AngelScript と C++の連携記述を支援する C++ライブラリの設計と実装

2231084　高橋 裕司　　成見研究室

1　背景

1.1　AngelScript

AngelScript (以下 AS) [1] は、静的なオブジェクト指向のスクリプト言語で、C++でコア部分を実装したプログラムの設定を記述するのに使用される．AS の使用例として、ゲーム開発会社におけるゲームプログラムの実装を挙げる．社内で使用するゲームエンジンを C++で実装し、ゲームの登場人物の挙動を AS で記述することで、登場人物の挙動を変更する際、AS の記述を変更するだけで済むため、ゲームエンジンを含む C++コード全体を再コンパイルする必要がない．

1.2　AngelScript の仮想機械

AS コードを扱う C++プログラムは、図 1 のように AS の仮想機械 (以下 VM) オブジェクトを、AS が提供する C++API (以下 C++API) を用いて内部に作る．プログラムは、この VM を用いて AS コードを読み込み、VM 上で AS 関数を呼び出す．以降 AS の VM オブジェクトを、単に VM と呼ぶ．

C++プログラムは、C++関数やクラスに対応する AS の定義を VM に追加する．以降、このような VM への AS 定義の追加を、登録と呼ぶ．図 2 の C++構造体 `Pos` を登録する C++コードを、図 3 に示す．ここで、本研究に関連しない箇所は省略した．登録された `Pos` を用いる AS 関数 `MovePos` の呼び出しを、図 4 に示す．これらの図では、AS の記述を赤字で示している．

1.3　問題点

C++プログラマは、C++または AS の定義に従って、図 3、4 における API 関数の呼び出しを記述する．そのため、C++API 関数を用いる処理の記述に手間がかかる．

API 関数の呼び出しでは、AS クラスの名前や AS 関数の宣言を含む文字列を用いる．これらの AS 記述は実行時に検査される．そのため、記述中の誤りはコンパイル時に検出されず、実行時エラーを引き起こす．以上のように、既存の AS と C++の連携記述は煩雑であり、間違いを起こしやすい．

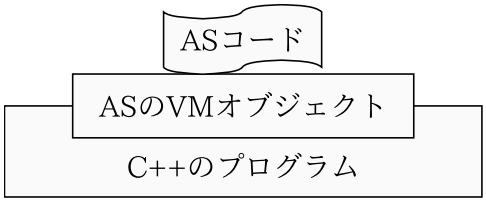


図 1: C++プログラムにおける AS コードの実行環境

```
struct Pos {
  Pos(int x, int y);
  ~Pos();
  void translate(int dx, int dy);
};
```

図 2: 平面座標を扱う C++クラス (構造体)`Pos` の定義

```
vm->RegisterObjectType(
  "Pos", sizeof(Pos), asGetTypeTraits<Pos>() ...);
vm->RegisterObjectBehaviour(
  "Pos", asBEHAVE_CONSTRUCT, "void f(int,int)", ...);
vm->RegisterObjectMethod(
  "Pos", "void translate(int, int)", ...);
```

図 3: `Pos` とそのメンバ関数の登録処理

```
asIScriptFunction* func = mod->GetFunctionByDecl(
  "const Pos MovePos(Pos, int, int)");
asIScriptContext* ctx = vm->CreateContext();
ctx->Prepare(func);
Pos pos = Pos(100, 200); ctx->SetArgObject(0, &pos);
ctx->SetArgDWord(1, 200); ctx->SetArgDWord(2, 300);
ctx->Execute();
const Pos R = *(const Pos*)(ctx->GetReturnObject());
ctx->Unprepare();
```

図 4: AS 関数の呼び出し処理

2　目的と方針

本研究は、C++と AS の連携記述を支援する C++ライブラリ CoHAS を実装し、プログラマの負担と誤りの可能性を減らす．CoHAS は、VM への登録と AS 関数の呼び出しを、短い記述で実装できるようにする．また、AS 記述の誤りを可能な限りコンパイル時に検出する．

CoHAS は、C++関数や型から、対応する AS の関数宣言や型の記述を持つ文字列と、API 関数に渡す C++の値を生成する．また、C++の型情報を用いて、複雑な API 関数を呼び出す記述を生成する．これらは、C++プログラムのコンパイル時に生成されるため、実行時オーバーヘッドを生じない．また、標準的な C++コンパイラとライブラリ以外には、特別な言語処理系を必要としない．

3　本機構の使用例

CoHAS を用いて図 3、4 の処理を記述すると、図 5、6 のようになる．コード中の C++関数は CoHAS が提供する登録関数で、図 3、4 の API 関数を内部で呼び出す．

```
cohas::RegisterValueClass<Pos>(vm);
cohas::RegisterCtr<Pos, int, int>(vm);
cohas::RegisterMethod<Pos, &Pos::translate>(vm);
```

図 5: CoHAS を用いた Pos と translate の登録

```
auto R = cohas::CallModuleFunctionR<"script",
    "MovePos", const Pos>(vm , Pos(100, 200), 200,
    300);
```

図 6: CoHAS を用いた MovePos の呼び出し

このように、CoHAS が提供する C++関数を用いることで、図 3、4 と同等の処理を誤りなく記述することができる.

4　評価

4.1　評価方法

3 つのプログラム Chat、Game、STK を用いて、以下の評価項目を、C++API を直接用いた場合と CoHAS を用いた場合で比較した．ここで、コンパイル時間の計測では、make コマンドの実行時間を測った.

- C++と AS の連携記述の行数 (以下 LOC).
- 連携記述中の AS 記述文字列の数 (以下 AS 記述数).
- プログラムのコンパイル時間.

4.2　評価結果

C++API 利用時に対する CoHAS 利用時の LOC と AS 記述数の割合を図 7 に示す．LOC は CoHAS 利用時で、Chat と Game では約 70%、STK では約 40% となった．また AS 記述数は、3 つのプログラムで約 20% となった．これらより、CoHAS の利用によって記述性が向上したことが確認できた.

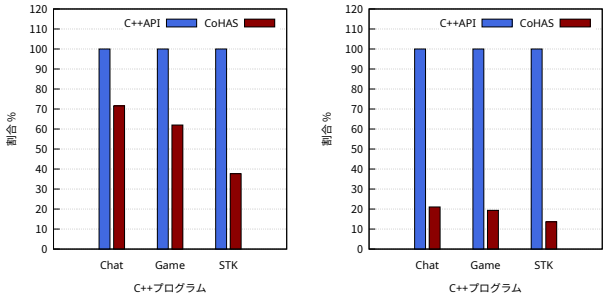
C++API 利用時に対する CoHAS 利用時のコンパイル時間の割合を図 8 に示す．コンパイル時間の増加量は、Chat と Game で 10% 未満だが、STK では約 60% であった．CoHAS が生成する AS 記述の数は、Chat と Game では 54、34 個だが、STK では 246 個であった．これより、AS 記述文字列の生成によって、STK のコンパイル時オーバーヘッドが増大したと考えた.

5　関連研究

Porkolab ら [4] は、C++コードのコンパイル時に動作する DSL パーサライブラリを提案した．CoHAS の実装では、このライブラリの実装を参考にした.

Beazley ら [2] は、C/C++ライブラリの記述からそのラッパーコードを生成するツール SWIG を提案した．CoHAS は、プログラマによる AS と C++の連携記述の実装を支援するため、SWIG と異なる目標を持つ.

Tanimura ら [3] は、Lua と C の連携を簡潔に実装するドメイン特化型言語 LuCa を提案した．LuCa コードをコンパイルする際、LuCa のコードを C のコードに変換する変換器を用いる．CoHAS は、C++のコンパイラを用いて AS 記述文字列を生成するため、特別な処理系を必要としない.



(a) LOC　　(b) AS 記述数

図 7: C++API、CoHAS 利用時の LOC、AS 記述数

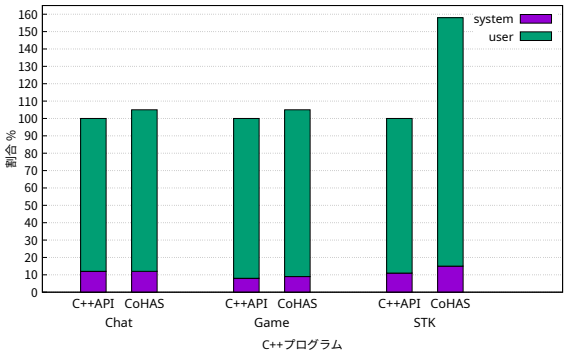


図 8: C++API、CoHAS 利用時のコンパイル時間の割合

6　終わりに

本研究では、AS と C++の連携処理の記述を支援する C++ライブラリ CoHAS を設計し、その実装を行った．C++API を直接利用した場合と比較したところ、コンパイル時のオーバーヘッドはあるものの、プログラムの記述性の向上を確認できた．今後は、CoHAS の未実装機能の実現と、コンパイル時のオーバーヘッドの削減を行う.

参考文献

[1] Andreas Jönsson. AngelScript - AngelCode.com. https://www.angelcode.com/angelscript/

[2] David M. Beazley. “SWIG: an easy to use tool for integrating scripting languages with C and C++”. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, pp.15. USENIX Association, 1996. https://www.swig.org/papers/Tcl96/tcl96.html

[3] Akira Tanimura and Hideya Iwasaki. “Integrating lua into C for embedding lua interpreters in a C application”. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp.1936–1943. Association for Computing Machinery, 2016. https://dl.acm.org/doi/10.1145/2851613.2851747

[4] Zoltán Porkolab and Ábel Sinkovics. “Domain-specific language integration with compile-time parser generator library”. In *ACM SIGPLAN Notices, Volume 46, Issue 2*, pp.137–146. Association for Computing Machinery, 2010. https://doi.org/10.1145/1942788.1868315