du langage SML une façon d'accéder aux fonctions fournies par l'API Win32 et par divers objets COM, il est utile d'identifier une notation qui nous permette de spécifier l'interface d'un système d'une manière indépendante de la FFI du compilateur SML que nous avons sous la main, et de réaliser un outil qui nous permettrait de convertir une description dans cette notation en une implantation correspondant à la FFI qui nous intéresse. L'utilisation d'une FFI nécessitant généralement du verbiage important et étant susceptible à des erreurs difficiles à retracer, cette méthode a l'avantage de centraliser dans l'outil la génération de code de support pour l'appel de fonctions étrangères, ainsi que toute décision portant par exemple sur la représentation des données (types de base, types construits tels que structures and unions).

La notation que nous adoptons est naturelle, et Microsoft l'utilise déjà pour décrire l'interface aux objets COM. Il s'agit d'IDL (*Interface Definition*

Language), un langage dérivé du modèle IDL de [14] utilisé par l'OSF pour spécifier des services RPC. IDL permet de définir des structures de données semblables à celles de C, et permet de définir des opérations d'une manière similaire à C. Une différence importante concerne le fait que chaque paramètre d'une opération possède un attribut qui décrit si le paramètre est *in* (le paramètre est utilisé pour passer une valeur à l'opération), ou *out* (le paramètre est utilisé pour retourner une valeur de l'opération). L'IDL que l'outil accepte est en fait une version réduite de l'IDL pour COM: la base de DCE, sans les attributs pour la distribution par RPC, et la base de COM et Automation.

Voici un example de description IDL pour une librairie hypothétique:

```
interface Time {

  typedef struct {
     long sec;
     long usec;
  } timeval_t;

  void gettime ([out] timeval_t *cpu,
                [out] timeval_t *gc,
                [out] timeval_t *sys);

  void timeofday ([out] timeval_t *tod);
}
```

Le mode de fonctionnement de notre outil, ML-IDL [10], est représenté à la Fig. 1. Comme nous l'avons dit, c'est un traducteur de description IDL en code SML. L'outil est paramétré par le type de FFI pour laquelle le code doit être généré, et par le mode d'interface désiré — un fichier IDL peut décrire une librairie statique, une librairie dynamique, une interface COM. Le code généré repose sur la capacité d'abstraction du système de modules de SML. Une signature et un module sont générés pour chaque interface. La signature est indépendante de la FFI visée, et représente l'interface abstraite. Le module généré varie selon la FFI visée.

L'outil ML-IDL est de plus paramétré par ce que l'on pourrait appeler un "niveau" d'interface. L'idée de base est de représenter l'interface vers la librairie décrite en IDL de manière abstraite, en utilisant des types abstraits correspondant aux types de base et aux types définis, avec des fonctions créant et lisant ces valeurs abstraites, et des fonctions agissant sur ces valeurs abstraites correspondant aux fonctions de la librairie. Il est possible d'interagir avec la librairie de cette manière, en convertissant "à la main" les valeurs SML à passer à la librairie. Il est aussi possible de générer automatiquement ces conversions, à la manière de H/Direct [2]. Mais l'utilisation d'une interface abstraite permet d'implanter différents niveaux de génération automatique, par exemple en convertissant des valeurs abstraites retournées par la librairie de manière paresseuse. L'outil ML-IDL peut donc être vu comme un généralisation de H/Direct dans ce contexte, en ce qu'il permet de multiples modes de traduction. Une description complète de l'outil

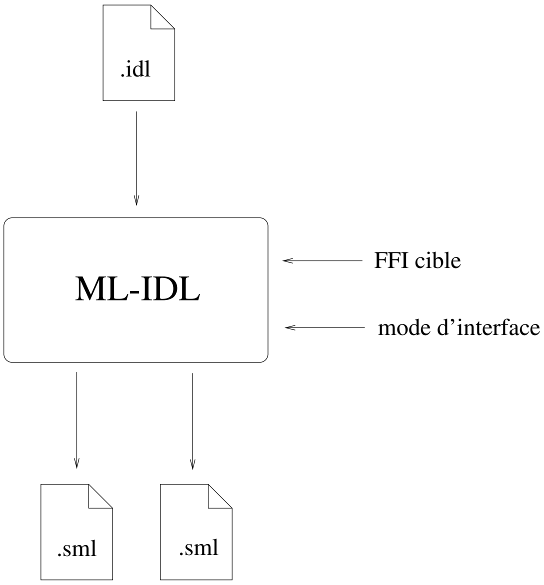

FIG. 1 – *L'outil ML-IDL*

étant impossible, nous référons le lecteur vers [10].

Pour les besoins de cet article, nous considérons une traduction d'IDL dans le style de H/Direct, qui nous donne pour la description IDL donnée plus haut la signature suivante. Noter que les paramètres *out* sont retournés comme résultat d'opérations.

```
structure Time : sig
  datatype timeval_t = timeval_t of {sec : Int32.int,
                                     usec : Int32.int}
  val gettime : unit -> (timeval_t * timeval_t * timeval_t)
  val timeofdat : unit -> timeval_t
end
```

5. Support pour l'API Win32

L'API Win32 étant fixe, créer une implantation de l'API devient simplement une question de produire sa description IDL. Avec cette description, nous pouvons utiliser l'outil ML-IDL pour générer une implantation de l'API compatible avec la FFI du système utilisé.

L'appendice A présente un extrait de la description IDL de l'API Win32 que nous utilisons. L'appendice B présente la signature générée par ML-IDL correspondant à l'extrait.

Nous ne montrons pas le module généré correspondant à l'implantation de la signature par la FFI de la section 3 pour des raisons d'espace. Plutôt qu'aller dans les détails, notons quelques aspects de la programmation Win32 en SML par le biais d'un exemple. L'appendice C présente un programme Win32 écrit en SML via l'interface de l'API Win32. Il s'agit du logo de SML/NJ rebondissant à l'intérieur d'une fenêtre. Le code, une simple traduction du programme C correspondant, montrent certaines caractéristiques de la programmation Win32 en SML.

Comme nous l'avons indiqué à la Section 2, la structure d'un programme inclut la définition de fonctions en charge de traiter les messages addressés aux différentes fenêtres de l'application. Nous appelons ces fonctions des wndprocs. De telles wndprocs sont utilisées lors de l'enregistrement de la classe d'une fenêtre. Une wndproc est un des éléments de la struct passée comme argument à la fonction `W32.User.RegisterClassExA`. L'outil ML-IDL génère les abstractions nécessaires pour nous permettre de passer une fonction de type

```
W32.hwnd * int * Word32.word * Word32.word -> int
```

comme wndproc à la fonction `RegisterClassExA`. Étant donné que la struct passée comme argument à la fonction `RegisterClassExA` de l'API s'attend à recevoir un pointeur vers une fonction comme élément correspondant à la wndproc, notre code SML généré se doit de convertir la fonction SML en pointeur de fonction. Ceci est achevé dans notre FFI en utilisant la fonction `Wffi.funToAddr`.

Une conséquence de l'utilisation d'une wndproc comme structure centrale d'un programme Win32 est particulièrement ennuyante. La wndproc est appelée une fois pour chaque message reçu de la fenêtre correspondante. Ce qui veut dire que si des données doivent être préservées d'un message à l'autre, par exemple des données sur les dimensions de la fenêtre, une cellule `ref` doit être allouée pour l'information. Ce qui conduit à du code extrêmement impératif, avec beaucoup de variables globales. L'argument n'est pas si grave pour Standard ML qui possède un support respectable pour la programmation impérative, mais rend l'expression de programmes Win32 natifs en Haskell (ou tout langage fonctionnel pur) problématique.

Il est possible de contourner ce problème en utilisant Concurrent ML [11]. Nous pouvons exprimer notre wndproc de la façon suivante:

```
local
  val ch = channel (W32.nullHwnd,
                    0,
                    0w0 : Word32.word,
                    0w0 : Word32.word)
  val retCh = channel (0)
in
  fun WndProc (hwnd, msg, wparam, lparam) =
         (send (ch,(hwnd,msg,wparam,lparam));
          recv (retCh))
```

```
    fun WndProc' () = let
      val loop (state) = let
        val (hwnd,msg,wparam,lparam) = recv (ch)
      in
        (* body *)
      end
    in
      loop (initial_state)
    end
  end
```

Le corps de la boucle interne de `WndProc'` doit communiquer la valeur de retour par le canal `retCh`, et s'appeler récursivement avec un nouvel état (contenant, par exemple, les dimensions de la fenêtre courante). Le code d'initialisation de l'application doit appeler `spawn (WndProc')` pour créer la *thread* qui va écouter sur le canal `ch`.

Il ne reste qu'une question: pourquoi nous attardons-nous à vouloir programmer à ce niveau si bas? Après tout, la plupart des programmeurs d'applications Win32 ne programment pas directement avec l'API Win32, mais plutôt avec une librairie de classes C++ telle que MFC ou OWL. En fait, la question se pose de même en C++: pourquoi devrait-on connaître l'API Win32 s'il existe des librairies comme MFC et OWL? La réponse dans les deux cas est similaire: parce ce qu'aucun vernis de haut-niveau ne peut traiter tous les cas possibles. Si quelque chose de spécial doit être effectué, il est souvent nécessaire de revenir au niveau le plus bas, c'est-à-dire l'API Win32. Il est donc impératif de pouvoir fonctionner complètement à ce niveau. Il est aussi intéressant de fournir une interface à un niveau tel qu'il est possible d'utiliser directement des textes pédagogiques existant. Il est en effet facile de traduire des programmes tirés de [8] et de les implanter en SML avec l'interface dérivée plus haut. Par exemple, l'exemple de l'appendice C est tiré directement du chapitre 7 de [8].

6. Support pour COM

Il a deux problèmes relativement distincts reliés à la programmation COM: utiliser des objets COM lors de la programmation d'applications Win32, et l'implantation d'objets COM utilisable par d'autres applications. Nous considérons ces problèmes séparément.

6.1. Utiliser des objets COM en SML

Le problème le plus simple à résoudre est celui d'utiliser des objets COM dans un programme SML. Étant donné un objet COM que nous voulons utiliser avec sa description IDL, nous pouvons utiliser l'outil ML-IDL pour générer du code qui va pouvoir créer des instances de cet objet.

Nous avons tout d'abord besoin d'une petite librairie COM fournissant l'infrastructure nécessaire. Les fonctions de cette librairie se retrouvent principalement dans la librairie dynamique `ole32.dll`, mais nous ne voulons pas utiliser la méthode de la section précédente pour implanter cette interface, puisque nous voudrions un système de type plus sophistiqué pour traiter les interfaces COM. Voici la signature d'une partie de la librairie COM dont nous avons besoin.

```
structure Com : sig

  type CLSID
  type 'a IID
  type 'a interface

  type IUnknown
  val IUnknown : IUnknown IID

  val CoCreateInstance : CLSID -> 'a IID -> 'a interface

end
```

La fonction `CoCreateInstance` prend comme argument le CLSID de l'objet que nous voulons créer, et l'interface que nous désirons obtenir. Nous utilisons des types-témoins (comme dans [9]) pour guarantir le fait que `CoCreateInstance` crée une interface (de type `interface`) correspondant à l'`IID` demandé. Chaque interface générée par le compilateur IDL se doit d'inclure un type-témoin pour l'interface.

Par exemple, considérons un objet COM `Bar` qui possède une interface `IX` et `IY` (toutes deux dérivées de `IUnknown`) avec une fonction `FooX` dans `IX` et une fonction `FooY` dans `IY`. La signature générée par l'outil IDL a la forme suivante.

```
structure Bar : sig

  val BarCLSID : Com.CLSID

  structure IX : sig
    type IX
    val IX : IX Com.iid

    val QueryInterface : IX Com.interface -> 'a Com.IID -> 'a Com.interface
    val FooX : IX Com.interface -> unit -> unit
  end

  structure IY : sig
    type IY
    val IY : IY Com.iid

    val QueryInterface : IY Com.interface -> 'a Com.IID -> 'a Com.interface
    val FooY : IY Com.interface -> unit -> unit
  end

end
```

Notons que les fonctions `AddRef` et `Release` de l'interface `IUnknown` ne sont pas incluses dans le code généré, puisqu'elles ne sont jamais utilisées par l'utilisateur dans l'implantation SML.

Dans notre prototype utilisant la FFI décrite à la Section 3, une interface est simplement un pointeur vers un bloc de mémoire contenant les pointeurs de fonctions. Extraire une méthode d'une interface correspond donc simplement à extraire le pointeur correspondant et de convertir le pointeur de fonction en une fermeture via `Wffi.addrToFun`.

```
type 'a interface = Wffi.addr

fun getMethod (interface,index) = let
  val i' = Wffi.offset (interface, index)
  val p = hd (Wffi.read (i',1))
in
  addrToFun (p)
end
```

6.2. Création d'objets COM en SML

À priori, c'est le problème le plus ardu. Mais la difficulté est principalement due à l'architecture du compilateur que nous utilisons. Le problème peut se subdiviser encore une fois en deux problèmes distincts. Tout d'abord, nous devons pouvoir créer des objets COM en SML. Ensuite, nous devons pouvoir distribuer ces objets COM pour que d'autres applications puissent les utiliser (programmation d'un serveur).

Créer des objets COM est fondamentalement simple, mais demande de la discipline. Comme on se le rappelle, un objet COM est simplement un ensemble d'interfaces, toutes derivées de l'interface `IUnknown`. Créer un objet COM correspond à determiner un ensemble d'interfaces à supporter, implanter les méthodes, et créer les interfaces elles-mêmes, en créant un tableau en mémoire et en y plaçant les pointeurs de fonctions correspondants. Comment construire une interface? Supposons que nous voulons implanter l'objet COM `Bar` décrit à la sous-section précédente. Voici un extrait de code utilisant notre FFI de la section 3.

```
fun fooX [] = print "executing FooX"

fun fooY [] = print "executing FooY"

fun makeInterface (l) = let
  val l' = [queryInterface,addRef,release]@l
  val i = Wffi.alloc (length (l'))
  val _ = Wffi.store (i, map Wffi.funToAddr l')
  val p = Wffi.alloc (1)
  val _ = Wffi.store (p,[i])
in
  p
end

val IUnknown = makeInterface []

val IX = makeInterface [fooX]

val IY = makeInterface [fooY]
```

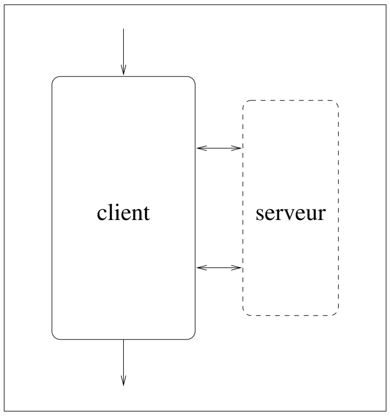

FIG. 2 – *Serveur* in-proc

La fonction `makeInterface` prend une liste de fonction SML à insérer dans l'interface, et construit le bloc mémoire, incluant les fonctions faisant partie de l'interface `IUnknown`. La fonction `queryInterface`, non montrée, est en charge de déterminer quelle interface retourner (`IUnknown`, `IX` ou `IY`), selon l'IID passé.

La création d'un serveur, c'est-a-dire d'un distributeur d'objets COM, est plus compliquée. Tout d'abord, une usine de classes (*class factory*) doit être implantée. Il s'agit là d'un objet COM ayant pour fonction de créer les objets COM implantés par le composant. Il n'y a pas d'interface exigée pour une usine de classes, mais l'interface typique est l'interface `IClassFactory`. Il existe aussi une interface `IClassFactory2` qui supporte des licenses et des authorisations. Un client qui utilise des objets COM fournis par un serveur implantant une usine de classes dérivant de `IClassFactory` peut utiliser la fonction `CoCreateInstance` de l'API Win32 pour créer les objets directement. Si une autre interface est utilisée, le client doit utiliser la fonction `CoGetClassObject` de l'API Win32, qui extrait l'usine de classes du composant et laisse au client le soin d'appeler la fonction de création requise. Il faut noter que `CoCreateInstance` est implanté en fonction de `CoGetClassObject`.

Deux types de serveurs sont possible: *in-proc* et *out-proc*. Un serveur *in-proc* (voir Fig. 2) est une librairie dynamique qui implante le code des objets COM fournis par le composant. Une fonction `DllGetClassObject` doit être exportée par la librairie, et crée l'usine de classes pour le composant (c'est la fonction qui est appelée par `CoGetClassObject`). D'autres fonctions, telles que `DllRegisterServer`, `DllUnregisterServer` et `DllCanUnloadNow` doivent être implantées pour l'enregistrement et la

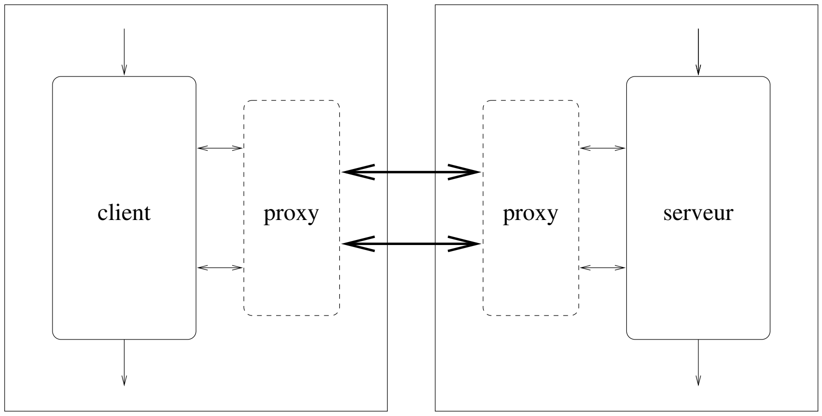

FIG. 3 – *Serveur* out-proc

libération de la librairie dynamique. Le problème de cette approche dans notre cas est que créer des librairies dynamiques avec Standard ML of New Jersey n'est pas réellement supporté. Un nouveau *runtime system* est en développement qui permettra la création de librairies dynamiques et donc de serveurs in-proc.

L'autre type de serveur, un serveur *out-proc*, est un serveur implanté par un exécutable (un *exe*). Si une librairie dynamique fournissant un composant se doit d'exporter une fonction `DllGetClassObject` qui retourne l'usine de classes du composant, un serveur *exe* doit manuellement enregistrer ses usines de classes en appelant la fonction `CoRegisterClassObject` de l'API Win32. Rien de bien compliqué jusqu'à présent. Le problème majeur, c'est que le système nécessite une librairie dynamique qui joue le rôle d'un *proxy* (voir Fig. 3). Le proxy communique avec une instance de lui-même par le biais d'un protocole LPC (*Local Procedure Call*). Construire cette librairie dynamique est long et offre d'innombrables possibilités d'erreur.

Le compilateur MIDL de Microsoft, qui prend comme source le fichier IDL correspondant au composant fourni par le serveur, génère du code C correspondant au proxy. Il est possible d'utiliser ce compilateur ainsi qu'un compilateur C pour créer le proxy. Il est regrettable par ailleurs que cela nous force à abandonner notre modèle de garder la création de programmes Win32 complètement centrée sur des programmes SML. Nous étudions présentement des méthodes alternatives pour achever le même effet.

Nous notons en terminant qu'un serveur *out-proc* peut être utilisé naturellement comme un serveur sur réseau en utilisant le mécanisme DCOM. Ceci ne nécessite aucune modification au code SML du serveur.

7. Conclusion

Nous décrivons dans cet article une infrastructure pour la programmation Win32 sous le langage fonctionnel Standard ML. Un prototype existe pour le compilateur Standard ML of New Jersey, et sert comme base expérimentale pour explorer différentes idées.

Où tout cela nous conduit-il? La majorité des infrastructures pour la programmation Win32 sous des langages tels que C et C++ introduisent des librairies de haut-niveau qui réduisent la complexité de la programmation, telles que les librairies de classes C++ MFC et OWL. Pour Standard ML, une direction intéressante consiste en l'implantation d'une librairie dans le style de eXene [3], une interface graphique pour le système X, qui utilise Concurrent ML [11] pour traiter le parallélisme implicite des interfaces graphiques. Un programme à long terme serait de généraliser ces différentes librairies et de fournir une infrastructure de programmation graphique uniforme sur différents systèmes, tout en préservant les caractéristiques fondamentales des différents systèmes (le *look-and-feel*). Bien sûr, ceci n'addresse que le problème de la programmation graphique, et laisse ouvert le problème d'utiliser des composants de type COM (ou CORBA) d'une manière généralisée.

Une remarque finale: notre infrastructure est basé sur le compilateur Standard ML of New Jersey, mais n'est certainement pas restreinte à ce système. En ajoutant à l'outil ML-IDL un *backend* qui génère du code pour la FFI de MLWorks ou de Ocaml [5], nous pouvons fournir la même interface et les mêmes capabilités à ces différents systèmes. Par ailleurs, dû à la nature des FFI de ces systèmes, nous devrons probablement aussi générer du code C pour pouvoir implanter l'équivalent de notre fonction `Wffi.funToAddr`.

Références

[1] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324. Springer-Verlag, September 1987.

[2] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the International Conference on Functional Programming*. ACM Press, 1998.

[3] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[4] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical memorandum, AT&T Bell Laboratories, November 1995.

[5] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system, 1996. Software and documentation available at http://pauillac.inria.fr/ocaml/.

[6] Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification, 1995. Available from http://www.microsoft.com/com.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[8] C. Petzold. *Programming Windows 95*. Microsoft Press, 1996.

[9] S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components from Haskell. In *Proceedings to the 5th International Conference on Software Reuse*. IEEE Computer Society Press, 1998.

[10] R. Pucella and J. Reppy. An IDL mapping for Standard ML, 1998. In preparation. For details, see the author's home page http://www.cs.bell-labs.com/∼riccardo/.

[11] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.

[12] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[13] P. Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 33(8):23–27, 1998.

[14] X/Open Company Ltd. X/Open Preliminary Specification X/Open DCE: Remote Procedure Call, 1993.

A. Extrait de la description IDL de l'API Win32

```
//
// IDL description of win32 API
//

sml_name ("W32");

typedef int INT;
typedef int HANDLE;
typedef int HRESULT;

typedef HANDLE HWND;
typedef HANDLE HDC;
typedef boolean BOOL;
typedef [string] char *STRING;
typedef [string] wchar_t *WSTRING;

typedef int *WNDPROC ([in] HWND hwnd, [in] INT msg,
                      [in] INT wParam, [in] INT lParam);

typedef struct _WNDCLASSEX {    // wc
    UINT    cbSize;
    int    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
```

```
    int      cbWndExtra;
    HANDLE   hInstance;
    HANDLE    hIcon;
    HANDLE hCursor;
    HANDLE   hbrBackground;
    STRING lpszMenuName;
    STRING lpszClassName;
    HANDLE    hIconSm;
} WNDCLASSEX;

typedef struct tagPOINT { // pt
    INT x;
    INT y;
} POINT;

const char *IDI_APPLICATION = "#32512";
const char *IDI_HAND = "#32513";
const char *IDI_QUESTION = "#32514";
const char *IDI_EXCLAMATION = "#32515";

const char *IDC_ARROW = "#32512";
const char *IDC_IBEAM = "#32513";
const char *IDC_WAIT = "#32514";
const char *IDC_CROSS = "#32515";

typedef enum {
 CS_VREDRAW = 1,
 CS_HREDRAW = 2,

 CW_USEDEFAULT = 0wx80000000,

 WS_OVERLAPPED =  0wx00000000,
 WS_POPUP =       0wx80000000,
 WS_CHILD =       0wx40000000,
 WS_MINIMIZE =    0wx20000000,
 WS_VISIBLE =     0wx10000000,

 ...

} OPTS;

typedef enum {
  SW_HIDE = 0,
  SW_SHOWNORMAL = 1,
  SW_NORMAL = 1,
  SW_SHOWMINIMIZED = 2,
  SW_SHOWMAXIMIZED = 3,
  SW_MAXIMIZE = 3,

  ...

} CONSTS;

typedef enum {
  WM_NULL           = 0wx0,
  WM_CREATE         = 0wx1,
  WM_DESTROY        = 0wx2,
  WM_MOVE           = 0wx3,
  WM_SIZE           = 0wx5,
  WM_SETFOCUS       = 0wx7,
  WM_PAINT          = 0wxf,
  WM_TIMER          = 0wx113,
  WM_HSCROLL        = 0wx114,
  WM_VSCROLL        = 0wx115,

  ...
```

```
} WM;

[sml_source ("user32.dll")]
interface User {

  // classes

  int RegisterClassExA ([in,ref] WNDCLASSEX *wndclass);

  BOOL UnregisterClassA ([in] STRING className,
                         [in] HANDLE hInstance);

  // window handling

  HWND CreateWindowExA ([in] INT dwExStyle,
                        [in] STRING classname,
                        [in] STRING windowname,
                        [in] INT style,
                        [in] INT x,
                        [in] INT y,
                        [in] INT width,
                        [in] INT height,
                        [in] HWND parent,
                        [in] HANDLE menu,
                        [in] HANDLE hinstance,
                        [in] LPVOID param);

  BOOL ShowWindow ([in] HWND hwnd,
                   [in] INT cmdshow);

  BOOL UpdateWindow ([in] HWND hwnd);

  // painting

  HDC BeginPaint ([in] HWND hwnd,
                  [out,ref] PAINTSTRUCT *ps);

  BOOL EndPaint ([in] HWND hwnd,
                 [in,ref] PAINTSTRUCT *ps);

  // icons

  HANDLE LoadIconA ([in] HANDLE h,
                    [in] STRING name);

}


[sml_source ("gdi32.dll")]
interface Gdi {

   BOOL LineTo ([in] HDC hdc,
                [in] int nXEnd,
                [in] int nYEnd);

   BOOL PolyLineTo ([in] HDC hdc,
                    [in,size_is (cPoints)] POINT *lppt,
                    [in] INT cPoints);

}
```

B. Extrait de la signature pour l'API Win32

```
(*****************************************************************
 *
 *  This file was automatically generated by ml-idl
 *  (Tue Dec  8 09:41:24 1998)
 *
 *****************************************************************)

signature W32_SIG =
  sig
    (*
     * Pervasives
     *)
    type 'a pointer
    val null : 'a pointer
    val free : 'a pointer -> unit


    val IDI_APPLICATION : String.string
    val IDI_HAND : String.string
    val IDI_QUESTION : String.string
    val IDI_EXCLAMATION : String.string

    val IDC_ARROW : String.string
    val IDC_IBEAM : String.string
    val IDC_WAIT : String.string
    val IDC_CROSS : String.string

    type INT = Int32.int
    type HANDLE = Int32.int
    type HRESULT = Int32.int

    type HWND = HANDLE
    type HDC = HANDLE
    type BOOL = Bool.bool
    type STRING = String.string
    type WSTRING = String.string

    type WNDPROC = ((HWND * INT * INT * INT) -> Int32.int)

    datatype WNDCLASSEX = WNDCLASSEX of {cbSize:UINT,
                                         style:Int32.int,
                                         lpfnWndProc:WNDPROC,
                                         cbClsExtra:Int32.int,
                                         cbWndExtra:Int32.int,
                                         hInstance:HANDLE,
                                         hIcon:HANDLE,
                                         hCursor:HANDLE,
                                         hbrBackground:HANDLE,
                                         lpszMenuName:STRING,
                                         lpszClassName:STRING,
                                         hIconSm:HANDLE}

    datatype POINT = POINT of {x:INT,y:INT}

    datatype OPTS = CS_VREDRAW
                  | CS_HREDRAW
                  | CW_USEDEFAULT
                  | WS_OVERLAPPED
                  | WS_POPUP
                  | WS_CHILD
                  | WS_MINIMIZE
```

```
                 | WS_VISIBLE
                 | ...
   structure OPTS : sig
     val toInt : OPTS -> Int32.int
     val fromInt : Int32.int -> OPTS option
   end

   datatype CONSTS = SW_HIDE
                   | SW_SHOWNORMAL
                   | SW_NORMAL
                   | SW_SHOWMINIMIZED
                   | SW_SHOWMAXIMIZED
                   | SW_MAXIMIZE
                   | ...
   structure CONSTS : sig
     val toInt : CONSTS -> Int32.int
     val fromInt : Int32.int -> CONSTS option
   end

   datatype WM = WM_NULL
               | WM_CREATE
               | WM_DESTROY
               | WM_MOVE
               | WM_SIZE
               | WM_SETFOCUS
               | WM_PAINT
               | WM_TIMER
               | WM_HSCROLL
               | WM_VSCROLL
               | ...
   structure WM : sig
     val toInt : WM -> Int32.int
     val fromInt : Int32.int -> WM option
   end

   structure User : sig
     val RegisterClassExA : WNDCLASSEX -> Int32.int
     val UnregisterClassA : (STRING * HANDLE) -> BOOL
     val CreateWindowExA : (INT * STRING * STRING * INT *
                            INT * INT * INT * INT * HWND *
                            HANDLE * HANDLE * LPVOID) -> HWND
     val ShowWindow : (HWND * INT) -> BOOL
     val UpdateWindow : HWND -> BOOL
     val BeginPaint : HWND -> (PAINTSTRUCT * HDC)
     val EndPaint : (HWND * PAINTSTRUCT) -> BOOL
     val LoadIconA : (HANDLE * STRING) -> HANDLE
   end

   structure Gdi : sig
     val LineTo : (HDC * Int32.int * Int32.int) -> BOOL
     val PolyLineTo : (HDC * POINT list * INT) -> BOOL
   end

 end
```

C. Exemple de programme Win32

```
structure BounceWin = struct

  val zero = 0 : Int32.int
```

```
val ballTimer = 2 : Int32.int

val moveRate = 10 : Int32.int
val timerRate = 20 : Int32.int

val hInstance = ref (zero)

val cxClient = ref (zero)
val cyClient = ref (zero)
val xCenter = ref (zero)
val yCenter = ref (zero)
val cxTotal = ref (zero)
val cyTotal = ref (zero)
val cxRadius = ref (zero)
val cyRadius = ref (zero)
val cxMove = ref (zero)
val cyMove = ref (zero)
val hBitmap = ref (zero)

fun destroy (hwnd) =
  (W32.User.KillTimer (hwnd,ballTimer);
   if (!hBitmap <> 0)
     then ignore (W32.Gdi.DeleteObject (!hBitmap))
   else ();
     W32.User.PostQuitMessage (0);
   zero)

fun size (hwnd, xsize, ysize) =
  (cxClient := xsize;
   cyClient := ysize;
   xCenter := xsize div 2;
   yCenter := ysize div 2;
   cxMove := moveRate;
   cyMove := moveRate;
   cxTotal := 158;  (* from PAINT *)
   cyTotal := 131;
   cxRadius := 118 div 2;
   cyRadius := 90 div 2;
   if (!hBitmap <> 0)
     then ignore (W32.Gdi.DeleteObject (!hBitmap))
   else ();
   hBitmap := W32.User.LoadImageA
                  (0,"smlnj.bmp",
                   W32.CONSTS.toInt
                   (W32.IMAGE_BITMAP),
                   0,0,W32U.or [W32.LR_LOADFROMFILE]);
   zero)

fun timerBall (hwnd) =
  if (!hBitmap = 0) then zero
  else let
    val hdc = W32.User.GetDC (hwnd)
    val hdcMem = W32.Gdi.CreateCompatibleDC (hdc)
  in
    W32.Gdi.SelectObject (hdcMem, !hBitmap);
    W32.Gdi.BitBlt (hdc,
                    (!xCenter) - (!cxTotal) div 2,
                    (!yCenter) - (!cyTotal) div 2,
                    !cxTotal, !cyTotal,
                    hdcMem, 0,0,
                    W32.CONSTS.toInt (W32.SRCCOPY));
    W32.User.ReleaseDC (hwnd, hdc);
    W32.Gdi.DeleteDC (hdcMem);
    xCenter := (!xCenter) + (!cxMove);
    yCenter := (!yCenter) + (!cyMove);
    if ((!xCenter) + (!cxRadius) >= (!cxClient)) orelse
```

```
      ((!xCenter) - (!cxRadius) <= 0)
      then cxMove := ~(!cxMove)
    else ();
    if ((!yCenter) + (!cyRadius) >= (!cyClient)) orelse
      ((!yCenter) - (!cyRadius) <= 0)
      then cyMove := ~(!cyMove)
    else ();
      zero
  end

fun timer (hwnd, timerID) =
  if (timerID = ballTimer)
    then timerBall (hwnd)
  else zero


fun create (hwnd) = let
  val hdc = W32.User.GetDC (hwnd)
in
  W32.User.ReleaseDC (hwnd,hdc);
  W32.User.SetTimer (hwnd, ballTimer, timerRate, NONE);
  zero
end

and wndproc (hwnd, Msg, wParam, lParam) =
  case (W32.WM.fromInt (Msg))
    of SOME (W32.WM_CREATE) => create (hwnd)
     | SOME (W32.WM_SIZE) =>
         size (hwnd, W32U.loWord (lParam), W32U.hiWord (lParam))
     | SOME (W32.WM_DESTROY) => destroy (hwnd)
     | SOME (W32.WM_TIMER) => timer (hwnd, wParam)
     | _ => W32.User.DefWindowProcA (hwnd,Msg,wParam,lParam)

fun winmain (hinstance,_) = let
  val _ = hInstance := hinstance
  val szAppName = "BouncingSMLNJ"
  val hicon = W32.User.LoadIconA (0, W32.IDI_APPLICATION)
  val hcursor = W32.User.LoadCursorA (0, W32.IDC_ARROW)
  val hbrush = W32.Gdi.GetStockObject (W32.CONSTS.toInt (W32.WHITE_BRUSH))
  val wndclassex = W32.WNDCLASSEX
                    {cbSize = 48,
                     style = W32U.or [W32.CS_HREDRAW,
                                      W32.CS_VREDRAW],
                     lpfnWndProc = wndproc,
                     cbClsExtra = 0,
                     cbWndExtra = 0,
                     hInstance = hinstance,
                     hIcon = hicon,
                     hCursor = hcursor,
                     hbrBackground = hbrush,
                     lpszMenuName = "",
                     lpszClassName = szAppName,
                     hIconSm = hicon}
  val _ = W32.User.RegisterClassExA (wndclassex)
  val hwnd = W32.User.CreateWindowExA
                      (0,
                       szAppName,
                       "Bouncing SML/NJ",
                       W32U.or [W32.WS_OVERLAPPEDWINDOW],
                       W32U.or [W32.CW_USEDEFAULT],
                       W32U.or [W32.CW_USEDEFAULT],
                       500,
                       300,
                       0,
                       0,
                       hinstance,
```

```
                          W32.null)
    fun f () = (W32.User.ShowWindow (hwnd, 1);
                W32.User.UpdateWindow (hwnd);
                (* the following call is required because creating
                 * the window from the interactive loop seems to
                 * not put it on top
                 *)
                W32.User.SetForegroundWindow (hwnd);
                W32U.simpleMsgLoop ())
  in
    W32U.checkHwnd (hwnd,f);
    W32.User.UnregisterClassA (szAppName, hinstance);
    ()
  end

  fun doit () = RunW32.doit (winmain)

end
```