Validating Optimizations of the PEGASUS IR in the CASH compiler framework

Ajay Mathews, Sumit Kumar Jha

April 16, 2007

Contents

Abstract

This draft explores the possibility of validating optimizations in the CASH compiler. Three different methods for increasing trust in the correctness of the compiled program are presented. These include the equivalence checking of the optimized PEGASUS intermediate representation against the unoptimized PEGASUS IR using (equality with uninterpreted functions) EUF decision procedures like UCLID and the model checking based validation of compiler optimizations using temporal logic. The theory for these techniques is developed and their benefits and trade-offs are compared. Finally, a case is presented for implementing the approach.

Chapter 1

Introduction

Compilers are an extremely complex piece of software and are, hence, prone to errors. One particularly crucial aspect of the compilation process is the use of optimization techniques. This draft focusses on the problem of validating optimizations in the CASH compiler framework.

The CASH compiler translates a C language specification into the PEGASUS intermediate representation, and then into one of the many target languages like assembly code, ASYNC or the TARTAN framework.

In Chapter 2, we study the PEGASUS language and present a translation from PEGASUS to the Equality with Uninterpreted Function (EUF) logic appended with the increment operator. This is the class of languages accepted by many decision procedures like UCLID [?]. In Chapter 3, we present a **bounded model checking** algorithm for checking the equivalence between two different PEGASUS intermediate representations. In Chapter 4, we built upon earlier work and develop a temporal logic approach for checking correctness of compiler optimizations. In particular, we indicate that the PEGASUS intermediate representation is particularly suited for **temporal logic model checking** because of the abundance of information and the simplicity of the transformations in this framework.

This draft develops two different approaches for the validation of the CASH compiler.

- Equivalence of PEGASUS intermediate representations using Decision Procedures
- Temporal Logic Model Checking for the Validation of Compiler Optimizations

Before we delve into our study of the PEGASUS intermediate representation, we outline a simple technique for checking the equivalence of the source code and the target produced. Given a source code P, a description in the target language T, and a set of shared observable O in the two programs, we sketch the possibility of performing a *bounded* equivalence testing of the observables O in the source S and the target T respectively.

The CASH compiler takes C as its input language. One of the target languages produced is an assembly code. The rapid growth of SAT solvers has reached a point where it is possible to ask a SAT solver whether two programs are equivalent up to a bounded depth.

The principle of equivalence checking of two programs is shown in Fig. 1.1. Statements in the program are translated into bit vector expressions and the bounded execution of a program is turned into a bit vector formula. The source program S is turned into a bit vector formula BV (S) and the target language program is turned into a bit vector formula BV (T). We finally ask the question $BV(S) \wedge BV(T) \wedge O_S = O_T$, where O_S are the observables in S and O_T are the observables in T.

This *satisfiability problem* can be answered by several tools like CBMC, Yices, UCLID, CVC, etc.

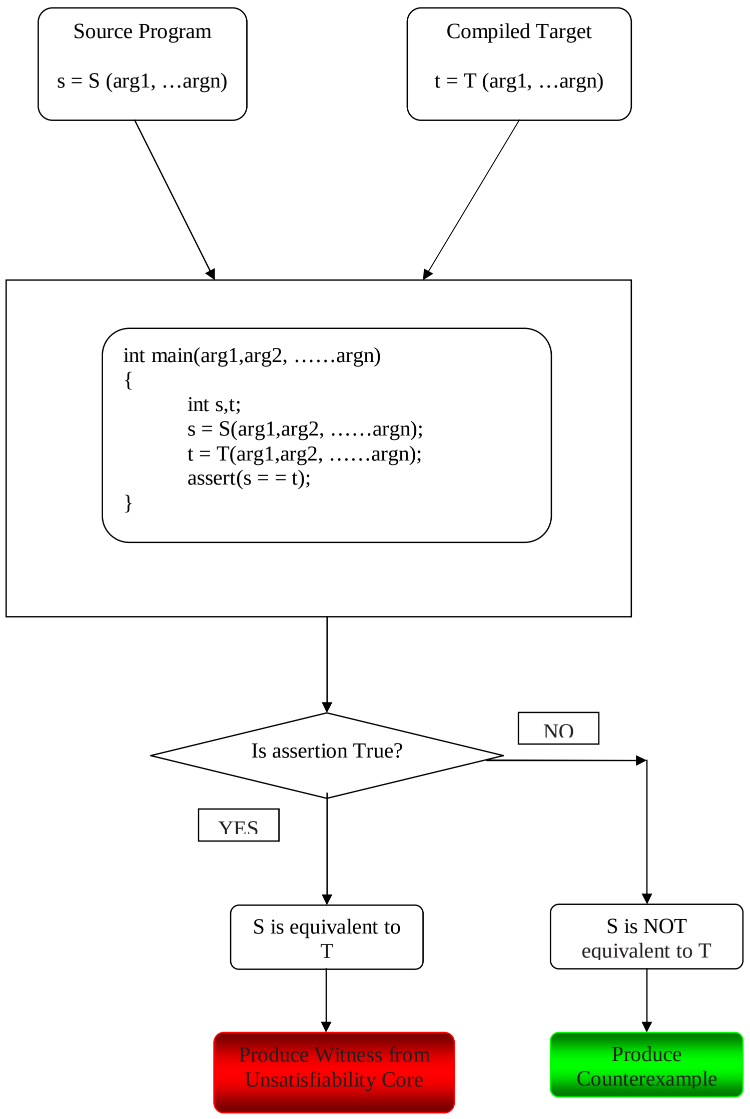

Figure 1.1: Equivalence of source and target programs

Chapter 2

Translating PEGASUS to EUF

The CASH compiler uses the PEGASUS internal representation which is a graph based intermediate representation. PEGASUS has a well defined formal execution semantics [**?**] and this intermediate representation is utilized to perform compiler optimizations.

Under the PEGASUS formal semantics model, the various operations are only related by a partial order. If the inputs to an operation are available, the output is produced. We associate with each operation a time at which its output is "ready" to be used or consumed. Thus, an operation can be performed **iff** all its inputs are ready.

We translate the formal semantics of PEGASUS to quantifier free logic of equality with uninterpreted functions (EUF) appended with the increment operator. This translation presented in Appendix 1 makes use of a degenerate uninterpreted function *null*, and is closely related to the formal semantics of PEGASUS presented in [**?**].

However, we then introduce logical clocks in each of the operations; an operation is executed only if all its inputs are already available and this execution also increments the local clock of the operation. Another execution of the same operations (perhaps in a loop) would hence produce a new value. The translation is presented in Table. 2.1 and Table. 2.2.

We will now briefly explain our translation philosophy. Each operation has a precondition which checks if all its inputs are ready. This is achieved by checking that the local clock of the operation is one more than the maximum of all the input clocks. When we perform the operation, we also increment the local clock of the operation. This prevents us from re-executing the same operation multiple number of times.

Our translation into EUF is an **eager** translation - an operation is executed as soon as it can be. In reality, the formal definition of PEGASUS allows many possible executions because of the partial order it imposes among instructions.

Prove a theorem saying that if the eager semantics of PEGASUS preserves equivalence, then so does any lazy execution...take eager IRs L1 and L2 - first argue L1 = E1 and L2 = E2 where E1 amd E2 are eager executions..then appeal to the fact that E1=E2.

PEGASUS IR	**Eager Translation into SMV/UCLID/Yices Input Language**
o = un_op(i)	$(ot == it) \Rightarrow (o_{ot} = \text{un\_op}(i_{it-1}));$ Increment ot
o = bin_op(i,j)	$(ot == max(it, jt)) \Rightarrow (o_{ot} = \text{bin\_op} (i_{it-1}, j_{jt-1}));$ Increment ot
$o_1, \ldots, o_n$ = fanout(i)	$(o_1t == it \wedge \ldots \wedge o_nt == it) \Rightarrow (o_{1_{o_1t}} = i_{it-1} \wedge \ldots \wedge o_{n_{o_nt}} = i_{it-1});$ Increment $o_1t, \ldots, o_nt$
$o = c$, c is constant	$(ot == 0) \Rightarrow o_{ot} = c;$ Increment ot
$o = merge(i_1, \ldots, i_n)$	$\bigvee_j [(ot == i_jt) \Rightarrow (o_{ot} = i_{j_{i_jt-1}})] ;$ Increment ot
$o = eta(p, i)$	$(ot == max(it, pt)) \Rightarrow [(p_{pt-1} \Rightarrow (o_{ot} = i_{it-1}))] ;$ Increment ot
$t_0 = combine(t_1, \ldots, t_n)$	$(t_0t == max(t_1t, \ldots, t_nt)) \Rightarrow [t_{0_{t_0t}} =$ newtoken]; Increment t_0t
$o = mux(i_1, p_1, \ldots i_n, p_n)$	$(ot == max_j\{i_jt, p_jt\}) \Rightarrow [((\bigwedge_j \neg p_{j_{p_jt-1}}) \Rightarrow o_{ot} = \Delta) \wedge (\bigwedge_j p_{j_{p_jt-1}} \Rightarrow (o_{ot} = i_{j_{i_jt-1}}))] ;$ Increment ot
$o = store(addr, p, v, t)$	$(ot == max(addrt, pt, vt, tt)) \Rightarrow [o_{ot} = newtoken \wedge (p_{pt-1} \Rightarrow M(addr_{addrt-1}) = v_{vt-1})];$ **Increment** ot

Table 2.1: Translation of PEGASUS to UCLID

PEGASUS IR	**Eager Translation into SMV/UCLID/Yices Input Language**
$(o, t_0) = load(addr, p, t)$	$(\ ot\ == max(addrt, pt, tt)\ \wedge\ t_0t\ == max(addrt, pt, tt)\ )\ \Rightarrow\ [\ t_{0_{t_0t}} = newtoken \wedge (\ \neg p_{pt-1} \Rightarrow (o_{ot} = \Delta)\ ) \wedge (\ p_{pt-1} \Rightarrow (o_{ot} = M(addr_{addrt-1}))\ )\ ]$; Increment ot
o = i	$ot == it \Rightarrow [\ o_{ot} = i_{it-1}\ ]$; Increment ot
return(i,t,p,pc)	$(\ pct\ == max(it, tt, pt)\ )\ \Rightarrow (p_{pt-1} \Rightarrow (pc.ot = pct \wedge \mathbf{pc.o_{pc.ot}} = i_{it}))$; Increment pct
$(o, t_0) = call_k(t, p, pc, i_1, \ldots, i_n)$	$(ot, t_0t,\ pc.arg1t, \ldots,\ pc.argnt, pc.tt, pc.pcint* == max_j\{tt,\ pt, pct, i_jt, call_kt\})$ $\Rightarrow [\ (\neg p_{pt-1} \Rightarrow (o_{ot} = \Delta \wedge t_{0_{t_0t}} = newtoken)) \wedge (p_{pt-1} \Rightarrow (pc.arg1_{pc.arg1t} = i_{1_{i_1t-1}} \wedge \ldots \wedge pc.argn_{pc.argnt} = i_{n_{i_nt-1}} \wedge pc.t_{pc.tt} = newtoken \wedge pc.pcin_{pc.pcint} = call_{k_{call_kt}}))\ ]$; Increment $ot, t_0t, pc.arg1t, \ldots, pc.argnt, pc.tt,\ pc.pcint$

Table 2.2: Translation of PEGASUS to UCLID

Chapter 3

Optimization Equivalence using Decision Procedures

As The PEGASUS IR has a formal execution semantics, the question whether two different PEGASUS IR are equivalent is a formally well defined problem.

Definition: A PEGASUS IR P_2 is said to be **t-bounded equivalent** to another PEGASUS IR P_1 with respect to a set of observable variables $O_1 \in P_1$, $O_2 \in P_2$ if and only if for each variable $o_1 \in O_1$, there exists a variable $o_2 = \mathit{equivalence\_map}(o_1) \in O_2$ such that o_1 and o_2 are identical during the execution of P_1 and P_2 for t operations.
We write $P_2 \cong_{t,O_1,O_2} P_1$.

The above definition is conservative and is also useful to circumvent the undecidability barrier of deciding the termination of programs.

We unroll the execution of the PEGASUS IR for n execution time steps. We create a symbolic program $S(P_i)$ that models the execution of the PEGASUS IR P_i for n steps using the translation of the PEGASUS IR into EUF presented in Chapter 2.

In order to show that the unoptimized PEGASUS IR P_0 is equivalent to the optimized PEGASUS IR P_n, we present two different approaches.

3.1 Direct Bounded Model Checking

We translate the unoptimized initial PEGASUS IR P_0 and the final optimized version P_n into symbolic programs $S(P_0)$ and $S(P_n)$.

We then ask a decision procedure if $S(P_0)$ and $S(P_n)$ are equivalent with respect to the observable set O_0 and O_n, where $O_i = \{o_i^0, o_i^1, \dots o_i^m\}$.

$$\boxed{S(P_0) \bigwedge S(P_n) \Rightarrow (\ \mathit{equivalent}(o_0^0, o_n^0) \wedge \dots \wedge \mathit{equivalent}(o_0^m, o_n^m)\ )}$$

The same may be written more succinctly as:

$$\boxed{S(P_0) \bigwedge S(P_n) \Rightarrow \mathit{equivalent}(O_0, O_n)}$$

3.2 Incremental Bounded Model Checking

We have a sequence of PEGASUS IRs obtained during the optimization phases :

$P_0 \rightarrow P_1 \rightarrow \dots\dots \rightarrow P_n$

Clearly, P_0 and P_n may be very different. So, instead of proving that $P_0 \equiv P_n$, one may argue that it may be easier to discharge a larger number of more simpler equivalences:

$$\boxed{\begin{array}{c} S(P_0) \bigwedge S(P_1) \Rightarrow \mathit{equivalent}(O_0, O_1) \\ S(P_1) \bigwedge S(P_2) \Rightarrow \mathit{equivalent}(O_1, O_2) \\ S(P_2) \bigwedge S(P_3) \Rightarrow \mathit{equivalent}(O_2, O_3) \\ \dots \\ \dots \\ S(P_{n-1}) \bigwedge S(P_n) \Rightarrow \mathit{equivalent}(O_{n-1}, O_n) \end{array}}$$

3.3 Distributed Equivalence Checking

If we have multiple cores or processors to run our verification algorithms, the above approach of proving equivalence incrementally may further be adapted to the distributed architecture.

We build a distributed algorithm which makes sure that we spend no more time than a single core implementation would. Observe that a distributed implementation seeking to solve the problem in parts may actually take more time than a single process implementation.

Assumptions: It is no harder to prove the equivalence of P_i and P_j than the equivalence of P_i' and P_j', where $i' \geq i, j' \leq j$.

$E(i,j) \geq E(i+\alpha, j-\beta)$, where $\alpha, \beta > 0$.

Aim: To design an algorithm which solves the problem for nm PEGASUS IRs within t time for n processors if we could prove equivalence within time t on a single processor.

Algorithm: Ask E(0,m), E(m,2m), ... E((n-1)m,nm) on processor $0, 1, \ldots n$ respectively.

Proof:

$E(0, nm) \geq E(0, m)$
$E(0, nm) \geq E(m, 2m)$
$\ldots$
$\ldots$
$E(0, nm) \geq E((n-1)m, nm)$

Now, the best a single core implementation could do is to show the equivalence of P_0 and P_{nm} in no less than E(0,nm) time. Hence, each of the queries in the distributed system would be answered by atmost E(0,nm) time.

Chapter 4

Temporal Logic Model Checking of PEGASUS Optimizations

4.1 Constructing the Kripke Structure from the PEGASUS IR

Given a PEGASUS IR, we construct an equivalent transition system or Kripke structure $T = (S, S_0, R, L)$, where

- $S = (N \times 2^A) \cup (N' \times 2^{A'})$ is the the set of states of the transition system, where N is the set of nodes of one of the PEGASUS IR and each node $n \in N$ is labeled with the set of information atoms A. The set of atoms labeling n is denoted by A(n). Similarly, N' denoted the set of nodes of the optimized PEGASUS IR and each node $n' \in N'$ is labeled with the set of information atoms A'
- $S_0 = (n_0, A(n))$: The initial state of the transition system in the initial entry node of the unoptimized PEGASUS IR along with the atoms that label the node.
- $R \subseteq S \times S$ s.t. $(s, s') \in R$ where $s = (n, A(n)) \in (N \times 2^A)$, $s' = (n', A(n')) \in (N' \times 2^{A'})$, and n' is obtained from n after the optimization.
- L is a labelling function which labels each state s with the set of atomic propositions that are true in that state. In our case, $L(s) = A(n)$, where $s = (n, A(n))$.

[1]

[1] Obviously, only a polynomial fraction of this exponential state space is actually reachable.

4.2 Temporal Logic as a language for PEGASUS Optimizations

4.2.1 Dead Code Elimination

Under the PEGASUS intermediate representation, each side effect free operation whose output is not connected is dead.

The corresponding (temporal) logic formula is

$AG[(output(v) == null \wedge \neg(side - effect(v) == null)) \Rightarrow (dead(\ \mathbf{X}\ v) = True)]$

4.2.2 Multiplexer Optimizations

Multiplexors with constant True predicate

If an input to a multiplexor is always true, the optimization transformation changes the node from a multiplexor to a direct connection.

$AG[(muxpredicate_i(v) == true \wedge nodetype(v) == mux) \Rightarrow (nodetype(\ \mathbf{X}\ v) == direct - connection \wedge input(\ \mathbf{X}\ v) == input_i(v))]$

Multiplexors with constant False predicate

If an input to a multiplexor is known to be false, the optimization transformation deletes the corresponding input and predicate from the multiplexor, while retaining the other inputs and predicates.

$AG(muxpredicate_i(v) == false \wedge nodetype(v) == mux) \Rightarrow (nodetype(\ \mathbf{X}\ v) == mux \wedge \bigwedge_{j<i}(muxpredicate_j(\ \mathbf{X}\ v) == muxpredicate_j(v) \wedge input_j(\ \mathbf{X}\ v) == input_j(v)) \wedge \bigwedge_{j>i}(muxpredicate_{j-1}(\ \mathbf{X}\ v) == muxpredicate_j(v) \wedge input_{j-1}(\ \mathbf{X}\ v) == input_j(v)))$

4.2.3 Global Common Subexpression, PRE, Redundant Memory Operation

These optimizations are implemented in PEGASUS by merging two nodes whose inputs all originate from identical sources.

$AG(((nodetype(v1) == nodetype(v2)) \wedge (\bigwedge_j input_j(v1) = input_j(v2))) \Rightarrow ((nodetype(\ \mathbf{X}\ v1) == nodetype(v1) \wedge nodetype(\ \mathbf{X}\ v2) == null) \wedge (\bigwedge_j input_j(\ \mathbf{X}\ v1) = input_j(v1))(\bigwedge_j input_j(\ \mathbf{X}\ v2) = null)))$

4.2.4 Constant Folding

Given an operation whose all inputs are constant, the operation can be performed at compile time and the node replaced by a simple constant.

$$AG((nodetype(v) == op_type \wedge \bigwedge_i (input_i(v) == constant)) \Rightarrow (nodetype(\mathbf{X}\ v) == constant \wedge \bigwedge_i (input_i(\mathbf{X}\ v) == null) \wedge (output(\mathbf{X}\ v) == op_type(input_0(v), input_1(v) \ldots\ldots))))$$

4.3 Checking PEGASUS graphs for Invariant Generation

Chapter 5

Appendix 1

PEGASUS IR	Translation into EUF logic with Increments
o = un_op(i)	$(o_{ot-1} == \text{null} \wedge i_{it-1} \neq \text{null}) \Rightarrow (o_{ot} = \text{un\_op}(i_{it-1}) \wedge i_{it}{:=}\text{null}$ [1] $\wedge (ot = ot+1))$
o = bin_op(i,j)	$(j_{jt-1} \neq \text{null} \wedge i_{it-1} \neq \text{null} \wedge o_{ot-1} == \text{null}) \Rightarrow (o_{ot} = \text{bin\_op}\ (i_{it-1}, j_{jt-1}) \wedge j_{jt}{:=}\text{null} \wedge i_{it}{:=}\text{null} \wedge \wedge (ot = ot+1))$
$o_1, \ldots, o_n$ = fanout(i)	$(i_{it-1} \neq \text{null} \wedge \bigwedge_k (o_{k_{o_k t-1}} == \text{null})) \Rightarrow (o_{1_{o_1 t}} = i_{it-1} \wedge \ldots \wedge o_{n_{o_n t}} = i_{it-1}) \wedge i_{it}{:=}\text{null} \wedge \ \wedge \bigwedge_j (o_j t = o_j t + 1))$
$o = c$, c is constant	$(o_{ot-1} == null) \Rightarrow o_{ot} = c \wedge ot = ot{+}1$
$o = merge(i_1, \ldots, i_n)$	$(o_{ot} == null \Rightarrow \bigwedge_j [(i_{j_{i_j t-1}} \neq null) \Rightarrow (o_{ot} = i_{j_{i_j t-1}} \wedge i_{j_{i_j t}} = null)]) \wedge ot = ot + 1$
$o = eta(p, i)$	$(i_{it-1} \neq null \wedge p_{pt-1} \neq null) \Rightarrow [(p_{pt-1} \Rightarrow (o_{ot} = i_{it-1})) \wedge (\neg p_{pt-1} \Rightarrow (1)) \wedge i_{it} = null \wedge p_{pt} = null \wedge ot = ot + 1]$
$t_0 = combine(t_1, \ldots, t_n)$	$(t_{0_{t_0 t}} == \text{null} \wedge t_{1_{t_1 t}} \neq null \wedge \ldots \wedge t_{n_{t_n t}} \neq null) \Rightarrow [t_{0_{t_0 t}} = \text{newtoken} \wedge t_{1_{t_1 t}} = null \wedge \ldots \wedge t_{n_{t_n t}} = null \wedge t_0 t = t_0 t + 1]$
$o = mux(i_1, p_1, \ldots i_n, p_n)$	$\bigwedge_j (i_{j_{i_j t-1}} \neq null \wedge p_{j_{p_j t-1}} \neq null) \Rightarrow [(\bigwedge_j \neg p_{j_{p_j t-1}} \Rightarrow o_{ot} = \Delta) \wedge (\bigwedge_j p_{j_{p_j t-1}} \Rightarrow o_{ot} = i_{j_{i_j t-1}}) \wedge \bigwedge_j (i_{j_{i_j t}} = null \wedge p_{j_{p_j t}} = null) \wedge ot = ot + 1]$
$o = store(addr, p, v, t)$	$addr_{addrt-1} = null \wedge p_{pt-1} = null \wedge t_{tt-1} = null \wedge v_{vt-1} = null \wedge o_{ot-1} = null \Rightarrow [o_{ot} = newtoken \wedge addr_{addrt} = null \wedge p_{pt} = null \wedge t_{tt} = null \wedge v_{vt} = null \wedge (p_{pt-1} \Rightarrow M(addr_{addrt-1}) = v_{vt-1}) \wedge$ **Increment** $addr, p, v, t\ ]$

PEGASUS IR	Translation into EUF logic
$(o, t_0) = load(addr, p, t)$	$addr_{addrt-1} \neq null \wedge p_{pt-1} \neq null \wedge t_{tt-1} \neq null \wedge v_{vt-1} \neq null \wedge t_{0_{t_0 t-1}} == null \Rightarrow [\, t_{0_{t_0 t}} = newtoken \wedge addr_{addrt} = null \wedge p_{pt} = null \wedge t_{tt} = null \wedge v_{vt} = null \wedge (\neg p_{pt-1} \Rightarrow (o_{ot} = \Delta)) \wedge (p_{pt-1} \Rightarrow (o_{ot} = M(addr_{addrt-1}))) \,]$
o = i	$i_{it-1} \neq null \Rightarrow [\ o_{ot} = i_{it-1} \wedge\ i_{it}$:=null $] \wedge (ot = ot + 1)$
return(i,t,p,pc)	$i_{it-1} \neq null \wedge t_{tt-1} \neq null \wedge p_{pt-1} \neq null \wedge pc_{pct} \neq null \Rightarrow i_{it} = null \wedge t_{tt} = null \wedge p_{pt} = null \wedge (p_{pt-1} \Rightarrow \mathbf{pc.o_{pc.ot}} = i_{it})$
$(o, t_0) = call_k(t, p, pc, i_1, \ldots, i_n)$	$t_{tt-1} \neq null \wedge p_{pt-1} \neq null \wedge pc_{pct-1} \neq null \wedge \wedge i_{1_{i_1 t-1}} \neq null \wedge \ldots \wedge i_{n_{i_n t-1}} \neq null \Rightarrow [\ (\neg p_{pt-1} \Rightarrow (o_{ot} = \Delta \wedge t_{0_{t_0 t}} = newtoken)) \wedge (p_{pt-1} \Rightarrow (pc.arg1_{pc.arg1t} = i_{1_{i_1 t-1}} \wedge \ldots \wedge pc.argn_{pc.argnt} = i_{n_{i_n t-1}} \wedge pc.t_{pc.tt} = newtoken \wedge pc.pcin_{pc.pcint} = call_{k\,call_k t})) \wedge\ t_{tt} = null \wedge p_{pt} = null \wedge pc_{pct} = null \wedge \wedge i_{1_{i_1 t}} = null \wedge \ldots \wedge i_{n_{i_n t}} = null \wedge (ot = ot + 1)\ ]$