Verification of Parametric Counter Automata in a Dependently Typed Language

Work-in-progress Report

Christoph Herrmann Edwin Brady Kevin Hammond

University of St Andrews, School of Computer Science, Scotland/UK

{ch,eb,kh}@cs.st-andrews.ac.uk

Abstract

Counter Automata are Finite State Automata which include a set of registers containing natural numbers, and where state transitions occur based on these register values. The contribution of our work is to extend previous work on verification of counter automata properties to cover arbitrary computable functions and predicates (in principle). In order to achieve this, we use the dependently typed language Agda to verify properties. The advantage of this approach is that it allows us to verify both functional and extra-functional properties such as resource consumption, in the presence of iteration and control parameters. However, we are not able to achieve fully automatic verification. We illustrate our approach using three examples: i) verifying that a particular finite state automaton accepts exactly those binary sequences divisible by 3, if they are interpreted as numbers; ii) verifying the worst-case execution time of a system of three boxes, depending on an unknown control parameter; and iii) verifying the number of box executions for a system of two boxes, which is non-linear.

Categories and Subject Descriptors D [*2*]: 4

General Terms boxes, certificates, control parameters, execution time, functional programming, induction, iteration, non-linear costs, Presburger arithmetic, relations, resource usage

Keywords Agda, compositional verification, counter automata, dependent types, Hume

1. Introduction

Safety-critical systems will always benefit from formally specified, independently machine-checkable certificates of resource usage. The general context of our research is the resource analysis of programs written in Hume[12], a very high-level language for programming asynchronous resource-critical systems. However, in this paper we abstract from most of the special properties of Hume, focusing on analysing compositions of interacting hardware and software components, represented as *counter automata*. Our results are therefore applicable to a large class of systems written in a variety of programming languages.

[Copyright notice will appear here once 'preprint' option is removed.]

In [13], we have presented a formal model for a system of reactive components communicating over static wires. This can be used to formally verify resource bounds using a dependently-typed language framework. Our goal is to make it a *requirement* for there to be a formal certificate of resource usage as part of a program's *type*, and that this be tightly connected to the program itself. In [13], we have also pointed out the specific challenge presented to such a verification in the presence of unknown parameters, which require additional proofs of algebraic identities. In this paper, we begin to address this challenge. We make the following two contributions:

- We describe the symbolic calculation of resource usage of inductively defined models for components performing iteration.
- Our approach permits use of parameters as well as arbitrary computable functions in predicates, i.e., beyond Presburger arithmetic, which is important because they originate from the programs that we want to verify.

1.1 Modelling Hume Box Compositions in Agda

Ultimately, we intend to model programs written in Hume[12] by automatically translating them to Agda programs, and by annotating these programs with resource bounds corresponding to the resource usage of the original Hume programs. We therefore initially present our examples as Hume programs, togther with their corresponding translation into Agda [15]. Agda is a dependently typed programming language. It can be used to specify properties of reactive components such that the proper behaviour of a composition is guaranteed by the type system. A particular challenge we address is to include program values in the type of the Agda programs in order to be able to verify upper bounds on time usage in an accurate way.

In Hume, programmable boxes are connected to form a (potentially cyclical) process network, using static single-buffered communication channels (wires). A scheduler checks all boxes in a round-robin fashion. It executes a box if all its required input is available, and the box is not blocked due to non-consumption of any previous outputs.

We use Agda functions to model the behaviour of Hume boxes and their composition at an abstract level, using distinguished program values such as iteration counters and timing information in the type. Agda's inductive reasoning capability thenn allows us to verify the reaction times of a Hume program when given some input, even if this involves tracking values across multiple scheduling cycles.

In order to focus on the more interesting type-related issues, and to keep the associated expressions simple, we use a simpler model than that of [13]. Unlike our earlier work, the model used here abstracts from issues of runnability and blocking of boxes and

also does not require an extra phase for asserting wire values at the end of a superstep. This is possible because of the examples we consider in this paper. The abstraction we use also ignores all program values which do not impact whether a component will be executed. We call the remaining values *control parameters*. These include iteration counters, data structures sizes, and result values from particular execution modes.

As a first step towards a general solution, we try to obtain a closed form that expresses the values of the result parameters in terms of the input parameters; possible candidates for a solution can be obtained by profiling the automaton. In this paper, we show how to verify that such a formula is valid for all possible input parameter combinations, which is infinite.

1.2 Related Approaches

Several previous approaches deal with the analysis and verification of counter automata (or other, more specific, kinds of automata) for real time constraints. These approaches are mostly based on model-checking or constraint verification, and are generally limited in the kinds of functions that can be applied to calculate new register values. We overcome these limitation by using the dependently-typed functional programming language Agda. This allows any computable (total) functions to be expressed and to be integrated into the verification process. The challenge is to provide appropriate manual assistance to Agda, e.g., by providing induction proofs at particular places, in order to make the verification process tractable.

1.3 Overview

The remainder of this paper is organised as follows. In the next section we give a short introduction to parameterised counter automata and Agda. We explain the basic structure of a Hume program in Section 3, which also introduces an example to verify that a particular finite state automaton accepts exactly those binary sequences divisible by 3, if they are interpreted as numbers. In Section 4, we verify the worst-case execution time of a system of three boxes, depending on an unknown control parameter. In Section 5, we consider a system of two boxes that incurs execution time which is a polynomial of degree two in a control parameter, i.e., which is not linear. Section 6 discusses related work. Finally, Section 7 concludes. The Agda specifications used here are accessible from `https://www.cs.st-andrews.ac.uk/node/4146`.

2. Preliminaries

2.1 Parameterised Counter Automata

Counter automata extend finite-state automaton by including working counters. The transitions of a counter automaton can be guarded by (made dependent on) predicates of counters, and state transitions can change counter values. Formally, the treatment of counter values is expressed by relations between the pre- and postcondition of a state transition, in which the value of a counter before the transition is referenced by using its name and the value after the transition by using a primed version of its name. A relation such as $x' = x + 1$ could therefore be viewed as incrementing the counter x.

A counter automaton can be parametric. In the presence of parameters, all statements we make about such an automaton depend on these parameters. For example, n could be a parameter that is used in transition guards such that the accepting state is always reached, say within $2 * n + 3$ transitions. We could reduce this problem to the standard problem of reachability of the final state by introducing a counter x which is incremented in each transition and make the final transition dependent on $x < 2 * n + 3$.

More background information on parameterised counter automata, their formal treatment in automata-theoretic background (which we do not need for this paper), and the reachability problem can be found in [3].

2.2 The Dependently Typed Language Agda

Agda is a functional programming language with dependent types[1] Dependent types can certify program behaviour much more precisely than simple types; in particular they can express functional properties, such as invariants, as well as extra-functional properties, such as execution time.

In addition to Agda's rôle as a general-purpose programming language, it can also be used as an interactive theorem prover based on constructive type theory. Here, the types of functions express propositions, and the associated implementations of the functions act as proof components rather than being executed at run time. Such propositions can then be composed to construct certificates for functions executed at run time.

We employ Agda to prove properties about abstract specifications, so our Agda programs are not one-to-one translations of Hume programs but are rather specifications of abstract properties of Hume boxes. The propositions we show are, for example, that the execution time of a scenario of repeated box executions controlled by unknown parameters is below a given limit. This exploits only Agda's support for theorem proving. A more challenging task would be to let an Agda program carry out an automatic resource analysis of a Hume program and to use its dependent type system to prove that the analysis was correct.

3. Example 1 : Automaton for Divisibility by 3

Capturing abstract program properties can be important to decide which boxes are executed and which program branches are taken. Traditional analysis methods for counter automata are restricted to simple conditions, which might not be sufficient for the program that needs to be analysed. Our first example verifies a single box system which checks whether a given natural number is divisible by 3. Depending on this condition, the rest of the system could proceed with very different actions.

Figure 1 gives an implementation in Hume. Using `data` declarations the user can define new application-specific types and list their elements on the right-hand side. A Hume program consists of a set of box definitions with input and output ports (keywords `in` and `out`) and a set of rules (*pattern* `->` *result*). The top-level components of the pattern are matched against the values at the input ports and the top-level components of the result to the output ports. For a particular input, the first pattern that matches binds any free variables in the pattern, evaluates the expression at the right-hand side and assigns the components to the outgoing wires. In this example, rules correspond to state transitions; in general Hume permits the full power of functional programming. Asterisks (`*`) appearing on the left-hand side of a rule denote that the corresponding input value is not consumed (remains on the wire for the next box execution). On the right-hand side they denote that a value is not written to the outgoing wire, which might cause a consumer to wait for the final result, as in our example. This partial consumption of input and generation of output make Hume suitable to deal with asynchronicity at a high level. At the end of the program there are wiring instructions telling which outgoing port of a box is to be connected to which ingoing port.

We abstract from the details of Hume and represent this system by an automaton with three states. Each box activation will be associated with a state transition. Since our task is to verify a given system, or its abstraction by an automaton, we have to establish a connection between the concrete property of the automaton being in a certain state and its connection to the input being divisible by 3,

[1] We use Agda to refer to the most recent version, Agda2 [15].

```
data dState = a0 | a1 | a2;
data inT    = i0 | i1;

box check
 in (state :: dState , insymbol :: inT)
 out (result :: bool , newState :: dState)
match
   (a0, i0) -> (*,a0)
 | (a0, i1) -> (*,a1)
 | (a1, i0) -> (*,a2)
 | (a1, i1) -> (*,a0)
 | (a2, i0) -> (*,a1)
 | (a2, i1) -> (*,a2)
 | (a0, * ) -> (true, a0)
 | (_ , * ) -> (false, a0)
;
wire instream       to check.insymbol;
wire check.result   to consumer.input;
wire check.newState to check.state initially a0;
```

Figure 1. Hume Program for Checking Divisibility by 3

which is useful information at the high level to express the program behaviour by a case distinction.

Figure 2 shows the automaton associated with the Hume box. For simplicity, it does not have any counters, but the program behaviour is reflected very precisely to demonstrate the principle possibility to follow program calculations to obtain important information for the analysis.

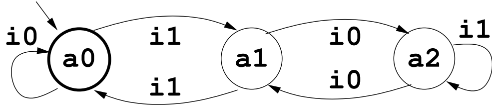

Figure 2. Divisibility by 3, Finite State Automaton

The number to be checked for divisibility by 3 is input as a prefix of a stream of the digits of its binary representation. The box has one input wire for receiving this stream bit-by-bit, an outgoing wire for sending the result whether the number is divisible by 3, in case that the input stream stops, and a feedback wire for storing the state. The state is a three-element set representing the remainder modulo 3 of the input received so far.

The state transition function is given by the Agda function `step`, where the state `a0`, `a1` and `a2` tells whether the remainder by 3 is 0, 1 or 2, respectively. The input symbols 0 and 1 are represented by `i0` and `i1`. Start and the only accepting state is `a0`. Since we are working with remainder classes, we define a new type M3 and make this the base type for indexing state and input symbols.

```
M3 : Set
M3 = Fin 3

data dState : M3 → Set where
 a0 : dState zero
 a1 : dState (suc (zero))
 a2 : dState (suc (suc (zero)))

data inT : M3 → Set where
 i0 : inT zero
 i1 : inT (suc (zero))
```

The important thing is that both state and input information are available in the type. We also define addition modulo 3, using operator $\oplus$. Then, the impact on the state transition is reflected in the index: if we are in remainder class `m` and we read the next digit `i`, then the number has been doubled by the shift of the new digit and we have to add the value of this digit. Since we are only interested in the remainder which is preserved by doubling and addition, we can calculate the value of the new state as $((\mathtt{m}\oplus\mathtt{m})\oplus\mathtt{i})$.

```
step : {m i : M3} → dState m → inT i
                    → (dState ((m ⊕ m) ⊕ i))
step a0 i0 = a0
step a0 i1 = a1
step a1 i0 = a2
step a1 i1 = a0
step a2 i0 = a1
step a2 i1 = a2
```

Because we are applying the step function several times, we also accumulate application of $\oplus$ in the type index. The `reduceState⊕` function reduces these:

```
reduceState⊕ : {a b c : M3} → (a ⊕ b ≡ c)
               → dState (a ⊕ b) → dState c
reduceState⊕ proof st = subst dState (≡-to-≅ proof) st
```

We can then specify the complete work of the automaton with its certificate in function `steps`, which tells that if the automaton reads a number `v` in State `m`, it will be in State $\mathtt{m} \oplus (\mathtt{v\ mod\ 3})$ afterwards.

```
steps : {m : M3}{v : ℕ} → number v → dState m
      → dState (m ⊕ (v mod 3))
```

4. Example 2: System of Three Boxes with Iteration

In our second example, we prove the computation time of an abstract system of three boxes A, B, and C in terms of an unknown control parameter n. In this system, A is executed once, B is executed for n iterations and then C once. The structure of the composition and the corresponding abstract automaton are given in Figure 4. The automaton has four states: `S0`: start; `S1`: after the box A has been executed; `S2`: after each iteration of the middle box B; and `S3`: end/after the final box C. We can then prove that the total cost will be (cost C + (n * costB) + costA).

The Hume program is given in Figure 3, with the implementation of the application-specific functions `f`, `g` and `h` left open.

We define the state explicitly as a data type `State`. The counters and costs are represented by the natural numbers ℕ in Agda. Since we want to include dependent information about the state and the counters into the certificates, i.e., the type information, we define two types `PState` and `SNat` which are indexed by the state and the counter value, respectively. The transition function `tr` works on the state and the counter but also increments the cost information, depending on the transition. E.g. the cost could be the execution time of a Hume box.

```
data State : Set where
 S0 : State
 S1 : State
 S2 : State
 S3 : State

data PState : State → Set where
 P : (s : State) → PState s

data SNat : ℕ → Set where
 ze : SNat 0
 su : m : ℕ → SNat m → SNat (suc m)

Cost : Set
Cost = ℕ   -- costA, costB and costC are of this type
Counter : Set
Counter = ℕ

tr : {m : ℕ} → (State × Counter × Cost)
```

```
f :: Int -> T1 -> T2;
g :: Int -> T2 -> T2;
h :: T2 -> T3;

box BoxA
 in (x :: (Int, T1))
 out (y :: T2, z :: (Int, T2))
match
   (0,x) -> (f 0 x, *)
 | (m,x) -> (*, (m, f m x))
 ;
box BoxB
 in (x :: (Int, T2), y :: (Int, T2))
 out (z :: T2, y1 :: (Int, T2))
match
   ((m,u), *) -> (* , (m-1, g m u))
 | (*, (0,u)) -> (u , *)
 | (*, (m,u)) -> (* , (m-1, g m u))
 ;
box BoxC
 in (x :: T2, y :: T2)
 out (z :: T3)
match
   (x, *) -> h x
 | (*, y) -> h y
 ;
wire extinput to BoxA.x;
wire BoxA.y   to BoxC.x;
wire BoxA.z   to BoxB.x;
wire BoxB.z   to BoxC.y;
wire BoxB.y1  to BoxB.y;
wire BoxC.z   to extoutput;
```

Figure 3. Three Boxes, Hume Code

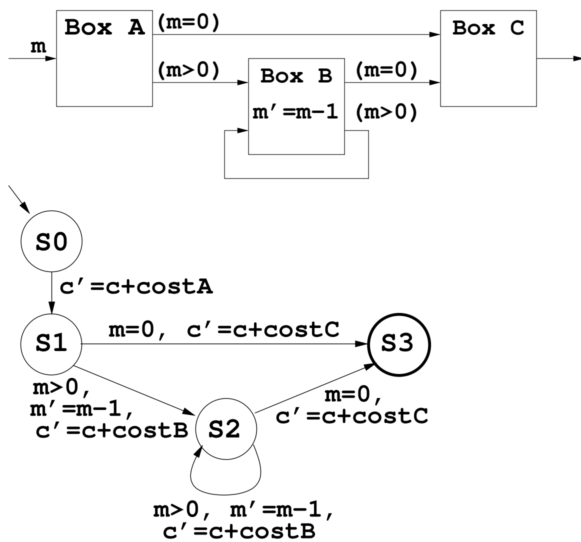

Figure 4. A Three Box Structure and Counter Automaton

```
                → (State × Counter × Cost)
tr (S0 , m , c) = (S1 , m , c + costA)
tr (S1 , 0 , c) = (S3 , 0 , c + costC)
tr (S1 , suc n , c) = (S2 , n , c + costB)
tr (S2 , 0 , c) = (S3 , 0 , c + costC)
tr (S2 , suc n , c ) = (S2 , n , c + costB)
tr (S3 , m , c) = (S3 , m , c)
```

We then calculate functions for the remaining costs. The functions are specialised by the state to simplify termination detection.

```
rmCosts3 : ℕ → ℕ
rmCosts3 _ = 0

rmCosts2 : ℕ → ℕ
rmCosts2 0 = costC
rmCosts2 (suc n) = rmCosts2 n + costB

rmCosts1 : ℕ → ℕ
rmCosts1 0 = costC
rmCosts1 (suc n) = rmCosts2 (suc n)

rmCosts0 : ℕ → ℕ
rmCosts0 n = rmCosts1 n + costA
```

We can then prove a closed form expression for the total cost depending on the parameter n:

```
lemma_rmCosts0 : {n : ℕ}
       → (rmCosts0 n ≡ (costC + (n * costB)) + costA)
lemma_rmCosts0 {0} = refl
lemma_rmCosts0 {suc m} =
 begin
  rmCosts0 (suc m)
 ≡⟨ refl ⟩
  rmCosts1 (suc m) + costA
 ≡⟨ refl ⟩
  rmCosts2 (suc m) + costA
 ≡⟨ cong (λ x → x + costA) (lemma_rmCosts2 {suc m})  ⟩
  (costC + (suc m) * costB) + costA
 ∎
```

Specialising the traversal function for the cost result:

```
tr3 : (State × Counter) → Cost
tr3 (S0 , m) = rmCosts0 m
tr3 (S1 , m) = rmCosts1 m
tr3 (S2 , m) = rmCosts2 m
tr3 (S3 , m) = rmCosts3 m
```

After lifting the state by index information (using $\oplus$ instead of + on the new domain) we can integrate the certificate tr3, which is equal to rmCosts for which we have proven the equality with its closed form for the start state S0:

```
ptr : {m : ℕ}{s : State} → (PState s × SNat m)
    → (SNat (tr3 (s , m)))
ptr (P S0 , m) with ptr (P S1 , m)
... | c = c ⊕ scostA
ptr (P S1 , ze) with ptr (P S3 , ze)
... | c = c ⊕ scostC
ptr (P S1 , su n) with ptr (P S2 , n)
... | c = c ⊕ scostB
ptr (P S2 , ze) with ptr (P S3 , ze)
... | c = c ⊕ scostC
ptr (P S2 , su n) with ptr (P S2 , n)
... | c = c ⊕ scostB
ptr (P S3 , m) = toSNat 0
```

5. Example 3: Verification of Non-Linear Execution Times

This example demonstrates how verification can be performed for a non-linear cost expression. The example consists of two boxes A and B. Box A has an external input and output, a feedback loop and a wire to and from Box B. In addition to these wires, Box B also has a feedback wire. Box A performs as many repetitive executions as a control parameter n prescribes; during this it decrements n. Box B takes the current value of n and performs n iterations before returning an intermediate result to Box A such that Box A can continue, i.e., enter its next iteration. The Hume program is given in Figure 5 and the box structure and automaton in Figure 6.

```
f :: Int -> T1 -> T1;
g :: Int -> T1 -> T1;

box BoxA
 in (exin :: (Int, T1), fromB :: T1, n :: Int)
 out (exout :: T1, toB :: (Int, T1), n' :: Int)
match
   ((0,x),*,*) -> (x,*,*)
 | ((n,x),*,*) -> (*, (n, f n x), n-1)
 | (*,x,0)     -> (x,*,*)
 | (*,x,n)     -> (*, (n, f n x), n-1)
 ;
box BoxB
 in (fromA :: (Int, T1), state :: (Int, T1))
 out (toA :: T1, state' :: (Int, T1))
match
   ((i,x),*)  -> (*, (i-1, g i x))
 | (*, (0,x)) -> (x, *)
 | (*, (i,x)) -> (*, (i-1, g i x))
 ;
wire extinput    to BoxA.exin;
wire BoxA.exout  to extoutput;
wire BoxA.n'     to BoxA.n;
wire BoxA.toB    to BoxB.fromA;
wire BoxB.toA    to BoxA.fromB;
wire BoxB.state' to BoxB.state;
```

Figure 5. Non-Linear Execution Time, Hume Program

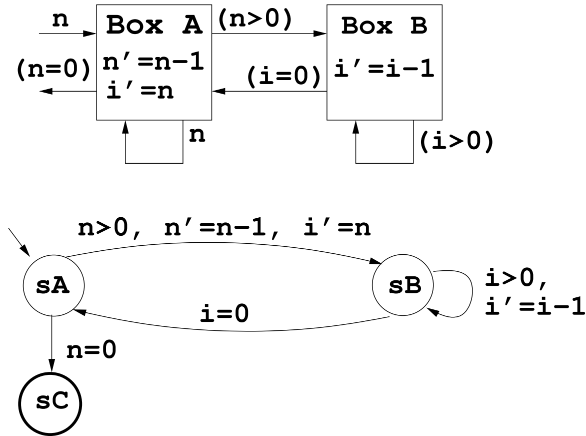

Figure 6. Non-Linear Execution Time, Box Structure and Automaton

We model this behaviour with a counter automaton with three states. State `sA` is the start state and the state Box `A` is in between two iterations. The State `sB` expresses that Box `B` performs iterations. State `sC` is the final state which we will omit in the Agda specification. The automaton also has two counters: n for the remaining outer iterations and i for the remaining inner iterations for the current value of n.

State transitions depend solely on the values of the counters instead of additional external inputs. We go from State `sA` to State `sB` if $n > 0$, set i to n and decrement n. In State `sB`, we go back to State `sA` if $i = 0$. Otherwise we are decrementing i in a loop at State `sB`.

The definition of data type `State` now contains `sA` and `sB` and based on this we define `PState` and `SNat` like in the previous section.

Furthermore, we abstract from a function which would simulate the automaton by successive application of the state transition function a step-counting function similar to the work by Rosendahl [16]. Function `rmSteps` tells us the number of remaining steps to be done for the particular state and the counters until we reach the final state. The simulation function `sim` integrates this cost function as a certificate in the type.

```
rmSteps : (State × ℕ × ℕ) → ℕ
rmSteps (sA , 0 , _) = 1
rmSteps (sA , (suc n) , _) = 1 + rmSteps (sB , n , suc n)
rmSteps (sB , n , 0) = 1 + rmSteps (sA , n , 0)
rmSteps (sB , n , (suc i)) = 1 + rmSteps (sB , n , i)

sim :{n i : ℕ}{s : State}→ (PState s × SNat n × SNat i)
                         → (SNat (rmSteps (s , n , i)))
sim (P sA , ze , _) = su ze
sim (P sA , su n , _) = su (sim (P sB , n , su n))
sim (P sB , n , ze ) = su (sim (P sA , n , ze))
sim (P sB , n , su i) = su (sim (P sB , n , i))
```

Having `rmSteps` as a certificate is only an intermediate step. Since it is recursive it contains all the overhead of the simulation. Therefore, we prove an equivalence between `rmSteps` and its closed form for the start state `sA`, which is $\frac{1}{2}n^2 + \frac{5}{2}n + 1$. After that, we could give this as a certificate for a specialisation of `sim` for that state being the start state. To simplify the proof, we multiply both sides of the equation by 2.

```
lemma2aux : {n i : ℕ} →
 (2 * rmSteps (sA , n , i) ≅ ((n + 5) * n + 2))
lemma2aux {0} {i} = ≅-refl
lemma2aux {suc n} {i} =
 begin
  2 * (rmSteps (sA , suc n , i))
 ≅⟨ ≅-refl ⟩
  2 * (1 + rmSteps(sB , n , suc n))
 ≅⟨ cong (λ x → 2 * (1 + x)) (lemma1 {n} suc n) ⟩
  2 * (1 + (suc n + rmSteps (sB , n , 0)))
 ≅⟨ cong (λ x → 2 * (1 + (suc n + x))) ≅-refl ⟩
  2 * (1 + (suc n + (1 + rmSteps (sA , n , 0))))
 ≅⟨ lemma_aux1 {1} suc n 1 rmSteps (sA , n , 0) ⟩
  2 * (1 + (suc n + 1)) + 2 * rmSteps (sA , n , 0)
 ≅⟨ cong (λ x → (2 * (1 + (suc n + 1))) + x)
        (lemma2aux {n} {0}) ⟩
  2 * (1 + (suc n + 1)) + ((n + 5) * n + 2)
 ≅⟨ normalise_special1 {n} ⟩
  n * n + 7 * n + 8
 ≅⟨ ≅-sym (normalise_special2 {n}) ⟩
  ((suc n) + 5) * (suc n) + 2
 ■
```

6. Related Work

There have been several previous approaches to representating resource usage of functional programs in the type system. In our own previous work, we have investigated the encoding of Hume programs in a dependently-typed language for analysing heap costs [4], but have not considered execution times, as in this paper, nor have we considered compositions of boxes, or parametric automata. Danielsson [9] has suggested counting execution time ticks in the type of a monad, thereby hiding it from the user, pointing out that it is highly desirable to replace additional manual proofs by automatic equality checking of arithmetic expressions in the monad type. Bove et al. [2] demonstrated how this can be achieved by normalising such terms. Earlier studies of formally-bounded time and space behaviour in a functional setting, e.g. [7, 14, 17], are based on restricted language constructs to ensure that bounds can be placed on time/space usage, and require considerable programmer expertise to exploit effectively. Other dependent type systems have been developed which allow an upper bound to be expressed

on the number of steps to complete a computation [8, 11], but these do not consider the kind of iterative programs we require in Hume.

In other previous work [6], we have investigated the possibilities for verification of invariance properties, such as size preservation of sorting, using a combination of *generalised algebraic data types* (GADTs) and *type families*, as provided by the current Glasgow Haskell Compiler, but have found that the fully dependently-typed approach used here is more elegant.

Finally, Agda has been used for a combined testing/proving process [10] and as a special backend for verifying Haskell programs automatically [1].

7. Conclusions

The main contribution of this paper is that it describes a new approach for verifying parameterised counter automata using a dependently typed language. This goes significantly beyond the limitations of Presburger Arithmetic or single counter approaches, but at the expense of full automation.

In our first example, the integer division/remainder in the arithmetic language, which we consider to be an appropriate counterpart to consuming a list of digits in a fully functional language, does not fit Presburger formulae, since integer division even by a constant is not linear: $(0 = (1 \text{ div } 3) = (2 \text{ div } 3) \neq (3 \text{ div } 3) = 1)$. In the second example, we have an external parameter which requires application of the induction principle to verify the property of resource consumption. In the third example, if we expressed the cost bound to be verified by a predicate, this would again not fit Presburger formulae, and we have the additional complication of an external parameter.

Our motivation is to verify properties of Hume box compositions. While we have previously developed good resource analysis for the purely functional program parts of Hume, in this paper we have addressed the crucial point in which inductive reasoning is required to verify the behaviour of a parameterised discrete system with feedback.

Our approach is based on the topical idea of having program certificates within the type information and of writing additional non-executed program components to prove that these certificates are correct.

We have not yet hit any fundamental problems when following this approach. If a system is designed carefully, then verification should go smoothly along with the design, with some manual interaction. However, we did encounter some artificial complications when using Agda. In particular, we needed to spell out rewriting of type information in some places where we would have preferred to use more sophisticated pattern matching.

We believe that part of the verification process described here could be automated. This could be done in two ways: i) in Agda, reflection techniques could be used to calculate normal forms of arithmetic expressions automatically, e.g., extensions of the decision procedure for commutative monoids [2]; and ii) based on a description of the design in a formal language, simple induction proofs in Agda could be generated automatically. We intend to investigate these issues in future work.

References

[1] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *ACM SIGPLAN 2005 Haskell Workshop, Haskell'05, Tallinn, Estonia, September 30, 2005*, pages 62–73. ACM Press, 2005. ISBN 1-59593-071-X.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda – a functional language with dependent types. In S. B. et al., editor, *TPHOLs*, LNCS 5674, pages 73–78, 2009.

[3] M. Bozga, R. Iosif, and Y. Lakhnech. Flat parametric counter automata. In *Automata, Languages and Programming*, Lecture Notes in Computer Science 4042, pages 577–588. Springer-Verlag, 2006.

[4] E. Brady and K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Proc. Implementation of Functional Languages (IFL 2005)*, volume 4015 of *LNCS*, pages 74–90. Springer, 2006.

[5] E. Brady and K. Hammond. Ensuring correct-by-construction resource usage by using full-spectrum dependent types. Available from `http://www.cs.st-and.ac.uk/ eb`, March 2009. Draft.

[6] E. Brady, C. Herrmann, and K. Hammond. Lightweight invariants with full dependent types. In P. Achten, P. Koopman, and M. T. Morazan, editors, *Trends in Functional Programming*, pages 161–176. Intellect, 2008.

[7] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, Dept. of Comp. Sci., Univ. of Edinburgh, April 1987.

[8] K. Crary and S. Weirich. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)*, pages 184–198, N.Y., Jan. 19–21 2000. ACM Press.

[9] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *POPL*, pages 133–144, 2008.

[10] P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information and Software Technology*, 46(15):1011–1025, December 2004.

[11] B. Grobauer. *Topics in Semantics-based Program Manipluation*. PhD thesis, BRICS, Department of Computer Science, University of Aarhus, August 2001.

[12] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[13] C. Herrmann and K. Hammond. Compositional timing verification with dependently typed program abstraction (extended abstract). In *Trends in Functional Programming, Draft Proceedings*, volume 10, pages 24–27, 2009.

[14] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, 1999.

[15] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[16] M. Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.

[17] D. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Functl. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.