Practical Demand-Driven Program Analysis with Recursion *

Leandro Facchinetti †
The Johns Hopkins University
leandro@jhu.edu

Abstract

Demand-Driven Program Analysis (DDPA) is a context-sensitive control-flow analysis for higher-order functional programming languages in the same space as k-CFA and related analyses. DDPA is provably sound and polynomial, but that alone does not guarantee its applicability to real-world programs. This project studies DDPA's practicality in two dimensions: expressiveness (the analysis closely approximates the run-time) and performance (the analysis is fast to compute). To address expressiveness, we extended the analysis to directly support recursion, eliminating the need for encodings. To address performance, we developed a higher-level programming language that compiles to DDPA's core language, ported over benchmarks that exercise all its features, and measured the reference implementation's performance, comparing to other state-of-the-art program analyses. DDPA is competitive and, in some cases, faster than the alternatives.

1. Introduction

Demand-Driven Program Analysis (DDPA) is a form of context-sensitive control-flow analysis for higher-order functional programming languages [17]. DDPA takes a demand-driven approach to program analysis, which works by incrementally building a control-flow graph of the program and performing lookups by traversing it backwards [4, 5], from end to start. The extent of the context sensitivity is a parameter k – higher k translates to more space available in the abstract context stack and *potentially* more context sensitivity; but also *potentially* more work, which slows down the analysis.

DDPA is polynomial for a fixed k, in contrast with k-CFA, which is EXPTIME-complete [21]. This suggests DDPA has the potential of being practical for real-world usage. To achieve this goal, the analysis has to be expressive enough to represent a wide variety of programs, predicting their run-time behavior with precision, and its implementation has to be fast. The contributions of this project span both expressiveness and performance: we extended DDPA to directly support recursive functions, without relying on encodings; and, we measured its implementation's performance against other state-of-the-art analyses. The result was an assessment of the practicality of DDPA as a program analysis for real-world programs.

In terms of expressiveness, recursion is a fundamental construct of functional programs, but DDPA's core language does not support it directly. Programs that require recursive functions must use encodings such as self-passing and the Y-combinator, which introduce extra function calls. More function calls in the program means more work for the analysis – the problem gets even worse in programs that require mutually recursive functions and objects, as they heavily depend upon recursion. Adding first-class support for recursive functions in the analysis allows for programs without the extra function calls from encodings.

Section 2 specifies an extension to DDPA that adds support for recursion. To illustrate the enhanced expressiveness provided by this modification, we cover an encoding of objects in which all methods are mutually recursive.

Section 3 covers the other dimension of practicality with performance tests of DDPA's reference implementation and other related control-flow analyses. It outlines a higher-level programming language that compiles to DDPA's core language; the benchmarks that we translated to it, which exercise all the analysis features; and the results of experiments.

Finally, sections 4 and 5 discuss related and future work, respectively.

2. Natural recursion

This section describes an extension to DDPA that adds direct support for recursion. We begin with an intuition for the feature, then we explain the changes necessary to the operational semantics of the core language and to the analysis. The section's conclusion is an exploration of an use case: object encoding.

2.1 Intuition

The only way to write recursive functions in DDPA's core language is to use an encoding. For example, the following program uses self-passing to define a recursive function and then calls it once:

```
fpp = fun fp -> (
  fx = fun x -> (
    fxr = fp fp;
    fr = fxr x;
  );
);
f = fpp fpp;
a = 0;
r = f a;
```

The goal is to allow for the following more *natural* rewrite:

```
f = fun x -> (
  fr = f x;
);
a = 0;
r = f a;
```

Note that the original program contained four function calls – at `fxr`, `fr`, `f` and `r` – while the rewritten program contains only the two the programmer intended the write – at `fr` and `r`. In this example, half the function calls existed only to encode recursion via self-passing, but resolving them still requires lookups and results in a bigger control-flow graph. An alternative to self-passing for encoding recursion is the Y-combinator. Using it can reduce the number

* Research project report to fulfill a qualifying requirement of the Ph.D. program at The Johns Hopkins University.

† The author's work is supported by a CAPES fellowship. Process number: 13477/13-7.

of call sites, but is not ideal for the analysis because, upon context stack exhaustion, DDPA loses context sensitivity – i.e., functions calls and returns no longer align. When that happens, any lookup for a particular recursive function results in all recursive functions in the program. Therefore, the effort to extend the analysis with support for recursion is still valid.

The following subsections describe how to achieve this goal in two parts: first at the language's operational semantics, then with the corresponding modifications to the analysis.

2.2 Operational semantics

Consider the rewritten program from the previous subsection. In DDPA's core language, it is ill-formed, because line 2 uses the variable `f`, which is out of scope. But, setting aside the well-formedness condition and supposing the program was well-formed, what would its run-time behavior be?

According to the operational semantics from [17], Section 4.1, the only rule that matches the program is APPLICATION. The program point `r` matches x_1 and the result is to WIRE function `f`'s body in place. Subsequent matches can only occur in the introduced clauses, not before, so the program point `f` is already in the environment E. When the use of `f` at line 2 causes a LOOKUP, it succeeds.

The small-step evaluation machine does not get stuck in this program – it steps forward. The intended semantics is well-defined with the existing operational semantics relation; no changes are necessary. The only requirements are to relax the well-formedness condition to allow for restricted cases of recursive functions and to adapt the analysis accordingly. This is the subject of the following subsections.

2.3 Well-formedness condition

To support programs with recursive functions, modifications to the well-formedness condition are necessary. The original definition allowed programs in which all binding occurrences of variables had unique names and all use occurrences of variables referred to names in the lexical scope. The change is to no longer use the rules of lexical scoping, but those of a non-standard definition of scoping that permits limited forms of references to identifiers before their definitions. It is neither lexical scoping nor dynamic scoping, but somewhere in between. The closest analogy is the treatment of variables given by the *variable hoisting* feature in languages like JavaScript [2].

This non-standard definition of scoping is sufficient to admit programs with recursive functions – including mutually recursive functions. The current subsection presents an intuition for the non-standard notion of scoping, the next formalizes the intuition in a complete specification.

Valid programs are those that do not bind the same variable more than once and are *closed*. This means variables must be in scope before use, but the notion of scoping does not have to be the usual – lexical scoping. We present the variations by example, starting with the following program:

```
1 f = fun x -> (
2   r = f x;
3 );
4 a = 0;
5 j = f a;
```

The binding of the variable `f` occurs after considering the function body `fun x ->(...)`, which uses `f`. Nevertheless, the program is well-formed: the interpreter does not have a problem with it because it processes function calls by *inlining* the function body at the call site. In the example above, it means the program point `r` is *inlined* at the call site `j`, and `f` is already in the environment by then.

Furthermore, references to variables that occur further down in the program are also considered well-formed, as depicted by the following program:

```
1 f = fun x -> (
2   b = a;
3 );
4 a = 0;
5 r = f a;
```

At the program point `b`, there is a reference to the variable `a`, whose definition appears after the function containing `b`. Again, this works because at the call site `r` the value for `a` is already in the environment. A slight variation in this program makes it ill-formed, though:

```
1 f = fun x -> (
2   b = a;
3 );
4 c = 0;
5 r = f c; # ILL-FORMED
6 a = 0;
```

Here, the program point `r` includes a call to the function `f`, but `f` refers to `a`, whose value is so far undefined. In order to call a function, all its external dependencies must already be available. This restriction extends to any use of the function, not only calling it, as the code below shows:

```
1 f = fun x -> (
2   b = a;
3 );
4 g = f; # ILL-FORMED
5 a = 0;
```

This is an ill-formed program even though its semantics would be well-defined. To detect this in the general case requires data-flow analysis and, even though DDPA is a such an analysis, for simplicity we decided not use it for this purpose. There exists an idea of using the binding structure detected by the well-formedness condition to inform the analysis – see the section on future work – which would introduce mutual dependency between well-formedness condition and the analysis, and we want to avoid that complexity. With this restriction in place, the treatment of external dependencies can be purely syntactical and, for all interesting uses – for example, defining mutually recursive functions and closure encoding of objects – it is not an important limitation.

Note that variables defined in conditional branches are not available during scope resolution, so the following code is ill-formed:

```
1 f = fun x -> (
2   b = a; # ILL-FORMED
3 );
4 c = 0;
5 r = c ~ {} ? fun fm -> (
6   a = 0;
7   e = f a;
8 ) : fun fa -> (
9   d = 0;
10 );
```

The program would have well-defined semantics because, at the program point `e` that includes a call to `f`, `a` is already available. But, if `f`'s call site had occurred after the conditional and evaluation had proceeded by the anti-match branch, `a` would not be in the environment. The treatment of this edge case is to consider all programs of this kind ill-formed.

A reference to a function that still has unsatisfied external dependencies is only possible if itself happens *frozen* in another function body. The code below exemplifies such a case:

```
1 f = fun x -> (
```

```
  b = a;
);
g = fun y -> (
  c = f y;
);
a = 0;
r = g a;
```

In the program above, note how `g` can refer to `f` even before `a`'s definition. But this binds `g` to the same obligations `f` had – namely, an *unfrozen* reference to `g` can not occur before `f`'s external dependency, `a`, is in the environment. The following program is ill-formed because it does not respect that condition:

```
f = fun x -> (
  b = a;
);
g = fun y -> (
  c = f y;
);
d = 0;
r = g d; # => ILL-FORMED
a = 0;
```

Finally, the last condition is that external dependencies are not allowed for variables outside the enclosing immediate block. Or, in other words, hoisting only works one lexical level deep. This is an artificial restriction in place to simplify the analysis – there exists a guarantee that the REWIND operation only needs to result in a JUMP as far as the end of the immediate block. It also simplifies the implementation, in which the control-flow graph grows incrementally, and the target of the JUMP may not be immediately available. For most interesting examples, this is not a limiting restriction – removing it is future work, see section 5.

Consider the example:

```
f = fun x -> (
  g = fun y -> (
    b = a; # => ILL-FORMED
  );
);
a = 0;
```

The program is ill-formed because, at the point `b`, the reference to the variable `a` is outside `g`'s enclosing immediate block – i.e., `f`'s body. Note the implication this has in η-conversion: refactoring a program by enclosing a block with an immediately invoked function does not work if the block has an unsatisfied external dependency.

This restriction only applies to direct external dependency. The well-formedness condition allows indirect external dependency resulting from the use of a variable that itself has unsatisfied external dependencies. Regardless of the nesting level, respecting the notion of non-locals in lexical scoping. The following well-formed program exemplifies this case:

```
f = fun x -> (
  b = a;
);
g = fun y -> (
  h = fun z -> (
    c = f f;
  );
);
a = 0;
r = g a;
```

With all this in place, programs with recursive functions are natural to write, without the need for self-passing or other encoding. Consider the following real-world examples, starting with a recursive summation function:

```
# let rec sum = fun number ->
#   if number = 0 then
#     0
#   else
#     (sum (number - 1)) + number
# in
# sum 5 (* => 15 *)

zero = 0;
one = 1;
five = 5;

sum = fun number -> (
  numberequal0 = number == zero;
  sumresult = numberequal0 ~ true ? fun numberequal0match ->
(
    numberequal0matchresult = zero;
  ) : fun numberequal0antimatch -> (
    numberminus1 = number - one;
    partialsum = sum numberminus1;
    numberequal0antimatchresult = partialsum + number;
  );
);

r = sum five; # => 15
```

And a program that finds out if a number is even or odd with mutually recursive functions:

```
# let rec even n =
#   if n = 0 then true
#   else odd (n - 1)
# and odd n =
#   if n = 0 then false
#   else even (n - 1)
# ;;
#
# even 0;; (* => true *)
# odd 0;; (* => false *)
#
# even 3;; (* => false *)
# odd 3;; (* => true *)
#
# even 6;; (* => true *)
# odd 6;; (* => false *)

zero = 0;
one = 1;
three = 3;
six = 6;

even = fun evennumber -> (
  evennumberiszero = evennumber == zero;
  evenresult = evennumberiszero ~ true ?
  fun evennumberiszeromatch -> (
    evennumberiszeromatchreturn = true;
  ) : fun evennumberiszeroantimatch -> (
    evennumberminusone = evennumber - one;
    evennumberiszeroantimatchreturn = odd evennumberminusone;
  );
);

odd = fun oddnumber -> (
  oddnumberiszero = oddnumber == zero;
  oddresult = oddnumberiszero ~ true ?
  fun oddnumberiszeromatch -> (
    oddnumberiszeromatchreturn = false;
  ) : fun oddnumberiszeroantimatch -> (
    oddnumberminusone = oddnumber - one;
    oddnumberiszeroantimatchreturn = even oddnumberminusone;
  );
);

```

```
45 evenzero = even zero; # => true
46 oddzero = odd zero; # => false
47
48 eventhree = even three; # => false
49 oddthree = odd three; # => true
50
51 evensix = even six; # => true
52 oddsix = odd six; # => false
```

The next subsection formalizes the introduced intuitions.

2.4 Formalization

The approach to formalize the notion of well-formedness is to divide the problem in two independent parts: the first is to check whether all variable bindings are unique, the second is to check whether the program respects the modified notion of lexical scoping introduced in the previous subsection.

To accomplish the first part, we list all the bindings in the program and check for duplicates. The second part is further subdivided in two responsibilities: build a relation of external dependencies and their requirements for each clause in the program, and check that all the external dependencies' requirements are satisfied at each variable use occurrence.

The presentation of the formalization starts with some useful notation, continues with the definition auxiliary definitions that collection information by traversing the syntax tree, and finally builds up to well-formedness predicate.

Notation 2.1. *Let $\overrightarrow{x} = [x, \dots]$ be a list of variables. Let $x \in \overrightarrow{x}$ iff $\overrightarrow{x} = [\dots, x, \dots]$. Similarly, let $c \in e$ iff $e = [\dots, c, \dots]$.*

The FT – "flatten" – function *linearizes* the expression, *pulling* the clauses nested within function bodies and conditionals to the top level:

Definition 2.2. *Let* FT *be the least function such that:*

- $\textsc{Ft}([]) = []$;
- $$\textsc{Ft}(c \,||\, e) = \begin{cases} c \,||\, \textsc{Ft}(e' \,||\, e) & \text{if } c = (x = \texttt{fun}\, x' \texttt{->} (\, e' \,)) \\ c \,||\, \textsc{Ft}(e_1 \,||\, e_2 \,||\, e) & \text{if } c = (x = x' \sim p \,?\, f_1 : f_2), \text{ where } f_1 = \texttt{fun}\ x_1 \texttt{->} (\, e_1 \,), \text{ and } f_2 = \texttt{fun}\ x_2 \texttt{->} (\, e_2 \,) \\ c \,||\, \textsc{Ft}(e) & \text{otherwise} \end{cases}$$

The $\textsc{Ft}^i$ – "flatten immediate block" – function *linearizes* the immediate block expression, *pulling* the clauses nested within conditionals – but not functions – to the top level (note that $\textsc{Ft}^i(e) \subseteq \textsc{Ft}(e)$):

Definition 2.3. *Let* $\textsc{Ft}^i$ *be the least function such that:*

- $\textsc{Ft}^i([]) = []$;
- $$\textsc{Ft}^i(c \,||\, e) = \begin{cases} c \,||\, \textsc{Ft}^i(e_1 \,||\, e_2 \,||\, e) & \text{if } c = (x = x' \sim p \,?\, f_1 : f_2), \text{ where } f_1 = \texttt{fun}\ x_1 \texttt{->} (\, e_1 \,), \text{ and } f_2 = \texttt{fun}\ x_2 \texttt{->} (\, e_2 \,) \\ c \,||\, \textsc{Ft}^i(e) & \text{otherwise} \end{cases}$$

The DEFINES function returns the variables immediately bound by an expression, shallowly traversing its list of clauses:

Definition 2.4. *Let* DEFINES *be the least function such that:*

- $\textsc{Defines}([]) = []$;
- $\textsc{Defines}((x = b) \,||\, e) = x \,||\, \textsc{Defines}(e)$.

The BINDINGS function returns all variables bound by an expression, deeply traversing its syntax tree (note that $\textsc{Defines}(e) \subseteq \textsc{Bindings}(e)$):

Definition 2.5. *Let* $\textsc{Bindings}(e) = \textsc{Defines}(e') \,||\, \textsc{Bindings}'(e')$, *where* $e' = \textsc{Ft}(e)$ *and* $\textsc{Bindings}'$ *is the least function such that:*

- $\textsc{Bindings}'([]) = []$;
- $\textsc{Bindings}'((x = \texttt{fun}\, x' \texttt{->} (\, e' \,)) \,||\, e) = x' \,||\, \textsc{Bindings}'(e)$;
- $\textsc{Bindings}'((x = x' \sim p \,?\, \texttt{fun}\ x_1 \texttt{->} (\, e_1 \,) : \texttt{fun}\ x_2 \texttt{->} (\, e_2 \,)) \,||\, e) = [x_1, x_2] \,||\, \textsc{Bindings}'(e)$;
- $\textsc{Bindings}'(c \,||\, e) = \textsc{Bindings}'(e)$, *otherwise.*

The USES function returns all variables used by an expression, deeply traversing its syntax tree:

Definition 2.6. *Let* $\textsc{Uses}(e) = \textsc{Uses}'(\textsc{Ft}(e))$, *where* $\textsc{Uses}'$ *is the least function such that:*

- $\textsc{Uses}'([]) = []$;
- $\textsc{Uses}'((x' = \{\ell_1 = x_1, \dots, \ell_n = x_n\}) \,||\, e) = [x_1, \dots, x_n] \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = \texttt{fun}\ x \texttt{->} (\, e' \,)) \,||\, e) = \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = \texttt{ref}\ x) \,||\, e) = x \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x) \,||\, e) = x \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x_1\ x_2) \,||\, e) = [x_1, x_2] \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x \sim p \,?\, f_1 : f_2) \,||\, e) = x \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x_1 \texttt{<-} x_2) \,||\, e) = [x_1, x_2] \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = \texttt{!}\, x) \,||\, e) = x \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = \square x) \,||\, e) = x \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x_1 \odot x_2) \,||\, e) = [x_1, x_2] \,||\, \textsc{Uses}'(e)$;
- $\textsc{Uses}'((x' = x_1\, .\, (\, x_2 \,)) \,||\, e) = [x_1, x_2] \,||\, \textsc{Uses}'(e)$.

The UNIQUE predicate holds when there are no repetitions among the variables:

Definition 2.7. *Let* UNIQUE *be the least predicate satisfying the following assertions:*

- $\textsc{Unique}([])$;
- $\textsc{Unique}(x \,||\, \overrightarrow{x})$ *iff* $x \notin \overrightarrow{x}$, *and* $\textsc{Unique}(\overrightarrow{x})$.

The definitions that follow all assume that $\textsc{Unique}(\textsc{Bindings}(e))$ hold and that, when given an expression e and a clause c, then the clause occurs in the expression: $c \in \textsc{Ft}(e)$.

The $x_d \prec_e x_r$ – "x_d immediately requires x_r in e" – relation holds if the x_r variable occurs but is not bound in the body of x_d declaration. Note that a variable can require itself if it is part of its own definition, as happens in the case of *immediate* recursion.

Definition 2.8. *Let the $x_d \prec_e x_r$ relation hold iff $c \in \textsc{Ft}(e)$, $c = (x_d = b)$, $x_r \in \textsc{Uses}([c])$, and either $x_d = x_r$ or $x_r \notin \textsc{Bindings}([c])$.*

The $\prec\!\!\!\prec$ relation is the transitive closure of $\prec$:

Definition 2.9. *Let $x_1 \prec\!\!\!\prec_e x_2$ hold iff $x_1 \prec_e \dots \prec_e x_2$.*

The DEPS – "dependencies" – function returns the variables upon which a clause depends. Those are the variables that *must* be in scope in a well-formed clause; the variables that are immediate requirements and are not *frozen* (note that $\textsc{Deps} \subseteq \prec$). Note that the only construct capable of *freezing* variable uses is a function declaration.

Definition 2.10. *Let* DEPS *be a function such that:*

$$\textsc{Deps}_e(c) = \begin{cases} [] & \textit{if } c = (x \texttt{ = fun } x \texttt{ -> (} e \texttt{)}) \\ [x_1, \ldots, x_n] & \textit{for all } x_i \\ & \textit{such that } c = (x = b), \\ & x \prec_e x_i, \\ & i \in \{1, \ldots, n\}, \\ & \textit{otherwise} \end{cases}$$

The UFD – "up to first dependent" – function returns the clauses up to (before) the first clause that depends on a given variable (not inclusive).

Definition 2.11. *Let* UFD *be a function such that:*

$$\textsc{UFD}_e(x) = \begin{cases} e & \textit{if there does not exist } c \in e \\ & \textit{such that } x \in \textsc{Deps}_e(c) \\ e_1 & \textit{where } e = e_1 \,||\, c \,||\, e_2, \\ & x \in \textsc{Deps}_e(c), \textit{ and} \\ & \textit{there does not exist } c_1 \in e_1 \\ & \textit{such that } x \in \textsc{Deps}_e(c_1) \end{cases}$$

The $\textsc{Avail}^r$ – "available for requirement" – function returns the variables that are available – i.e., already defined – at a given clause to satisfy requirements.

Definition 2.12. *Let* $\textsc{Avail}^r_e(c)$ *be the least function such that:*

$$\textsc{Avail}^r_e(c) = \begin{cases} \textsc{Defines}(e_1) & \textit{if } e = e_1 \,||\, c \,||\, e_2 \\ x' \,||\, \vec{x}_1 \,||\, \vec{x}_2 & \textit{where} \\ & \vec{x}_1 = \textsc{Defines}(\textsc{UFD}_e(x)), \textit{ and} \\ & \vec{x}_2 = \textsc{Avail}^r_{e'}(c), \textit{ if} \\ & e = e_1 \,||\, c' \,||\, e_2, \\ & c' = (x \texttt{ = fun } x' \texttt{ -> (} e' \texttt{)}), \\ & \textit{and } c \in \textsc{Ft}(e') \\ x' \,||\, \vec{x}_1 \,||\, \vec{x}_2 & \textit{where} \\ & \vec{x}_1 = \textsc{Defines}(e_1), \textit{ and} \\ & \vec{x}_2 = \textsc{Avail}^r_{e'}(c), \textit{ if} \\ & e = e_1 \,||\, c' \,||\, e_2, \\ & c' = (x_1 = x_2 \texttt{ ~ } p \texttt{ ? } f_1 \texttt{ : } f_2), \\ & \texttt{fun } x' \texttt{ -> (} e' \texttt{)} \in \{f_1, f_2\}, \\ & \textit{and } c \in \textsc{Ft}(e') \end{cases}$$

The $\textsc{Avail}^d$ – "available for dependency" – function returns the variables available – i.e., already defined – at a given clause to satisfy dependencies. Note that, in order to support the artificial restriction that external dependencies are not allowed for variables outside the enclosing immediate block, there is a special case in the treatment of function bodies. If we were to remove this artificial restriction, the notions of $\textsc{Avail}^d$ and $\textsc{Avail}^r$ would converge: $\textsc{Avail}^d_e(c) = \textsc{Avail}^r_e(c)$. But, as it is, neither $\textsc{Avail}^d_e(c) \subseteq \textsc{Avail}^r_e(c)$ nor $\textsc{Avail}^d_e(c) \supseteq \textsc{Avail}^r_e(c)$. In the following definition, the special case uses $\textsc{Ft}^i$ so that forward external dependencies within conditional bodies work:

Definition 2.13. *Let* $\textsc{Avail}^d_e(c)$ *be the least function such that:*

$$\textsc{Avail}^d_e(c) = \begin{cases} \textsc{Defines}(e_1) & \textit{if } e = e_1 \,||\, c \,||\, e_2 \\ x' \,||\, \vec{x}_1 \,||\, \vec{x}_2 & \textit{where} \\ & \vec{x}_1 = \textsc{Defines}(\textsc{UFD}_e(x)), \textit{ and} \\ & \vec{x}_2 = \textsc{Avail}^d_{e'}(c), \textit{ if} \\ & e = e_1 \,||\, c' \,||\, e_2, \\ & c' = (x \texttt{ = fun } x' \texttt{ -> (} e' \texttt{)}), \\ & \textit{and } c \in \textsc{Ft}^i(e') \\ x' \,||\, \vec{x}_1 \,||\, \vec{x}_2 & \textit{where} \\ & \vec{x}_1 = \textsc{Defines}(e_1), \textit{ and} \\ & \vec{x}_2 = \textsc{Avail}^d_{e'}(c), \textit{ if} \\ & e = e_1 \,||\, c' \,||\, e_2, \\ & c' = (x \texttt{ = fun } x' \texttt{ -> (} e' \texttt{)}), \\ & c \in \textsc{Ft}(e') \\ & \textit{and } c \notin \textsc{Ft}^i(e') \\ x' \,||\, \vec{x}_1 \,||\, \vec{x}_2 & \textit{where} \\ & \vec{x}_1 = \textsc{Defines}(e_1), \textit{ and} \\ & \vec{x}_2 = \textsc{Avail}^d_{e'}(c), \textit{ if} \\ & e = e_1 \,||\, c' \,||\, e_2, \\ & c' = (x_1 = x_2 \texttt{ ~ } p \texttt{ ? } f_1 \texttt{ : } f_2), \\ & \texttt{fun } x' \texttt{ -> (} e' \texttt{)} \in \{f_1, f_2\}, \\ & \textit{and } c \in \textsc{Ft}(e') \end{cases}$$

The INSCOPE predicate holds when the given variable dependency and all its requirements are available at the particular clause in the expression.

Definition 2.14. *Let* $\textsc{InScope}_e(c, x_d)$ *hold iff* $x_d \in \textsc{Avail}^d_e(c)$*, and for all* x_r *such that* $x_d \prec\!\!\prec_e x_r$*,* $x_r \in \textsc{Avail}^r_e(c)$*.*

Finally, the WELLFORMED predicate holds when the bindings are unique and the external dependencies of every clause are in scope.

Definition 2.15. *Let* $\textsc{WellFormed}(e)$ *hold iff* $\textsc{Unique}(\textsc{Bindings}(e))$*, and for all* c *and* x *such that* $c \in \textsc{Ft}(e)$ *and* $x \in \textsc{Deps}_e(c)$*,* $\textsc{InScope}_e(c, x)$*.*

Given this definition of well-formed programs, there is a guarantee that evaluation does not get stuck due to looking for a variable that is not in the environment. In other words, no JavaScript-like `undefined` values show up during evaluation.

Lemma 2.16. *If* $\textsc{WellFormed}(e)$*, then* $e \longrightarrow^* e'$ *will not get stuck as a result of a failed predicate of the form* $x = v \in E$*.*

That concludes the work to add support for natural recursion to the core DDPA language and its semantics. The next step is to adapt the analysis accordingly, so that it approximates the run-time behavior.

2.5 Analysis

For the analysis to support recursive functions, the treatment of non-local variable lookup needs modification. The core DDPA strategy for looking up variables not local to a function is to start a subordinate lookup for the function definition and resume the main lookup from there. With the additions of the previous subsections, the definition of such external variable dependencies might occur *after* the definition of the function using them, within the same immediate block. Because the analysis traverses the control-flow graph backwards, it would proceed, eventually reach START, and not find the external dependency. The solution is to REWIND to the end of the immediate block when the subordinate lookup finds the definition of the function whose external dependency the analysis is trying to find.

A REWIND is a JUMP with a predetermined target: a node in the control-flow graph past the end of the current immediate block. REWIND is a new *continuation* in the analysis grammar:

$\hat{k} \quad ::= \quad \ldots \mid \textsc{Rewind} \qquad$ *continuations*

To calculate the target of the JUMP generated by a REWIND, there exists the ENDOFSCOPE function:

Definition 2.17. *Let* $\textsc{EndOfScope}(\hat{a}_1)$ *be the least function such that:*

$$\textsc{EndOfScope}(\hat{a}_1) = \begin{cases} \hat{a}_1 & \text{if } \hat{a}_1 = \textsc{End} \\ & \text{or } \hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}') \\ \textsc{EndOfScope}(\hat{a}_0) & \text{such that } \hat{a}_1 \lll \hat{a}_0 \\ & \text{and } \hat{a}_0 \neq (\hat{x} \overset{\hat{c}\downarrow}{=} \hat{x}'), \\ & \text{otherwise} \end{cases}$$

To handle the REWIND operation, it is necessary to extend the lookup function $\hat{D}(\hat{a}_0, \hat{K}, \hat{C})$. A new condition translates a REWIND into a JUMP with a fixed target determined by the function ENDOFSCOPE:

- If $\hat{K} = [\textsc{Rewind}] \,||\, \hat{K}'$ then
  $\hat{D}(\hat{a}_0, [\textsc{Jump}(\textsc{EndOfScope}(\hat{a}_0), \hat{C})] \,||\, \hat{K}', \hat{C}) \subseteq \hat{V}$.

Finally, we replace the lookup function condition that matches in the case of non-local variable lookup – condition 4d on the definition [5]. The updated condition adds a REWIND continuation onto the stack of operations – it happens right after looking up the definition of a function that has an external dependency:

- If $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_1'' = \hat{x}_2'' \; \hat{x}')$, $\hat{K} = [\hat{x}^{\Pi^+}_{\Pi^-}] \,||\, \hat{K}'$, $\hat{x}'' \neq \hat{x}$, and $\textsc{IsTop}(\hat{c}, \hat{C})$, then
  $\hat{D}(\hat{a}_1, [\hat{x}''^{\emptyset}_{2\,\emptyset}, \textsc{Rewind}] \,||\, \hat{K}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.

With those modifications, the analysis is capable of approximating the run-time of functions that make any external reference allowed by the well-formedness condition. The next section builds on this idea exploring a use case: an object encoding that relies on natural recursion.

2.6 Use case: object encoding

Start with a object-oriented program such as the following Java code:

```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.setX(x); this.setY(y);
    }
    public boolean isZero() {
        return this.magnitude() == 0;
    }
    public double magnitude() {
        return Math.sqrt((this.getX() * this.getX()) +
                         (this.getY() * this.getY()));
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public static void main(String[] args) {
        Point point = new Point(3, 4);
        assert ! point.isZero();
        assert point.magnitude() == 5;
        point.setX(0);
        point.setY(0);
        assert point.isZero();
        assert point.magnitude() == 0;
    }
}
```

It is possible to translate it to DDPA's core language using objects encoded via natural recursion:

```
zero = 0; three = 3; four = 4; five = 5;
ignoredParameter = {};
PointClass = fun pointConstructorParameters -> (
  pointConstructorParameterX = pointConstructorParameters.x;
  pointConstructorParameterY = pointConstructorParameters.y;
  xCell = zero; x = ref xCell; yCell = zero; y = ref yCell;
  pointConstructor = fun pointConstructorIgnored -> (
    pointConstructorSetX = setX pointConstructorParameterX;
    pointConstructorSetY = setY pointConstructorParameterY;
  );
  isZero = fun isZeroIgnored -> (
    isZeroMagnitude = magnitude ignoredParameter;
    isZeroR = isZeroMagnitude == zero;
  );
  magnitude = fun magnitudeIgnored -> (
    magnitudeX = getX ignoredParameter;
    magnitudeY = getY ignoredParameter;
    # calculation ...
  );
  getX = fun getXIgnored -> ( getXR = !x; );
  getY = fun getYIgnored -> ( getYR = !y; );
  setX = fun setXX -> ( setXR = x <- setXX; );
  setY = fun setYY -> ( setYR = y <- setYY; );
  pointConstructorRun = pointConstructor ignoredParameter;
  pointObject = {
    isZero = isZero, magnitude = magnitude,
    getX = getX, getY = getY, setX = setX, setY = setY
  };
);
pointParameters = {x = three, y = four};
point = PointClass pointParameters;
pointSetX = point.setX;
pointSetY = point.setY;
pointIsZero = point.isZero;
pointMagnitude = point.magnitude;
pointIsZeroResult = pointIsZero ignoredParameter;
assertPointIsNotZero = not pointIsZeroResult;
pointMagnitudeResult = pointMagnitude ignoredParameter;
assertPointMagnitude = pointMagnitudeResult == five;
pointModifyX = pointSetX zero;
pointModifyY = pointSetY zero;
assertModifiedPointIsZero = pointIsZero ignoredParameter;
modifiedPointMagnitudeResult = pointMagnitude ignoredParameter;
assertModifiedPointMagnitude =
    modifiedPointMagnitudeResult == zero;
```

The variables `xCell` and `yCell` are necessary to give explicit names to the reference cells, as required by A-Normal Form (ANF). The class becomes a function that returns an object, a record encodes the object, and each field in the record is a function that represents a method. Note that the order of method declarations is not relevant to the correctness of the program. A method is free to call other methods defined after itself – e.g., `isZero` calls `magnitude`. This is possible because they are all in the same immediate block – the class – so they are all accessible to one another by the non-standard notion of scoping presented above.

The introduction of native support for recursion in DDPA allows natural representations of complex programs such as the one above, which encodes objects. At the same time, it reduces the number of call sites that would exist solely to encode recursion – for example, by using self-passing. The result is to reduce the burden on the analysis, which has less call sites to resolve and, consequently, less lookups for the potential functions to WIRE.

More object-oriented programming features are encodable in DDPA's core language: inheritance, interface segregation, dynamic dispatch and so on. The strategies from the literature [3, 8] are applicable. They benefit from the possibility of mutually recursive function definitions, multiplying the gains from the results of this section.

This concludes the exploration of one dimension of practicality: expressiveness. The next section covers performance, the other dimension.

3. Benchmarks

To assess DDPA's performance it is necessary to run its reference implementation [4] on programs designed to stress the analysis – to measure the worst-case behavior – as well as programs that represent real-world usage of programming languages – to measure the average-case behavior. The core DDPA language is too low-level for humans to write significant amounts of code, so the first step was to develop a language at a higher level of abstraction that compiled to the core language. Then, we hand-translated selected benchmarks to this language. Finally, running the benchmarks showed how DDPA compares to other state-of-the-art program analyses.

The following subsections regard this process as well as the results of the tests.

3.1 Swan

Swan is a surface language that compiles to DDPA's core language. This report only contains a high-level explanation of Swan's goals and most important features, not a full specification. It is similar to OCaml in syntax and semantics, which helped in the translation of real-world programs. The most noteworthy Swan feature is the nesting of expressions, instead of having to give each subexpression a unique name. This has interesting implications, for example, it is not necessary to give explicit names to reference cells in stateful code as was the case in the code sample from section 2.6 – in particular, the translation process would generate the variables `xCell` and `yCell`.

Swan also has syntactic sugar for common programming idioms, such as conditionals and multi-way pattern matching; but it is a thin layer above the core language, so it carries along some of the restrictions, for example, variable name uniqueness. All the expressiveness of the core language is available to Swan, including the natural recursion extension from section 2. Some data structures are unique to Swan, they translate to the core language via encodings. For example, Swan supports *lists* both as values and patterns, which translate to records in the core language – the translation uses *cons* cells, akin to Scheme. Finally, Swan allows for binding variables in patterns.

Combined, these features result in high-level, readable code, as the following snippet illustrates (except for the explicit `end` in blocks, the code is also valid OCaml):

```
let takef (takefList, takefFunction) =
  match takefList with
  | [] -> []
  | takefListHead :: takefListTail ->
    if takefFunction takefListHead then
      takefListHead :: takef (takefListTail, takefFunction)
    else
      []
    end
  end
in
# ...
```

The equivalent program in the core language spans dozens of lines of code. With Swan, it was possible to hand-translate programs relevant to testing DDPA's performance. The next subsection explains the decision process behind the benchmarks selection and how they exercise different aspects of the analysis.

3.2 Translated programs

We hand-translated programs to Swan in order to fulfill two goals: to stress particular aspects of the analysis and to simulate real-world usage of a programming language. Put together, they exercise all of DDPA's features, including the natural recursion extension from the previous section. To facilitate the comparison with other program analyses, some of the selected programs came from their benchmark suites; others we wrote from scratch in *idiomatic Swan*, as a way to check the analysis behavior in a scenario closer to the real-world.

Here are the programs that compose the benchmark suite:

ack (from Larceny R6RS and [7]) A highly recursive function on numbers. Uses natural recursion.

alex (original) Uses lists and the application of higher-order recursive functions on them. Explores the precision of the representation of recursion in the analysis.

blur (from [7]) A polymorphic recursive function. Also explores the precision of the representation of recursion in the analysis.

church (from [12]) Church encoded numerals and operations on them. Checks the distributive property of multiplication over addition. Uses higher-order and polymorphic recursive functions. Stresses the analysis ability to handle function application and its representation of recursion.

cpstak (from Larceny R6RS and [7]) Continuation passing style (CPS) version of *tak*. Stresses the context stack due to the use of continuations.

eta (from [7]) An identity function that makes an spurious function call before returning. Assesses the analysis ability to handle the context stack without losing precision in irrelevant function calls.

kcfa-generator (from [7]) Generates growing programs that represent the worst-case for k-CFA based on *kcfa2* and *kcfa3*. Explores the analysis ability to handle variables non-local to the bodies of arbitrarily nested functions.

loop2 (from [7]) Nested loops. Uses natural recursion.

mbrotZ (from [12]) Calculates numbers in the Mandelbrot set. Uses arithmetical operations (e.g., multiplication defined as a recursive function that performs repeated addition), complex numbers (encoded as pairs of numbers) and vectors (encoded as lists of reference cells). It is a big example that explores different aspects of the analysis, resembling a real-world program.

mj09 (from [7]) Higher-order functions on numbers.

recursion (original) The simplest non-trivial recursive function.

sat (from [7]) A SAT solver. It is a worst-case scenario for the analysis due to the exponential nature of the problem.

state (from [10]) Operations on reference cells. Exercises the support for state and how it interacts with natural recursion.

tak (from [6, 7]) A highly recursive function on numbers. Uses natural recursion.

growing programs (original) A test generator that produces copies of the source of another program. Exercises the analysis behavior over controlled program growth. Used with *sat* – a demanding program – and *eta* – representative of the class of non-trivial average programs.

growing number of calls (original) A generator for tests with growing number of function calls. It differs from the previous *growing programs* generator by not copying the whole source, but only generating more function calls to the same function. Used with the identity function as a baseline and a function that requires the natural recursion feature from the previous section for stress testing.

DDPA performance tests ran over the programs listed above. The next sections discuss which experiments we ran and their results, and concludes with an analysis that places DDPA in the space of program analysis in terms of practicality.

3.3 Experiments

The biggest challenge in testing the performance of program analyses is comparing expressiveness. Different analyses are able to express different features, which leads to discrepancies in precision. Even within the same program, one analysis might be able to perfectly capture one function and over-approximate another; and a second analysis might do the exact opposite. This project does not aim to settle the issue, which often has no clear solution. The comparisons solely rely in similarities that allow for measurements of related properties. Then, the main measurements of interest are execution time and number of explored states. The *states* in different analyses mean different things, so they are not directly comparable. But, within experiments in the same analysis, they give a good indication of the complexity of the algorithm, both in terms of space and time.

The analyses to which DDPA compares are P4F [7, 19] and OAAM [11, 12]. They are state-of-the-art program analyses from the same space as DDPA: context-sensitive control-flow analyses for higher-order functional programming languages. Note that the analyses' implementations do not use the same language: DDPA's reference implementation [4] is in OCaml; P4F's [19], in Scala; and OAAM's [11], in Racket. But all these programming languages fare within the same order of magnitude performance-wise, so the experiments disregarded the differences.

The following is a description of the performed experiments. The experiments ran on an Intel Xeon (3.10GHz) with 8GB of RAM, the operating system was the Debian 8.0 distribution of GNU/Linux:

baseline Disabled context-sensitivity, i.e., $[k = 0]$-DDPA, $[k = 0]$-P4F and $[k = 0]$-OAAM. (The meaning of $[k = 0]$-P4F and $[k = 0]$-OAAM is $[k = 0]$-CFA in those systems.) In this scenario, all three analyses lose call-return alignment or the equivalent thereof for a theory based on *continuation passing style* (CPS).

higher k The precision of $[k = 1]$-DDPA and $[k = 1]$-P4F do not compare, the latter is able to capture the meaning of programs that the former cannot. The most fair comparison is to choose $k = 1$ for P4F and the lowest k necessary for perfect precision on DDPA – we did that for all examples except the biggest, *church* and *mbrotZ*, in which the parameter was $k = 1$ for both analyses. OAAM is not in the experiment because its reference implementation only supports $k = 0$. The only programs exercised in this experiment are those for which a higher k yields better precision – there were benchmarks for which $k = 0$ was enough to achieve perfect precision and there were benchmarks for which no choice of k would result in perfect precision; in those cases, the results of the previous experiment suffice.

growing k To measure the effect of increasing the context sensitivity, the same program *tak* ran through the analysis with higher choices for k, ranging from 1 to 5. Only DDPA participates in this experiment because the reference implementations for P4F and OAAM do not support higher choices of k.

growing programs For a k big enough to achieve perfect call-return alignment, vary the program size. The variations result from copying the whole source of the program and renaming the variables accordingly. The programs used in this experiment were *sat* – a worst-case scenario – and *eta* – an example of non-trivial but average scenario. OAAM is not in the experiment because its reference implementation only supports $k = 0$.

growing function calls A variation of the previous experiment in which the function bodies are not copied, only the function calls are. This allows for sharing of the function body in the states of the analysis. The programs in this experiment were an identity function – the baseline – and a function with natural recursion that performs multiplication in terms of repeated addition – a stress test. The only analysis benchmarked in this experiment was DDPA because natural recursion is a feature particular to it.

natural recursion To assess the impact of adding support for recursion in DDPA – see previous section – the same program ran with the feature enabled and disabled. This experiment uses the *sat* program because it is a stress test that does not include recursive functions, so no loss in precision occurs. Only DDPA is in this experiment, as it covers a feature particular to it.

The next section analyzes the results of these experiments.

3.4 Results

Figures 3.1–3.6 list the results of the experiments. The rest of this section analyzes them, situating DDPA among other program analyses.

The first experiment – *baseline*, in figure 3.1 – shows that DDPA is competitive with P4F and OAAM in terms of performance. On some programs it is faster (e.g., *eta*; *mj09*; and, most notably, *sat*); on some programs it is slower (e.g., *csptak*); but, on average, it stays within the same order of magnitude. The outstanding bad result in this experiment is *cpstak*, which reveals DDPA does not perform well on programs in continuation passing style. The reason is CPS introduces extra deeply nested function calls, which causes DDPA to perform more function lookups and exhausts its context stack faster. This is not a big concern, though, since programmers tend not to write programs in that style, only compilers commonly generate CPS code.

The results from OAAM show almost no difference across the benchmark suite. The reason is none of them significantly exercise OAAM, a task which requires bigger programs [12] – up to two orders of magnitude bigger than the ones used in the reported experiments. We conjecture that OAAM is not as expressive as DDPA and P4F in these programs, but unfortunately the author could not find a way to inspect the outputs of OAAM's reference implementation in that dimension, so the actual reason remains an open question.

In the *higher* k experiment, when increasing k such that the analysis achieves perfect precision in the small programs – all except *church* and *mbrotZ* – DDPA is consistently better than P4F – though not by a statistically significant amount. The only exception is *sat*, in which DDPA's bad performance is probably explained by the introduction of support for natural recursion – see discussion below about the last experiment.

Inspection of DDPA's outputs for recursive programs revealed an issue in the treatment of *context stacks*. The analysis generates all possible context stacks involving the recursive function. For example, suppose that $k = 4$ and `f` is a recursive function, then DDPA generates the context stacks `f`, `ff`, `fff` and `ffff`. If `f` and `g`

	DDPA				P4F			OAAM		
Program	**CFG nodes**	**PDR nodes**	**PDR edges**	**Time (ms)**	**States**	**Edges**	**Time (ms)**	**States**	**Points**	**Time (ms)**
ack	69	817	21433	605	17	20	452	186	50	4721
alex	61	422	5452	177	—	—	—	—	—	—
blur	66	602	10160	295	58	66	496	141	59	3872
cpstak	135	4348	271245	50663	53	68	705	5	5	1359
eta	25	106	842	31	26	25	140	60	28	1166
kcfa2	47	232	1955	53	35	34	114	201	50	1187
kcfa3	60	332	2934	86	53	52	156	427	62	1164
loop2	93	939	21609	862	46	49	527	142	50	1152
mj09	36	177	1568	51	32	31	104	98	42	1149
recursion	23	121	1007	22	—	—	—	—	—	—
sat	79	545	7181	320	248	311	2374	460	87	1169
state	27	82	686	17	—	—	—	—	—	—
tak	81	1610	72070	8343	29	36	425	208	56	1168
church	155	4456	304437	27509	73486	103629	1801462	6273	292	1286
mbrotZ	807	19913	2760869	663718	—	—	—	59	59	1163

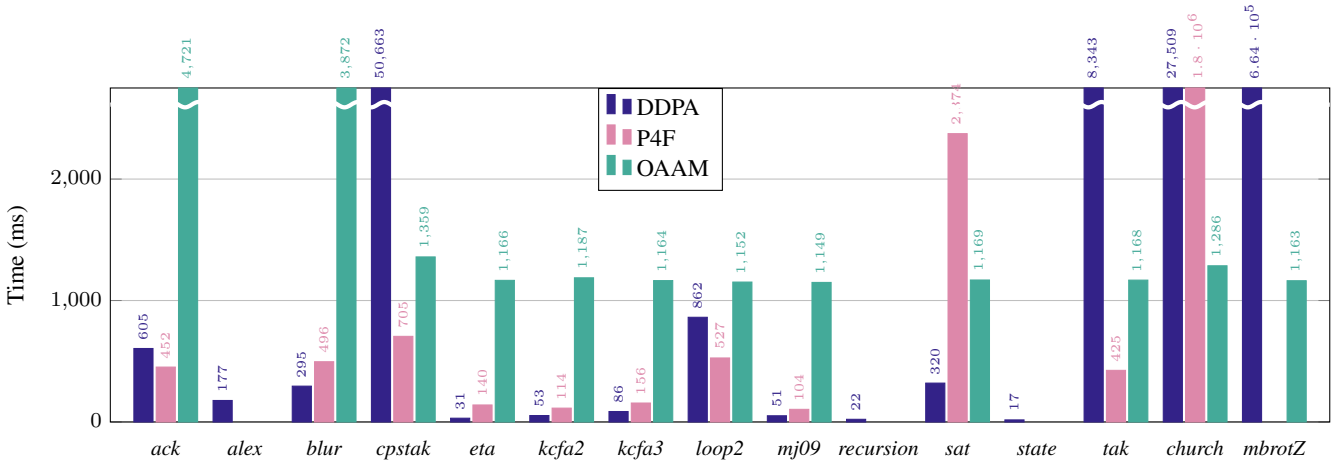

Figure 3.1. Results of the *baseline* experiment. In all experiments, $k = 0$. The time measurements for DDPA are the average of 100 runs. Measurements for the programs *alex*, *recursion* and *state* in P4F and OAAM are not available because these programs are original to Swan, and Scheme translations do not exist. Measurements for *mbrotZ* in P4F are not available because the analysis does not support the features necessary to analyze the program – e.g., complex numbers and mutable vectors.

are mutually recursive functions, the problem gets worse, as DDPA explores the context stacks `f`, `ff`, `fg`, `fgfg`, `fggf`, `fggg`, and so on. A new context stack model to address this problem is future work.

The next experiment – *growing k* – shows how DDPA performs in the same program – *tak* – with increasing choices for k. An exponential blowup is noticeable – see graph in figure 3.3 – but it does not contradict the polynomial algorithm complexity associated with DDPA: the assertion about complexity refers to variable program size with a fixed k, not the other way around.

That opposite scenario is the subject of the next experiment – *growing programs*. In the worst case – *sat* – DDPA and P4F behave similarly. But in the average case – *eta* – DDPA is consistently better than P4F. The difference is evident in the *eta* × *32* test, which P4F could not complete due to exhausting resources – the run-time threw an *StackOverflowError* – and DDPA analyzed under 20 seconds.

A variation in the previous experiment is to add more calls to the same function, without copying the function body – *growing function calls*. What stands out in the results of this experiment is that, in *recursive*, while the number of *control-flow graph* (CFG) nodes increases linearly with the number of function calls, the execution time grows super-linearly. The effect is more pronounced when $k = 1$ than when $k = 0$. The overhead suggests the implementation's performance degrades when the scale of the work gets larger.

Another noteworthy outcome of this experiment is the difference between $k = 0$ and $k = 1$ in the *identity* programs: when $k = 1$, DDPA runs *faster* than when $k = 0$ – see graph in figure 3.5. Both achieve perfect precision, context sensitivity is not necessary to analyze these examples. But $k = 0$ seems to generate spurious paths during lookup, which it then has more work to check and invalidate. This is an instance of the theoretically predicted effect of better precision yielding better performance.

Finally, the last experiment – *natural recursion* – shows the overhead of introducing support for natural recursion in DDPA. Note that *sat* does not contain recursive functions, so disabling the feature has no effect in the precision. Compare the results to those of the *growing programs* experiment – see graph in figure 3.6 – the feature causes an average slowdown of 192%. The effect is present even in programs that contain no recursion, because, as initially proposed, the REWIND continuation appears after the lookup of *any* variable non-local to a function – including the cases in which such continuation is not necessary. A refined revision of this feature addressing this issue is future work – see section 5.

The exploration of the performance of the analysis shows that it has potential to be practical. It fares within the same performance

	DDPA					P4F			
Program	k	**CFG nodes**	**PDR nodes**	**PDR edges**	**Time (ms)**	k	**States**	**Time Edges**	**(ms)**
eta	1	25	115	907	29	1	26	25	113
kcfa2	5	47	240	1969	98	1	35	34	141
kcfa3	7	60	340	2948	176	1	53	52	180
sat	4	79	561	7237	1528	1	65	67	327
church	1	155	5040	274352	35109	1	39218	69669	1841680
mbrotZ	1	807	37552	6218451	1943991	—	—	—	—

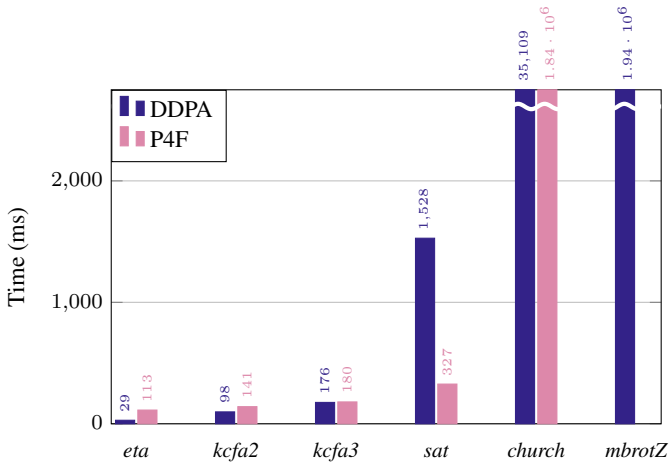

Figure 3.2. Results of the *higher k* experiment. The time measurements for DDPA in the programs *eta*, *kcfa2*, *kcfa3* and *sat* are the average of 10 runs. Measurements for *mbrotZ* in P4F are not available because the analysis does not support the features necessary to analyze the program – e.g., complex numbers and mutable vectors.

range as the other state-of-the-art control-flow analyses and can be faster in some examples. The benchmark also pointed at the points that need improvement and suggested directions for future work.

This concludes the presentation of the main contributions of the reported project. The following sections explores related and future work.

4. Related work

The works most closely related to DDPA are P4F [7] and OAAM [12], both of which are control-flow analyses for higher-order functional programming languages. They take a forwards approach to the task, defining abstract interpreters [1] that approximate the program run-time. Thus, they generate spurious states that can never happen during execution, which prompts the necessity of abstract garbage collection techniques [13, 15]. DDPA, in opposition, performs lookups by traversing the control-flow graph *backwards*, in a demand-driven fashion. Consequently, *by construction* it does not generate spurious states and does not need abstract garbage collection.

DDPA is not the only program analysis that performs lookups in a demand-driven way. CFL-reachability [9, 18], an analysis for first-order languages, for example, uses the same strategy. But it is not as expressive as DDPA, lacking flow-sensitivity and the treatment of non-local variables, among other features. So it is possible to think of DDPA as an extension of the CFL-reachability analyses to the realm of higher-order languages.

From this perspective, DDPA is bridging the gap between techniques that come from first-order program analyses and the research in higher-order program analyses. It solves problems from the same space as k-CFA [20] and related developments [14–16]. But it is a polynomial algorithm for a fixed k [17], instead of being exponential like k-CFA [21].

The object encoding is related to [3, 8]. Those works go beyond the simple objects presented in this report, including features such as interface segregation and inheritance. It is possible to bring those encodings to DDPA, exploring the expressiveness of the core language and analysis – in particular, the expressiveness of the natural recursion extension.

5. Future work

This project's goal was to assess the practicality of DDPA. The exploration that resulted from this pursuit gives insight into how the analysis works in complex use cases and suggests possible improvements. The first and most outstanding improvement is to bring even more tests and benchmarks to the system: translating more programs and writing more Swan code, not limited to programs designed to stress certain parts of the analysis, but also real-world applications.

In the same direction, another venue of future work is to extend Swan and the core language, adding support for more data types such as complex numbers and vectors, more control-flow constructs such as exceptions and iterations, more system capabilities such as input/output to the terminal, more object-oriented programming features such as interface segregation, more operations such as mathematical functions and record concatenation, and so on. (Record concatenation is of special interest because it allows for a natural encoding of object extensions – e.g., inheritance.) A long-term goal is to extend Swan to a point where it is a fully-featured language for real-world programs, as that would be the ideal testbed for the analysis. Alternatively, we propose a different front-end for the core language, targeting an existing language – for

k	CFG nodes	PDR nodes	PDR edges	Time (ms)
1	27	1307	15135	704
2	27	5837	80442	5335
3	27	27112	480811	83536
4	27	138392	3312841	1692891
5	—	—	—	>171244463

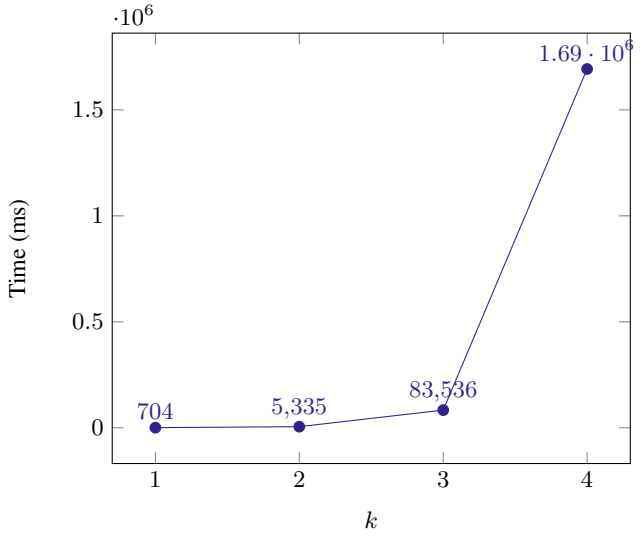

Figure 3.3. Results of the *growing* k experiment. All experiments ran the *tak* program. Measurements for $k = 5$ are not available because the execution did not terminate before the reported time. The graph shows the exponential blowup.

example, JavaScript. It would compile existing code into the core language and allow for analysis of programs in the wild.

The discussion thus far has focused in expressiveness. There are also possibilities of future work in the dimension of performance. For example, the experiments showed that adding natural recursion had a negative impact on the execution time on non-recursive programs. To address the issue, we could use the binding information collected during the well-formedness check to inform the analysis. As presented in section 2, the analysis pushes the REWIND continuation to the stack in the treatment of *any* non-local variable lookup, including those that do not need it because they do not use natural recursion. Whether a non-local variable needs the natural recursion feature or not is statically determined, as part of the well-formedness condition and this information could be part of the analysis so it would only REWIND where necessary.

With that binding structure available to the analysis, a further optimization is possible: instead of REWIND meaning a JUMP to the end of the immediate block, it could be a JUMP targeted directly to the occurrence of the non-local definition. This has two consequences: the first is saving the steps during lookup that would traverse the graph to find the definition; the second is to remove the artificial restriction that natural recursion can only occur within the same immediate block – functions could refer to variables beyond the definition block, as long all their dependencies are available upon first use.

Finally, future work includes closing the gaps in the core analysis' expressiveness. DDPA as it stands is not able to capture the meaning of some programs from the benchmark suite, regardless of the choice for k. One known reason is polymorphic recursion, of which the *predecessor* function in the *church* program is an example. The possible solution is to use a subgraph of the control-flow graph as a replacement for the *context stack* in DDPA; then the analysis would not exhaust the stack upon recursion and, thus, would not lose precision. It is still uncertain whether this fixes the issue and whether there are other sources of incompleteness.

6. Conclusion

This project explored the practicality of DDPA. Two dimensions of practicality were relevant: expressiveness and performance. For expressiveness, we extended DDPA's core language and analysis to support natural recursion – a feature that allowed for simpler definitions of recursive and mutually recursive functions, with less call sites. This work did not require changes to the rules in the operational semantics from [17], Section 4.1, which already behaved in a way that supported natural recursion. It was necessary to modify only the well-formedness condition and the analysis. It is possible to leverage natural recursion to write an encoding of simple objects that resembles an equivalent Java code, attesting to the enhanced expressiveness of DDPA with the extension.

In the second dimension of practicality – performance – tested the performance of the DDPA reference implementation. To make tractable the task of writing large programs, we created a higher-level language that compiles to the core language – Swan. Then, we hand-translated benchmarks used by other program analyses and developed applications that resemble real-world usage of a programming language. Finally, we ran a series of experiments that included comparisons with other state-of-the-art program analyses – P4F and OAAM.

The experiments showed that DDPA is on par with other program analyses in terms of execution time and precision. Some test cases resulted in orders-of-magnitude improvements. It also revealed that the analysis is not able to express some programs, in particular those that use polymorphic recursion. Furthermore, recursion makes the analysis exhaust the context stack, losing precision and generating spurious states along the way. And the natural recursion extension adds a penalty in execution time to programs that do not include recursive functions.

There are venues of future research in both aspects of practicality. The project's work revealed lacks in precision and performance bottlenecks. From getting a better understanding of how the analysis behaves in real-world programs, we also glanced at the possible solutions for the reported issues.

Acknowledgments

Thanks to Dr. Scott F. Smith and Dr. Zachary Palmer for discovering DDPA, and giving support and feedback. Thanks to Hari Menon and Alex Rozenshteyn for the conversations about program analysis.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, 1977.

[2] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN 0596517742.

[3] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *OOPSLA Conference Proceedings*, pages 16–30, 1994.

[4] L. Facchinetti, Z. Palmer, and S. F. Smith. Higher-order demand-driven program analysis—Implementation. `https://github.com/JHU-PL-Lab/odefa`.

	DDPA					P4F				
Program	k	**CFG nodes**	**PDR nodes**	**PDR edges**	**Time (ms)**	k	**States**	**Time Edges**	**(ms)**
sat × *1*	4	35	205	2224	889	1	65	67	402
sat × *2*	4	68	461	5014	1420	1	134	139	1332
sat × *4*	4	134	1135	12322	4073	1	272	283	6704
sat × *8*	4	266	3131	33850	13011	1	548	571	52949
sat × *16*	4	530	9715	104554	47682	1	1100	1147	536471
sat × *32*	4	1058	33251	356554	188304	1	1456	1512	1800579
eta × *1*	1	18	103	775	69	1	26	25	55
eta × *2*	1	34	225	1740	98	1	60	59	264
eta × *4*	1	66	535	4270	293	1	128	127	475
eta × *8*	1	130	1419	11730	917	1	264	263	1332
eta × *16*	1	258	4243	36250	4130	1	536	535	6915
eta × *32*	1	514	14115	123690	19654	—	—	—	—

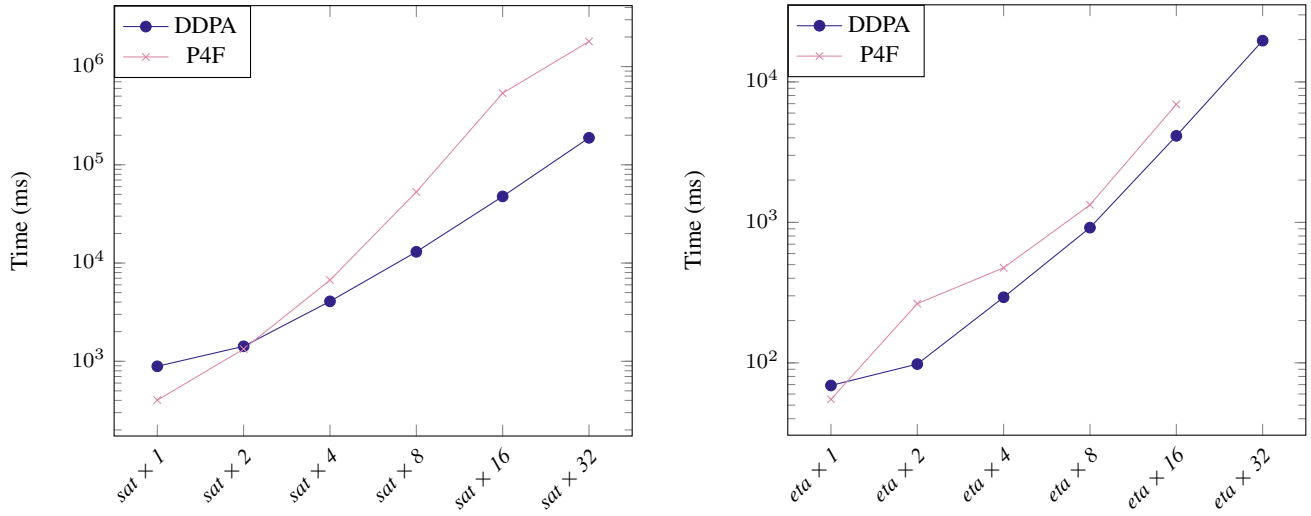

Figure 3.4. Results of the *growing programs* experiment. The difference between the results for the programs *sat* × *1* and *eta* × *1* and those from the *higher k* experiment happened because in *higher k* we ran Swan programs, and in this experiment we ran programs directly hand-translated to the core DDPA language. Measurements for *eta* × *32* in P4F are not available because the execution terminated with a *StackOverflowError*.

[5] L. Facchinetti, Z. Palmer, and S. F. Smith. Implementing higher-order demand-driven program analysis—Draft. The Johns Hopkins University Programming Languages Laboratory, Mar 2016.

[6] R. P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1985. ISBN 9780262571937.

[7] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. Van Horn. Pushdown control-flow analysis for free. In *POPL*, 2016.

[8] M. Grant, Z. Palmer, and S. Smith. *Principles of Programming Languages*. `http://www.cs.jhu.edu/~scott/pl/book/dist`, 2010.

[9] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, 1995.

[10] S. Jagannathan, P. Thiemann, S. Weeks, and A. K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *POPL*, 1998.

[11] J. I. Johnson, N. Labich, M. Might, and D. V. Horn. Optimizing abstract abstract machines—Implementation. `https://github.com/dvanhorn/oaam/tree/7823131`.

[12] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. ICFP. ACM, 2013.

[13] J. I. Johnson, I. Sergey, C. Earl, M. Might, and D. V. Horn. Pushdown flow analysis with abstract garbage collection. *JFP*, 2014.

[14] M. Might and O. Shivers. Environment analysis via ΔCFA. In *POPL*, 2006.

[15] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.

[16] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI*, 2010.

[17] Z. Palmer and S. Smith. Higher-order demand-driven program analysis. In *ECOOP*, 2016.

[18] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, New York, NY, USA, 2001.

[19] I. Sergey, C. Earl, M. Might, D. V. Horn, S. Lyde, T. Gilray, and M. D. Adams. Pushdown control-flow analysis for free—Implementation. `https://bitbucket.org/ucombinator/p4f-prototype`.

[20] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

[21] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP*, 2008.

Program	k	CFG nodes	PDR nodes	PDR edges	Time (ms)
identity × *1*	0	10	14	110	45
identity × *2*	0	14	30	243	57
identity × *4*	0	22	66	565	76
identity × *8*	0	38	138	1325	165
identity × *16*	0	70	282	3229	679
identity × *32*	0	134	570	8573	3140
identity × *64*	0	262	1146	25405	14869
identity × *128*	0	518	2298	83645	69036
identity × *1*	1	10	14	110	39
identity × *2*	1	14	30	243	45
identity × *4*	1	22	68	565	47
identity × *8*	1	38	144	1325	141
identity × *16*	1	70	296	3229	447
identity × *32*	1	134	600	8573	1899
identity × *64*	1	262	1208	25405	9178
identity × *128*	1	518	2424	83645	40643
recursive × *1*	0	71	648	22694	1716
recursive × *2*	0	80	768	25726	2624
recursive × *3*	0	89	888	28720	3980
recursive × *4*	0	98	1008	31732	5187
recursive × *5*	0	107	1128	34762	6761
recursive × *6*	0	116	1248	37810	8477
recursive × *7*	0	125	1368	40876	10729
recursive × *8*	0	134	1488	43960	12905
recursive × *9*	0	143	1608	47062	15211
recursive × *10*	0	152	1728	50182	18090
recursive × *1*	1	71	649	22694	4233
recursive × *2*	1	80	771	25726	6424
recursive × *3*	1	89	893	28720	9074
recursive × *4*	1	98	1015	31732	11592
recursive × *5*	1	107	1137	34762	14607
recursive × *6*	1	116	1259	37810	18128
recursive × *7*	1	125	1381	40876	21779
recursive × *8*	1	134	1503	43960	25962
recursive × *9*	1	143	1625	47062	30602
recursive × *10*	1	152	1747	50182	35338

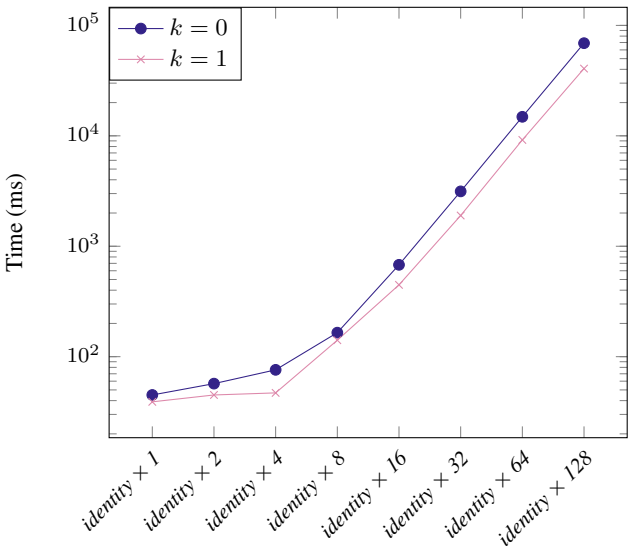

Figure 3.5. Results of the *growing function calls* experiment. The graph shows a case in which $k = 1$ is *faster* than $k = 0$.

Program	CFG nodes	PDR nodes	PDR edges	Time (ms)
sat × *1*	35	130	1372	661
sat × *2*	68	279	2916	1241
sat × *4*	134	631	6490	2832
sat × *8*	266	1551	15582	7015
sat × *16*	530	4255	41542	19135
sat × *32*	1058	13119	124566	58189

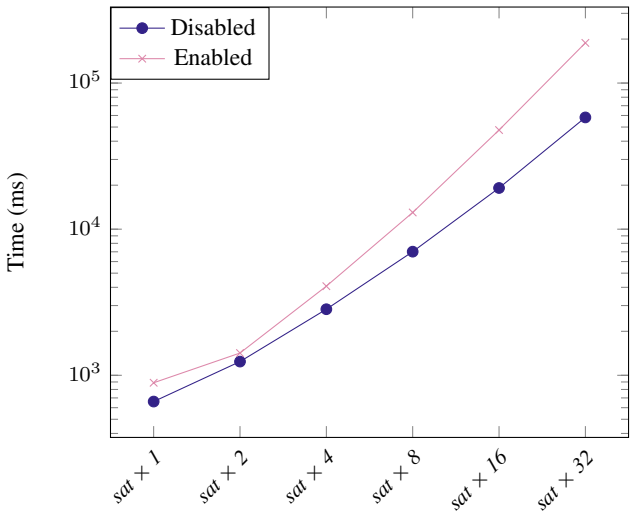

Figure 3.6. Results of the *natural recursion* experiment. In all experiments, $k = 4$. The graph shows the impact of natural recursion on the analysis performance, comparing with results from figure 3.4.