C言語ヘッダからのATS言語関数シグニチャの自動生成と段階的な線形型の導入

岡部 究 [1]

[1] 理化学研究所 計算科学研究機構

kiwamu@debian.or.jp

概 要 C言語を用いた低レベルなプログラミングにおいてメモリ破壊やリソースリークはやっかいな問題です。C言語の代替としてATS言語を採用し、線形型を用いてメモリアクセス権限やリソース生存範囲を表明することで、これらの違反をコンパイル時検査できることが知られています。しかし既存のC言語コード資産の上でATS言語実装を動かすには、当該C言語関数をATS言語にインポートする必要がありました。本論文では、この明らかに人為的なミスを誘発するプロセスをコード生成によって自動化するc2atsというツールを提案します。さらに、自動生成された関数シグニチャを使って危険なATSコードを作成した後、当該関数シグニチャに手動で線形型を導入することで、段階的にATSコードの安全性を向上できることを示します。

1 はじめに

近年プログラミング言語の進化はめざましく、ガベージコレクション/型推論/関数型プログラミングのような機能をそなえた言語を用いて高い生産性で安全なアプリケーションを作ることが可能になりました。しかし組み込み開発におけるハードウェアに近い領域でのプログラミングではC言語を用いた開発を余儀なくされています。C言語を用いた場合、その設計はバッファオーバフローのようなメモリ破壊やリソースリークのような不具合を容易にひきおこします。またC言語の機能は一般的な関数型言語よりも貧弱であり、代数的データ型のような今日一般的な機能でさえ使うことができません。

この大きな課題に対して様々な手法が提案されています。1つ目は静的コード解析 [1] と呼ばれる手法で、C言語コードをそのまま静的解析して不正なメモリ使用やリソースリークを防止します。しかしこのような解析器は多くのフォールスポジティブ (誤検知) やフォールスネガティブ (見逃し) を誘発してしまいます。またこの手法では設計基盤にC言語を使っているため、C言語と同等の型表現しか使うことができません。

2つ目はC言語のコード中に表明を手動で注入して、その表明を静的に検証する方法 [2][3] です。1つ目の手法と異なり、この手法を用いることでメモリの不正使用やリソースリークを正確に検出できます。しかし当然この手法でもC言語と同等の型表現しか使うことができません。

3つ目は低レベルプログラミング向けにC言語ではない新しい言語を導入する手法 [4][5] です。その新しい言語の機能を使うことで、メモリの不正使用やリソースリークを正確に検出でき、さらに型推論やより豊かな型表現を使うことも可能になります。しかし既存のC言語資産と協調動作をする場合にはC言語関数を当該言語にインポートする必要があります。このインポートは主に手動で行なわれるため人為的なミスを誘発する可能性があります。

本論文では上記3つ目の手法の具体例としてATS言語を紹介した後、この手法が持つ問題点を解決するために当該インポートをコード生成によって自動化するc2atsというツールを提案します。このc2atsを用いて生成された関数シグニチャを使って危険なアプリケーションをATS言語で書いた後、当該関数シグニチャに手動で段階的に線形型を導入することで当該アプリケーションに発生しうるメモリの不正使用やリソースリークの危険性を除去することができることを示します。

2　ATS 言語の紹介

2.1　概要

ATS 言語 [6][7] は静的型付けのプログラミング言語です。ATS 言語のソースコードは拡張子 .sats と .dats の 2 種類のファイルで構成されています。前者には C 言語の拡張子 .h のヘッダファイルのように関数の宣言を格納し、後者には C 言語の拡張子 .c のソースコードのように関数本体の定義を格納します。

ATS 言語のプログラムの値には ML 言語と同様に型がわりあてられており、値の性質を規定できます。その型は全称量化と存在量化で導入できる静的な値に依存することができ、その静的な値の型は種と呼ばれます。さらにコンパイル後に実体を持たない証明の値を持つこのもでき、その証明の型も通常の型と同様に静的な値に依存することができます。この両者からの依存によって、通常の値と証明の値を静的な値を通して結びつけることができます。

上記の証明とは具体的には次の 2 つです。1 つ目は古典論理を用いたコードの証明で、ソートアルゴリズムの証明などに使うことができます。2 つ目は線形論理を用いたリソースの追跡で、ポインタの生存範囲をコンパイル時に検査することができます。

ATS 言語の関数定義には証明の変数と動的な変数があり、前者はコンパイル時でのみの検査のために使われ、後者はコンパイル後の実コードにそのまま使われます。同様に関数宣言にも証明の引数/動的な引数/証明の返り値/動的な返り値を取ることができます。例えばリスト 1 のキーワード fun で定義された関数は fun_a という名前がついていています。この関数は PF1 と PF2 という型の証明引数と、char 型の動的な引数を取り、PF3 という型の証明返り値と int 型の動的な返り値を返します。

```
1 fun fun_a: (PF1, PF2 | char) -> (PF3 | int)
```

リスト 1. 証明の引数と動的な引数

動的な引数と動的な返り値のみを取り、証明の値を扱わない関数も作ることができ、さらに prfun や praxi キーワードを使うことで証明引数だけを取って証明返り値だけを返す関数を定義することもできます。

2.2　線形型による安全なメモリ使用とリソースの追跡

ATS 言語では線形論理の命題を「観」と呼び、中でもメモリへのアクセス権限を表わす観を「駐観」と呼びます。種 addr の静的な値アドレス l があり、その静的なアドレスに依存したポインタの型が ptr(l) であるとします。すると駐観 a@l は当該アドレス l に a 型の値があることを意味しています。すなわち先のポインタの値はデリファレンスすることができ、a 型の値を読み書きできることになります。もしそのコンテキストが l に対する駐観を持たない場合、型 ptr(l) のポインタはデリファレンスできません。

観や駐観は古典論理と異なり、関数の呼び出しにおいて生成と消費ができます。例えばリスト 2 の関数 fun_b は、全称量化で導入された静的なアドレス l1 に依存した駐観 char@l1 と同じく l1 に依存したポインタ ptr(l1) を取ります。そのため fun_b 関数を呼ぶ前にはコンテキストに駐観があるために ptr(l1) 型のポインタをデリファレンスして char 型の値を読み書きすることができました。ところが、fun_b 関数を呼んだ後は駐観 char@l1 は消費されてしまうために ptr(l1) 型のポインタをデリファレンスできなくなります。さらにこの関数は存在量化で導入された静的なアドレス l2 に依存した駐観 int@l2 と同じく l2 に依存したポインタ ptr(l2) を返します。関数 fun_b を呼び出した後は int@l2 が生成されているために ptr(l2) をデリファレンスして int 型の値を読み出すことができます。

```
1 fun fun_b: {l1:addr} (char@l1 | ptr(l1)) -> [l2:addr] (int@l2 | ptr(l2))
```

リスト 2. 駐観の生成と消費を行なう関数

このように関数の引数に取った駐観を消費したくない場合にはリスト 3 のように消費しない駐観の前に ! を付けます。このマークのおかげで関数 fun_b2 の呼び出しが終わった後も駐観 char@l1 は生きているので、ポインタ ptr(l1) をデリファレンスすることができます。

```
1 fun fun_b2: {l1:addr} (!char@l1 | ptr(l1)) -> [l2:addr] (int@l2 | ptr(l2))
```

リスト 3. 駐観を消費しない関数

さらに関数の呼び出し前後で駐観の形を変化 (観変化と呼ばれます) させることもできます。例えばリスト 4 の関数 fun_c を呼び出す前ではアドレス l に対応する駐観は int@l なので、ポインタ ptr(l) をデリファレンスすると int 型の値が得られますが、問うがい 関数を呼び出した後ではその駐観が消費されて新しい駐観 char@l に変化しているために、ポインタ ptr(l) をデリファンスすると char 型の値が得られることになります。

```
1 fun fun_c: {l:addr} (!int@l>>char@l | ptr(l)) -> void
```

リスト 4. 観変化

unsafe.sats というファイルをインクルードすると、(便利ですが) 危険な API を使うことが可能です。例えばその中で定義されている ptr_vtake 関数はなんの証明もないポインタを取り、任意の駐観を返すキャスト関数です。

C 言語では文字列は char * 型として扱われますが、ATS 言語ではリテラルとしての文字列である string 型、線形型としての文字列である strptr 型、静的な長さに依存した線形型の文字列である strnptr 型という 3 種類の文字列型があります。

2.3 ATS 言語から C 言語関数の呼び出し

C 言語の関数を ATS 言語から呼び出す場合には当該関数に型を付けた上で、インポートする必要があります。例えばリスト 5 の ATS コードを考えます。このコードの 4 行目から 8 行目までは C 言語の関数定義です。ATS 言語ではこのようにコード中に C 言語コードを埋め込むことができます。10 行目で C 言語で定義した関数のシンボル名 c_plus に ATS 言語での関数名 ats_c_plus を与えます。mac# から後続の文字列がシンボル名です。関数 ats_c_plus には ATS での型が明記されているので、13 行目のように ATS コードから呼び出すことができます。

```
1 #include "share/atspre_define.hats"
2 #include "share/atspre_staload.hats"
3
4 %{^
5 int c_plus(int a, int b) {
6   return a + b;
7 }
8 %}
9
10 extern fun ats_c_plus: (int, int) -> int = "mac#c_plus"
11
12 implement main0() = {
13   val r = ats_c_plus(1, 2)
14   val () = println!(r)
15 }
```

リスト 5. ATS 言語から C 言語関数の呼び出し

ここでは簡単のため C 言語コードを ATS 言語コードに埋め込んだ例を示しましたが、C 言語ソースファイルと ATS 言語ソースファイルが分割されている場合も同様に `mac#` を使って関数のシンボル名明記することで ATS 言語から C 言語関数を呼び出すことができます。

より詳しい ATS 言語の情報については”Introduction to Programming in ATS”[8] (筆者による翻訳 [9]) を参照してください。

3 c2ats 概要

前章で C 言語関数を ATS 言語から呼び出せることがわかりましたが、そのインポートは手動で行なわれました。この手作業でのインポートにおいて人為的なミスがあった場合、どんなに ATS コンパイラが正しく型検査したとしても、インポート元の C 言語関数とインポートされた ATS 関数シグニチャの型が合わないために不具合が混入してしまいます。

そこで筆者は c2ats[10] というツールを開発中です。本章では c2ats の詳細を解説する前にその使い方を通して概要を説明します。まずリスト 6 のような example.h ファイルがカレントディレクトリにあるとします。

```
1 #include <stdio.h>
```

リスト 6. example.h ファイル

このときコンソールからリスト 7 のように c2ats を実行することによって example.sats ファイルを自動生成できます。

```
1 $ cd example/hello
2 $ c2ats gen example.h > example.sats
```

リスト 7. example.sats の生成

生成された example.sats ファイルには先の example.h からインクルードしていた stdio.h 以下で宣言されている全ての関数がインポートされています。例えば C 言語の `printf` 関数はリスト 8 のように自動的に example.sats ファイルにインポートされます。

```
1 $ less example.sats
2 --snip--
3 viewdef ptr_v_1 (a:t@ype, l:addr) = a @ l
4 --snip--
5 fun fun_c2ats_printf: {l1:addr} (!ptr_v_1(char, l1) | ptr l1) -> int = "mac#printf"
6 --snip--
```

リスト 8. 自動インポートされた printf 関数

インポートされた `printf` 関数には、既存の ATS ライブラリと干渉しないように、`fun_c2ats_printf` という別名が付けられています。その第 1 引数は証明引数で、アドレス `l1` に型 `char` のデータがあること表明しています。通常の駐観の形式ではなく、`ptr_v_1` という別名を使っているのは多段ポインタに対応するためですが、詳しくは次章で解説します。第 2 引数は動的な引数でポインタです。これら 2 つの引数は同時に `l1` という静的な変数に依存していて、その種は `addr` です。ATS 言語では C 言語の可変長引数を扱う良い方法がないために、この `fun_c2ats_printf` 関数は単に 1 つの文字列を印字することしかできません。

この `fun_c2ats_printf` 関数を用いた ATS コードをリスト 9 のように作ることができます。このコードは、`string` 型の文字列リテラルを `string2ptr` 関数を使って一度何の証明もない `ptr` 型に変換した後、当該ポインタの先に `ptr_v_1(char, l1)` の形の駐観を `ptr_vtake` 関数を使って無理矢理生成します。さらにその駐観と `ptr` 型に変換された文字列リテラルを `fun_c2ats_printf` 関数に渡してコンソールに印字しています。

```
#include "share/atspre_define.hats"
#include "share/atspre_staload.hats"

staload UN = "prelude/SATS/unsafe.sats"

staload "example.sats"

fun my_printf (s: string): void = {
  val p = string2ptr(s)
  val (pfat, fpfat | p) = $UN.ptr_vtake(p)
  val ret = fun_c2ats_printf(pfat | p)
  prval () = fpfat(pfat)
}

implement main0 () = {
  val s = "Hello, world!\n"
  val () = my_printf(s)
}
```

リスト 9. main.dats ファイル

リスト 9 の main.dats ファイルはコンソールからリスト 10 のように ATS コンパイラでコンパイルした後実行することができます。

```
$ cd example/hello
$ patscc -o test_prog main.dats
$ ./test_prog
Hello, world!
```

リスト 10. main.dats のコンパイル

しかしリスト 9 の ATS コードは以下の点で危険です。

- 危険な API の入った unsafe.sats をインポートしている
- 何の証明もない単なるポインタから任意の駐観を取り出せる `ptr_vtake` 関数を使っている

これらの問題は、c2ats が C 言語のポインタをそのままに解釈して ATS コードを生成した結果、自動生成された `fun_c2ats_printf` 関数が `char` への駐観を取るように定義されてしまっているために生じています。そこで、自動生成された example.sats をリスト 11 のように、`fun_c2ats_printf` が ATS 言語の `string` 型を取るように手動で修正してみます。

```
$ cp example.sats example_welltyped.sats
$ vi example_welltyped.sats
--snip--
fun fun_c2ats_printf: (string) -> int = "mac#printf"
--snip--
```

リスト 11. fun_c2ats_printf 関数の引数を string に

これで main.dats をリスト 12 のようにシンプルかつ安全に書き直すことができます。

```
#include "share/atspre_define.hats"
#include "share/atspre_staload.hats"

staload "example_welltyped.sats"

implement main0 () = {
  val s = "Hello, world!\n"
  val _ = fun_c2ats_printf(s)
}
```

リスト 12. より安全な main.dats

上記の新しいコードには先のような危険な問題はありません。このように、C 言語と ATS 言語を協調させたプログラミングにおいて、c2ats が直接自動的にインポートした関数シグニチャは ATS プログラマが真に欲しい型表現ではありません。これは c2ats がまだ開発途上であることにも原因がありますが、より大きな課題は C 言語の関数定義が ATS 言語で使いたいレベルの不変条件を表明していないことに起因しています。この C 言語における表明の不足については次章で具体的に解説します。

そのため c2ats の作る sats ファイルは Ruby on Rails[11] の Scaffold のようなものだと考えることができます。c2ats で自動生成された sats ファイルは手っ取り早く ATS アプリケーションを作るための足場にすぎません。当該 sats ファイル内の関数シグニチャに問題があれば、手動でその型を強化すべきです。当該 sats ファイルは、決っして製品コードにそのまま無変更で採用されるべきではありません。その sats ファイルは多くの危険なアプリケーション実装を誘発する関数シグニチャを含んでいるからです。しかしその危険性は元の C 言語関数定義と同等のレベルです。

別の言い方をすると、元々危険であった C 言語関数インターフェイスを c2ats によって ATS 言語に写像することで、ATS プログラマはその危険性を unsafe.sats などの含む危険なキャストの使用という形で明確に知覚できるようになります。一旦知覚された危険性は ATS プログラマがその危険なコードを手動除去する過程で自然に安全になります。結果自動生成された sats ファイルの安全性を段階的に高めることが可能です。

4　c2ats による自動生成ルール

前章では c2ats の使い方について説明しました。この時 c2ats が C 言語ヘッダからどのような sats ファイルを生成するのか、そのルールを本章で説明します。また本論文では解説しませんが、構造体と構造体メンバーへのアクセサや関数ポインタも c2ats は自動変換します。

4.1　プリミティブ型

表 1. ATS 言語と C 言語のプリミティブ型

C 言語の型名	ATS 言語の型名
bool	bool
char	char
signed char	schar
unsigned char	uchar
short	sint
unsigned short	usint
int	int
unsigned int	uint
long int	lint
unsigned long int	ulint
long long int	llint
unsigned long int	ullint
float	float
double	double
long double	ldouble

ATS 言語のプリミティブ型は C 言語の型と対応しています。表 1 にその対応を示します。c2ats はこの表にしたがって C 言語のプリミティブ型を ATS 言語に写像します。

4.2 関数宣言

リスト 13 のような C 言語の関数宣言があったとき、

```
1 int func_a(int, char);
```

リスト 13. C 言語関数 func_a

c2ats はリスト 14 のような ATS の関数宣言に変換します。

```
1 fun fun_c2ats_func_a: (int, char) -> int = "mac#func_a"
```

リスト 14. c2ats によって変換された関数 func_ac

生成された関数名には `fun_c2ats_` という接頭辞が付けられます。`=` の左辺においては C 言語から ATS 言語への単純な文法の変換です。`"mac#func_a"` の部分には対応する C 言語の関数名を指定します。この指定によって当該 C 言語関数を ATS 言語から呼び出し可能になります。

4.3 ポインタ

C 言語のポインタを ATS 言語では `ptr` 型として単に扱うこともできます。しかしこの方法ではポインタを C 言語における `void *` のように扱うために、危険であると言えます。そこで c2ats は全てのポインタに対する駐観を自動生成します。

例えば、リスト 15 のような C 言語関数があるとき、

```
1 int my_getopt (int, char **, char *);
```

リスト 15. C 言語関数 my_getopt

c2ats はリスト 16 のような ATS の関数宣言に変換します。

```
1 fun fun_c2ats_my_getopt: {l1,l1_1,l2:addr} (!ptr_v_2(char, l1, l1_1), !ptr_v_1(char
    , l2) | int, ptr l1, ptr l2) -> int = "mac#my_getopt"
```

リスト 16. c2ats によって変換された関数 my_getopt

この時、型へのポインタを表わす駐観 `ptr_v_1` と、型へのポインタのポインタを表わす駐観 `ptr_v_2` と、型へのポインタのポインタのポインタを表わす駐観 `ptr_v_3` はリスト 17 のように同時に sats ファイルに付随して出力されます。

```
1 viewdef ptr_v_1 (a:t@ype, l:addr) = a @ l
2 dataview ptr_v_2 (a:t@ype+, l0: addr, l1: addr) =
3   | ptr_v_2_cons(a, l0, l1) of (ptr l1 @ l0, ptr_v_1 (a, l1))
4 dataview ptr_v_3 (a:t@ype+, l0:addr, l1:addr, l2:addr) =
5   | ptr_v_3_cons(a, l0, l1, l2) of (ptr l1 @ l0, ptr_v_2 (a, l1, l2))
```

リスト 17. c2ats における駐観の表現

ATS 言語では帰納的な駐観を作ることも可能ですが、c2ats では依存するアドレスの範囲を別々に指定したい場合を想定して型宣言の外に依存したアドレス全てを見せています。すなわちこの方式を使って、先の `my_getopt` 関数の変換結果において、リスト 18 のように種 `agz` を用いて、1 段目のポインタのアドレスである `l1` と `l2` のみが NULL ではなく、2 段目のポインタである `l1_1` は NULL であることを許容するような型シグニチャを書くことができます。

```
1 fun fun_c2ats_my_getopt: {l1,l2:agz}{l1_1:addr} (!ptr_v_2(char, l1, l1_1), !ptr_v_1
    (char, l2) | int, ptr l1, ptr l2) -> int = "mac#my_getopt"
```

リスト 18. 一部のポインタに NULL を許容しない ATS 関数 my_getopt

このように c2ats はポインタが NULL であることを許容されうるかどうかを推測しません。c2ats は全てのポインタには NULL が許容されるにもかかわらず、駐観があるものと見なすため、当該関数は当該ポインタを (NULL であるかもしれないにもかかわらず) デリファレンスできると想定していることになります。これは C 言語と同等に危険です。

本来、C 言語関数の実装者は引数の入出力や引数と返り値のポインタの関連などについて意図を持って設計したはずですが、その意図を c2ats が C 言語ヘッダのみから読み取ることは困難です。

さらに、引数としての駐観はその依存する全てのアドレスを全称量化で導入し、さらに関数が返っても消費しません。また返り値としての駐観はその依存する全てのアドレスを存在量化で導入します。これは POSIX の `FCLOSE(3)` のように、実際には駐観を消費するであろう関数シグニチャでも同様です。これもまた本来は C 言語設計者はポインタの生存区間を意図して設計したはずですが、その意図を c2ats が C 言語ヘッダから読み取ることは困難です

そこで c2ats で出力された sats ファイルを利用して ATS プログラミムを書く際には、より強い型を手動で割り当てることが奨励されます。

5 現実的なアプリケーション例: 自動生成された危険な関数の使用

より現実的なアプリケーション例として「自分自身のソースコードを標準出力に印字するアプリケーション」を考えます。リスト 19 のような example.h ファイルがあるとします。

```
1 #include <stdio.h>
```

リスト 19. example.h ファイル

このときコンソールからリスト 20 のように example.sats ファイルを自動生成すると、その中には以下の関数がインポートされています。

```
1 $ c2ats gen example.h > example.sats
2 $ less example.sats
3 --snip--
4 abst@ype struct_c2ats__IO_FILE // FIXME! Forward declaration.
5 typedef type_c2ats_FILE = struct_c2ats__IO_FILE
6 fun fun_c2ats_fclose: {l1:addr} (!ptr_v_1(type_c2ats_FILE, l1) | ptr l1) -> int = "
    mac#fclose"
7 fun fun_c2ats_fopen: {l1,l2:addr} (!ptr_v_1(char, l1), !ptr_v_1(char, l2) | ptr l1,
     ptr l2) -> [l3:addr] (ptr_v_1(type_c2ats_FILE, l3) | ptr l3) = "mac#fopen"
8 fun fun_c2ats_fread: {l1:addr} (!ptr_v_1(type_c2ats_FILE, l1) | ptr,
    type_c2ats_size_t, type_c2ats_size_t, ptr l1) -> type_c2ats_size_t = "mac#fread
    "
9 --snip--
```

リスト 20. example.sats の生成とその中身

上記の 3 つの関数 `fun_c2ats_fopen`, `fun_c2ats_fread`, `fun_c2ats_fclose` を使うことで、アプリケーションをリスト 21 のように構築することができます。このコードはファイル `"main.dats"` を `FOPEN(3)` で開いた後、当該ファイルをファイル末尾まで 128 バイトずつ読み込み、コンソールに印字します。

```
1 #include "share/atspre_define.hats"
2 #include "share/atspre_staload.hats"
3
4 staload UN = "prelude/SATS/unsafe.sats" // Unsafe!
5 staload STRING = "libats/libc/SATS/string.sats"
6
7 staload "example.sats"
8
9 extern praxi __create_view {to:view} ():<prf> to // Unsafe!
```

```
10 extern praxi __consume_view {from:view} (pf: from):<prf> void // Unsafe!
11
12 fun my_fopen (file: string, mode: string):
13     [l:agz] (type_c2ats_FILE@l | ptr(l)) = ret where {
14   val pn = string2ptr(file)
15   val (pfnat, fpfnat | pn) = $UN.ptr_vtake(pn)
16   val pm = string2ptr(mode)
17   val (pfmat, fpfmat | pm) = $UN.ptr_vtake(pm)
18
19   val (pffp | fp) = fun_c2ats_fopen(pfnat, pfmat | pn, pm)
20
21   prval () = fpfnat(pfnat)
22   prval () = fpfmat(pfmat)
23   val () = assertloc(fp > 0)
24   val ret = (pffp | fp)
25 }
26
27 fun my_fread {l:agz}{n:nat}
28     (pffp: !type_c2ats_FILE@l | fp: ptr(l), len: size_t(n)):
29     [m:int] (size_t(m), strnptr(m)) = ret where {
30   implement{} string_tabulate$fopr(s) = '_'
31   val buf_strptr = strnptr2strptr(string_tabulate(len))
32   val buf_ptr = strptr2ptr(buf_strptr)
33   val _ = $STRING.memset_unsafe(buf_ptr, 0, len)
34
35   val r = fun_c2ats_fread(pffp | buf_ptr, 1UL, $UN.cast2ulint(len), fp)
36   val r = $UN.cast(r)
37   val buf_strnptr = strptr2strnptr(buf_strptr)
38   val ret = (r, buf_strnptr)
39 }
40
41 fun my_fclose {l:agz} (pffp: type_c2ats_FILE@l | fp: ptr(l)): int
42     = ret where {
43   val ret = fun_c2ats_fclose(pffp | fp)
44   prval () = __consume_view(pffp)
45 }
46
47 fun readshow {l:agz} (pffp: !type_c2ats_FILE@l | fp: ptr(l)): void = {
48   val (r, str) = my_fread(pffp | fp, i2sz(128))
49   val str = strnptr2strptr(str)
50   val () = print(str)
51   val () = free(str)
52   val () = if r > 0 then readshow(pffp | fp)
53 }
54
55 implement main0 () = {
56   val (pffp | fp) = my_fopen("main.dats", "r")
57   val () = readshow(pffp | fp)
58   val r = my_fclose(pffp | fp)
59 }
```

リスト 21. main.dats ファイル

リスト 21 の main.dats ファイルはリスト 20 で自動生成した example.sats ファイルと共にコンパイルできますが、以下の点で危険なコードです。

- 危険性 1: 危険な API の入った unsafe.sats をインポートしている
- 危険性 2: 任意の駐観を消費できる `__consume_view` という証明関数を定義して使っている
- 危険性 3: 何の証明もない単なるポインタから任意の駐観を取り出せる `ptr_vtake` を使っている

- 危険性 4: `fun_c2ats_fread` 関数が駐観も長さ情報も持たない生のポインタ `buf_ptr` を取っている
- 危険性 5: 任意の型を任意の型にキャストできる `cast` という関数を使っている
- 危険性 6: `my_fread` 関数の返り値は静的な変数 `m` に依存しているがその値の範囲は `m <= n` であるべき

次の章では上記の危険性を取り除くことで、段階的に安全な ATS コードが手に入る様子を観察します。

6 現実的なアプリケーション例: 段階的な線形型の導入

前章での main.dats と example.sats を少しずつ修正して危険なコードを除去します。

6.1 fun_c2ats_fopen が string 型を取るように

危険性 3 は `fun_c2ats_fopen` 関数が `char` ポインタを引数に取るために当該ポインタのための駐観を作る必要性から生じたものです。そこで、リスト 22 のように関数シグニチャを `string` 型を取るように変更します。

```
1 $ cp example.sats example_welltyped.sats
2 $ vi example_welltyped.sats
3 --snip--
4 fun fun_c2ats_fopen: (string, string) -> [l3:addr] (ptr_v_1(type_c2ats_FILE, l3) |
      ptr l3) = "mac#fopen"
5 --snip--
```

リスト 22. string 型を取るように fun_c2ats_fopen 関数を修正

これで `fun_c2ats_fopen` 関数は引数を `string` 型で取るようになりました。結果 main.dats をリスト 23 のように書き換えて、`ptr_vtake` の使用を除去することができます。

```
1 $ vi main.dats
2 --snip--
3 fun my_fopen (file: string, mode: string):
4     [l:agz] (type_c2ats_FILE@l | ptr(l)) = ret where {
5   val (pffp | fp) = fun_c2ats_fopen(file, mode)
6   val () = assertloc(fp > 0)
7   val ret = (pffp | fp)
8 }
9 --snip--
```

リスト 23. ptr_vtake 関数を使わないように my_fopen 関数を修正

6.2 fun_c2ats_fclose で駐観を消費

危険性 2 は `fun_c2ats_fclose` 関数が返った後でもファイルポインタの駐観 `pffp` が消費されないために発生していました。C 言語の関数定義からは直接解釈することはできませんが、`fclose` の定義では一度クローズしたファイルポインタは使用不能になるはずです。そのため `fun_c2ats_fclose` は当該の駐観を消費すべきです。そこで関数シグニチャをリスト 24 のように修正します。

```
1 $ vi example_welltyped.sats
2 --snip--
3 fun fun_c2ats_fclose: {l1:agz} (ptr_v_1(type_c2ats_FILE, l1) | ptr l1) -> int = "
      mac#fclose"
4 --snip--
```

リスト 24. 駐観を消費するように fun_c2ats_fclose 関数を修正

また同時に fclose するファイルポインタは非 NULL であることが望ましいので、上記修正では当該ファイルポインタが依存している静的なアドレスの種を非 NULL である agz に設定しています。

すると、リスト 25 のように __consume_view 関数の使用が不要になり、main0 関数から直接 fun_c2ats_fclose 関数を呼び出せるようになります。

```
$ vi main.dats
--snip--
implement main0 () = {
  val (pffp | fp) = my_fopen("main.dats", "r")
  val () = readshow(pffp | fp)
  val r = fun_c2ats_fclose(pffp | fp)
}
--snip--
```

リスト **25.** __consume_view 関数を使わないように main0 関数を修正

6.3 fun_c2ats_fread が strnptr 型を取るように

危険性 4 を修正するために buf_ptr の元になった、string_tabulate 関数で確保した、長さ情報を静的に持つ strnptr 型の値を直接 fun_c2ats_fread 関数に渡せるように、リスト 26 関数シグニチャを修正します。

```
$ vi example_welltyped.sats
--snip--
fun fun_c2ats_fread: {l1:agz}{m:nat} (!ptr_v_1(type_c2ats_FILE, l1) | !strnptr(m)>>
    strnptr(o), type_c2ats_size_t, type_c2ats_size_t, ptr l1) -> #[o:nat | o <= m]
    size_t(o) = "mac#fread"
--snip--
```

リスト **26.** strnptr 型を使うように fun_c2ats_fread 関数を修正

上記の修正では fun_c2ats_fread 関数の動的な第 1 引数に strnptr(m) 型を渡し、この関数が返る時にはその型が strnptr(o) であることを表明しています。ただし、このとき o は o <= m にならなければなりません。またこの関数の返り値も o に依存しています。つまり fread した後のバッファの長さはその返り値と等しいことが期待されています。

また同時に fread するファイルポインタは非 NULL であることが望ましいので、上記修正では当該ファイルポインタが依存している静的なアドレスを非 NULL である agz に設定しています。

するとリスト 27 のように my_fread 関数を修正して、fun_c2ats_fread 関数に直接 strnptr 型を渡せるようになります。

```
$ vi main.dats
--snip--
fun my_fread {l:agz}{n:nat}
    (pffp: !type_c2ats_FILE@l | fp: ptr(l), len: size_t(n)):
    [m:int] (size_t(m), strnptr(m)) = ret where {
  implement{} string_tabulate$fopr(s) = '_'
  val buf_strnptr = string_tabulate(len)
  val buf_ptr = strnptr2ptr(buf_strnptr)
  val _ = $STRING.memset_unsafe(buf_ptr, 0, len)

  val r = fun_c2ats_fread(pffp | buf_strnptr, 1UL, $UN.cast2ulint(len), fp)
  val ret = (r, buf_strnptr)
}
--snip--
```

リスト **27.** strnptr 型を使うように my_fread 関数を修正

このとき、返り値と buf_strnptr が依存する静的な長さと返り値 r が依存する静的な長さが一致するので、cast による危険なキャストも不要になり、危険性 5 も解消することができました。

6.4 fun_c2ats_fread で size_t 型を使う

危険性 6 を解消するために、リスト 28 のように fun_c2ats_fread 関数の引数を size_t 型にして、さらにその値の範囲を明示します。

```
$ vi example_welltyped.sats
--snip--
fun fun_c2ats_fread: {l1:agz}{n,m:nat} (!ptr_v_1(type_c2ats_FILE, l1) | !strnptr(n*
    m)>>strnptr(o), size_t(n), size_t(m), ptr l1) -> #[o:nat | o <= n*m] size_t(o)
    = "mac#fread"
--snip--
```

リスト **28.** size_t 型を使うように fun_c2ats_fread 関数を修正

静的な変数 n と m を全称量化で導入し、その値は 0 以上です。fun_c2ats_fread 関数の動的な第 2 引数と動的な第 3 引数はそれぞれ n と m に依存し、さらに動的な第 1 引数である strnptr 型の文字列は当該関数の呼び出し前は n*m の長さです。本関数が呼び出された後では、strnptr 型の文字列の長さと size_t 型の返り値の値は o で、その範囲は p <= n*m です。

するとリスト 29 のように my_fread 関数を修正して、当該関数の返り値である size_t と strnptr 型の文字列に m <= n という制約をつけれるようになります。

```
$ vi main.dats
fun my_fread {l:agz}{n:nat}
    (pffp: !type_c2ats_FILE@l | fp: ptr(l), len: size_t(n)):
    [m:nat | m <= n] (size_t(m), strnptr(m)) = ret where {
  implement{} string_tabulate$fopr(s) = '_'
  val buf_strnptr = string_tabulate(len)
  val buf_ptr = strnptr2ptr(buf_strnptr)
  val _ = $STRING.memset_unsafe(buf_ptr, 0, len)

  val r = fun_c2ats_fread(pffp | buf_strnptr, i2sz(1), len, fp)
  val ret = (r, buf_strnptr)
}
```

リスト **29.** より強い制約を my_fread 関数に与える

さらにもはや unsafe.sats のインポートも不要になり、危険性 1 も解消することができました。

7 現実的なアプリケーション例: 安全になった関数シグニチャ

前章による段階的なコード修正によって、リスト 30 のようにより安全なアプリケーションが構築できました。

```
#include "share/atspre_define.hats"
#include "share/atspre_staload.hats"

staload STRING = "libats/libc/SATS/string.sats"

staload "example_welltyped.sats"

fun my_fopen (file: string, mode: string):
    [l:agz] (type_c2ats_FILE@l | ptr(l)) = ret where {
  val (pffp | fp) = fun_c2ats_fopen(file, mode)
  val () = assertloc(fp > 0)
  val ret = (pffp | fp)
}

fun my_fread {l:agz}{n:nat}
    (pffp: !type_c2ats_FILE@l | fp: ptr(l), len: size_t(n)):
    [m:nat | m <= n] (size_t(m), strnptr(m)) = ret where {
```

```
18   implement{} string_tabulate$fopr(s) = '_'
19   val buf_strnptr = string_tabulate(len)
20   val buf_ptr = strnptr2ptr(buf_strnptr)
21   val _ = $STRING.memset_unsafe(buf_ptr, 0, len)
22
23   val r = fun_c2ats_fread(pffp | buf_strnptr, i2sz(1), len, fp)
24   val ret = (r, buf_strnptr)
25 }
26
27 fun readshow {l:agz} (pffp: !type_c2ats_FILE@l | fp: ptr(l)): void = {
28   val (r, str) = my_fread(pffp | fp, i2sz(128))
29   val str = strnptr2strptr(str)
30   val () = print(str)
31   val () = free(str)
32   val () = if r > 0 then readshow(pffp | fp)
33 }
34
35 implement main0 () = {
36   val (pffp | fp) = my_fopen("main.dats", "r")
37   val () = readshow(pffp | fp)
38   val r = fun_c2ats_fclose(pffp | fp)
39 }
```

リスト 30. より安全な main.dats

また、リスト 30 アプリケーションはリスト 31 の sats ファイル中のより安全な関数シグニチャを使います。

```
1 $ vi example_welltyped.sats
2 --snip--
3 abst@ype struct_c2ats__IO_FILE // FIXME! Forward declaration.
4 typedef type_c2ats_FILE = struct_c2ats__IO_FILE
5 fun fun_c2ats_fclose: {l1:agz} (ptr_v_1(type_c2ats_FILE, l1) | ptr l1) -> int = "
    mac#fclose"
6 fun fun_c2ats_fopen: (string, string) -> [l3:addr] (ptr_v_1(type_c2ats_FILE, l3) |
    ptr l3) = "mac#fopen"
7 fun fun_c2ats_fread: {l1:agz}{n,m:nat} (!ptr_v_1(type_c2ats_FILE, l1) | !strnptr(n*
    m)>>strnptr(o), size_t(n), size_t(m), ptr l1) -> #[o:nat | o <= n*m] size_t(o)
    = "mac#fread"
8 --snip--
```

リスト 31. より安全な example.sats

このコードからは先の危険性 1-5 の全てを排除されています。

8 制限と今後

c2ats はまだ開発途上のために以下いくつか制限があります。

1. 単一の sats ファイルしか出力できません。大きなプログラムを扱う場合にはモジュール毎に sats ファイルを分割してインクルードしたい要求があります。
2. C 言語の前方宣言を ATS 言語に上手く写像できません。C 言語では中身の定義のない構造体を宣言した後、後続でその構造体へのポインタを使用し、さらにその後で実際の当該構造体の中身を定義することが可能でした。ところが ATS 言語ではこのような前方宣言を行なうことができません。現状は前方宣言に対応する ATS コードとして同名の抽象データ型宣言を挿入することでコンパイル可能にしています。
3. C 言語の関数宣言ではなく関数本体を ATS 言語に翻訳できません。
4. ATS 言語で共用体を構造体と同様に扱うのは明らかに危険です。代数データ型を使うべきです。

5. `const` などの特殊な修飾子やビットフィールドなどには ATS 言語に等価な表現がありません。

これらの制約の内、1,2,3 については以下のように今後の c2ats の開発で解消できると考えます。

1. ATS 言語の宣言がどの C 言語ヘッダで宣言されていたか調べることで、C 言語ヘッダと同じ構造の sats ファイル群を出力する機能を作成中です。
2. 前方宣言をできうるかぎり ATS 言語で宣言できる順序に整列します。整列できなかった前方宣言は本質的に循環しているため、手動で翻訳する必要があります。
3. goto のない C 言語コードは原理的には ATS 言語に翻訳可能です。しかし goto を含む C 言語コードは例外のない ATS 言語コードに翻訳するにはなんらかのモデルが必要です。つまり goto の用法をいくつかのモデル (大域脱出、リトライ、など) に分類した上で対応する ATS 言語のイディオムに変換することが可能かもしれません。

残る制限については上記が解消した後、ATS コンパイラ原作者と協議の上対応策を決定する予定です。

9 関連研究

c2ats は C 言語から高級言語を作成する唯一のプロジェクトではありません。c2hs[12] は C 言語のヘッダファイルと chs ファイルという注釈入りの Haskell コードから Haskell の FFI 定義を自動生成します。c2ats と異なり、chs ファイルで注釈された対象の C 言語コードに対してのみ FFI を生成するために効率的であると言えます。大きな C 言語コードの一部を ATS コード化したい場合には c2hs のような「興味のある対象だけを切り取って変換」する機能が必要でしょう。

corrode[13] は C 言語のヘッダファイルやソースファイルを入力として Rust 言語のソースコードを出力します。Rust 言語には ATS 言語がそなえる依存型を持たないため、Rust 言語に変換したコードの手動による正確さの強化は行ないにくいでしょう。一方、このツールは C 言語関数本体も Rust 言語に変換するため、c2ats はこのツールを将来の機能拡張の手本とすることができます。

10 結論

比較的単純な変換ルールを決めることで、C 言語ヘッダ中の宣言を ATS 言語の宣言 (sats ファイル) に自動的翻訳することが可能であることを示しました。自動変換された sats ファイルを使って型キャストを用いた危険な ATS 言語アプリケーションを作成できることを示しました。その sats ファイルと危険なアプリケーションに段階的に線形型を導入することで、より安全な関数シグニチャとアプリケーションが手に入ることを示しました。最後に本ツールの現状での制限と開発計画を展望しました。

謝辞

language-c という素晴しいライブラリを紹介してくれた村主崇行氏と、C 言語の型表現をいかにして ATS 言語の型表現に解釈しなおすか根気強いサポートをしてくれた Hongwei Xi に感謝します。

参考文献

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[2] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Matthew Danish. *Terrier: An embedded operating system using advanced types for safety.* PhD thesis, BOSTON UNIVERSITY, 2015.

[5] Alex Light. Reenix: Implementing a unix-like operating system in rust. 2015.

[6] Hongwei Xi. Applied type system. In *International Workshop on Types for Proofs and Programs*, pages 394–408. Springer, 2003.

[7] Hongwei Xi. The ATS programming language. `http://www.ats-lang.org/`.

[8] Hongwei Xi. Introduction to Programming in ATS. `http://ats-lang.sourceforge.net/DOCUMENT/INT2PROGINATS/HTML/`.

[9] Hongwei Xi. ATS プログラミング入門. `http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/`.

[10] Metasepi team. c2ats - generate ats interface from c code. `https://github.com/metasepi/c2ats/`.

[11] David Heinemeier Hansson. Ruby on Rails. `http://rubyonrails.org/`.

[12] Manuel M T Chakravarty. c2hs. `https://github.com/haskell/c2hs`.

[13] Jamey Sharp. corrode. `https://github.com/jameysharp/corrode`.