Abstract
Given the growing interest in the JVM and Microsoft’s CLI as programming language implementation targets, code generation techniques for efficient stack-code are required. Compiler infrastructures such as LLVM are attractive for their highly optimizing middleend. However, LLVM’s intermediate representation is register-based, and an LLVM code generator for a stack-based virtual machine needs to bridge the fundamental differences of the register and stack-based computation models.
In this paper we investigate how the semantics of a register-based IR can be mapped to stack-code. We introduce a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses, which is inherently incompatible with the consuming semantics of stack-based operators. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Code-generation then reduces to scheduling of treegraph nodes in the most cost-effective way. We implemented a treegraph-based instruction scheduler for the LLVM compiler infrastructure. We provide experimental results from our implementation of an LLVM backend for TinyVM, which is an embedded systems virtual machine for C.

Keywords: bytecode instruction scheduling, stack-code, treegraphs, DAGs, LLVM
1 Introduction

Stack-code provides a compact instruction encoding, because instructions only encode the operation being performed, while its operands are implicitly consumed from the stack. Likewise, result-values of operations are implicitly produced on the stack. Network computing is facilitated by a compact program representation, because smaller programs take less time to transmit. For this reason the Java virtual machine (JVM, [12]) and various virtual machines (VMs) for wireless sensor networks, e.g., [14,19,9] employ a stack-based instruction set. Stack-based VMs are an attractive target for code generation, because several non-trivial tasks such as register allocation and callstack management are not required. All-in-all, 58 language implementations targeting the JVM and more than 52 programming languages targeting Microsoft’s .NET’s common language runtime (CLR, [17]) have been reported [24,23].

The renewed interest in stack-machines requires compiler support for the generation of efficient stack-code. The LLVM [11,10] compiler infrastructure is attractive for its highly-optimizing middle-end, but LLVM uses static single assignment form (SSA, [4]) as its intermediate representation (IR). Targeting LLVM to a stack-machine is complicated by the fact that SSA assumes an underlying register machine. Operands in registers can be used several times, whereas stack-based operations consume their operands.

Because the last-in, first-out stack protocol cannot support random operand access and unlimited operand lifetimes, operands that are not immediately used can be temporarily stored as local variables (temporaries) instead of keeping them on the stack. Generating bytecode to both minimize code size and improve performance by avoiding temporaries has been referred to as stack allocation [8,13]. A stack allocation method is required to provide the operands of each operation on the top of stack (TOS) when operations need them. If the underlying stack machine provides instructions to manipulate the stack, operands can be fetched from below the TOS. E.g., the JVM provides SWAP and DUP instructions to manipulate the stack. If the cost of a stack manipulation is less than the costs from stores and loads from a temporary, keeping a value on the stack is profitable.

In this paper we introduce a novel intermediate representation called treegraphs to facilitate stack allocation from register-based IRs. Treegraph nodes encapsulate computations that are compatible with a stack-based computation model. Edges between treegraph nodes represent dependencies on operands that have multiple uses. Multiple operand uses are inherently incompatible with the consuming semantics of stack-based operators. Instead of saving multiply-used values in temporaries, our method keeps all values on the stack. We perform stack allocation for treegraphs through scheduling of treegraph-nodes such that (1) dependencies between nodes are satisfied, and (2) the...
number of nodes that don’t require stack manipulation because their operands are already on the TOS is maximized.

The remainder of this paper is organized as follows. In Section 2 we provide background information and survey the related work. Section 3 introduces our treegraph IR and the basic treegraph scheduling algorithm. In Section 4 we discuss the minimization of overall stack manipulation costs. Section 5 contains experimental results. We draw our conclusions in Section 6.

2 Background and Related Work

Generating register-code for arithmetic expressions was first studied by Andrei Ershov [5]. Sethi and Ullman used Ershov numbers to devise an algorithm that they prove to generate optimal code for arithmetic expressions [18]. Aho and Johnson used dynamic programming to generate optimal code for expression trees on CISC machines [1].

Compiler back-ends for stack machines perform a DFS traversal of expression trees (the abstract syntax tree or other IR equivalent) and generate code as a side-effect. Common subexpression elimination finds expressions with multiple uses. The resulting DAGs complicate stack code generation: unlike values in registers, operands on the stack are consumed by their first use and are thus unavailable for subsequent uses. Fraser and Hanson’s LCC [6] compiler for C comes with its own backend for a stack-based virtual machine. LCC converts DAGs to trees by storing multiply-used values in temporaries and replacing references by references to the corresponding temporaries. This approach is taken with the LLVM backend for Microsoft .NET MSIL code, and it can be observed with code produced by Sun’s javac compiler.

A significant amount of research has been conducted to reduce the number of redundant store/load combinations by post-compilation transformations of bytecode [8,13,21,22]. In [15], dynamic instruction scheduling is performed to reduce the stack usage of a JVM. These methods are different from our approach in that they operate on the generated bytecode itself. Our approach avoids stores to temporaries altogether by keeping all values on the VM’s evaluation stack.

TinyVM is a stack-based embedded systems VM for C [3]. TinyVM’s instruction set closely resembles the instruction set of the bytecode backend of the LCC compiler [6]. TinyVM’s instruction set is given in the appendix. Examples throughout this paper use the TinyVM instruction set.

---

4 Different from the JVM
3 Treegraph IR and Treegraph Scheduling

Our optimization for the stack allocation problem is a local optimization, i.e., it is restricted to single basic blocks (BBs), where a BB is a sequence of statements with a single entry and a single exit [2].

![Dependence-graph and instruction schedules for assignment statement](image)

Fig. 1. Dependence-graph and instruction schedules for assignment statement \( x = 5 \times x \)

An acyclic dependence graph \( G(V, E, L, a) \) consists of a set of vertices \( V \) that represent instances of operations, and a set of edges \( E \subseteq V \times V \) representing data dependencies among operations in \( V \). Each each operation \( o \in O \) has an arity \( a : O \to \mathbb{N}_0 \). The mapping \( L : V \to O \) assigns an operation to a vertex. In abuse of notation, we will refer to node \( u \) and \( L(u) \) synonymously. A data dependency \( (u, v) \in E \) is defined as a dependency between operation \( u \) and \( v \) such that operation \( u \) depends on operation \( v \).

The set of successors \( S_u = \{w : (u, w) \in E\} \) of vertex \( u \) constitutes the operands of \( u \), and they are totally ordered by relation \( \prec_u \subseteq S_u \times S_u \), i.e., \( v_1 \prec_u \ldots \prec_u v_k \) for successors \( \{v_1, \ldots, v_k\} \). Note that the total order is given by the expected order of the operands on the stack. A dependence graph \( V \) is well-formed if \( |S_u| = a(L(u)) \), for all \( u \in V \). The set \( P_u \) that denotes the predecessors of vertex \( u \) is defined similarly. We refer to operations \( u \) whose set of predecessors \( P_u \) is empty as result nodes of \( G \). Figure 1(a) depicts a dependence graph for assignment statement \( x = 5 \times x \).

A stack machine can execute a sequence of instructions \( \langle i_1, \ldots, i_l \rangle \) where an instruction belongs to one of the following instruction classes:

- **Instruction** \( op_o \) is an instruction that performs operation \( o \in O \) and has the signature \( E^{a(o)} \to E \). The instruction pops \( a(o) \) elements from the stack, generates one element as result, and pushes the result on top of the stack. If there are less than \( a(o) \) elements on the stack, the stack machine will go into the error state and will terminate the execution.
- **Instruction** \( DUP \ k \) duplicates the top of stack element \( k \) times. Omitting argument \( k \) is equivalent to \( DUP \ 1 \). If there are no elements on the stack, the

---

\[^5\text{The arity is the number of operands of an operation.}\]
stack machine will go into the error state and will terminate the execution.

- Instruction FETCH k duplicates element k (counted from the top) and pushes
  the duplicate onto the TOS. The stack size is incremented by one. If there
  are less than k elements on the stack, the machine will go into the error
  state and will terminate the execution.

A sequence of instructions \(\langle i_1, \ldots, i_l \rangle\) is admissible, if for an empty stack none
of the statements make the machine go into the error state and terminate the
execution. Graph \(G\) is computed by the sequence of instructions \(\langle i_1, \ldots, i_l \rangle\)
iff the resulting stack contains the correct result.

**Problem statement**

**Input:** An acyclic dependence graph \(G(V, E, L, a)\).

**Output:** code for a stack machine that performs the computations of \(G\)
at minimum cost, i.e., with the minimum number of DUP and FETCH stack
reordering instructions.

### 3.1 Special Case: \(G\) is a Tree

If \(G\) forms a tree, there exists only a single result node \(r \in V\) which represents
the root node of the tree. We can perform a depth-first search (DFS) as
shown in Algorithm 1 where \(\text{operand}(u, i)\) gives the \(i\)-th element of the total
order \(\prec_u\).

**Algorithm 1:** DFS \((G, u)\)

1. foreach \(i \in \{1, \ldots, |S_u|\} \) do
2. \(\quad\) DFS \((G, \text{operand}(u, i))\)
3. emit \(\text{op}_{L(u)}\)

**Proposition 3.1** Sequence DFS \((G, r)\) is admissible and code optimal.

**Proof.** The sequence is optimal because for each vertex a single instruction
is emitted and this is a lower bound on the number of computations. The
generated sequence is admissible and correct: This can be shown by structural
induction, i.e., for each sub-tree it is shown that it is admissible and correct.
The induction start are leaves of the tree.

Note that the DFS sequence is unique in the absence of commutative op-
erators. For a commutative operator like addition or multiplication, we get
two unique trees, depending on which subtree of the commutative operator is
traversed first by Algorithm 1.

**Theorem 3.2** No other instruction sequence for \(G\) is code optimal.

**Proof.** The other sequence is either longer or destroys correctness.
3.2 Code Generation for DAGs

If \( G \) is a DAG, there are no cycles but there exists a vertex \( v \) that has more than one predecessor in \( G \).

**Definition 3.3** Cut set \( C \subseteq E \) is the set of all in-coming edges of nodes \( v \) which have more than one predecessor, i.e., \( C = \{ (u, v) \in E : |P_v| > 1 \} \).

**Lemma 3.4** Graph \( G(V, E - C) \) is a forest \( F \).

**Proof.** By definition every node in a forest has at most one predecessor. Hence, the lemma follows.

The resulting forest is not well-formed in the sense that some of the operations depend on computations that are not performed inside their tree. To generate code for a DAG, we find the roots \( r_1, \ldots, r_m \) for all trees \( T_1, \ldots, T_m \) in \( F \), i.e., this is the set of all nodes which have more than one predecessor in \( G \) or have no predecessors in \( G \). We construct a tree-graph \( H(F, A) \) whose set of vertices \( F = \{ T_1, \ldots, T_m \} \) are the trees of the forest and whose arcs \( A \subseteq F \times F \) denote data dependencies between the trees, i.e., \((T_i, T_j) \in A\) iff there exists an edge \((u, v) \in E\) such that \( u \in T_i \) and \( v \in T_j \).

**Lemma 3.5** The tree-graph is a DAG.

**Proof.** The properties of DAG \( G \) can only be destroyed by adding an edge that introduces a cycle. Condensating DAG \( G \) to a tree-graph \( H(F, A) \) does not introduce additional edges, because every tree of forest \( F \) becomes a single node representing the root of the tree. The edges of the tree-graph \( A \) correspond to the cut-set \( C \).

For a tree \( T_i \in F \) the set of successors are denoted by \( \tilde{S}_{T_i} \). The successors are ordered by the depth-first-search order of \( T_i \) considering the root nodes of the successors in the order as well.

**Lemma 3.6** All trees \( T_i \in F \) have either no predecessor or more than one predecessor in \( H \).

**Proof.** By construction of \( H \).

Algorithm 2 generates code for a tree-graph \( H \) given a topological order \( R \) of \( H \). The treegraph is traversed in reverse topological order. For every treegraph node \( T_i \), the operands are fetched onto the TOS in reverse DFS order (lines 2–4)\(^6\). Then code for the tree represented by \( T_i \) is generated. The tree represented by \( T_i \) is a sub-graph of \( G \) and hence DFS gives the

---

\(^6\) Fetching a value onto the TOS does not increase stack space requirements compared to storing values in temporaries, because at any one time at most one copy of a stack value is fetched to the top. Misplaced operands are popped after a treegraph has been executed, using a \texttt{POP k} instruction provided for this purpose.

---
Algorithm 2: Treegraph scheduler

1. `foreach` $T_i \in F$ in reverse topological order $R$ of $H$ do
2. `foreach` $T_j \in \tilde{S}_{T_i}$ in reverse DFS order of $\tilde{S}_{T_i}$ do
3. `if` result value of $T_j$ not in correct stack slot `then`
4. `emit` FETCH $x_j$
5. `DFS` ($G, r_i$);
6. `if` $|P_{r_i}| > 0$ `then`
7. $N = \text{Oracle} (T_i, H, R) - 1$;
8. `if` $N > 0$ `then`
9. `emit` DUP $N$

optimal code (line 5). Note that execution of the code generated from $T_i$ will produce exactly one value $\nu$ on the stack unless $T_i$ is a result node. If node $T_i$ has predecessors (i.e., it is not a result node), then $P_{r_i} > 1$, i.e., node $T_i$ has at least 2 predecessors (by the definition of tregraphs).

For all but the last predecessor $v$ for which the value $\nu$ computed by $T_i$ is in the correct stack slot, we duplicate $\nu$ (line 9). The last predecessor will consume $\nu$ itself. All other predecessors will have to fetch a copy of $\nu$ to the TOS before they are scheduled (line 4 for each of those predecessors). Clearly, the number of predecessors for which the value $\nu$ is in the correct stack slot depends on (1) the treegraph $H$, (2) the topological order $R$ of $H$, and (3) the treenode $T_i \in H$. Algorithm 2 receives this number via an oracle (line 7). In Section 4, we will discuss our realization of this oracle via a symbolic execution of the stack state for given $T_i, H, R$.

As an example, consider the dependence graph of Figure 1(a). The corresponding treegraph consists of two nodes, one representing ADDR x, and one representing the remaining dependence graph nodes. The reverse topological sorting for this example schedules ADDR x first. The value $\nu$ produced by this treegraph node constitutes the memory address of global variable x (see the instruction set specification of TinyVM in the appendix). Value $\nu$ is already in the correct place for both uses of $\nu$, so we duplicate once. The resulting TinyVM bytecode is depicted in Figure 1(d).

Figures 1(b) shows the bytecode where the multiply-used value is recomputed, and Figure 1(c) shows the approach where the multiply-used value is stored/loaded from a temporary variable t0 (this is the approach chosen e.g., by the LCC compiler).

Theorem 3.7 There exists a topological sort order $R^*$ of $H$, that generates an admissible and code optimal instruction sequence.

Proof. Local property: trees can only be generated code optimal by DFS,
i.e., for any re-ordering of instructions inside trees we can find an instruction sequence that does better (the DFS search), in which case the solution can be improved locally. \hfill \Box

Optimality will be achieved by a topological sort order where the number of values computed in the correct stack slot will be maximized. Section 4 will present an enumeration algorithm over all topological sortings of a treegraph.

4 Minimizing Stack Manipulation Costs

To enumerate all topological sorts of a given basic block’s treegraph $H$, we use Knuth and Szwarcfiter’s algorithm from [7]. For each topological sort, Algorithm 3 is invoked. This algorithm computes the number of DUP and FETCH instructions induced by a topological sort. The sum of the number of DUP and FETCH instructions (line 31) constitutes the cost of a topological sort. The second piece of information computed by Algorithm 3 is the oracle that tells for each treegraph node how many times the value $\nu$ computed by this node must be duplicated on the stack (the oracle was introduced with Algorithm 2).

Algorithm 3 maintains the state of the stack in variable Stack. When a treegraph node is about to get scheduled, the stack is consulted to check the positions of the treegraph’s operands. We distinguish operations with one and two operands. Operands that are on the TOS are consumed (lines 4–5 for the two-operand case, and lines 6 and 10 for the one-operand case. For each operand not in the required position on the TOS we increase the overall counter for the required number of FETCH instructions by one (lines 7, 12 and 21).

To maintain the oracle-function that tells the number of times a treegraph node’s result $\nu$ needs to be duplicated on the stack, the algorithm optimistically pushes one copy of $\nu$ on the stack for each predecessor. This number is saved with the oracle (lines 24–27). Every time we encounter an operand value which is not in the correct stack position, we decrement this operand’s duplication counter in the oracle (lines 9, 15–16 and 23). The superfluous copy is then removed from the stack (lines 8, 13–14 and 22).

The topological sort $R$ of minimum cost and the associated oracle are then used in Algorithm 2 to schedule treegraph nodes and emit bytecode.

We used a timeout to stop enumeration for basic blocks with too many topological sorts. For those cases the solution of minimum cost seen so far was used.
Algorithm 3: computeBasicBlockCosts

Input: toposort sequence Seq of treegraph nodes
Output: cost of Seq, Oracle[...] for number of dupped values

Oracle[...] ← {}; Dup ← 0; Fetch ← 0; Stack ← empty;
foreach treegraph node SU in reverse Seq do
    if NumberOfOperands(SU) = 2 then
        if Stack.top() = SU.op[1] and Stack.second() = SU.op[0] then
            Stack.pop(); Stack.pop();
            Fetch = Fetch + 1;
            Oracle[SU.op[1]] −−;
            Stack.pop();
        else if Stack.top() = SU.op[0] then
            Fetch = Fetch + 1;
            Stack.eraseOne(SU.op[1]);
            Oracle[SU.op[1]] −−;
        else
            Fetch = Fetch + 2;
            Stack.eraseOne(SU.op[0]);
            Oracle[SU.op[0]] −−;
            Oracle[SU.op[1]] −−;
    else if NumberOfOperands(SU) = 1 then
        if Stack.top() = SU.op[0] then
            Stack.pop();
        else
            Fetch = Fetch + 1;
            Stack.eraseOne(SU.op[0]);
            Oracle[SU.op[0]] −−;
    else if SU computes result ν then
        for i to SU.NumberOfPreds do
            Stack.push(SU)
        Oracle[SU] = SU.NumberOfPreds;
    end if
end foreach

if Oracle[SU] > 1 then
    Dup = Dup + 1;
end if
return Dup + Fetch, Oracle[...]

5 Experimental Results

Figure 2 shows our experimental setup. C source code was compiled to LLVM IR by llvm-gcc ("Frontend-end"). LLVM’s optimizing middle-end was by-
passed for this experiment. We implemented an LLVM backend for the generation of TinyVM bytecode from LLVM IR, based on LLVM version 2.5. The generated bytecode was then benchmarked on TinyVM.

Figure 3 depicts the structure of our TinyVM bytecode backend. We implemented the "Instruction Selection" pass to lower LLVM IR to TinyVM bytecode instructions\(^7\). These bytecode instructions use pseudo-registers to accommodate SSA virtual registers. At this stage, the IR consists of DAGs where each DAG-node corresponds to one scheduable unit of bytecode instructions (i.e., a sequence of instructions that should be scheduled together). "Instruction Scheduling" denotes our treegraph instruction scheduler. For each basic block DAG we create the corresponding treegraph and run our treegraph scheduling algorithm. We use LLVM's 'local' register allocator to ensure that there are no live ranges between basic blocks. (Our optimization works on a per basic block basis; we did not consider global stack allocation yet.) Our "Stackification" pass converts pseudoregister code to stack code. This is a straight-forward conversion where operands that correspond to stack slots are dropped. For example, ADDRG R1 x would become ADDRG x. For further details we refer to [16].

All experiments were performed on an Intel Xeon 5120 server running Linux CentOS 5.4 with kernel version 2.6.18. We selected 24 C benchmark programs from the testsuite that comes with the LLVM compiler infrastruct-

\(^7\) “Lowering” denotes the change of abstraction-level by converting LLVM IR to TinyVM bytecode.
Figure 4 contains the size reductions and the speedsups obtained for our benchmarks programs. Size reductions where computed as $1 - \frac{\text{newsize}}{\text{oldsize}}$. As a yardstick for our comparison, we implemented the store/load mechanism to temporaries for our LLVM backend (i.e., every multiply-used value is written to a temporary and loaded from there when the value is required). Figure 4 depicts the improvements of our store/load mechanism over the LCC bytecode. It should be noted that LCC also applies store/load of temporaries and that the improvements are largely due to the superior code quality achiev-
able with LLVM. Note also that benchmarks with 0 improvement for ST/LD denote cases where a benchmark was not ANSI C and thus not compilable by LCC. The “Treegraph scheduling” data in Figure 4 denotes the improvement of our treegraph scheduling technique over ST/LD.

For 93% of all basic blocks our treegraph scheduler could derive the optimal solution, for the remaining 7% the enumeration of topological sorts hit the 2 second timeout. For 86% of basic blocks the solve-time was below 0.08 seconds.

6 Conclusions

In this paper we investigated how the semantics of a register-based IR can be mapped to stack-code. We introduced a novel program representation called treegraphs. Treegraph nodes encapsulate computations that can be represented by DFS trees. Treegraph edges manifest computations with multiple uses. Instead of saving a multiply-used value in a temporary, our method keeps all values on the stack, which avoids costly store and load instructions. Values that are in the correct stack slot for (some of) their users are duplicated so that they can be consumed without stack manipulation. All other values are lifted to the top of stack via a FETCH instruction. We implemented our treegraph scheduler with the LLVM compiler infrastructure for TinyVM, a stack-based embedded systems VM for C.

References


### A TinyVM Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IS-Op.</th>
<th>Suffixes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD SUB</td>
<td>—</td>
<td>FIUP...</td>
<td>integer addition, subtraction</td>
</tr>
<tr>
<td>MUL DIV</td>
<td>—</td>
<td>FIU...</td>
<td>integer multiplication, division</td>
</tr>
<tr>
<td>NEG</td>
<td>—</td>
<td>FI......</td>
<td>negation</td>
</tr>
<tr>
<td>BAND BOR BXOR</td>
<td>—</td>
<td>.IU....</td>
<td>bitwise and, or, xor</td>
</tr>
<tr>
<td>BCOM</td>
<td>—</td>
<td>.IU......</td>
<td>bitwise complement</td>
</tr>
<tr>
<td>LSH RSH MOD</td>
<td>—</td>
<td>.IU.....</td>
<td>bit shifts and remainder</td>
</tr>
<tr>
<td>CNST</td>
<td>a</td>
<td>.IUP...</td>
<td>push literal a</td>
</tr>
<tr>
<td>ADDRG</td>
<td>p</td>
<td>...P...</td>
<td>push address p of global</td>
</tr>
<tr>
<td>ADDRF</td>
<td>l</td>
<td>...P...</td>
<td>push address of formal parameter, offset l</td>
</tr>
<tr>
<td>ADDRL</td>
<td>l</td>
<td>...P...</td>
<td>push address of local variable, offset l</td>
</tr>
<tr>
<td>BADDRG</td>
<td>index</td>
<td>...P...</td>
<td>push address of mc entity at index</td>
</tr>
<tr>
<td>INDIR</td>
<td>—</td>
<td>FIUP...</td>
<td>pop p; push ∗p</td>
</tr>
<tr>
<td>ASGN</td>
<td>—</td>
<td>FIUP...</td>
<td>pop arg; pop p; ∗p = arg</td>
</tr>
<tr>
<td>ASGN_B</td>
<td>a</td>
<td>.....B...</td>
<td>pop q, pop p; copy the block of length a at ∗q to p</td>
</tr>
<tr>
<td>CVI</td>
<td>—</td>
<td>FIU...</td>
<td>convert from signed integer</td>
</tr>
<tr>
<td>CVU</td>
<td>—</td>
<td>.IUP...</td>
<td>convert from unsigned integer</td>
</tr>
<tr>
<td>CVF</td>
<td>—</td>
<td>FI......</td>
<td>convert from float</td>
</tr>
<tr>
<td>CVP</td>
<td>—</td>
<td>..U....</td>
<td>convert from pointer</td>
</tr>
<tr>
<td>LABEL</td>
<td>—</td>
<td>....V..</td>
<td>label definition</td>
</tr>
<tr>
<td>JUMP</td>
<td>target</td>
<td>....V...</td>
<td>unconditional jump to target</td>
</tr>
<tr>
<td>IJUMP</td>
<td>—</td>
<td>....V...</td>
<td>indirect jump</td>
</tr>
<tr>
<td>EQ GE GT LE LT NE</td>
<td>target</td>
<td>FIU...</td>
<td>compare and jump to target</td>
</tr>
<tr>
<td>ARG</td>
<td>—</td>
<td>FIUP...</td>
<td>top of stack is next outgoing argument</td>
</tr>
<tr>
<td>CALL</td>
<td>target</td>
<td>....V...</td>
<td>vm procedure call to target</td>
</tr>
<tr>
<td>ICALL</td>
<td>—</td>
<td>....V...</td>
<td>pop p; call procedure at p</td>
</tr>
<tr>
<td>INIT</td>
<td>l</td>
<td>....V...</td>
<td>allocate l stack cells for local variables</td>
</tr>
<tr>
<td>BCALL</td>
<td>—</td>
<td>FIUPVb</td>
<td>mc procedure call</td>
</tr>
<tr>
<td>RET</td>
<td>—</td>
<td>FIUPVb</td>
<td>return from procedure call</td>
</tr>
<tr>
<td>HALT</td>
<td>—</td>
<td>....V...</td>
<td>exit the vm interpreter</td>
</tr>
<tr>
<td>POP</td>
<td>k</td>
<td>....V...</td>
<td>pop k values from the TOS</td>
</tr>
</tbody>
</table>

Table A.1
TinyVM bytecode instruction set

Table A.1 depicts the TinyVM bytecode instruction set. The TinyVM instruction set is closely related to the bytecode interface that comes with LCC [6]. Instruction opcodes cover the leftmost column whereas the column headed “IS-Op.” lists operands derived from the instruction stream (all other instruction operands come from the stack). The column entitled “Suffixes” denotes the valid type suffixes for an operand (F=float, I=signed
integer, U=unsigned integer, P=pointer, V=void, B=struct). In this way instruction ADDR receives its pointer argument p from the instruction stream and pushes it onto the stack. Instructions ADDR and ADDRL receive an integer argument literal from the instruction stream; this literal is then used as an offset to the stack frame pointer to compute the address of a formal or local variable. Unlike the JVM, TinyVM uses an ADDR* / INDIR instruction sequence to load a value onto the stack. To store a value, TinyVM uses the ASGN instruction.

---

8 Operators contain byte size modifiers (i.e., 1, 2, 4, 8), which we have omitted for reasons of brevity.