Practical Implementation of ωDDPAc

ACHINTYA GOPAL, The Johns Hopkins University

ACM Reference format:
Achintya Gopal. 2018. Practical Implementation of ωDDPAc. 1, 1, Article 1 (May 2018), 17 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the [1] paper, to prove the soundness of DDPA in a fashion common to other higher order program analyses, a non-standard operational semantics, termed ωDDPAc, is introduced which is proven to be equivalent to a standard operational semantics. ωDDPAc is shown to be equivalent to closure-based operational semantics which is well-known to be equivalent to call-by-value λ-calculus.

ωDDPAc is a graph-based operational semantics which represents expressions as concrete (runtime) control flow graphs. ωDDPAc is an interesting presentation of operational semantics as it lacks closures, substitution, fresh variables, or an environment, and reductions are all polynomially bounded in length.

The primary goal of this paper is to show the viability of ωDDPAc by extending the operational semantics to incorporate integers, booleans, and conditional clauses, as well as incorporate other modifications to the operational semantics of ωDDPAc to improve the run-time of the algorithm which retaining the core concept behind ωDDPAc.

In the theoretical presentation of the ωDDPAc rule and its modifications, the paper focuses on pure λ-calculus to not get overwhelmed by details. In Section 2, the operational semantics of ωDDPAc is given and the modifications of the operational semantics are defined. In Section 3, implementation specific modifications to ωDDPAc operational semantics are defined. In Section 4, the paper outlines how additional language features may be incorporated. In Section 5, the paper outlines how computation with unsigned integers would be made more lazy. In Section 6, the paper compares the runtime of multiple different experiments. The paper discusses related work in Section 7 and conclude in Section 8.

2 A GRAPH-BASED OPERATIONAL SEMANTICS

2.1 ωDDPAc

The original presentation of ωDDPAc uses a simple functional language defined in Figure 1. It is a call-by-value λ-calculus. It is required that programs are closed, and that variable identifiers are unique ("alphatized"). ωDDPAc uses an A-normal form (ANF) [4] intermediate representation, so a clause c denotes a program point.

$$\begin{array}{lllr}
e & ::= & [c, \ldots] & \textit{expressions} \\
c & ::= & x = b & \textit{clauses} \\
b & ::= & v \mid x \mid x\ x & \textit{clause bodies}
\end{array}
\qquad
\begin{array}{lllr}
v & ::= & f & \textit{values} \\
f & ::= & \texttt{fun}\ x\ \texttt{->}\ (\ e\ ) & \textit{functions} \\
x & & & \textit{variables}
\end{array}$$

Fig. 1. Expression Grammar

$$\begin{array}{lllr}
a & ::= & c \mid x \overset{(\!|c}{=} x \mid x \overset{|\!)c}{=} x \mid & \textit{annotated clauses} \\
 & & \textsc{Start} \mid \textsc{End} &
\end{array}
\qquad
\begin{array}{lllr}
g & ::= & a \lll a & \textit{concrete control flow edges} \\
G & ::= & \{g, \ldots\} & \textit{concrete control flow graphs} \\
C & ::= & [c, \ldots] & \textit{clause stacks} \\
\mathbf{C} & ::= & \{C, \ldots\} & \textit{clause stack sets}
\end{array}$$

Fig. 2. ωDDPAc Evaluation Grammar

The calling contexts in ωDDPAc are fixed to be the full call stack without approximation (the ω in the name), and the wiring rule takes the current context into account (the additional "c" at the end of ωDDPAc).

The definition of the grammar for ωDDPAc is given in Figure 2.

The nodes of the graph are individual program clauses and the special nodes Start and End, representing the start and end of the overall program. Edges in the graph will represent control flow which occurs *at least once* during execution. A consequence of this model is that, for any program, the set of possible control flow decisions (here, graph edges) is finite and monotonically increasing. However, the number of program states (here, paths in the graph) may not be. As a result, a particular node in the graph is insufficient to describe the current state of execution: we must also be able to identify how we reached that point. This is achieved by the use of a stack of clauses C which for the simple language in this section, C corresponds to the runtime call stack of the program.

To define ωDDPAc, a lazy lookup function is required. In this case, the graph affords the ability to skip over out-of-scope bindings without an integer counter by relying on a wiring process. In the original set of rules, a set of values is returned; however, in [1], it was proven that the set will never be of size more than 1. In the following presentation of those rules, the value set is replaced with simply a value. This leads to the definition:

Definition 2.1. Given control flow graph G, $G(X, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \lll a_0$:

(1) $\boxed{\textsc{Value Discovery}}$
If $a_1 = (x = v)$ and $X = [x]$, then v.

(2) $\boxed{\textsc{Value Discard}}$
If $a_1 = (x_1 = \hat{f})$ and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $G([x_2, \ldots, x_n], a_1, C)$.

(3) $\boxed{\textsc{Alias}}$
If $a_1 = (x = x')$ and $X = [x] \,||\, X'$ then $G([x'] \,||\, X', a_1, C)$.

(4) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (x \overset{(\!|c}{=} x')$, $X = [x] \,||\, X'$, and $C = [c] \,||\, C'$, then $G([x'] \,||\, X', a_1, C')$.

(5) $\boxed{\textsc{Function Enter Non-Local}}$
If $a_1 = (x'' \overset{(\!|c}{=} x')$, $X = [x] \,||\, X'$, $x'' \neq x$, $c = (x_r = x_f\ x_v)$, and $C = [c] \,||\, C'$, then $G([x_f, x] \,||\, X', a_1, C')$.

$$\dfrac{\begin{array}{l}\textsc{Application}\\ c = (x_1 = x_2\ x_3) \qquad C \in \textsc{Active}(c, G) \qquad f = G([x_2], c, C) \qquad v = G([x_3], c, C)\end{array}}{G \longrightarrow^1 G \cup \textsc{Wire}(c, f, x_3, x_1)}$$

Fig. 3. The ωDDPAc Operational Semantics

(6) $\boxed{\textsc{Function Exit}}$

If $a_1 = (x \overset{)c}{=} x')$, $X = [x] \,||\, X'$, and $c = (x_r = x_f\ x_v)$, then $G([x'] \,||\, X', a_1, [c] \,||\, C)$, provided `fun` x'' `->` `(` e `)` $= G([x_f], c, C)$ and $x' = \text{RV}(e)$.

(7) $\boxed{\textsc{Skip}}$

If $a_1 = (x'' = b)$, $X = [x] \,||\, X'$, and $x'' \neq x$, then $G(X, a_1, C)$.

Given the above lookup function, we define ωDDPAc with an incremental construction of the graph structure based upon the values discovered by demand-driven lookup. See Figure 3 for the (sole) rule. The precise definition of $\textsc{Active}$ is as follows.

Definition 2.2. Let $\textsc{Active}(G, a)$ be least set $\boldsymbol{C}$ conforming to the following conditions:

- If $c \ll a$ then $\textsc{Active}(G, c) \subseteq \boldsymbol{C}$.
- If $a' \ll a$ for $a' = (x \overset{(c}{=} x')$ and $C \in \textsc{Active}(G, a')$, then $(C \,||\, [c]) \in \boldsymbol{C}$.
- If $\textsc{Start} \ll a$ then $[] \in \boldsymbol{C}$.

Note that $\boldsymbol{C}$ may not be finite.

$\textsc{Wire}$ is defined as follows:

Definition 2.3. Let $\textsc{Wire}(c', \texttt{fun}\ x_0\ \texttt{->}\ \texttt{(}\,[c_1, \ldots, c_n]\,\texttt{)}, x_1, x_2) =$
$\textsc{Preds}(c') \ll (x_0 \overset{(c'}{=} x_1) \ll c_1 \ll \ldots \ll c_n \ll (x_2 \overset{)c'}{=} \text{RV}([c_n])) \ll \textsc{Succs}(c')$
where $\textsc{Preds}(a) = \{a' \mid a' \ll a\}$ and $\textsc{Succs}(a) = \{a' \mid a \ll a'\}$.

c' here is the call site, and $c_1 \ll \ldots \ll c_n$ is the wiring of the function body. The $\textsc{Preds}$/$\textsc{Succs}$ functions reflect how we simply wire to the existing predecessor(s) and successor(s).

2.2 Lazy ωDDPAc

Given the above definitions, ωDDPAc easily lends itself to lazy evaluation, replacing a call-by-value operational semantics to call-by-name operational semantics. Whereas in the previous section, the rules require wiring the graph completely, the following rules can remove the necessity of this and only wire in functions as need be when evaluating the program lazily.

Whenever the Function Exit rule is used, the value of x_f is looked up. It can be seen that the ωDDPAc Operational Semantics wiring the graph will have done this exact lookup. So, instead of wiring in the function during the construction of the graph, the function can be wired in a separate lookup rule that replaces the Function Exit rule. Because the graph is wired up during the lookup rules, the graph is initialized such that the top level of the program is wired in when the lookup starts. Along with this, we can remove the value lookup to make the language call-by-name. From this point on, all the lookup rules will be lazy and will use call-by-name semantics.

Definition 2.4. Given control flow graph G, $G(X, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$ (where rules (1) - (5) and (7) are the same as in Definition 2.1):

a	$::=$	$c \mid \text{◖}\, x \mid \text{◗}\, x \mid$ START $\mid$ END	*annotated clauses*
g	$::=$	$a \ll a$	*concrete control flow edges*
G	$::=$	$\{g, \ldots\}$	*concrete control flow graphs*
C	$::=$	$[c, \ldots]$	*clause stacks*
$\boldsymbol{C}$	$::=$	$\{C, \ldots\}$	*clause stack sets*

Fig. 4. Lazy ωDDPAc Evaluation Grammar

(6) $\boxed{\text{Application Clause}}$
If $a_1 = (x_c = x_f\ x_v)$, $X = [x_c] \,||\, X'$, and $\texttt{fun}\ x \texttt{ -> (} e \texttt{)} = G([x_f], a_1, C)$, then $(G \cup \textsc{Wire}(a_1, \texttt{fun}\ x \texttt{ -> (} e \texttt{)}, x_v, x_c))([x'] \,||\, X', (x_c \overset{\text{◗} a_1}{=} x'), [a_1] \,||\, C)$ where $x' = \text{RV}(e)$.

With the modification of the Function Exit Rule, the wiring definition also changes to:

Definition 2.5. Let $\textsc{Wire}(c', \texttt{fun}\ x_0 \texttt{ -> ([} c_1, \ldots, c_n \texttt{])}, x_1, x_2) = c' \ll (x_0 \overset{\text{◖} c'}{=} x_1) \ll c_1 \ll \ldots \ll c_n \ll (x_2 \overset{\text{◗} c'}{=} \text{RV}([c_n]))$

2.3 Simplifying the Graph

Given the current lookup, finding the correct Function Enter Node is difficult as c_1 can have multiple predecessors of the form $(x_0 \overset{\text{◖} c'}{=} x_1)$. However, the information about the clause c' is not needed given that in the lookup rules, c' is used to align with the context stack, and x_1 is simply x_v which is information that is at the top of the context stack. So to further simplify the graph nodes, we remove all but the formal parameter from the Function Enter Node.

Along with this, the information in the Exit Clause is not needed as all we need is a graph node that is right after the return clause since by the definition of the lookup, we always look at the node before the current a_0.

Given the description above, the grammar for the graph changes which is described in Figure 4, as well as the lookup rules and the definition of wiring given below.

Definition 2.6. Given control flow graph G, $G(X, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$ (where rules (1) - (3) and (7) are the same as in Definition 2.1):

(4) $\boxed{\text{Function Enter Parameter}}$
If $a_1 = (\text{◖}\, x)$, $X = [x] \,||\, X'$, $C = [c] \,||\, C'$, and $c = (x_r = x_f\ x_v)$, then $G([x_v] \,||\, X', c, C')$.

(5) $\boxed{\text{Function Enter Non-Local}}$
If $a_1 = (\text{◖}\, x')$, $X = [x] \,||\, X'$, $x' \neq x$, $C = [c] \,||\, C'$, and $c = (x_r = x_f\ x_v)$, then $G([x_f, x] \,||\, X', c, C')$.

(6) $\boxed{\text{Application Clause}}$
If $a_1 = (x_c = x_f\ x_v)$, $X = [x_c] \,||\, X'$, $\texttt{fun}\ x \texttt{ -> (} e \texttt{)} = G([x_f], a_1, C)$, and $e = \texttt{([} c_1, \ldots, c_n \texttt{])}$, then $(G \cup \textsc{Wire}(x, e))([x'] \,||\, X', (\text{◗}\, x), [a_1] \,||\, C)$ where $x' = \text{RV}(e)$.

Definition 2.7. Let $\textsc{Wire}(x_0, [c_1, \ldots, c_n]) = (\text{◖}\, x_0) \ll c_1 \ll \ldots \ll c_n \ll (\text{◗}\, x_0)$

Given the above changes to lazy evaluation of ωDDPAc, one interesting side effect of these changes is that every node only has one predecessor. In comparison to the original analysis of ωDDPAc, application clauses are traversed to decide which path in the graph to take instead of originally taking paths in the graph to avoid the application clauses.

$$\frac{c = (x = \texttt{fun}\ x_0 \texttt{ -> } e)}{G \longrightarrow^1 G \cup \textsc{Wire}(x_0, e)}$$

Fig. 5. Modified ωDDPAc Operational Semantics

$$\begin{array}{rcll}
l & ::= & (v, a, C) & \textit{lookup value} \\
a & ::= & c \mid ◖\, x \mid ◗\, x \mid \textsc{Start} \mid \textsc{End} & \textit{annotated clauses}
\end{array}
\qquad
\begin{array}{rcll}
g & ::= & a \ll a & \textit{concrete control flow edges} \\
G & ::= & \{g, \ldots\} & \textit{concrete control flow graphs} \\
s & ::= & (c, l) & \textit{context} \\
C & ::= & [s, \ldots] & \textit{context stacks} \\
\mathbb{C} & ::= & \{C, \ldots\} & \textit{context stack sets}
\end{array}$$

Fig. 6. Improving Non Local Lookup ωDDPAc Evaluation Grammar

Another side effect of these changes is that since the only clauses wired together are those in the same context, the graph can be constructed before running the lookup. We modify the definition of a small step in Figure 5. The graph is initialized such that for every clause at the top level of the program is wired together.

Definition 2.8. Given control flow graph G, $G(X, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$ (where rules (1) - (5) and (7) are the same as in Definition 2.6):

(6) $\boxed{\textsc{Application Clause}}$
If $a_1 = (x_c = x_f\ x_v)$, $X = [x_c] \,||\, X'$, $\texttt{fun}\ x \texttt{ -> (} e \texttt{)} = G([x_f], a_1, C)$, and $e = \texttt{(}\,[c_1, \ldots, c_n]\,\texttt{)}$, then $G([x'] \,||\, X', (◗\, x), [a_1] \,||\, C)$ where $x' = \textsc{rv}(e)$.

2.4 Improving non-local lookups

An observation that can be made with the definition of lookup rules for lazy evaluation of ωDDPAc is that the Function Enter Non-Local Rule looks up a function which has already been looked up in the Application Clause Rule. So, to reduce the amount of redundant lookups, we can modify the context stack to contain the information about the function lookup so that the Function Enter Non-Local Rule does not have to repeat this lookup.

However, one difficulty that arises in trying to save the function lookup done in the Application Clause Rule is that when we continue the variable lookup, the context stack and node in the graph will have changed. To accomodate this, the returned value type must be modified to keep track of the context stack and node at the point the value was found and the Context Stack must also be modified to allow the saving of the function lookup.

Along with this change, it can be seen the only time the lookup stack is utilized is for continuing a lookup after doing a non-local lookup. So, by storing the information of the non-local lookup on the context stack, the lookup stack will only contain one variable at any point in time and hence can be replaced with simply looking up a variable x. These changes will basically cache the information where the next set of non-locals are defined.

The new grammar rules with the new lookup value type and context stack modified are defined in Figure 6.

To accomodate these changes, all the rules are modified.

Definition 2.9. Given control flow graph G, $G(x, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$:

$$\begin{array}{rcll} l & ::= & (v, a, C) & \textit{lookup value} \\ a & ::= & c \mid \lhd x \mid \rhd x \mid & \textit{annotated clauses} \\ & & \textsc{Start} \mid \textsc{End} & \end{array} \qquad \begin{array}{rcll} g & ::= & a \ll a & \textit{concrete control flow edges} \\ G & ::= & \{g, \ldots\} & \textit{concrete control flow graphs} \\ s & ::= & (a, c, C) & \textit{context} \\ C & ::= & [s, \ldots] & \textit{context stacks} \\ C & ::= & \{C, \ldots\} & \textit{context stack sets} \end{array}$$

Fig. 7. Transpose Context Stack ωDDPAc Evaluation Grammar

(1) $\boxed{\textsc{Value Discovery}}$
If $a_1 = (x = v)$, then (v, a_1, C).
$\boxed{\textsc{Value Discard}}$
~~If $a_1 = (x_1 = \hat{f})$ and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $G([x_2, \ldots, x_n], a_1, C) \subseteq V$.~~
(2) $\boxed{\textsc{Alias}}$
If $a_1 = (x = x')$ then $G(x', a_1, C)$.
(3) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (\lhd x)$, and $C = [(c, \_)] \,||\, C'$, then $G(x', c, C')$.
(4) $\boxed{\textsc{Function Enter Non-Local}}$
If $a_1 = (\lhd x')$, $x' \neq x$, $C = [(c, l)] \,||\, C'$, and $(\_, a_2, C'') = l$, then $G(x, a_2, C'')$.
(5) $\boxed{\textsc{Application Clause}}$
If $a_1 = (x = x_f\ x_v)$, $l = G(x_f, a_1, C)$, and $(\texttt{fun}\ x'' \texttt{ -> (} e \texttt{)}, \_, \_) = l$, then $G(x', (\rhd x''), [(a_1, l)] \,||\, C)$ where $x' = \text{RV}(e)$.
(6) $\boxed{\textsc{Skip}}$
If $a_1 = (x' = b)$, $x' \neq x$, then $G(x, a_1, C)$.

2.5 Improving local lookups

With the current construction of the context stack, by popping off the stack, we do a 'local traversal' i.e. doing a lookup to find what the value of the passed in value to formal parameter is; by traversing the context stack through the context stack referenced at the top of the original stack, we traverse all the referencable formal parameters and contexts which we can call a 'non local traversal'.

The first observation given the new context stack structure is at a program point, any variable that is reachable without any local traversals is a variable that can be referenced at that program point. This is interesting because if we were to modify the lookup rules to be call-by-value and store the value of x_v on the context stack, the context stack would only need to contain non local traversals. This fact can be utilized by modifying the context stack to save local lookups to save both computation time and space.

Since the eventual goal is to remove local traversals, the context stack will look more like a linked list instead of stack. To 'remedy' this, we can 'transpose' the context stack, i.e. make non-local lookups a stack traversal and local-lookups (function enter local) a linked list traversal. The only change to allow this is to change the lookup rules for Function Enter and Application Clause and the grammar definition to Figure 7.

Definition 2.10. Given control flow graph G, $G(x, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$ (where rules (1), (2) and (6) are the same as in Definition 2.9):

(3) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (\lhd x)$, and $C = [(\_, c, C'')] \,||\, C'$, and $c = (x_r = x_f\ x_v)$, then $G(x_v, c, C'')$.

l	$::=$	(v, a, C)	*lookup value*
a	$::=$	$c \mid \lhd x \mid \rhd x \mid$ START $\mid$ END	*annotated clauses*

g	$::=$	$a \ll a$	*concrete control flow edges*
G	$::=$	$\{g, \ldots\}$	*concrete control flow graphs*
L	$::=$	$(c, C) \mid l$	*local value*
s	$::=$	$(a, \texttt{ref}\ L)$	*context*
C	$::=$	$[s, \ldots]$	*context stacks*
$\mathbf{C}$	$::=$	$\{C, \ldots\}$	*context stack sets*

Fig. 8. Improve Local Lookup ωDDPAc Evaluation Grammar

(4) $\boxed{\text{FUNCTION ENTER NON-LOCAL}}$
If $a_1 = (\lhd x')$, $x' \neq x$, and $C = [(a_2, \_, \_)] \mid\mid C'$, then $G(x, a_2, C')$.

(5) $\boxed{\text{APPLICATION CLAUSE}}$
If $a_1 = (x = x_f\ x_v)$, and $(\texttt{fun}\ x'' \texttt{->}\ (\ e\ ), a_2, C') = G(x_f, a_1, C)$, then $G(x', (\rhd x''), [(a_2, a_1, C)] \mid\mid C')$ where $x' = \text{RV}(e)$.

Now that the context stack has been transposed, the goal is to save local lookups on the context stack. The third value in the object that is pushed to the context stack is only there to allow the lookup rules to do a function enter local lookup. Once the local lookup has been completed, the clause and context stack can be completely replaced with what the lookup rule returns during the local lookup.

To implement this, we simply have to replace the context stack in a context with a 'ref' that either points to a context stack or lookup value. This then modifies the Function Enter Parameter rule as there are now two cases.

The grammar is defined to Figure 8.

Definition 2.11. Given control flow graph G, $G(x, a_0, C)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$ (where rules (1), (2) and (6) are the same as in Definition 2.9):

(3a) $\boxed{\text{FUNCTION ENTER PARAMETER, LOOKUP}}$
If $a_1 = (\lhd x)$, and $C = [(\_, L)] \mid\mid C'$, $L = \texttt{ref}\ (c, C'')$ where C'' is a context stack, and $c = (x_r = x_f\ x_v)$, then $L = \texttt{ref}\ G(x_v, c, C'')$ and $G(x_v, c, C'')$

(3b) $\boxed{\text{FUNCTION ENTER PARAMETER, STORED}}$
If $a_1 = (\lhd x)$, and $C = [(\_, L)] \mid\mid C'$, and $L = \texttt{ref}\ l$ where l is a lookup value , then l.

(4) $\boxed{\text{FUNCTION ENTER NON-LOCAL}}$
If $a_1 = (\lhd x')$, $x' \neq x$, and $C = [(a_2, \_)] \mid\mid C'$, then $G(x, a_2, C')$.

(5) $\boxed{\text{APPLICATION CLAUSE}}$
If $a_1 = (x = x_f\ x_v)$, $(\texttt{fun}\ x'' \texttt{->}\ (\ e\ ), a_2, C') = G(x_f, a_1, C)$, and $e = (\ [c_1, \ldots, c_n]\ )$, then $G(x', (\rhd x''), [(a_2, \texttt{ref}\ (a_1, C))] \mid\mid C')$ where $x' = \text{RV}(e)$.

This set of rules however is not the mathematical way to describe this form of memory. However, these rules give an intuitive way to understand what the following rules do.

To add statefulness to the operational semantics, we have to use a store. The store needs to be threaded through the different evaluations. The usage of stores also modifies the grammar which is defined in Figure 9.

Definition 2.12. Given control flow graph G, $G(x, a_0, C, S)$ is the function returning a value satisfying the following conditions given some $a_1 \ll a_0$:

(1) $\boxed{\text{VALUE DISCOVERY}}$
If $a_1 = (x = v)$, then (v, a_1, C, S).

t			*cell names*	g	$::=$	$a \ll a$	*concrete control flow edges*
S	$::=$	$\{t \mapsto L\}$	*store*	G	$::=$	$\{g, \ldots\}$	*concrete control flow graphs*
l	$::=$	(v, a, C, S)	*lookup value*	L	$::=$	$(c, C) \mid l$	*local value*
a	$::=$	$c \mid \lhd x \mid \rhd x \mid$ START $\mid$ END	*annotated clauses*	s	$::=$	(a, t)	*context*
				C	$::=$	$[s, \ldots]$	*context stacks*
				$\mathbf{C}$	$::=$	$\{C, \ldots\}$	*context stack sets*

Fig. 9. Improve Local Lookup ωDDPAc Evaluation Grammar with Store

(2) $\boxed{\text{Alias}}$
If $a_1 = (x = x')$ then $G(x', a_1, C, S)$.

(3a) $\boxed{\text{Function Enter Parameter, Lookup Local}}$
If $a_1 = (\lhd x)$, and $C = [(\_, t)] \,||\, C'$, $S(t) = (c, C'')$ where C'' is a context stack, $c = (x_r = x_f \; x_v)$, $l = G(x_v, c, C'', S)$ and $(v, a_2, C''', S') = l$, then $S'' = S'\{t \mapsto l\}$ and (v, a_2, C''', S'')

(3b) $\boxed{\text{Function Enter Parameter, Stored Local}}$
If $a_1 = (\lhd x)$, and $C = [(\_, t)] \,||\, C'$, $S(t) = l$ where l is a lookup value , then l.

(4) $\boxed{\text{Function Enter Non-Local}}$
If $a_1 = (\lhd x')$, $x' \neq x$, and $C = [(a_2, \_)] \,||\, C'$, then $G(x, a_2, C', S)$.

(5) $\boxed{\text{Application Clause}}$
If $a_1 = (x = x_f \; x_v)$, and $(\texttt{fun } x'' \texttt{->} (\, e \,), a_2, C', S') = G(x_f, a_1, C, S)$, then $G(x', (\rhd x''), [(a_2, t)] \,||\, C', S'\{t \mapsto (a_1, C)\})$ where $x' = \textsc{rv}(e)$ and $t \notin Dom(S')$.

(6) $\boxed{\text{Skip}}$
If $a_1 = (x'' = b)$, $x'' \neq x$, then $G(x, a_1, C, S)$.

To do a lookup of the return value x of the program defined at clause c_n, the lookup value is $G(x, c_n, [], \{\})$.

3 IMPLEMENTING LOOKUP

We saw in the previous section how lookup was done in the original set of rules for ωDDPAc and how these rules can be modified to remove some redundancies in the graph and remove redundant lookups.

In this section, we discuss how the modified rules allow for modifications to the implementation of the graph and context stack which significantly speed up the interpreter.

3.1 Graph Implementation

In the original graph implementation, both the successors and predecessors had to be stored due to the need to traverse forward to construct the graph and traverse backwards to do the lookup. Along with this, each node in the graph could have multiple successors and multiple predecessors even though the traversal is deterministic, meaning only one of the successors/predecessors was a valid path at any one point in time. By modifying the language to be lazy, the information about successors could be removed.

However, there was still the problem of multiple predecessors. This is solved by the modifications outlined in Section 2.3. The main problem was to notice that the cause of multiple predecessors was that there could be multiple Exit Clauses reachable from an application clause. This was fixed by modifying Enter and Exit Clauses. Due to these modifications, the rules were further updated so that only variables in the same context are wired together. This gives the ability to efficiently implement the graph by using hash tables to define which graph node is before what node.

t			*cell names*
S	$::=$	$\{t \mapsto L\}$	*store*
l	$::=$	(v, C, S)	*lookup value*
a	$::=$	$c \mid ◖\, x \mid$	*annotated clauses*
		$\textsc{Start} \mid \textsc{End}$	

ctx	$::=$	$x \mid None$	*context var*
g	$::=$	$x \rightarrow (\, ctx, c \,)$	*concrete control flow edges*
G	$::=$	$\{g, \ldots\}$	*concrete control flow graphs*
L	$::=$	$(ctx, C) \mid l$	*local value*
s	$::=$	(ctx, t)	*context*
C	$::=$	$[s, \ldots]$	*context stacks*
$\boldsymbol{C}$	$::=$	$\{C, \ldots\}$	*context stack sets*

Fig. 10. Modified ωDDPAc Evaluation Grammar

With this implementation, a large portion of the code is spent on the Skip Rule. To avoid this, we can further modify the graph structure to store not the predecessor but in which context a variable is defined in and the index of said variable. Then, when we are doing the lookup, we can traverse the context stack through valid non local contexts until we see the variable we are looking for. In simpler terms, if we know that the current location is a_1 and that the variable we are looking for is in a_2, if they are in separate contexts, then we will pop from the stack and arrive in a new context; if the two locations are in the same context, then by knowing the index of the two variables in the context, we can easily check if $a_2 \ll a_1$.

3.2 Context Stack Implementation

In [1], along with the definition of the language, the well-formedness of the program is defined rigorously. Using this definition of well-formedness, we do not need to know the index of a variable. The reason for this is we know that any variable referenced at a program point is within scope. So lookup is simpler now since by knowing what context the variable is defined in, we just have to pop through context stack until we find the context. So, we can modify the lookup rule to first modify the context stack to find the correct context and then do the corresponding lookup rule. This modification will remove the Non Local Rules as those are 'refactored' into another rule.

Along with this change, when we push to the context stack, we do not need to know the specific program point anymore and instead need to know which context the program was in, which this information can be stored on the context stack. Because of this, we have to modify the evaluation grammar, the lookup rules, wiring rule, and define a new function called Align. We can modify the wiring rules such that the graph contains information about the set of contexts variables are defined in.

The graph is initialized such that for the list of clauses e at the top level of the program, then $G \longrightarrow^1 G \cup \textsc{Wire}(None, e)$.

Definition 3.1. Let $\textsc{Wire}(x_0, (\,[c_1 = (x_1 = b_1), \ldots, c_n = (x_n = b_n)]\,)) = \{x_i \rightarrow (x_0, c_i) | 1 \leq i \leq n\}$

Definition 3.2. *align C ctx* is the function given the context stack C and starting the search for ctx, where $C = [(ctx', \_)] \,||\, C'$:

(1) $\boxed{\textsc{Found Context Variable}}$
If $ctx' = ctx$, then C
(2) $\boxed{\textsc{Not Found Context Variable}}$
If $ctx' \neq ctx$, then *align* C' ctx

Definition 3.3. Given control flow graph G, $G(x, C_0, S)$ is the function returning a value where, given $x \rightarrow (ctx, a_1)$ in G and $C = align\ C_0\ ctx$:

v	::=	$\ldots \mid \mathbb{Z} \mid \mathbb{B}$	*values*
p	::=	`fun` | `int` | `true` | `false` | `any`	*patterns*
b	::=	$\ldots \mid \square x \mid x \odot x \mid x \sim p \,?\, f : f$	*clause bodies*
$\odot$	::=	`+` | `-` | `<` | `<=` | `==` | `and` | `or`	*binary operators*
$\square$	::=	`not`	*unary operators*

Fig. 11. Expression Grammar Modifications

(1) $\boxed{\text{Value Discovery}}$
If $a_1 = (x = v)$, then (v, C, S).

(2) $\boxed{\text{Alias}}$
If $a_1 = (x = x')$ then $G(x', C, S)$.

(3a) $\boxed{\text{Function Enter Parameter, Lookup Local}}$
If $a_1 = (\lhd\, x)$, and $C = [(\_, t)] \,||\, C'$, and $S(t) = (x_v, C'')$ where C'' is a context stack, $l = G(x_v, C'', S)$ and $(v, C''', S') = l$, then $S'' = S'\{t \mapsto l\}$ and (v, C''', S'')

(3b) $\boxed{\text{Function Enter Parameter, Stored Local}}$
If $a_1 = (\lhd\, x)$, and $C = [(\_, t)] \,||\, C'$, and $S(t) = l$ where l is a lookup value, then l.
$\boxed{\text{Function Enter Non-Local}}$
~~If $a_1 = (\lhd\, x')$, $x' \neq x$, and $C = [(a_2, \_)] \,||\, C'$, then $G(x, a_2, C', S) \subseteq V$.~~

(4) $\boxed{\text{Application Clause}}$
If $a_1 = (x = x_f\ x_v)$, and $(\texttt{fun}\ x'' \texttt{->} (\,e\,), C', S') = G(x_f, C, S)$, then $G(x', [(ctx, t)] \,||\, C', S'\{t \mapsto (x_v, C)\}))$ where $x' = \textsc{rv}(e)$ and $t \notin Dom(S')$
$\boxed{\text{Skip}}$
~~If $a_1 = (x'' = b)$, $x'' \neq x$, then $G(x, ctx, C)$.~~

The align rule is used to find the context variable we are looking for in the context stack. We were able to remove the Value Discard rule in lookup rules since by construction of the graph, x will always point to a clause $x = b$. The Function Enter Non-Local is removed for the same reason as Value Discard as that rule was just a special case of Value Discard where we reached the end of a function; the definition of the rule was in essence moved to the align rule.

One observation in terms of the alignment definition is that in the lookup rule its results are most specifically used only in the Function Enter Parameter Rules. In the Alias rule, the aligned context stack is not used constructively. So, to optimize the usage of the align definition, in the implementation of the lookup rule, the align definition is only used in the Function Enter Rules and the Value Discovery Rule. The reason to use it in the Value Discovery Rule is to prevent the context stack from growing too big.

4 EXTENSIONS

In this section, the ωDDPAc rules are extended to incorporate booleans, integers, binary operations, unary operations, and conditional branching. The language grammar extensions is depicted in Figure 11.

4.1 Integers, Booleans, and Operations

Since the binary and unary operations are very similar, the integer binary operation rules are grouped together. To encourage the lookup rules to be further lazy, the rules for `and` and `or` are specified separately from the other rules. The rules for Value Lookup does not change since integers and booleans are not handled differently from functions in terms of being values.

Definition 4.1. Given control flow graph G, $G(x, C_0, S)$ is the function returning a value where, given $x \rightarrow (ctx, a_1)$ in G and $C = align\ C_0\ ctx$:

(5) $\boxed{\text{Integer Binary Operation}}$
If $a_1 = (x_1 \odot x_2)$, $\odot \in \{\texttt{+}, \texttt{-}, \texttt{<}, \texttt{<=}, \texttt{==}\}$, $(v_1, \_, S') = G(x_1, C, S)$, $(v_2, \_, S'') = G(x_2, C, S')$, $v_1 \in \mathbb{Z}$, and $v_2 \in \mathbb{Z}$, then $(v_1 \odot v_2, C, S'')$.

(6a) $\boxed{\text{And Binary Operation Short Circuit}}$
If $a_1 = (x_1\, \texttt{and}\, x_2)$, and $(\texttt{false}, \_, S') = G(x_1, C, S)$, then $(\texttt{false}, C, S')$.

(6b) $\boxed{\text{And Binary Operation}}$
If $a_1 = (x_1\, \texttt{and}\, x_2)$, $(\texttt{true}, \_, S') = G(x_1, C, S)$, $(v_2, \_, S'') = G(x_2, C, S')$, and $v_2 \in \mathbb{B}$, then (v_2, C, S'').

(7a) $\boxed{\text{Or Binary Operation Short Circuit}}$
If $a_1 = (x_1\, \texttt{or}\, x_2)$, and $(\texttt{true}, \_, S') = G(x_1, C, S)$, then $(\texttt{true}, C, S')$.

(7b) $\boxed{\text{Or Binary Operation}}$
If $a_1 = (x_1\, \texttt{or}\, x_2)$, $(\texttt{false}, \_, S') = G(x_1, C, S)$, and $v_2 \in \mathbb{B}$, then (v_2, C, S'').

(8) $\boxed{\text{Not Unary Operation}}$
If $a_1 = (\texttt{not}\, x)$, $(v, \_, S') = G(x, C, S)$, and $v \in \mathbb{B}$, then $(\texttt{not}\, v, C, S')$.

4.2 Conditional Branching

Conditionals have syntax $x \sim p\, ?\, f : f$ and are straightforward: the bodies are wired in just like function calls. The conditional syntax has to eagerly match the pattern and then the function call is similar to Application Clause Rule.

Definition 4.2. Let $\text{ImmediatelyMatchedBy}(\hat{v})$ be defined as follows:

- $\text{ImmediatelyMatchedBy}(\hat{f}) = \{\texttt{fun}, \texttt{any}\}$
- $\text{ImmediatelyMatchedBy}(\texttt{int}) = \{\texttt{int}, \texttt{any}\}$
- $\text{ImmediatelyMatchedBy}(\texttt{true}) = \{\texttt{true}, \texttt{any}\}$
- $\text{ImmediatelyMatchedBy}(\texttt{false}) = \{\texttt{false}, \texttt{any}\}$

For all other values, ImmediatelyMatchedBy is undefined.

Definition 4.3. Given control flow graph G, $G(x, C_0, S)$ is the function returning the values where, given $x \rightarrow (ctx, a_1)$ in G and $C = align\ C_0\ ctx$:

(9a) $\boxed{\text{Conditional Clause Positive Rule}}$
If $a_1 = (x_c = x \sim p\, ?\, f_1 : f_2)$, $f_1 = \texttt{fun}\ x'' \texttt{->} (\, e\,)$, $l = G(x, C, S)$, $(v, \_, S') = l$, $p \subseteq \text{ImmediatelyMatchedBy}(v)$,
$G(x', [(a_1, t)] \,||\, C, S'\{t \mapsto l\})$ where $x' = \text{RV}(e)$ and $t \notin Dom(S')$.

(9b) $\boxed{\text{Conditional Clause Negative Rule}}$
If $a_1 = (x_c = x \sim p\, ?\, f_1 : f_2)$, $f_2 = \texttt{fun}\ x'' \texttt{->} (\, e\,)$, $l = G(x, C, S)$, $(v, \_, S') = l$, $p \not\subseteq \text{ImmediatelyMatchedBy}(v)$,
$G(x', [(a_1, t)] \,||\, C, S'\{t \mapsto l\})$ where $x' = \text{RV}(e)$ and $t \notin Dom(S')$.

An interesting part of the rule is the information that is pushed onto the stack. Since the function is defined in the same line as conditional clause, the location where the function is defined and the location where the function is called is the same. Also, since the value that is passed into the function is evaluated, the Function Enter Parameter Lookup Local Rule is never reached, so there is no need to define a new rule for Function Enter Parameter Lookup Local Rule for the Conditional Clause.

$$\begin{array}{lcll} v & ::= & \ldots \mid \mathbb{N}\, u & \textit{values} \\ \odot & ::= & \ldots \mid \texttt{+.} \mid \texttt{-.} \mid \texttt{<.} \mid \texttt{<=.} \mid \texttt{==.} & \textit{binary operators} \end{array}$$

Fig. 12. Expression Grammar Modifications for Unsigned Integers

4.3 Implementation of Extensions

The current lookup value set in the evaluation grammar rules contains a tuple of a value, a location in the graph, and a context stack. The context stack is used to continue looking up within the function, however this is not needed in the case of integers and booleans. For implementation, to decrease the amount of memory needed, we can remove the context stack and location in the graph for integers and booleans.

5 UNSIGNED INTEGERS

Looking above at the extensions section, the 4.3 rule in comparison to other rules is much more eager. More specifically, the rules for 'and' and 'or' are short circuited; for function application, only the function is looked up since the value passed in is not necessarily needed until later. However, for integer operations and comparison operations, both variables have to be completely evaluated to perform the operation.

An interesting observation is that unsigned integers can be made more lazy. The main observation is that for unsigned integers, if we add two numbers a and b, and evaluate the left side to get v_a, we know the sum $a + b \geq v_a$ because b is an unsigned integer i.e. it will never be negative. If we incorporate unsigned integers into the language, using this observation to our advantage, we can make the comparison of two unsigned integers more lazy and efficient.

To distinguish between unsigned integer operations and signed integer operations, new operations are added to grammar which can be seen in Figure 12. The evaluation of the grammar changes such that if the operation does not contain a '.' after it, the operation is on integers whereas if the operation contains a '.', the operation is specifically for unsigned integers.

Also, we define natural numbers such that 0 is included. A u is added next to the natural numbers so that grammar is able to distinguish signed integers from unsigned integers.

For the following discussion of the rules of unsigned integers, I will focus only on the addition of unsigned integers and the comparison of two unsigned integers. In the later sections, the implementation of subtraction is discussed.

The naive rule to incorporate these changes are

Definition 5.1. Given control flow graph G, $G(x, C_0, S)$ is the function returning the least set of values V where, given $x \to (ctx, a_1)$ in G and $C = align\ C_0\ ctx$:

(10) $\boxed{\text{Unsigned Integer Binary Operation}}$
If $a_1 = (x_1 \odot .\ x_2)$, $\odot \in \{+, <, <=, ==\}$, $(v_1, \_, S') \in G(x_1, C, S)$, $(v_2, \_, S'') \in G(x_2, C, S')$, $v_1 \in \mathbb{N}$, and $v_2 \in \mathbb{N}$, then $(v_1 \odot v_2, C, S'') \in V$.

(11a) $\boxed{\text{Unsigned Integer Minus}}$
If $a_1 = (x_1 - .x_2)$, $(v_1, \_, S') \in G(x_1, C, S)$, $(v_2, \_, S'') \in G(x_2, C, S')$, $v_1 \in \mathbb{N}$, $v_2 \in \mathbb{N}$, and $v_1 \geq v_2$, then $(v_1 - v_2, C, S'') \in V$

(11b) $\boxed{\text{Unsigned Integer Minus}}$
If $a_1 = (x_1 - .x_2)$, $(v_1, \_, S') \in G(x_1, C, S)$, $(v_2, \_, S'') \in G(x_2, C, S')$, $v_1 \in \mathbb{N}$, $v_2 \in \mathbb{N}$, and $v_1 < v_2$, then $(0, C, S'') \in V$.

5.1 Church Encoding

An interesting observation is that if we revert to the lambda calculus language and implement unsigned integers using Church Encoding, this laziness occurs naturally. For example, say we want to check if the 10-th Fibonacci number is equal to 20, then using the Church encoding, once the 10-th Fibonacci number is found to be greater than 20 (i.e. does sums up to 21), it returns false.

5.1.1 Internal Encoding. Since Church encoding seems to naturally make unsigned integers lazy, this section focuses on how to incorporate what a Church encoding does into the lookup rules of ωDDPAc. The overview of what the changes were to incorporate the Church encoding is internally we church encode each unsigned integer addition and each unsigned integer declaration, and when we do a comparison, we 'unwrap'/'expand' the unsigned integer step by step until we can give a sure answer to the comparison.

The church encoding of a number n is $\lambda f.\lambda x.f^n\ x$; the church encoding of the addition function is $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$. In a church encoding of a number, f is the successor function (the function that adds one) and x is 0. However, for the language, we do not need this form of granularity. It would be more efficient to replace f^n with another function that specifically adds n to x. Because of this change, we can think of the church encoding for a number as $\lambda x.f_n\ x$ (and for $n = 0$, $\lambda x.x$) and the church encoding of addition as $\lambda m.\lambda n.\lambda x.m\ (n\ x)$.

To incorporate these two encodings, the three variables x, m, and n have to have an internal representation. Since these are basically just variables in the language, the implementation has to allow the ability to push these internal representations of these three variables on the stack as well as use them in the lookup rule.

Along with this, there are four new return types. The three of them that are straightforward are the representations of the addition function (uint-add), the unsigned int function (uint-num), and another representation for the 0 unsigned int function (uint-zero). The reason there is a separate representation of 0 is that it mechanically works differently than for positive unsigned integers. The fourth return type is the number 0 that is not the church encoding of 0 (final-uint-zero). This is used to signify when the unsigned integer has been completely expanded.

5.1.2 Evaluation. When evaluating unsigned integers in the church encoding, the unsigned integer is only evaluated (whether partially or fully) when either doing a comparison between two unsigned integers or where the final result of the program is an unsigned integer. In the case where the unsigned integer is the final result of the program, the unsigned integer must be fully evaluated.

In the case where the program is not computing the value of an unsigned integer, there are two cases: one where the unsigned integer is defined and the other when two unsigned integers are added together. In the first case, the object that is returned is simply an encoding of $\lambda x.f_n\ x$ or $\lambda x.x$ (uint-num or uint-zero); the second case, since the encoding for addition is $\lambda m.\lambda n.\lambda x.m\ (n\ x)$, the information about m and n (which are the two variables that are being added along with the context stack) are pushed to the stack and the object returned is an encoding of $\lambda x.m\ (n\ x)$ (uint-add).

Once evaluation of an unsigned integer starts, 0 must be passed into the unsigned int encoding, more specifically x=final-uint-zero is pushed to the top of the context stack which will be used to signify the end of computation of the unsigned integer.

Evaluation of an unsigned integer is similar to the function application rule since Church encodings is done with multiple function applications (where the function application in terms of unsigned integers will be interpreted as addition).

In the case of unsigned integer evaluation, there are four return cases: when it's final-uint-zero, the computation is done and a 0 is returned; in a uint-zero, then looking up continues by looking up the encoding of the formal parameter x ($\lambda x.x$); in the case on uint-num, we can return the value

f_n and keep the context stack frozen so that the computation can be resumed later: in the case of uint-add, the lookup is continued by looking up the encoding of the formal parameter m because the encoding of addition is $\lambda x.m\ (n\ x)$.

To actually do a lookup of the formal parameters x, m, or n, we just have to retrieve the information from the context stack and then repeat the four cases of unsigned integer evaluation to see if we have found a non zero unsigned integer or final-uint-zero. When looking up m or n, we are following the rule of a function application lookup; the formal parameter is x since the functions are encodings of unsigned integers, and the passed in parameter is n or x, respectively. This information is put on the stack before we repeat the case of unsigned integer evaluation to see if we have found a non zero unsigned integer.

In the case of full evaluation, we just have to do a partial step, remember the result of the partial computation and add that to the next step in the computation, and keep repeating until we reach final-uint-zero.

In the case of comparison (for example $x_1 < x_2$), we partially evaluate x_1 and x_2 until we reach a case in which we can definitively give the result of the comparison. There are two cases in which we can finish comparing the two variables: one in which both are done evaluating and we just compare the numbers; in the other case, one is done evaluating and the other is number is larger than the evaluated number, so we know the number will only grow from there on. There are still two other cases that we can run into: one is where one is done evaluating but the other is not larger than the evaluated number, in this case, we have to continue partially evaluating the second number until it is either done evaluating or it becomes bigger than the one completely evaluated. In the case both are not done evaluating, then for efficiency, it is better to always evaluate only the smaller value, because it might not be beneficial to further evaluate the larger number.

5.1.3 Inefficiencies. There are a few inefficiencies in this implementation. The most glaring inefficiency is the local value caching mechanism. The information stored in the local value will be the location of the beginning of a unsigned integer computation; however, if we start expanding the computation, that information is not used to update the local information. Another inefficiency is the complexity in having four different representations of unsigned integers.

5.2 Simplification of Encoding

In the above example, the continuation of the computation is stored on the context stack. However, this can be simplified by creating a new stack called the computation stack. This stack will contain what values we need to add together to get the result (i.e. we need to store the variable name and the context stack).

Given this change, instead of needing four different objects to describe an unsigned integer, we need only one which contains the integer and the rest of the computation stack needed to compute the rest of the unsigned integer.

For implementation, there are two separate functions: one without the computation stack since until we are doing a comparison or a full evaluation, the computation does not need to be continued and another function which contains the computation stack to continue the computation. The second function is similar to the original lookup, however the only difference is that any computation that does not lead to an unsigned integer is an error and whenever an unsigned integer addition is reached $a + b$, b is pushed to the computation stack and a is looked up.

However, with these changes, the problem with local caching has not been fixed.

5.3 Improvement of Local Caching

At a high level, the change to improve local variable caching is by modifying the computation stack to contain information about which formal parameters variables are needed to compute the value. To save the work into the formal parameter, we need to be able to undo any work that was not a part of the computation of that variable.

The computation stack is further modified into a stack of 'computation frames'. Each frame contains the variables needed to calculate the variable either started from a local lookup or the original variable the program was looking up as well as the value calculated up until that point. During local lookups, the computation frame is saved on the context stack for each local variable along with the partial evaluated value.

Whenever we start a new local lookup, the current computation frame, the current value computed, and the variable that was being looked up is pushed to the computation frame stack and the new frame of the local frame is 'loaded' in and that computation is continued.

The partially computed value is kept track of so that when we get to a new value, it is added to the partially computed value. Another value is stored which is the value computed up to the start of a new local value lookup; this is stored so that this value can be subtracted away when the value is stored in the new local variable on the context stack.

When the computation must be continued, the computation frame stack is checked: the first check is to see if there're more computations needed to compute the local value; if there is no other computations for the local value needed, the top of the computation frame stack is popped and that local value is continued to computed until the stack is empty.

6 EVALUATION

There were four different interpreters I compared to measure performance. There is an implementation of the forward interpreter in OCaml based on a CEK machine, an implementation of ωDDPAc without lazy unsigned integers, an implementation of ωDDPAc with an internal representation of Church encodings, and an implementation of ωDDPAc with lazy unsigned integers given the changes described in 5.2 and 5.3.

For all the experiments, recursion is done through self-passing. Five different experiments were run: 'fibuint' is an implementation of calculating the 25-th fibonacci number using unsigned integers; 'fibuint-cmp' is an implementation of calculating the 25-th fibonacci number using unsigned integers and checks if it is equal to 10000; 'fibonacci' is an implementation of calculating the 28-th fibonacci number using signed integers; 'church' is an implementation of calculating the 18-th fibonacci number using a church encoding; 'ack' is an implementation of the Ackermann function A(3,7). The results are found in Figure 13.

Because of the lazy evaluation of unsigned integers, the Church Encoding interpreter and the improved unsigned integer interpreter heavily outperform the other two interpreters on 'fibuint-cmp'. The improved unsigned integer interpreter consistently does better than the church encoding interpreter because of the issues mentioned in 5.1.3.

Interestingly, the ωDDPAc interpreter and the improved unsigned integer interpreter outperformed the CEK machine forward interpreter in all the experiments. The two interpreters that have lazy unsigned integers do worse than the original ωDDPAc interpreter except for the 'fibuint-cmp' experiment. However, even in the 'fibuint' experiment, the improved unsigned integer interpreter is only about 15% worse than the ωDDPAc interpreter.

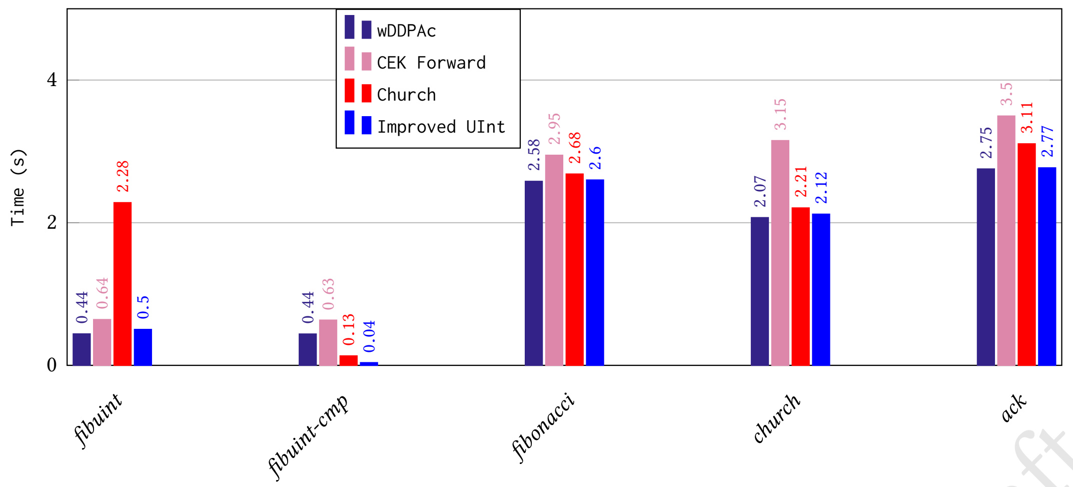

Fig. 13. Comparing Interpreters

7 RELATED WORK

ωDDPAc is a presentation of graph-based operational semantics which represents expressions as concrete (runtime) control flow graphs. For optimizations, an environment was implicitly added to the context stack. There are many other machines for interpreting programs efficiently which have interesting comparisons to ωDDPAc.

Haskell uses outermost graph reduction which is similar to the rules for ωDDPAc as when there is a function application, just the function is looked up. Haskell also shares the reference to the parameter that is passed into the function which is accomplished in ωDDPAc by adding a reference to the passed in parameter on the context stack. Haskell uses a Spineless Tagless G-Machine [6] for the interpreter whcih has an evaluation stack which is used to keep track of the variables that are passed into the function which is similar to the context stack. It seems that due to the outermost graph reduction, ωDDPAc seems to be just as lazy as Haskell. However, the incorporation of lazy evaluation of unsigned integers makes ωDDPAc more lazy than Haskell in some cases. However, testing the laziness from evaluating unsigned integers through Church encoding in Haskell is difficult because Haskell is a typed language.

Matthias Felleisen's CESK machine [2, 3] is an architecture that uses a control component (the code), an environment (associates variables to addresses in the store), a store (map from address to value), and a continuation stack (program stack). The role of the environment is used to differentiate program points which is similar to context stack which also maps to locations in the graph. The store contains information shared by all program points which is similar to the graph. The continuation stack is dealt by the fact that the lookup rules are recursive, and since information is stored in the stack of the program running the interpreter, that stack is similar to the continuation stack.

Like [5], ωDDPAc is a graph-based notion of evaluation, but where function bodies are never copied – a single body is shared. In [5], the graph is constructed and reduced such that functions are shared wherever possible. If a function is used in two separate parts of the program and different parameters are passed in, the function is shared in the graph with differences in the parameters that were passed in. It seems that the amount of sharing of functions is the same as in ωDDPAc in which the function is not copied when used multiple times, but more work is needed to determine the exact relationship as it seems [5] can do partial evaluation of functions whereas ωDDPAc does not.

8 CONCLUSION

In this paper, the viability of ωDDPAc as an interpreter was analyzed and the rules were modified to improve the performance of the interpreter. The original set of rules for ωDDPAc is interesting as it lacks closures, substitution, fresh variables, or an environment, and reductions are all polynomially bounded in length. The optimizations included an environment on the context stack to perform non-local lookups as well as lookups of parameters passed into functions. Along with this, the construction of the graph is a preprocessing step so there are fewer reductions. Given the benchmarks, ωDDPAc shows promising signs of having practical performance benefits.

REFERENCES

[1] Leandro Facchinetti, Zachary Palmer, and Scott Smith. 2017. Higher-Order Demand-Driven Program Analysis. (2017).
[2] Mattias Felleisen. 1987. The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Phd thesis)*.
[3] Mattias Felleisen and Daniel P. Friedman. 1987. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 314+.
[4] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*.
[5] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. 1992. The Geometry of Optimal Lambda Reduction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[6] Simon L Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine.