2006-02-06

Computational Properties of SNAFU

Computational Properties of SNAFU *

Yarom Gabay, Michael J. Ocean, Assaf J. Kfoury, and Likai Liu
{yarom, mocean, kfoury, liulk}@cs.bu.edu
Department of Computer Science, Boston University

Technical Report: BUCS-TR-2006-001

Abstract

Sensor applications in Sensoria [1] are expressed using STEP (*Sensorium Task Execution Plan*). SNAFU (*SensorNet Applications as Functional Units*) serves as a high-level sensor-programming language, which is compiled into STEP. In SNAFU's current form, its differences with STEP are relatively minor, as they are limited to shorthands and macros not available in STEP. We show that, however restrictive it may seem, SNAFU has in fact universal power; technically, it is a Turing-complete language, i.e., any Turing program can be written in SNAFU (though not always conveniently). Although STEP may be allowed to have universal power, as a low-level language not directly available to Sensorium users, SNAFU programmers may use this power for malicious purposes or inadvertently introduce errors with destructive consequences. In future developments of SNAFU, we plan to introduce restrictions and high-level features with safety guards, such as those provided by a type system, which will make SNAFU programming safer.

1 Introduction

SNAFU (*SensorNet Applications as Functional Units*) is a high-level functional language for programming sensor networks [1]. SNAFU is designed as an easy-to-use language which supports stateful, temporal, and persistent computation.

A SNAFU program is compiled into an intermediate abstract representation of the processing graph, called a Sensorium Task Execution Plan (STEP). The STEP graph will be linked to available computing resources by a *Sensorium Service Dispatcher* (SSD). The SSD's linking process loads STEP subgraphs to appropriate individual resources and binds the loaded subgraphs together with appropriate network protocols. This process of compiling, loading and linking is described in greater details in [1].

In this report we focus attention on the computational properties of SNAFU. Although it is seemingly quite restrictive by design, our main conclusion is that SNAFU is Turing-complete (i.e., it is powerful enough to implement any Turing machine).

This result leads to many important consequences, in particular, it is undecidable whether two arbitrary SNAFU programs are equivalent. Admittedly, such programs may be written in a rather convoluted and unusual style, uncharacteristic of legitimate uses of the Sensorium. Such pathological programs, regardless of the intent of the programmer, are the bane of system administrators.

We use the result herein to justify further restriction of SNAFU to make it harder, if not impossible, for SNAFU programs to disrupt normal operation of the Sensorium. Various established approaches may be applied to this end including adapting the language into a more restrictive DSL (*Domain-Specific Language*) or augmenting SNAFU with a type system that prevents "harmful" computations.

This report focuses on establishing the Turing-completeness of SNAFU. In Section 2, we devise a technique for implementing a special type of recursion in SNAFU. SNAFU does not support recursions. However, using SNAFU iterators (so called, *triggers*), we enable recursions in tail-position (or tail recursions) available to the programmer. With recursion, one may implement any algorithm in SNAFU. In particular, in Section 3, we present a SNAFU implementation of a call-by-value λ-calculus interpreter, which in turn implies the Turing-completeness of SNAFU.

*This work is partially supported by NSF award EIA-0202067.

2 Tail Recursion in SNAFU

Triggers are SNAFU iterators [1]. Although recursive functions are not supported in SNAFU, tail recursions can be acheived without language modification. Tail recursions are recursive functions in tail position, i.e., when the recursive call is made, no additional control information need be recorded – see Chapter 7 in [2]. Consequentially, when we call the next recursion instance, we can reuse the activation record of the current recursion instance. With this idea, tail recursions can be made available in SANFU.

At present, the only data structure available in SNAFU is a *pair*. `pairc`, `pairl` and `pairr` are pair construction, pair left-projection and pair right-projection, respectively. Currently, SNAFU is untyped and any two arbitrary values can be paired together. To ease presentation, we introduce a *tuple* notation that can be translated directly into pairs.

Definition 2.1 (Tuple Notation)**.**

1. $\langle a_1, a_2, \ldots, a_n \rangle \quad \triangleq \quad pairc(a_1, pairc(a_2, \ldots, pairc(a_n, nil)))$
2. *Given a tuple* t*,*

$$t.n \quad \triangleq \quad pairl(pairr(pairr(\cdots t))), \quad \textit{where } pairr \textit{ occurs } n-1 \textit{ times}$$

We demonstrate a general method of writing tail recursions in SNAFU. The following is a generalized tail-recursive function in C:

C style Recursive Function

```
// Assume, test1, test2, foo1a, foo1b, foo2a,
// foo2b and foo3 some simple functions.

int rec_function(int a, int b)
{
    if (test1(a,b))
        return rec_function(foo1a(a), foo1b(b));
    else if (test2(a,b))
        return rec_function(foo2a(a), foo2b(b));
    else
        return foo3(a,b);
}
```

We implement `rec_function` by using triggers. An iterative computation often needs to communicate values from one iteration to another. While, an imperative language allows this by using global variables, SNAFU provides the primitive *LTE* (Last_Trigger_Eval) that, when used inside a trigger, access the value returned by the last computation of the trigger action.

In order to implement a tail-recursive function that expects n arguments, we use a $n+2$ tuple as follows: $< \texttt{is-done}, \texttt{ret-value}, a_1, a_2, \ldots, a_n >$. The first component is a boolean flag indicating the end of computation, and the second component is the return value. The rest of the components are the function arguments. The symbol '$-$' is used in the code to indicate the value in that tuple component has no significance, since it will not be read.

SNAFU Implementation of "rec_function"

We use the following SNAFU constructs:

1. *cond* – followed by three expressions. the first is a test, the second will be carried out if the test evaluates to *true* and the last will be evaluated if the test evaluates to *false*.
2. *equals* – the equality relation.
3. *isNil* – tests whether a value is the special value NIL.

```
rec_function(a,b)
{
 level_trigger(   isNil(LTE) or (equals(LTE.1, false),
                  let_const  t = cond  isNil(LTE)
                                       <false,-,a,b>
                                       LTE
                  cond   test1(t.2,t.3)
                             <false,-,foo1a(a),foo1b(b)>
                  cond   test2(t.2,t.3)
                             <false,-,foo2a(a),foo2b(b)>
                             <true,foo3(a,b),-,->
}
```

The design of the SNAFU trigger constructs is such that three triggers exist. The use of *level_trigger* above provides a persistent computation in which the predicate is evaluated in perpetuity and the body of the trigger is only re-evaluated when the precondition evaluates to true. As a result of the use of the level_trigger construct in the above example our `rec_function` (1) continues evaluating until the function expires and (2) will be asynchronous insofar as it provides access to the result immediately despite the possibility that the result is not yet "complete". The caller of `rec_function` gets a 4-tuple as a result of which the first component is a boolean value indicating the completion of the computation.

Thus, to achieve a blocking call to `rec_function`, we can wrap a call to `rec_function` in the following construction:

```
let x = rec_function(a,b)
trigger( not(isNil(x)) and (equals(x.1, true) , x.2)
```

The *trigger* construct evaluates the precondition until it becomes true, at which case it returns the body and then terminates, thus solving both (1) and (2) above.

To conclude this section, we give an example of Fibonacci function implemented both in C and SNAFU. Both implementations are tail recursive.

Fibonacci Implementation in C

```
int fibonacci_aux(int n, int next, int result)
{
    if(n==0)
        return result;
    else
        return fibonacci_aux(n-1, next+result, next);
}

int fibonacci(int n)
{
    return fibonacci_aux(n, 1, 0);
}
```

Fibonacci Implementation in SNAFU

```
fibonacci_aux(n, next, result)
{
 level_trigger(  isNil(LTE) or (equals(LTE.1, false),
                 let_const  t = cond  isNil(LTE)
                                      <false,-,n,next, result>
                                      LTE
                 cond   equals(n,0)
                        <true,t.5,-,-,->
                        <false,t.3-1,t.4+t.5,t.4>
}

fibonacci(n)
{
 let x = fibonacci_aux(n ,1, 0)
 trigger( not(isNil(x)) and (equals(x.1, true) , x.2)
}
```

3 Interpreter for λ-calculus in SNAFU

Leveraging the method for writing tail recursions in SNAFU, we implement an interpreter for λ-calculus in SNAFU. The interpreter we present is a SNAFU function that takes as input a λ-calculus term and returns as result its normal-

form, if one exists. Our interpreter simulates a call-by-value reduction of the input. For simplicity, the input is assumed to be a closed term. Moreover, we avoid parsing the input by applying the interpreter function to a data structure that represents the AST (Abstract Syntax Tree) of the input term.

In order to proceed with the implementation, we describe a general datatype in SNAFU implemented using the tuple notation in definition 2.1. The datatype includes constructor, field selectors and a type predicate. Assume a datatype, my_type, with two fields, `f1` and `f2`. One way of implementing my_type might be:

```
my_type_create(f1,f2)  ≜  <"my_type",f1,f2>
my_type_to_f1(mt)      ≜  mt.2
my_type_to_f2(mt)      ≜  mt.3
is_my_type             ≜  equals(mt.1,"my_type")
```

This way we introduce the datatypes `var`, `lambda` and `app` to represent variable, lambda and application, respectively, with the following fields:

`var`	with field	`symbol`.
`lambda`	with two fields	`var` and `body`.
`app`	with two fields	`first` and `second`.

If we could have general recursions in SNAFU, our implementation might have been the following:

```
eval(e)
    cond   is_var(e)
           e
    cond   is_lambda(e)
           e
    cond   is_app(e)
           let    op = eval(app_to_first(e))
                  opd = eval(app_to_second(e))
           eval(substitute(lambda_to_body(op), lambda_to_var(op), opd))
    ERROR
```

where `eval` is the main function of our interpreter and `substitute` is implemented as follows:

```
substitute(e1, x, e2)
    cond   is_var(e1)
           cond   equals(var_to_symbol(e1),x)
                  e2
                  e1
    cond   is_lambda(e1)
           cond   equals(lambda_to_var(e1),x)
                  e1
                  lambda_create(lambda_to_var(e1),
                                substitute(lambda_to_body(e1), x, e2))
    cond   is_app(e1)
           app_create( substitute(app_to_first(e1), x, e2) ,
                       substitute(app_to_second(e1), x, e2))
    ERROR
```

The problem with `eval` and `substitute` is that they both have recursive calls that are not in tail position. We process our code in two steps as described in Chapters 7 and 8 in [2]. First, we translate our functions to tail-recursive functions by transforming the code to CPS (Continuation Passing Style). Next, we get rid of the high-order continuations by representing them as data structures. It is essential to do so since high-order functions are not available in SNAFU. This two-phase processing leads to implementation that can be done entirely in SNAFU. Below is the resulting code. Assume the following datatypes:

`k0`	with fields	–
`k1`	with fields	`k` and `e`.
`k2`	with fields	`k` and `e`.
`k3`	with fields	`k` and `e`.
`k4`	with fields	`k`, `e1`, `x` and `e2`.
`k5`	with fields	`k` and `e`.

Our implementation now includes three tail-recursive functions: `eval`, `substitute` and `apply_k`.

```
eval(e, k)
    cond   is_var(e)
           apply_k(k, e)
    cond   is_lambda(e)
           apply_k(k, e)
    cond   is_app(e)
           eval(app_to_first(e), k1_create(k,e))
    ERROR
```

```
substitute(e1, x, e2, k)
    cond   is_var(e1)
           cond   equals(var_to_symbol(e1),x)
                  apply_k(k, e2)
                  apply_k(k, e1)
    cond   is_lambda(e1)
           cond   equals(lambda_to_var(e1),x)
                  apply_k(k, e1)
                  substitute(lambda_to_body(e1), x, e2, k3_create(k,e1)))
    cond   is_app(e1)
           substitute(app_to_first(e1), x, e2, k4_create(k,e1,x,e2))
    ERROR



apply_k(k,e)
    cond   is_k0(k)
           e
    cond   is_k1(k)
           eval(  app_to_second(k1_to_e(k)), k2_create(k1_to_k(k),e))
    cond   is_k2(k)
           let new = substitute(   lambda_to_body(k2_to_e(k)),
                                   lambda_to_var(k2_to_e(k)), e, k0_create())
           eval(new, k2_to_k(k))
    cond   is_k3(k)
           apply_k(k3_to_k(k), lambda_create(lambda_to_var(k3_to_e(k)),e))
    cond   is_k4(k)
           substitute(   app_to_second(k4_to_e1(k)),
                         k4_to_x(k), k4_to_e2(k), k5_create(k2_to_k(k),e))
    cond   is_k5(k)
           apply_k(k5_to_k(k), app_create(k5_to_e(k),e))
    ERROR
```

We invoke the interpreter with `k0` as the continuation. `k0` behaves as the identity function. As an example of how to use the interpreter, the λ-calculus term $((\lambda x.x)\ (\lambda x.x))$ can be evaluated as follows:

```
let_const  e = app_create(   lambda_create("x", var_create("x")),
                             lambda_create("x", var_create("x")))
eval(e, k0_create())
```

Theorem 3.1. *SNAFU is Turing-complete, i.e., it is powerful enough to implement any Turing machine.*

Proof. The call-by-value λ-calculus is Turing-complete as shown in [3]. In this section we have shown how to encode closed λ-calculus terms in SNAFU and how to simulate their call-by-value β-reduction. This implies that SNAFU is also Turing-complete.

□

References

[1] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Michael Ocean. SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications. In *Proceedings of the IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets 2005)*, Boston, MA, October 2005.

[2] Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of programming languages (2nd ed.)*. Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[3] Mark Lillibridge. Unchecked exceptions can be strictly more powerful than call/CC. *Higher-Order and Symbolic Computation*, 12(1):75–104, 1999.