KOPUL – Kind Of Pack / Unpack Language

Sébastien Deveza

{EPITECH.}

06/03/2011

Kopul est un langage de représentation de structure binaire et de flux. Ce document vise à expliquer la conception de son back-end.

Sommaire

I. Introduction

Kopul est un langage de représentation de structure binaire et de flux. Interprété, il permet de gérer dynamiquement les parties segmentation, conversion, manipulation d'un flux de données binaire ou de structure statique. L'utilisateur disposera d'une API C permettant d'accéder de manière indexe ou par clef'*' (hashtable) au diffèrent éléments et structure du flux. Il permet de décrire de manière unique une représentation de données binaire et ce indépendamment de la machine. Enfin on pourra générer une fois pour tout un moteur de décodage en C/C++ pour l'utiliser dans tout type d'application.

Pour la conception de KOPUL nous sommes partis de la constatation qu'il pouvait être défini comme n'importe quel langage de programmation : il encode et décode des **variables typées,** en fonction de certaines **conditions.** Dans la suite de ce document nous allons donc voir la conception des 4 grands axes du back-end de KOPUL.

Toutefois avant de commercer a parler du code en lui-même, nous detaillerons l'architecture que nous avons mis en place pour le projet ainsi que deux trois normes que nous nous sommes imposées.

Pour détailler l'implémentation nous commencerons dans un premier temps par le « compilateur » en lui-même qui est représenté par la class Register**.** Dans cette partie nous nous attarderons sur comment cette class empêche les conflits entre les différents types enregistrés en son sein par un « pseudo mangling ».

Dans un deuxième temps nous verrons les différents types gérés par KOPUL et comment sont générés les fonctions d'encodage et de décodage qui leurs sont associées**.** Nous détaillerons la solution que nous avons choisis pour pouvoir « inliner » le code pour ne pas être oblige pour les types composés de faire plein d'appel de fonction (optimisation en terme de rapidité d'exécution, perte de lisibilité du code intermédiaire).

Dans un troisième temps nous parlerons des solutions que nous avons choisir pour gérer les variables. Elles ont deux utilisations différentes dans KOPUL. La première sert dans la représentation du langage sous la forme d'un AST (par exemple : *[#8 - >c ($c)]*). La deuxième sert au moment de l'exécution des fonctions d'encodage et de décodage (pour stocker ou récupérer des informations dans le flux).

Nous terminerons par les solutions que nous avons choisies pour la gestion des conditions. Nous les avons séparés en trois types différents : les conditions de type var1==var2 représentés par la class Condition, les conditions de type Condtion1&&Condition2 représentés par la class ConditionNode et enfin les conditions de type ?== INT #+32 représentés par la class SwitchCondition.

II. Architecture

KPLC++

```
|   libkopul.a
|   Makefile
|
+---includes
|       Bitfield.h
|       ComposedType.h
|       Condition.h
|       ConditionNode.h
|       ConstantValue.h
|       Container.h
|       DynamicArray.h
|       FunctionList.h
|       ICondition.h
|       IObject.h
|       Register.h
|       StaticArray.h
|       StaticStruct.h
|       StaticType.h
|       Struct.h
|       Switch.h
|       SwitchCondition.h
|       Type.h
|       Value.h
|       Variable.h
|       VariableIterator.h
|
\---srcs
        Bitfield.cpp
        Condition.cpp
        ConditionNode.cpp
        ConstantValue.cpp
        Container.cpp
        DynamicArray.cpp
        FunctionList.cpp
        Register.cpp
        StaticArray.cpp
        StaticStruct.cpp
        StaticType.cpp
        Struct.cpp
        Switch.cpp
        SwitchCondition.cpp
        Type.cpp
        Value.cpp
        Variable.cpp
        VariableIterator.cpp
```

KPLTest

```
|   Makefile
|   test
|
+---includes
|       test.h
|
\---srcs
        main.cpp
        testBitfield.cpp
        testConstantValue.cpp
        testDynamicArray.cpp
        testStaticArray.cpp
        testStaticStruct.cpp
        testSwitch.cpp
        testVariable.cpp
```

1) Environnement

OS : Ubuntu 10.04 LTS

LLVM version: 2.7

IDE : NetBeans IDE 6.8

2) KPLC++

KPLC++ est le dossier dans lequel se trouve le code source du back-end de KOPUL. Il est constitué de deux sous répertoires :

- Le répertoire srcs qui contient le fichier .cpp
- Le répertoire includes qui contient les fichiers .h

Le Makefile génère une lib (libkopul.a).

3) KPLCTests

KPLTests est le dossier dans lequel se trouvent tous les tests unitaires du back-end de KOPUL. Il est lui aussi constitue d'un répertoire srcs et d'un répertoire includes.

Chaque fichier source comprend les tests unitaires d'un type de KOPUL. Le programme de test est compilé avec la lib construite dans KPLC++. Puis il lance les tests les uns à la suite des autres et a la première erreur ou comportement non satisfaisant s'arrête.

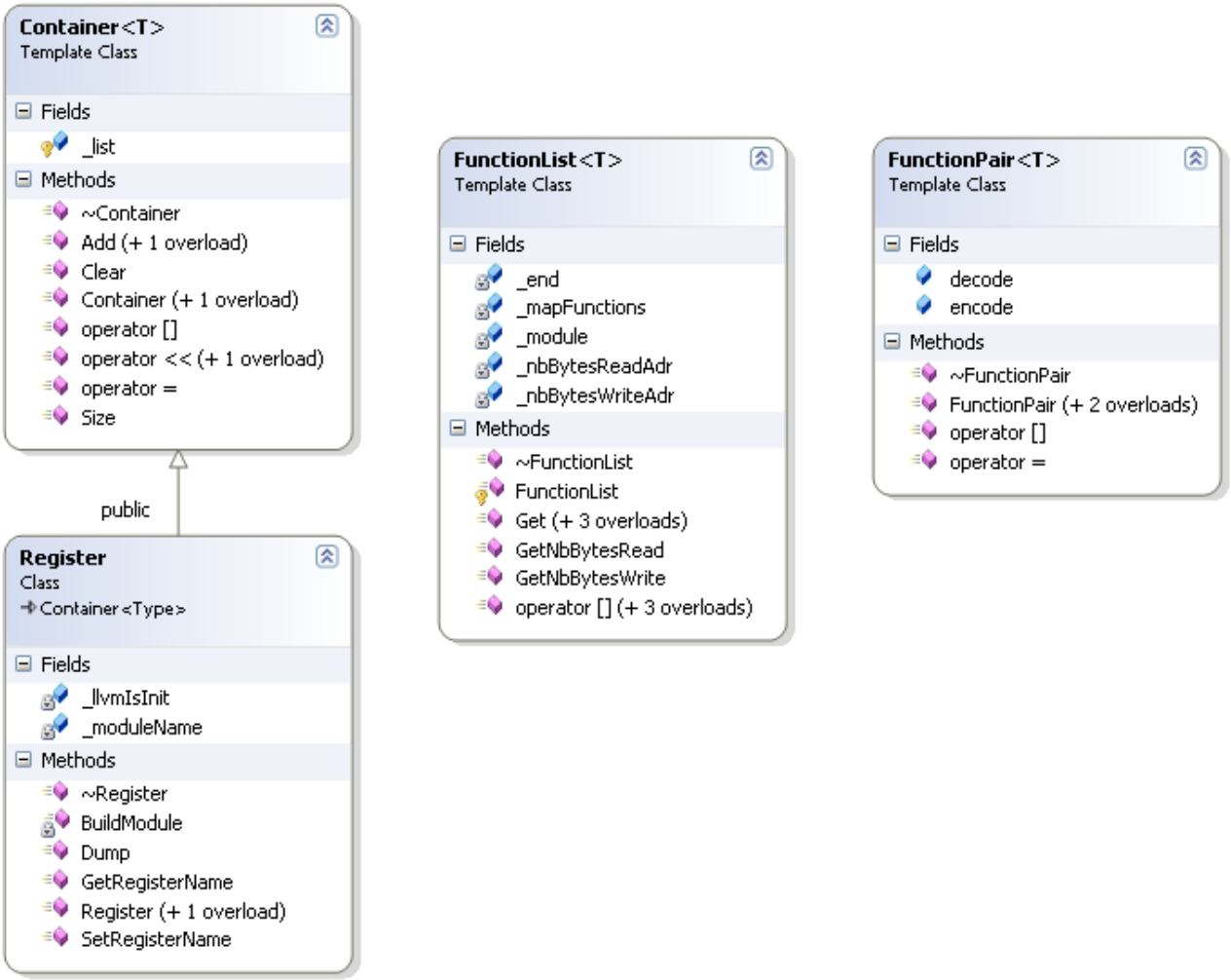
Les tests unitaires sont écrits avant l'implémentation du type à tester. Ils visent ainsi à définir a l'avance le comportement attendu.

Ces tests servent aussi de système anti régression. A chaque ajout, amélioration ou modification tous les tests sont effectues de nouveau. Ainsi une nouvelle version n'est mis a disposition sur le SVN qu'une fois que son intégrité a été complètement testé.

4) Norme

- L'indentation sera la même que celle de la norme de JAVA.
- Les noms des class et des méthodes commencent par une majuscule
- Les noms des attributs commencent par un '_' et sont suivis d'une minuscule
- Les parties contenant du code llvm devront être commentés.

III. Le Compilateur



1) Class Register

La class Register est en quelque sort une surcouche de LLVM. C'est cette class qui va enregistrer les types pour lequel vont être généré les fonctions d'encodage et de décodage.

```
Register::Register()
{
    _moduleName = "Kopul";
    if (!_llvmIsInit)
    {
      llvm::InitializeNativeTarget();
      _llvmIsInit = true;
    }
}
```

Dans le constructeur de Register on regarde si LLVM a déjà été initialisé. Si ce n'est pas le cas on initialise llvm et met _llvmIsInit à true.

Register permet de générer des fonctions d'encodage et de décodage pour 3 différents types d'utilisations : buffer, socket, et file descriptor.

```
FunctionList<int (*)(stream, ...)> *Register::CompileInMemoryMode() const
{
    llvm::Module    *_module = this->BuildModule(MEMORY_MODE);

    return (new FunctionList<int (*)(stream, ...)>(_module, this->_list));
}

FunctionList<int (*)(fd, ...)>     *Register::CompileInFileMode() const
{
    llvm::Module    *_module = this->BuildModule(FILE_MODE);

    return (new FunctionList<int (*)(fd, ...)>(_module, this->_list));
}

FunctionList<int (*)(socket, ...)> *Register::CompileInSocketMode() const
{
    llvm::Module    *_module = this->BuildModule(SOCKET_MODE);

    return (new FunctionList<int (*)(socket, ...)>(_module, this->_list));
}
```

Pour les trois différents cas d'utilisation on construit le module à partir de la méthode BuildModule :

```
llvm::Module                        *Register::BuildModule(MODE mode) const
{
    llvm::Module        *_module;
    std::ostringstream  oss;

      ...
      ...
    for (unsigned int i = 0; i < this->_list.size(); ++i)
    {
        oss << i;
        this->_list[i]->SetName("_type" + oss.str() + "_" +
                                this->_list[i]->ToString());
        this->_list[i]->Build(_module, mode);
        oss.str("");
    }
    return (_module);
}
```

Pour tous les Types enregistrés ont les renomme pour éviter que deux types ai le même nom. Dans Struct{i8,i8}, les deux i8 ne doivent pas avoir le même nom sinon il risque d'y avoir un conflit. Apres avoir décoré les noms on construit les fonctions d'encodage et de décodage des types en appelant la méthode Build (qui prends comme paramètre le module dans lequel construire les fonctions et le mode d'encodage/décodage).

Une fois le module construit on extrait les fonctions générées dans le module et on les retourne sous la forme de la class FunctionList.

2) FunctionList

```
template <typename T>
FunctionList<T>::FunctionList(llvm::Module *module, const std::vector<Type
*> &listType)
{
    llvm::ExecutionEngine   *exec = llvm::EngineBuilder(module).create();
    llvm::Module::iterator  itb = module->begin();
    llvm::Module::iterator  ite = module->end();
    void                    *pfencode;
    void                    *pfdecode;

    this->_module = module;
    this->_nbBytesWriteAdr = (char *)exec-
>getOrEmitGlobalVariable(static_cast<const llvm::GlobalVariable *>(module-
>getValueSymbolTable().lookup("_nbBytesWrite")));
    this->_nbBytesReadAdr = (char *)exec-
>getOrEmitGlobalVariable(static_cast<const llvm::GlobalVariable *>(module-
>getValueSymbolTable().lookup("_nbBytesRead")));
    for (unsigned int i = 0; itb != ite; ++itb, ++i)
    {
        pfencode = exec->recompileAndRelinkFunction(itb);
        ++itb;
        pfdecode = exec->recompileAndRelinkFunction(itb);
        this->_mapFunctions[static_cast<Type *>(listType[i]->Clone())] =
FunctionPair<T>((T)pfencode, (T)pfdecode);
    }
}
```

On commence par extraire les variables _nbBytesWrite et _nbBytesRead qui sont les variables qui nous servent à savoir combien de bit sont a cheval. Ensuite pour chaque type on extrait sa fonction d'encodage et de décodage qui sont représenté par la class FunctionPair. On obtient ainsi une collection de pair de function.

La class FunctionList est une class Template a cause des trois différents types de flux, a savoir buffer, socket, et file descriptor.

La class FonctionList fournis un certain nombre de méthode afin d'accéder aux fonctions d'encodage et de décodage le plus facilement possible.

```
const FunctionPair<T>&  operator [] (const Type *) const;
const FunctionPair<T>&  operator [] (const Type &) const;
const FunctionPair<T>&  operator [] (const std::string &) const;
const FunctionPair<T>&  operator [] (unsigned int i) const;

const FunctionPair<T>&  Get(const Type *) const;
const FunctionPair<T>&  Get(const Type &) const;
const FunctionPair<T>&  Get(const std::string &) const;
const FunctionPair<T>&  Get(unsigned int i) const;
```

On peut donc accéder a ces fonctions de trois manières différentes :

- Par le Type
- Par le nom du Type
- Par un index

3) FunctionPair

```
enum FUNCTIONTYPE
{
    ENCODE,
    DECODE,
};

template <typename T>
class FunctionPair
{
public:
    FunctionPair();
    FunctionPair(const T&, const T&);
    FunctionPair(const FunctionPair<T>&);
    ~FunctionPair();
    FunctionPair<T>&    operator = (const FunctionPair<T>&);
    T                   operator [] (FUNCTIONTYPE) const;

    T           encode;
    T           decode;
private:

};
```

Comme expliqué précédemment FunctionPair est un class Template qui représente les fonctions d'encodage et de décodage.

Elle comprend donc deux attributs qui sont des pointeurs sur ces fonctions :

- `T  encode;`
- `T  decode;`

T étant un pointeur sur fonction de type :

- `int (*)(stream, ...)`
- `int (*)(fd, ...)`
- `int (*)(socket, ...)`

En plus de ces deux attributs cette class fournit une surcharge de l'operateur « [] » pour accéder aux fonctions d'encodage et de décodage :

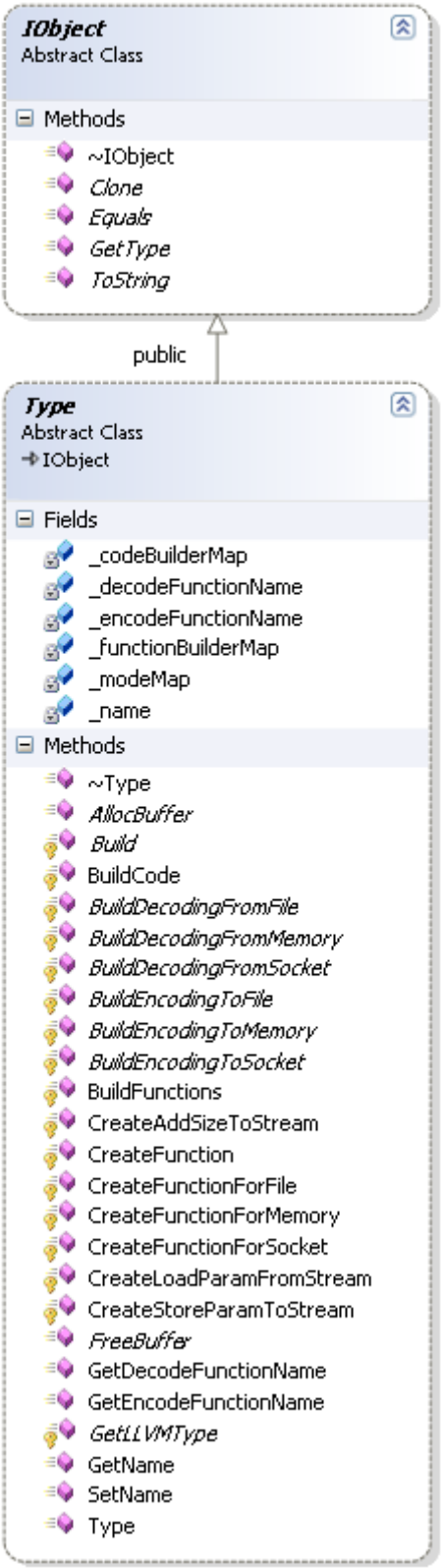
```
T                   operator [] (FUNCTIONTYPE) const;
```

Ainsi on peut facilement accéder aux fonctions d'encodage et de décodage a partir de la class FunctionList généré par la class Register :

```
listFunction[bit5][ENCODE]((char **) & test, &c1);
listFunction[bit6].encode((char **) & test, &c2);

listFunction[bit5][DECODE]((char **) & test, &c1);
listFunction[bit6].decode((char **) & test, &c2);
```

IV. Les Types

1) Class Type



Type est une class abstraite dont vont hériter tous les types de KOPUL. Nous allons dans la suite de ce document détailler les méthodes qu'offrent cette class.

a) IObject

On s'est rapidement rendu compte que les types de KOPUL ainsi que la plupart des autres objets devaient avoir des méthodes de base. C'est pourquoi on s'est inspire de JAVA pour créer l'interface IObject. Les méthodes suivantes doivent être implémentées dans les class qui en héritent.

```
virtual const std::string&  ToString() const = 0;
```

ToString retourne la représentation de l'objet sous la forme d'une chaine de caractère. On a vu que cette methode servait au moment de la decoration dans la class Register.

```
virtual const std::string&  GetType() const = 0;
```

GetType retourne une chaine de caractère qui nous renseigne sur le Type de l'objet. Par exemple Register manipule une collection de Type* il est donc intéressant d'avoir une méthode qui nous dise exactement de quel type il s'agit.

```
virtual bool                Equals(const IObject &value) const = 0;
```

Equals retourne prend en paramètre un IObject value et retourne vrai ou faux si this et value sont identiques. Nous entendons par identique que les deux objets soient « physiquement » identiques, les noms n'ont pas d'importances. Cette méthode sert notamment dans la class FunctionList pour retourner les fonctions d'encodage et de décodage associe a un Type.

```
virtual IObject*            Clone() const = 0;
```

Dans le back-end de KOPUL nous avons choisis de travailler avec des clones pas comme LLVM qui travaille avec des pointeurs. Nous avons choisis cette méthode car elle permet a son utilisateur de réutiliser les objets même après les avoir enregistre dans un ComposedType ou dans la class Register. Cela permet par exemple de faire ce qui suit :

```
bit.SetSize(5);
register << bit;
bit.SetSize(18);
register << bit;
```

Ceci est rendu possible car dans Register ce sont des clones qui sont sauvegardées.

b) Génération du code

```
virtual bool      Build(llvm::Module *, MODE) const = 0;
```

Comme on l'a vu dans la class Register au moment de la génération du code c'est cette fonction qui est appelée. Elle permet au type qui l'implémente de construire la liste des paramètres dont ses fonctions auront besoin. Puis cette dernière appelle la fonction BuildFunctions avec la liste des paramètres précédemment créé.

```
bool              BuildFunctions(llvm::Module *, const
std::map<std::string, const llvm::Type*>&, MODE) const;
```

Cette méthode va construire toute la fonction d'encodage et de décodage. Pour cela elle commence par créer les deux fonctions dans le module en appelant la méthode CreateFunction puis elle construit le corps de ces fonctions en appelant la méthode BuildCode. Elle termine par vérifier l'intégrité des fonctions ainsi créés et retourne le résultat sous la forme d'un booléen.

```
llvm::Function*   CreateFunction(llvm::Module *, const std::string &name,
const std::map<std::string, const llvm::Type*>&, MODE) const;
```

CreateFunction va appeler la bonne méthode en fonction du MODE. Si le mode est MEMORY_MODE on appelle la méthode CreateFunctionForMemory, si c est FILE_MODE on appelle CreateFunctionForFile et enfin si c est SOCKET_MODE on appelle CreateFunctionForSocket. Afin d'éviter de faire des « if » imbrique nous utilisons des pointeurs sur méthode:

Dans le constructeur de Type:

```
_functionBuilderMap[MEMORY_MODE] = &Type::CreateFunctionForMemory;
_functionBuilderMap[FILE_MODE] = &Type::CreateFunctionForFile;
_functionBuilderMap[SOCKET_MODE] = &Type::CreateFunctionForSocket;
```

Dans CreateFunction:

```
if (this->_functionBuilderMap.find(mode)==this->_functionBuilderMap.end())
        throw (std::logic_error("Unknown Mode"));
return((this->*this->_functionBuilderMap.find(mode)->second)(module, name, mapVariable));
```

```
llvm::Function*   CreateFunctionForMemory(llvm::Module *, const std::string
&name, const std::map<std::string, const llvm::Type*>&) const;

llvm::Function*   CreateFunctionForFile(llvm::Module *, const std::string
&name, const std::map<std::string, const llvm::Type*>&) const;

llvm::Function*   CreateFunctionForSocket(llvm::Module *, const std::string
&name, const std::map<std::string, const llvm::Type*>&) const;
```

Ces trois méthodes font à peu de chose près la même chose. A savoir elles construisent le squelette de la fonction à générer. La seul chose qui va les différencié c'est la définition du flux.

Ces méthodes vont donc commencer par ajouter un paramètre a ceux fournis par le Type, à savoir le flux. Ensuite elles vont créer trois blocks : entry, action, block

Entry

C'est le block d'entré de la fonction. On va y charger le flux ainsi que _nbBytesWrite et _nbBytesRead pour que les Types n'aient pas à le faire au moment ou ils construiront le corps de la fonction. Puis on crée un saut vers le block action.

Action

C'est dans ce block que les Types construiront leurs codes d'encodage et de décodage.

Error

En cas d'erreur rencontré lors de la construction du code les Types feront un saut vers ce block. Le contexte de départ y est restitué. Pour cela on restaure les paramètres qui ont été charges dans le block Entry.

```
llvm::BasicBlock* BuildCode(llvm::BasicBlock *actionBlock, llvm::Value
*streamAdr, llvm::Value *nbBytesAdr, llvm::Value *paramAdr, BUILDER_MODE)
const;
```

BuildCode appelle la bonne méthode en fonction du BUILDER_MODE. Tout comme pour CreateFunction pour éviter les « if » imbriqué nous utilisons des pointeurs sur méthode.

Tout d'abord un MODE est associe a deux BUILDER_MODE :

```
_modeMap[MEMORY_MODE] = pair(ENCODE_TO_MEMORY, DECODE_FROM_MEMORY);
_modeMap[FILE_MODE] = pair(ENCODE_TO_FILE, DECODE_FROM_FILE);
_modeMap[SOCKET_MODE] = pair(ENCODE_TO_SOCKET, DECODE_FROM_SOCKET);
```

Puis chaque BUILDER_MODE est associe a un pointeur sur méthode :

```
_codeBuilderMap[ENCODE_TO_MEMORY] = &Type::BuildEncodingToMemory;
_codeBuilderMap[DECODE_FROM_MEMORY] = &Type::BuildDecodingFromMemory;
_codeBuilderMap[ENCODE_TO_FILE] = &Type::BuildEncodingToFile;
_codeBuilderMap[DECODE_FROM_FILE] = &Type::BuildDecodingFromFile;
_codeBuilderMap[ENCODE_TO_SOCKET] = &Type::BuildEncodingToSocket;
_codeBuilderMap[DECODE_FROM_SOCKET] = &Type::BuildDecodingFromSocket;
```

```
virtual llvm::BasicBlock*     BuildEncodingToMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;

virtual llvm::BasicBlock*     BuildDecodingFromMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;

virtual llvm::BasicBlock*     BuildEncodingToFile(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;

virtual llvm::BasicBlock*     BuildDecodingFromFile(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;

virtual llvm::BasicBlock*     BuildEncodingToSocket(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;

virtual llvm::BasicBlock*     BuildDecodingFromSocket(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const = 0;
```

Ces 6 fonctions au dessus doivent être implémentées par chaque Type de KOPUL. Nous verrons en détail leurs implémentations un peu plus loin.

```
llvm::BasicBlock* CreateAddSizeToStream(llvm::Value *streamAdr, llvm::Value
*nbBytesAdr, int sizeInBytes, llvm::BasicBlock *whereToBuild) const;
```

CreateAddSizeToStream est une fonction utile pour tous les types de KOPUL. Elle permet d'avancer dans le flux en fonction d'un nombre de bit. Son algorithme en lui-même est assez simple :

- Chargement du flux
- Chargement du nombre de bit a cheval
- Cast du flux pour en faire un entier
- Ajout du nombre d'octet contenu dans le nombre de bit à ajouter à l'entier ainsi obtenue
- Ajout du reste de bit au nombre de bit a cheval
- Si le nombre de bit a cheval est supérieur a 8 on rajoute 1 a l'entier représentant le flux et on enlève 8 au nombre de bit a cheval
- Enfin on cast l'entier représentant le flux et on met a jour le flux avec le résultat obtenue

```
llvm::BasicBlock* CreateStoreParamToStream(llvm::Value *streamAdr,
llvm::Value *nbBytesAdr, llvm::Value *paramAdr, int sizeParamInBytes,
llvm::BasicBlock *whereToBuild, const std::string &newBlockName = "")const;
```

CreateStoreParamToStream est la méthode de base pour l'encodage. Son algorithme est rendu complique par la gestion des bits a cheval :

- Chargement du flux (i8** -> i8*)
- Chargement du nombre de bit a cheval
- On test le nombre de bit qui sont a cheval (il peut y en avoir maximum 7) par un Switch
    - 0 Bit
        - Cast du flux en i(`sizeParamInBytes`)*
        - Chargement du paramètre depuis la variable passée a la fonction
        - Ajout du paramètre chargé dan le flux
    - N bit (de 1 a 7)
        - Cast du flux en iN*
        - Chargement des bits a cheval dans une variable temporaire tmp
        - Cast de tmp en i(N + `sizeParamInBytes`)
        - Chargement du paramètre depuis la variable passée a la fonction
        - Cast du paramètre en i(N + `sizeParamInBytes`)
        - Décalage de bit pour pouvoir ajouter les bits a cheval devant le paramètre en le multipliant par 2^N : paramètre = paramètre * 2^N
        - On ajoute les bits à cheval par une simple addition : paramètre = paramètre + bits a cheval.
        - Ajout du résultat ainsi obtenue dans le flux

```
llvm::BasicBlock* CreateLoadParamFromStream(llvm::Value *streamAdr,
llvm::Value *nbBytesAdr, llvm::Value *paramAdr, int sizeParamInBytes,
llvm::BasicBlock *whereToBuild, const std::string &newBlockName = "")const;
```

CreateLoadParamFromStream est la méthode de base pour le décodage. Tout comme pour **CreateStoreParamToStream** son algorithme est rendu complique par la gestion des bits à cheval :

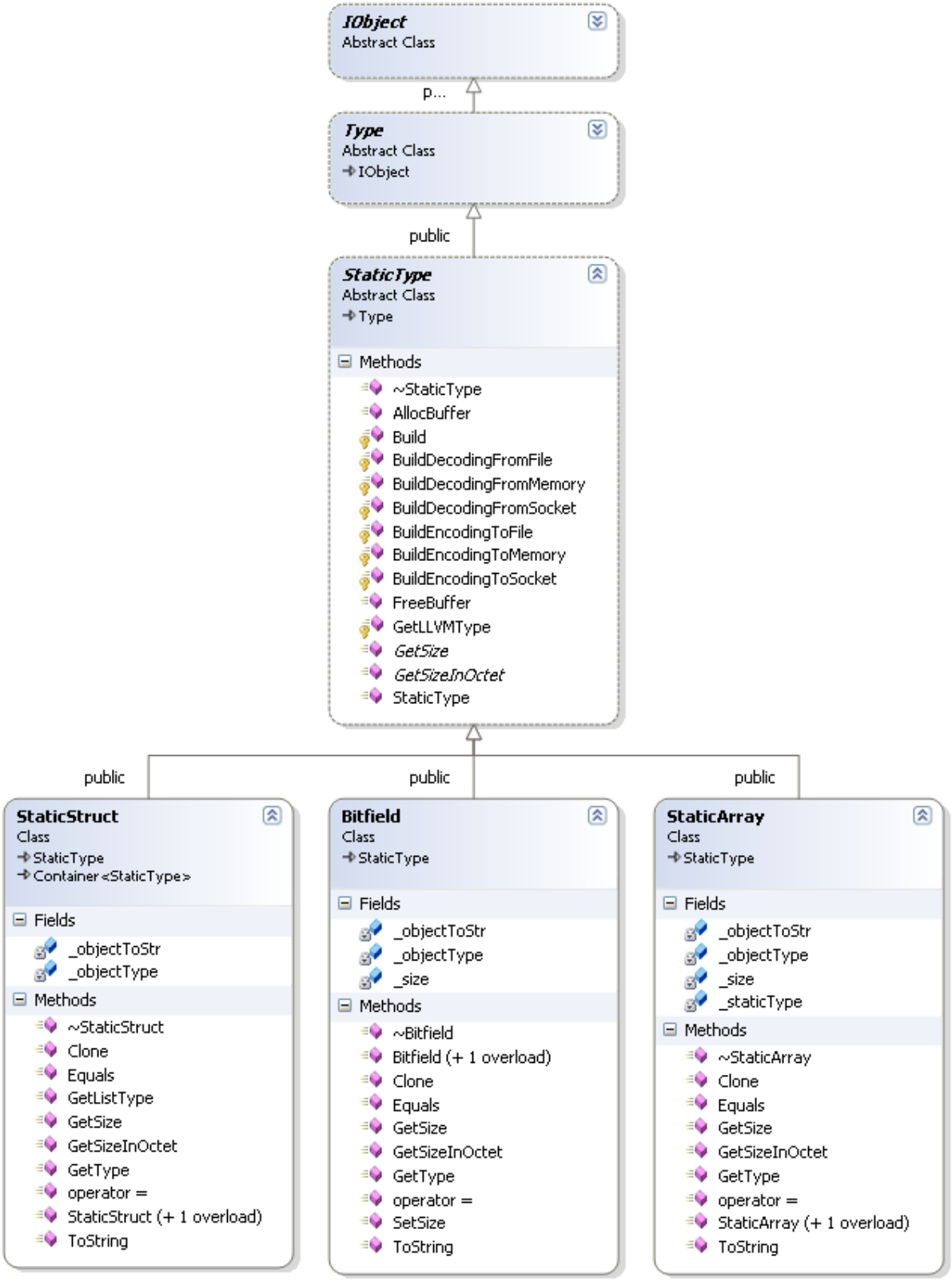
- Chargement du flux (i8** -> i8*)
- Chargement du nombre de bit a cheval
- On test le nombre de bit qui sont a cheval (il peut y en avoir maximum 7) par un Switch
    - 0 Bit
        - Cast du flux en i(`sizeParamInBytes`)*
        - Chargement depuis le flux de `sizeParamInBytes` bits
        - On met ce qui a été chargé dans la variable passée en paramètre.
    - N bit (de 1 a 7)
        - Cast du flux en i(N + `sizeParamInBytes`)*
        - Chargement dans une variable temporaire tmp
        - Decalage de bit afin d'enlever les bits a cheval devant ceux a chargés en divisant tmp par 2^N.
        - Decalage de bit afin d'enlever les bits se trouvant derrières en multipliant tmp par 2^(CONVERT_NBBYTES_TO_NBOCTET(sizeParamInBytes) * 8) - sizeParamInBytes)
        - On décale une dernière fois pour mettre les bits qui nous intéressent au début en divisant pas la même puissance de 2 que ci-dessus.
        - On met ce qui a été ainsi obtenue dans la variable passée en paramètre

c) Gestion des buffers

```
virtual void*      AllocBuffer(void *oldBuffer = NULL) const = 0;
virtual void       FreeBuffer(void *oldBuffer) const = 0;
```

Ces deux fonctions sont la pour fournir des opérations de base sur les buffers. Ces buffers sont la pour stocker les variables qui vont être encodées ou décodées grâce aux fonctions générées. Nous verrons plus en détails l'utilisation de ces méthodes au moment ou nous aborderons les variables.

2) Types Statiques



Les types dit statique sont ceux dont leurs tailles sont finis et non pas dynamique. Pour cette raison leurs fonctions d'encodage et de décodage sont assez simples :

```
llvm::BasicBlock* StaticType::BuildEncodingToMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const
{
    llvm::BasicBlock    *addSizeToStreamBlock =
    this->CreateStoreParamToStream(streamAdr, nbBytesAdr, paramAdr,
                                   this->GetSize(), actionBlock,
                                   std::string("addSizeTo"));

    return (this->CreateAddSizeToStream( streamAdr, nbBytesAdr,
                                         this->GetSize(),
                                         addSizeToStreamBlock));
}
```

On utile la méthode `CreateStoreParamToStream` vu précédemment puis on avance dans le flux grâce a la méthode `CreateAddSizeToStream`.

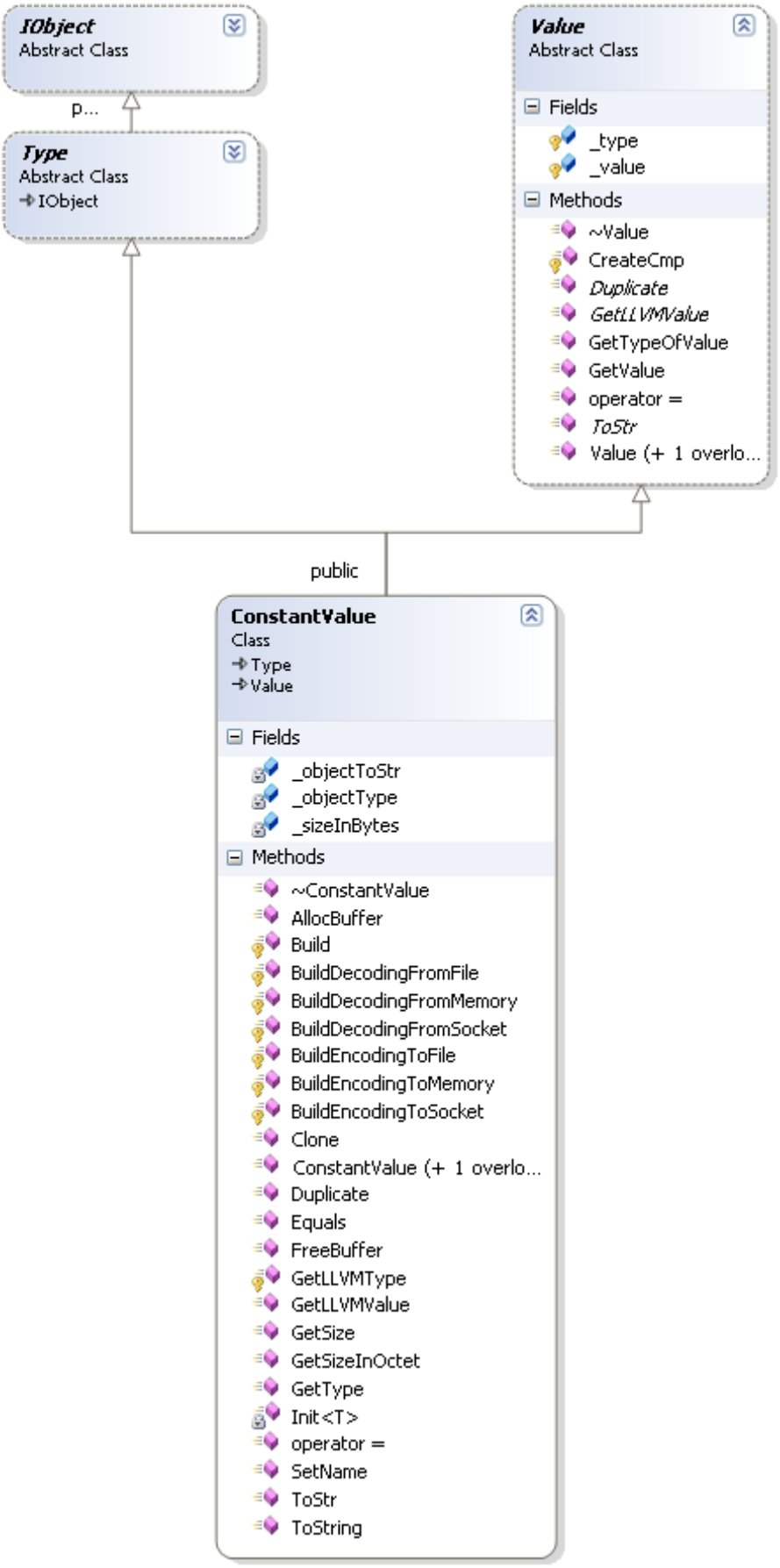
```
llvm::BasicBlock* StaticType::BuildDecodingFromMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const
{
    llvm::BasicBlock    *addSizeToStreamBlock =
    this->CreateLoadParamFromStream(streamAdr, nbBytesAdr, paramAdr,
                                   this->GetSize(), actionBlock,
                                   std::string("addSizeTo"));

    return (this->CreateAddSizeToStream(streamAdr, nbBytesAdr,
                                         this->GetSize(),
                                         addSizeToStreamBlock));
}
```

On utile la méthode `CreateLoadParamFromStream` vu précédemment puis on avance dans le flux grâce a la méthode `CreateAddSizeToStream`.

KOPUL propose trois différents types statiques:

- Bitfield
- StaticArray
- StaticStruct

3) ConstantValue



Les ConstantValue sont les types qui représentent les valeurs de types 42, 4.2, « coucou », etc. Par conséquent leurs fonctions d'encodage et de décodage seront assez similaires à celle des types statiques. A l'exception près de la fonction de décodage qui va verifier que la valeur chargée correspond bien à celle attendue.

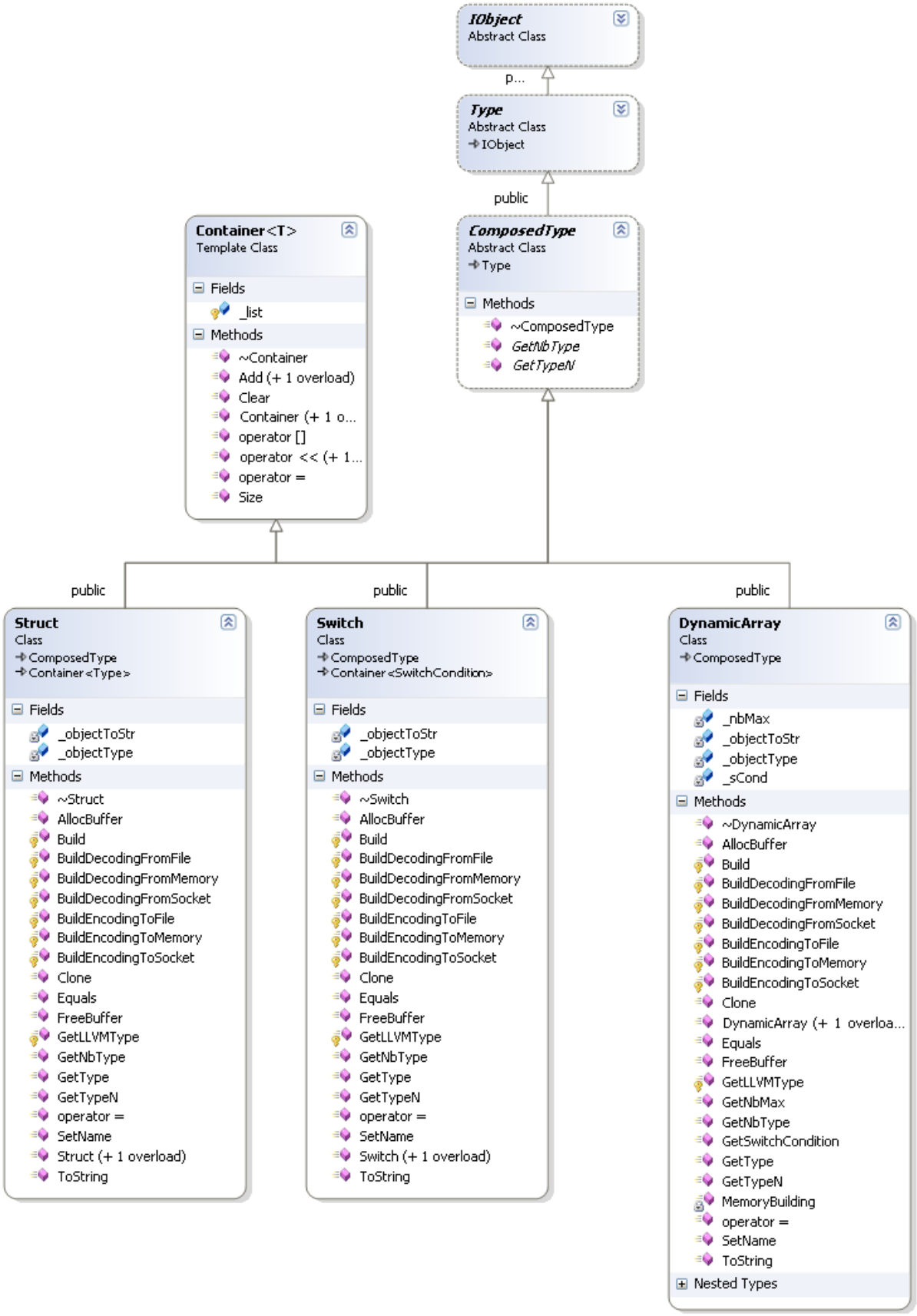
```
llvm::BasicBlock* ConstantValue::BuildEncodingToMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const
{
    llvm::IRBuilder<>   builder(llvm::getGlobalContext());
    builder.SetInsertPoint(actionBlock);
    llvm::Value         *bufferToStore =
builder.CreateAlloca(llvm::IntegerType::get(llvm::getGlobalContext(), this-
>GetSizeInOctet() * 8), llvm::ConstantInt::get(llvm::getGlobalContext(),
llvm::APInt(32, 1, false)), this->_type->GetName());

    builder.CreateStore(this->GetLLVMValue(actionBlock), bufferToStore);
    llvm::BasicBlock    *newActionBlock = this->_type-
>BuildEncodingToMemory(actionBlock, streamAdr, nbBytesAdr, bufferToStore);
    return (newActionBlock);
}
```

- Allocation d'un espace mémoire de la taille de la valeur à mettre dans le flux
- On met dans l'espace alloué la valeur
- On appelle `BuildEncodingToMemory` du type de la ConstantValue avec la variable ainsi créée

```
llvm::BasicBlock* ConstantValue::BuildDecodingFromMemory(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr) const
{
    llvm::IRBuilder<>   builder(llvm::getGlobalContext());

    builder.SetInsertPoint(actionBlock);
    llvm::Value         *bufferToStore =
builder.CreateAlloca(llvm::IntegerType::get(llvm::getGlobalContext(), this-
>GetSizeInOctet() * 8), llvm::ConstantInt::get(llvm::getGlobalContext(),
llvm::APInt(32, 1, false)), this->_type->GetName());
    llvm::BasicBlock    *newActionBlock = this->_type-
>BuildDecodingFromMemory(actionBlock, streamAdr, nbBytesAdr,
bufferToStore);

    builder.SetInsertPoint(newActionBlock);
    llvm::Value         *toCompare = builder.CreateLoad(bufferToStore,
"ToCompare");
    llvm::BasicBlock    *trueBlock =
llvm::BasicBlock::Create(llvm::getGlobalContext(), "action", actionBlock-
>getParent());

    this->CreateCmp(newActionBlock, trueBlock, static_cast<llvm::BasicBlock
*>(actionBlock->getParent()->getValueSymbolTable().lookup("error")),
toCompare);
    return (trueBlock);
}
```

- Allocation d'un espace mémoire de la taille de la valeur à mettre dans le flux
- On appelle `BuildDecodingFromMemory` du type de la ConstantValue avec la variable ainsi créée
- On ajoute une comparaison pour verifier que la valeur lu est bien celle attendu

4) Types Composés



ComposedType représente les types qui ont en leurs seins une collection de Type. KOPUL comprend 3 types compose a savoir : DynamicArray, Switch et Struct (struct est différent de StaticStruct parce qu'il peut avoir des Switch ou des DynamicArray)

a) DynamicArray

DynamicArray représente les expressions KOPUL suivantes :

- pascalString = {#8 - > len [#8 ($len)]}
- cString = {[#8 - >c ($c)]}
- etc

L'implémentation de ses fonctions d'encodage et de décodage est rendu difficile car on peut considérer que le paramètre est lui aussi un flux. De plus la fin de la lecture ou de l'écriture est conditionnée. L'algorithme est de ce fait assez complexe:

```
llvm::BasicBlock*              DynamicArray::MemoryBuilding(llvm::BasicBlock
*actionBlock, llvm::Value *streamAdr, llvm::Value *nbBytesAdr, llvm::Value
*paramAdr, MEMORY_TYPE flag) const
```

- Allocation d'un espace mémoire d'une case du tableau appelé lastBytes
- On renseigne a toutes les variables qui concernent le DynamicArray que les derniers bits lu seront stockes dans cette espace mémoire
- On crée l'index du tableau et on le met à zéro
- On crée le block loop et on va dedans
- On stock dans tmpAdr l'adresse de l'élément i du tableau
- On test le mode pour lequel on veut générer le code
  - ENCODE : on appelle la méthode `BuildEncodingToMemory` du type de l'élément qui compose le tableau avec comme paramètre tmpAdr
  - DECODE : on appelle la méthode `BuildDecodingFromMemory` du type de l'élément qui compose le tableau avec comme paramètre tmpAdr
- On charge le résultat contenue dans tmpAdr et on le met dans lastBytes
- On construit la condition de fin. Si la condition n'est pas atteinte on retourne dans loop. Ca nous permet ainsi de faire une boucle tant que la condition n'est pas remplie.

b) Switch

Switch représente les expressions KOPUL suivantes :

```
#8-> choice (
        ?==1 /* type chaine */ [#8 (?) ]-> theString
        ?==2 /* double */ #.64 - > theReal
        ?==4 /* struct user */ user -> theUser
        ?==8 {
                #128 - > magic
                #2-> flags_foo
                #5-> flags_bar
                #1-> flags_pad
                (? $flags_bar & 2 == 1 [[#8 (10) ]
                (? $flags_foo )])
             }
            )
```

L'implémentation de ses fonctions d'encodage et de décodage devrait être assez simple étant donne qu'il s'agit de construire une liste de SwitchCondition (cf Les conditions)

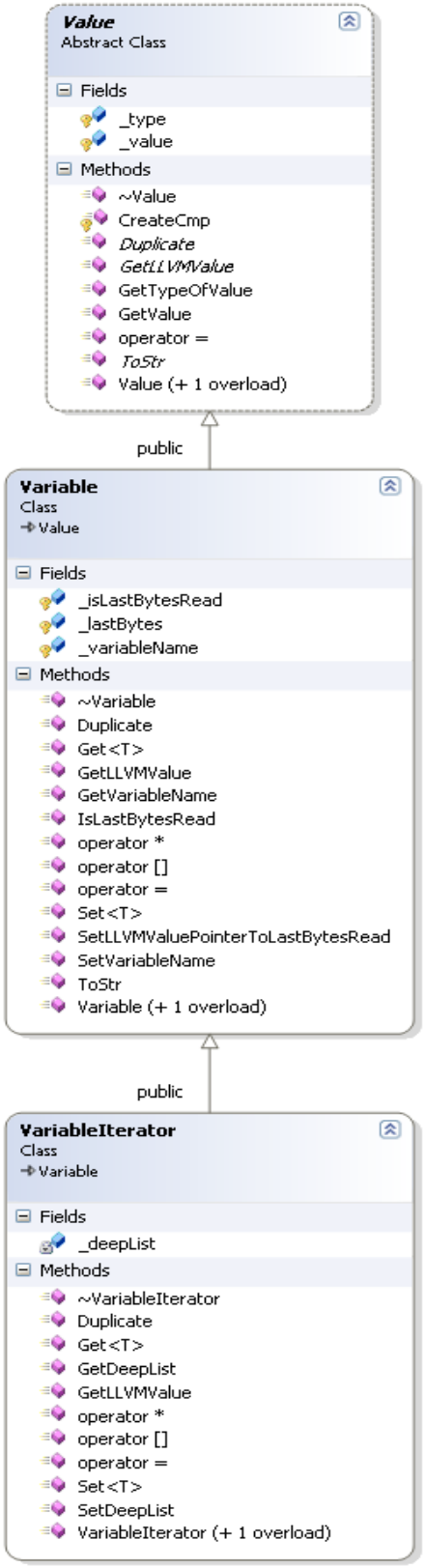
c) Struct

Struct représente les expressions KOPUL suivantes :

```
{
        #128 - > magic
        #2-> flags_foo
        #5-> flags_bar
        #1-> flags_pad
        (? $flags_bar & 2 == 1 [[#8 (10) ]
        (? $flags_foo )])
}
```

Comme pour le Switch ses fonctions d'encodage et de décodage seront assez simple a implémentées. Il s'agira de construire la liste des types qu'elle possède en son sein.

V. Les Variables



1) Utilisation dans l'AST

Une variable dans le back-end de KOPUL peut servir uniquement comme une représentation dans l'AST. On a par exemple précédemment vu dans la partie DynamicArray qu' « on renseigne a toutes les variables qui concernent le DynamicArray que les derniers bits lu seront stockes dans cette espace mémoire ». La class Variable possède un attribut de type booléen qui permet de savoir si la variable représente les derniers bits lus :

```
Bool  _isLastBytesRead;
```

On peut indiquer l'adresse de l'espace mémoire dans lequel seront stockes ces bit grâce a la méthode :

```
void  SetLLVMValuePointerToLastBytesRead(llvm::Value *);
```

2) Utilisation au moment de l'exécution

Une variable dans le back-end de KOPUL peut aussi servir au moment du run-time pour stocker les variables que l'ont va mettre dans le flux ou alors a partir de laquelle on va récupérer les informations présentent dans le flux. Les Variables ont donc besoin d'un buffer. Toutes les variables étant typées, elles peuvent récupérer les dites buffer a partir des méthodes de Type :

```
void*     AllocBuffer(void *oldBuffer = NULL) const;
void      FreeBuffer(void *oldBuffer) const;
```

Si aucun paramètre n'est passé à AllocBuffer alors un espace mémoire de la taille nécessaire pour stocker le Type est retourné, sinon une copie de ce paramètre est retournée. FreeBuffer libère un buffer crée par AllocBuffer.

La spécification de KOPUL renseigne que l'on pourra accéder de manière indexe ou par clef aux différents éléments et structures du flux. Pour cela la class Variable et VariableIterator implémentent toutes les deux la surcharge d'operateur « [] »

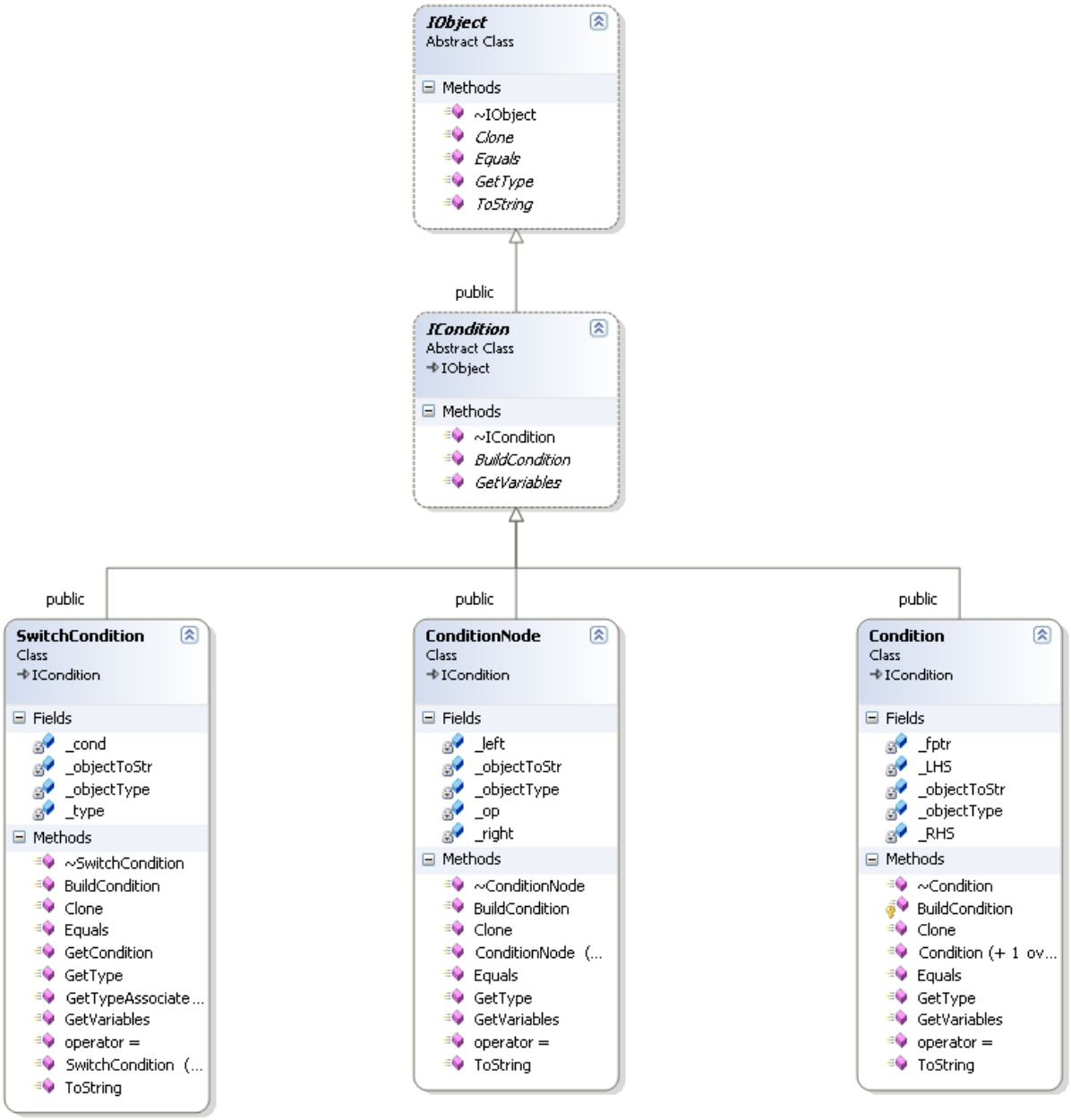
```
VariableIterator    Variable::operator [] (unsigned int i)
VariableIterator    VariableIterator::operator [] (unsigned int i)
```

Dans ces méthodes à chaque fois que l'on accède de façon indexe à une variable on vérifie si la variable en question est bien de type ComposedType et que l'index n'est pas « out of bound ». Le VariableIterator retourné contient une référence sur le buffer de base ainsi que la liste de tous les indexes précédents qui ont permis d'arriver jusqu'à cette variable (_deepList).

On obtient la valeur de la variable en appelant la surcharge de l'operateur « * » :

```
void*                   VariableIterator::operator *()
{
    void                *buffer = this->_value;

    for (unsigned int i = 0; i < this->_deepList.size(); ++i)
        buffer = ((void **)buffer)[this->_deepList[i]];
    return (buffer);
}
```

VI. Les Conditions



Toutes les conditions héritent de l'interface ICondition. Elles doivent donc en implémentés toutes les méthodes à savoir :

```
virtual void BuildCondition(llvm::BasicBlock *actionBlock, llvm::BasicBlock *trueBlock, llvm::BasicBlock *falseBlock) const = 0;
```

Cette méthode est appelée au moment de la génération du code des fonctions d'encodage et de décodage. Elle construit la représentation intermédiaire de la condition. Elle prend en paramètre trois block : actionBlock, trueBlock, et falseBlock. ActionBlock est le block dans lequel la condition va être construite. Si la condition est remplis trueBlock sera joint dans le cas contraire ce sera falseBlock.

```
virtual std::vector<const Variable *>   GetVariables() const = 0;
```

Cette méthode est appelée au moment de la création des fonctions d'encodage et de décodage. Permet de récupérer les variables dont la fonction a besoin.

1) Condition

Comme spécifié dans l'introduction, la class Condition représente les conditions de types var1==var2. Nous en avons recensés 6 différentes : ==, !=, <, >, <=, >=. Pour faciliter l'utilisation du back-end ainsi que son implémentation nous avons choisis d'utiliser la surcharge de ses operateurs pour créer des conditions.

```
Condition       kpl::operator == (const Value & LHS, const Value & RHS)
{
  //  icmp eq
  return (Condition(&llvm::IRBuilder<>::CreateICmpEQ, LHS, RHS, "_EQ_"));
}

Condition       kpl::operator != (const Value &LHS, const Value &RHS)
{
  //  icmp ne
  return (Condition(&llvm::IRBuilder<>::CreateICmpNE, LHS, RHS, "_NE_"));
}

Condition       kpl::operator <= (const Value &LHS, const Value &RHS)
{
  //  icmp sle
  return (Condition(&llvm::IRBuilder<>::CreateICmpSLE, LHS, RHS, "_SLE_"));
}

Condition       kpl::operator >= (const Value &LHS, const Value &RHS)
{
  //  icmp sge
  return (Condition(&llvm::IRBuilder<>::CreateICmpSGE, LHS, RHS, "_SGE_"));
}

Condition       kpl::operator < (const Value &LHS, const Value &RHS)
{
  //  icmp slt
  return (Condition(&llvm::IRBuilder<>::CreateICmpSLT, LHS, RHS, "_SLT_"));
}

Condition       kpl::operator > (const Value &LHS, const Value &RHS)
{
  //  icmp sgt
  return (Condition(&llvm::IRBuilder<>::CreateICmpSGT, LHS, RHS, "_SGT_"));
}
```

Le constructeur de la class Condition prends 4 paramètres :

Le premier est un pointeur sur méthode de la class IRBuilder. Cette dernière est celle qui construit le langage intermédiaire. Donc chaque condition prend le pointeur sur méthode de cette class qui lui est associée. Par exemple la condition == est associée a la méthode CreateICmpEq. Le pointeur sur méthode est stockes dans l'attribut _fptr.

Les deuxièmes et troisièmes paramètres sont les opérandes de cette condition. Ils sont tous les deux de types Value. Value est une base class dont héritent ConstantValue et Variable. Ils sont stockes dans les attributs _LHS et _RHS.

Le quatrième paramètre est une représentation de la condition sous la forme d'une chaine de caractère. Sert pour le ToString de cet objet.

Avec les trois premiers paramètres la construction sous la forme du langage intermédiaire LLVM devient plus facile. La méthode `BuildCondition` implemente depuis l'interface ICondition est appelée.

```
void Condition::BuildCondition(llvm::BasicBlock *actionBlock,
llvm::BasicBlock *trueBlock, llvm::BasicBlock *falseBlock) const
{
    llvm::IRBuilder<>   builder(llvm::getGlobalContext());

    builder.SetInsertPoint(actionBlock);
    llvm::Value* cmp = (builder.*this->_fptr)(this->_LHS-
>GetLLVMValue(actionBlock), this->_RHS->GetLLVMValue(actionBlock), "cmp");
    builder.CreateCondBr(cmp, trueBlock, falseBlock);
}
```

On crée un constructeur de code, on le place dans actionBlock puis on appelle sa méthode de construction de la condition à partir du pointeur de méthode stocké dans _fptr. Elle prend trois paramètres : Les deux opérandes de cette condition et le nom donné au résultat. La méthode GetLLVMValue de la class Value (kpl) retourne la Value (llvm) associe présente dans le langage intermédiaire. On finit enfin par créer un saut régis par une condition. Si la condition est remplie on saute vers trueBlock sinon on saute vers falseBlock.

2) ConditionNode

ConditionNode représente les conditions de types cond1&&cond2 et cond1||cond2. Encore une fois pour faciliter l'utilisation de la lib ainsi que d'en faciliter son implémentation nous avons choisis de surcharger ces operateurs :

```
ConditionNode   kpl::operator ||(const ICondition &left, const ICondition
&right)
{
    return (ConditionNode(left, right, OR));
}

ConditionNode   kpl::operator &&(const ICondition &left, const ICondition
&right)
{
    return (ConditionNode(left, right, AND));
}
```

Le constructeur de la class ConditionNode prend 3 paramètres. Les deux premiers paramètres sont les opérandes de la condition et sont de type ICondition. Le troisième paramètre est un enum pour représenter si la condition est AND ou OR.

La construction des ConditionNodes sont un peu plus complique que les Conditions :

```
void   ConditionNode::BuildCondition(llvm::BasicBlock *actionBlock, llvm::BasicBlock
*trueBlock, llvm::BasicBlock *falseBlock) const
{
    llvm::BasicBlock            *leftConditionBlock =
llvm::BasicBlock::Create(llvm::getGlobalContext(), "leftConditionBlock", actionBlock-
>getParent());
    llvm::BasicBlock            *rightConditionBlock =
llvm::BasicBlock::Create(llvm::getGlobalContext(), "rightConditionBlock", actionBlock-
>getParent());
    llvm::IRBuilder<>           builder(llvm::getGlobalContext());

    builder.SetInsertPoint(actionBlock);
    builder.CreateBr(leftConditionBlock);
    if (this->_op == AND)
    {
        this->_left->BuildCondition(leftConditionBlock, rightConditionBlock, falseBlock);
        this->_right->BuildCondition(rightConditionBlock, trueBlock, falseBlock);
    }
    else
    {
        this->_left->BuildCondition(leftConditionBlock, trueBlock, rightConditionBlock);
        this->_right->BuildCondition(rightConditionBlock, trueBlock, falseBlock);
    }
}
```

On crée deux nouveaux blocks, un pour la condition de gauche et l autre pour la condition de droite. On va commencer par verifier la condition de gauche donc on crée un saut dans actionBlcok vers leftConditionBlock. Ensuite deux cas de figures se présentent suivant si la ConditionNode est AND ou OR.

AND : On construit la Condition de gauche dans le block de gauche. Si la condition de gauche est remplie on doit alors vérifier la condition de droite (jump vers le block de droite), sinon on saute vers falseBlock. Si la condition de droite est remplie on saute vers trueBlock sinon on va vers falseBlock.

OR : Dans le cas du OR on fait l'inverse. Si la condition de gauche est remplie on saute directement vers trueBlock sans verifier la condition de droite. Par contre si la condition n'est pas remplie on va verifier que la condition de droite marche donc on saute vers rightConditionBlock. Si la condition de droite est remplie on saute vers trueBlock et falseblock dans le cas contraire.

3) SwitchCondition

SwitchCondition représente les conditions tel que ?==INT #+32 (Si la dernière variable lu indique INT alors lit un INT). Pour représenter cette relation entre une condition et la lecture d'un type nous avons encore une fois choisis la surcharge d'operateur.

```
SwitchCondition    kpl::operator , (const ICondition &cond, const Type
&type)
{
    return (SwitchCondition(cond, type));
}

SwitchCondition    kpl::operator , (const Type &type, const ICondition
&cond)
{
    return (SwitchCondition(cond, type));
}
```

L'operateur « : » et « -> » n'étant pas sur chargeable nous avons surchargé l'operateur «, ». Le constructeur d'une SwitchCondition prend donc en premier paramètre la condition et en deuxième le type à lire en cas de condition remplis.

L'implémentation de BuildCondition se fait pour le moment de manière très simple :

```
void SwitchCondition::BuildCondition(llvm::BasicBlock *actionBlock,
llvm::BasicBlock *trueBlock, llvm::BasicBlock *falseBlock) const
{
    this->_cond->BuildCondition(actionBlock, trueBlock, falseBlock);
}
```

Pour le moment seul la condition est construite et non pas l'écriture ou la lecture du type associe dans le flux. Ceci est dut a un souci de clarté du code. Pour mener à bien la lecture et l'écriture du type il faudrait rajouter un certain nombre de variable qui rendraient cette fonction difficilement compréhensible en plus de faire passer des paramètres inutiles pour les BuildCondition de ConditionNode et Condition. La lecture ou l'écriture du type associe est donc préalablement construit dans trueBlock.

VII. Class Utiles

1) Container

```
template <typename T>
class Container
{
public:
    Container();
    Container(const Container& orig);
    ~Container();
    Container&          operator = (const Container&);
    void                Add(const T&);
    void                Add(const T*);
    Container&          operator << (const T&);
    Container&          operator << (const T*);
    void                Clear();
    unsigned int        Size();
    T*                  operator[](unsigned int i);
protected:
    std::vector<T*>      _list;

};
```

La class Container est une class template dont peuvent hériter toutes les class qui sont des collections. Par exemple la class Register est une collection de Type, la class Switch est une collection de SwitchCondition, StatictStuct est une collection de StaticType, etc. Le but de cette class est de fournir des méthodes qui rendent la manipulation de ces class plus facile. Ainsi les operateurs « << » (ajouter un item a la collection) et « [] » (accéder a un item de la collection) ont été surchargés.

Ainsi créé une StaticStruct devient beaucoup plus simple et on gagne en lisibilité :

```
Bitfield        bit5(5);
Bitfield        bit6(6);
Bitfield        bit7(7);
Bitfield        bit14(14);
StaticStruct    myStruct;

// Construction du register
myStruct << bit5 << bit6 << bit7 << bit14;
```