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:

\[
\begin{align*}
1 & \text{pp} = \text{fun } \text{fp} \rightarrow ( \\
2 & \text{fx} = \text{fun } x \rightarrow ( \\
3 & \text{frx} = \text{fp } fp; \\
4 & \text{fr} = \text{fx } x; \\
5 & )); \\
6 & ); \\
7 & f = \text{pppp } fp; \\
8 & a = 0; \\
9 & r = f a; \\
\end{align*}
\]

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

\[
\begin{align*}
1 & f = \text{fun } x \rightarrow ( \\
2 & \text{fr} = f x; \\
3 & ); \\
4 & a = 0; \\
5 & r = f a; \\
\end{align*}
\]

Note that the original program contained four function calls – at \( \text{frx}, \text{fr}, f \) and \( r \) – while the rewritten program contains only the two the programmer intended the write – at \( \text{fr} \) 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

\[ \text{frx} = \text{fp } \text{fp} \]

\[ \text{fr} = \text{frx } x; \]

\[ f = \text{pppp } \text{fp}; \]

\[ a = 0; \]

\[ r = f a; \]

\[ f = \text{fun } x \rightarrow ( \]

\[ \text{fr} = f x; \]

\[ ); \]

\[ a = 0; \]

\[ r = f a; \]

\[ \text{frx} = \text{fp } \text{fp} \]

\[ \text{fr} = \text{frx } x; \]

\[ f = \text{pppp } \text{fp}; \]

\[ a = 0; \]

\[ r = f a; \]

\[ f = \text{fun } x \rightarrow ( \]

\[ \text{fr} = f x; \]

\[ ); \]

\[ a = 0; \]

\[ r = f a; \]
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:

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

The binding of the variable \( f \) occurs after considering the function body \( \text{fun } x \rightarrow (\ldots) \), which uses \( f \). Nevertheless, the program is well-formed: the interpreter does not have a problem with it because it processes function calls by \textit{lining} the function body at the call site. In the example above, it means the program point \( r \) is \textit{lined} 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:

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

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:

```ml
f = fun x -> ( b = a; ) ;
```

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:

```ml
f = fun x -> ( b = a; # ILL-FORMED ) ;
```

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 \textit{frozen} in another function body. The code below exemplifies such a case:

```ml
f = fun x -> ( g = f; # ILL-FORMED a = 0;
```

```
```
Consider the following real-world examples, starting with a recursive function that finds out if a number is even or odd with mutually recursive functions:

```plaintext
# let rec sum = fun number ->
#   if number = 0 then 0
# else
#   if number = 0 then false
# else odd (n - 1)
# ;
#    even 0;; (* => true *)
#    even 1;; (* => false *)
#    even 6;; (* => true *)
#    odd 0;; (* => false *)
#    odd 1;; (* => true *)
#    odd 3;; (* => false *)
#    odd 5;; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
# 1 # sum 5 (* => 15 *)
```

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:

```plaintext
# let rec sum = fun number ->
#   if number = 0 then 0
# else
#   if number = 0 then false
# else odd (n - 1)
# ;
#    even 0;; (* => true *)
#    even 1;; (* => false *)
#    even 6;; (* => true *)
#    odd 0;; (* => false *)
#    odd 1;; (* => true *)
#    odd 3;; (* => false *)
#    odd 5;; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
# 1 # sum 5 (* => 15 *)
```

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:

```plaintext
# let rec sum = fun number ->
#   if number = 0 then 0
# else
#   if number = 0 then false
# else odd (n - 1)
# ;
#    even 0;; (* => true *)
#    even 1;; (* => false *)
#    even 6;; (* => true *)
#    odd 0;; (* => false *)
#    odd 1;; (* => true *)
#    odd 3;; (* => false *)
#    odd 5;; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
#    # even 6 +; (* => true *)
#    # odd 6 +; (* => false *)
# 1 # sum 5 (* => 15 *)
```

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:
evenzero = even zero; # => true
oddzero = odd zero; # => false
eventhree = even three; # => false
oddthree = odd three; # => true
evensix = even six; # => true
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 \( \vec{x} = [x, \ldots] \) be a list of variables. Let \( x \in \vec{x} \) iff \( \vec{x} = [x, \ldots] \). Similarly, let \( c \in e \) iff \( e = \ldots, c, \ldots \).

The \( \text{Fr} \) – “flatten” – function \text{linearizes} the expression, \text{pulling} the clauses nested within function bodies and conditionals to the top level:

Definition 2.2. Let \( \text{Fr} \) be the least function such that:

\[
\begin{align*}
\text{Fr}([]) &= []; \\
\text{Fr}(c || e) &= \begin{cases} 
\text{Fr}(e') || e & \text{if } c = (x = \text{fun } x' \Rightarrow (e')) \\
\text{Fr}(e_1) || \text{Fr}(e_2) & \text{if } c = (x = x' - p ? f_1 : f_2), \\
& \text{where } f_1 = \text{fun } x_1 \Rightarrow (e_1), \text{ and } f_2 = \text{fun } x_2 \Rightarrow (e_2) \\
\text{Fr}(e) & \text{otherwise}
\end{cases}
\end{align*}
\]

The \( \text{Fr}^i \) – “flatten immediate block” – function \text{linearizes} the immediate block expression, \text{pulling} the clauses nested within conditionals – but not functions – to the top level (note that \( \text{Fr}^i(e) \subseteq \text{Fr}(e) \)).

Definition 2.3. Let \( \text{Fr}^i \) be the least function such that:

\[
\begin{align*}
\text{Fr}^i([]) &= []; \\
\text{Fr}^i(c || e) &= \begin{cases} 
\text{Fr}^i(e_1) || \text{Fr}^i(e_2) & \text{if } c = (x = x' - p ? f_1 : f_2), \\
& \text{where } f_1 = \text{fun } x_1 \Rightarrow (e_1), \text{ and } f_2 = \text{fun } x_2 \Rightarrow (e_2) \\
\text{Fr}^i(e) & \text{otherwise}
\end{cases}
\end{align*}
\]

The \text{Defines} function returns the variables immediately bound by an expression, shallowly traversing its list of clauses:

Definition 2.4. Let \text{Defines} be the least function such that:

\[
\begin{align*}
\text{Defines}([]) &= []; \\
\text{Defines}(x = b || e) &= x || \text{Defines}(e).
\end{align*}
\]

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

Definition 2.5. Let \text{Bindings}(e) = \text{Defines}(e') || \text{Bindings'}(e')
where \( e' = \text{Fr}(e) \) and \text{Bindings'} is the least function such that:

\[
\begin{align*}
\text{Bindings'}([]) &= []; \\
\text{Bindings'}((x = \text{fun } x' \Rightarrow (e')) || e) &= x' || \text{Bindings'}(e); \\
\text{Bindings'}((x = x' - p ? \text{fun } x_1 \Rightarrow (e_1) : \text{fun } x_2 \Rightarrow (e_2)) || e) &= [x_1, x_2] || \text{Bindings'}(e); \\
\text{Bindings'}(c || e) &= \text{Bindings'}(e), \text{otherwise}.
\end{align*}
\]

The \text{Uses} function returns all variables used by an expression, deeply traversing its syntax tree:

Definition 2.6. Let \text{Uses}(e) = \text{Uses}'(\text{Fr}(e))
where \text{Uses}' is the least function such that:

\[
\begin{align*}
\text{Uses}'([]) &= []; \\
\text{Uses}'((x' = \{x_1, x_2, \ldots, x_n\}) || e) &= [x_1, x_2, \ldots, x_n] || \text{Uses'}(e); \\
\text{Uses}'((x' = \text{fun } x \Rightarrow (e')) || e) &= \text{Uses'}(e); \\
\text{Uses}'((x' = \text{ref } x) || e) &= x || \text{Uses'}(e); \\
\text{Uses}'((x' = x) || e) &= x || \text{Uses'}(e); \\
\text{Uses}'((x' = x_1 \leftarrow x_2) || e) &= [x_1, x_2] || \text{Uses'}(e); \\
\text{Uses}'((x' = ! x) || e) &= x || \text{Uses'}(e); \\
\text{Uses}'((x' = \top x) || e) &= x || \text{Uses'}(e); \\
\text{Uses}'((x' = \bot x) || e) &= x || \text{Uses'}(e); \\
\text{Uses}'((x' = x_1 \circ x_2) || e) &= [x_1, x_2] || \text{Uses'}(e); \\
\text{Uses}'((x' = x_1 \cdot (x_2)) || e) &= [x_1, x_2] || \text{Uses'}(e).
\end{align*}
\]

The \text{Unique} predicate holds when there are no repetitions among the variables:

Definition 2.7. Let \text{Unique} be the least predicate satisfying the following assertions:

\[
\begin{align*}
\text{Unique}([]); \\
\text{Unique}(x || \vec{x}) \iff x \notin \vec{x}, \text{ and Unique}(\vec{x}).
\end{align*}
\]

The definitions that follow all assume that \text{Unique}(\text{Bindings}(e)) hold and that, when given an expression \( e \) and a clause \( c \), then the clause occurs in the expression: \( c \in \text{Fr}(e) \). The \( x_d \prec_c x_r \prec \ldots \prec_c x_d \) 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 \( x_d \prec_c x_r \) relation hold iff \( c \in \text{Fr}(e), \ c = (x_d = b), \ x_r \in \text{Uses}(c), \) and either \( x_d = x_r \) or \( x_r \notin \text{Bindings}(e) \).

The \( \prec \) relation is the transitive closure of \( \prec_c \):

Definition 2.9. Let \( x_1 \prec_c x_2 \) hold iff \( x_1 \prec \ldots \prec_c x_2 \).

The \text{Dep} – “dependencies” – function returns the variables upon which a clause depends. Those are the variables that \text{must} be in scope in a well-formed clause; the variables that are immediate requirements and are not \text{frozen} (note that \text{Dep} \subseteq \text{Uses}(e)). Note that the only construct capable of \text{freezing} variable uses is a function declaration.

Definition 2.10. Let \text{Dep} be a function such that:
Definition 2.11. Let $UFD$ be a function such that:

$$UFD_e(x) = \begin{cases} 
  e & \text{if there does not exist } c \in e \\
  e_1 & \text{where } e = e_1 \mid c \mid e_2, \\
  x \in \text{Dep}_c(e), \text{ and} \\
  \text{there does not exist } c_1 \in e_1 \\
  \text{such that } x \in \text{Dep}_{e}(c_1) 
\end{cases}$$

The $\text{Avail}^r$ – “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 $\text{Avail}^r(e)$ be the least function such that:

$$\text{Avail}^r(e) = \begin{cases} 
  \text{Def}(e_1) & \text{if } e = e_1 \mid c \mid e_2 \\
  x' \mid \vec{x}_1 \mid \vec{x}_2 & \text{where} \\
  \vec{x}_1 = \text{Def}(UFD_e(x)), \text{ and} \\
  \vec{x}_2 = \text{Avail}^r(e), \text{ if} \\
  e = e_1 \mid c' \mid e_2, \\
  c' = (x = \text{fun } x' \rightarrow (e')), \text{ and} \\
  e \in \text{Fr}(e') 
\end{cases}$$

The $\text{Avail}^d$ – “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 $\text{Avail}^d$ and $\text{Avail}^r$ would converge: $\text{Avail}^d(e) = \text{Avail}^r(e)$. But, as it is, neither $\text{Avail}^d(e) \subseteq \text{Avail}^r(e)$ nor $\text{Avail}^d(e) \supseteq \text{Avail}^r(e)$. In the following definition, the special case uses $\text{Fr}^t$ so that forward external dependencies within conditional bodies work.

Definition 2.13. Let $\text{Avail}^d(e)$ be the least function such that:
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:

\[ \overline{k} ::= \ldots | \text{Rewind} \quad \text{continuations} \]

To calculate the target of the jump generated by a Rewind, there exists the EndOfScope function:

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

\[
\text{EndOfScope}(\overline{a}_1) = \begin{cases} \\
\overline{a}_1 & \text{if } \overline{a}_1 = \text{End} \\
\text{EndOfScope}(\overline{a}_0) & \text{or } \overline{a}_1 = (\overline{x} \leftarrow \overline{x}') \\
such that \overline{a}_1 \ll \overline{a}_0 & \text{and } \overline{a}_1 \neq (\overline{x} \leftarrow \overline{x}'), \text{otherwise} \end{cases}
\]

To handle the Rewind operation, it is necessary to extend the lookup function \( D(\overline{a}_0, \overline{K}, \overline{C}) \). A new condition translates a Rewind into a Jump with a fixed target determined by the function EndOfScope:

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

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 \( \overline{a}_1 = (\overline{x}' \leftarrow \overline{x}') \), \( \overline{a}_1 \ll \overline{a}_0 \), \( \overline{c} = (\overline{x}' \leftarrow \overline{x}'' \leftarrow \overline{x}') \), \( \overline{K} = \overline{x}' || \overline{K}' \), \( \overline{z}' \neq \overline{x} \), and isTop(\( \overline{c} \), \( \overline{C} \)) then \( D(\overline{a}_1, \overline{x}' || \text{Rewind} \ || \overline{K}, \overline{C}) \subseteq \overline{V} \).

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:

```java
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.magnitude() == 0;
    }
}
```

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

```java
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;
        isZeroK = isZeroMagnitude == zero;
    };
    magnitude = fun magnitudeIgnored -> {
        magnitudeX = getX ignoredParameter;
        magnitudeY = getY ignoredParameter;
        calculation ... ;
    };
    getXY = fun getXYIgnored -> {
        getXR = 1x; }
    getXY = fun getXYIgnored -> {
        getYR = 1y; }
    setXY = fun setXY -> {
        setXR = x <- setXX; 
    };
    setXY = fun setYY -> {
        setYR = y <- setYY; 
    };

    pointConstructorRun = pointConstructor ignoredParameter;
    pointObject = {
        isZero = isZeroMagnitude == magnitude,
        getXY = getX, getXY = 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 = pointIsZeroResult;
    pointModifyY = pointSetY;
    assertModifiedPointIsZero = pointIsZero ignoredParameter;
    modifiedPointMagnitudeResult = pointMagnitude ignoredParameter;
    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):

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

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.
- **recreation** (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.
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 feature 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.

Within the same environment, it is possible to exactly capture one feature and over-approximate another; and 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, \texttt{church} and \texttt{mfr3}, 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 \texttt{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 \(k\).

The next sections discuss which experiments we ran and their results.
<table>
<thead>
<tr>
<th>Program</th>
<th>CFG nodes</th>
<th>PDR nodes</th>
<th>PDR edges</th>
<th>Time (ms)</th>
<th>States</th>
<th>Edges</th>
<th>Time (ms)</th>
<th>CFG PDR PDR Time</th>
<th>States</th>
<th>Points</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
<td>69</td>
<td>804</td>
<td>1913</td>
<td>605</td>
<td>17</td>
<td>20</td>
<td>452</td>
<td>70</td>
<td>68</td>
<td>114</td>
<td>1187</td>
</tr>
<tr>
<td>alex</td>
<td>61</td>
<td>422</td>
<td>5452</td>
<td>177</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>53</td>
<td>65</td>
<td>99</td>
<td>1359</td>
</tr>
<tr>
<td>blur</td>
<td>66</td>
<td>602</td>
<td>10160</td>
<td>295</td>
<td>58</td>
<td>66</td>
<td>496</td>
<td>705</td>
<td>5</td>
<td>5</td>
<td>3872</td>
</tr>
<tr>
<td>cpstak</td>
<td>135</td>
<td>4348</td>
<td>271245</td>
<td>50663</td>
<td>53</td>
<td>25</td>
<td>140</td>
<td>60</td>
<td>28</td>
<td>114</td>
<td>1166</td>
</tr>
<tr>
<td>eta</td>
<td>25</td>
<td>106</td>
<td>842</td>
<td>31</td>
<td>26</td>
<td>34</td>
<td>114</td>
<td>140</td>
<td>60</td>
<td>28</td>
<td>1166</td>
</tr>
<tr>
<td>kcfa2</td>
<td>47</td>
<td>232</td>
<td>1955</td>
<td>53</td>
<td>35</td>
<td>34</td>
<td>114</td>
<td>56</td>
<td>205</td>
<td>114</td>
<td>1167</td>
</tr>
<tr>
<td>kcfa3</td>
<td>60</td>
<td>332</td>
<td>2934</td>
<td>86</td>
<td>53</td>
<td>52</td>
<td>156</td>
<td>147</td>
<td>427</td>
<td>62</td>
<td>1164</td>
</tr>
<tr>
<td>loop2</td>
<td>93</td>
<td>939</td>
<td>21609</td>
<td>862</td>
<td>46</td>
<td>49</td>
<td>527</td>
<td>142</td>
<td>50</td>
<td>115</td>
<td>1152</td>
</tr>
<tr>
<td>nj09</td>
<td>36</td>
<td>177</td>
<td>1568</td>
<td>51</td>
<td>32</td>
<td>31</td>
<td>104</td>
<td>98</td>
<td>42</td>
<td>114</td>
<td>1149</td>
</tr>
<tr>
<td>recursion</td>
<td>23</td>
<td>121</td>
<td>1007</td>
<td>22</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>sat</td>
<td>79</td>
<td>545</td>
<td>7181</td>
<td>320</td>
<td>248</td>
<td>311</td>
<td>2374</td>
<td>460</td>
<td>87</td>
<td>116</td>
<td>1169</td>
</tr>
<tr>
<td>state</td>
<td>27</td>
<td>82</td>
<td>686</td>
<td>17</td>
<td>29</td>
<td>36</td>
<td>425</td>
<td>208</td>
<td>56</td>
<td>116</td>
<td>1168</td>
</tr>
<tr>
<td>church</td>
<td>155</td>
<td>4456</td>
<td>304437</td>
<td>2759</td>
<td>73846</td>
<td>103629</td>
<td>1801462</td>
<td>6273</td>
<td>292</td>
<td>1286</td>
<td>1286</td>
</tr>
<tr>
<td>mbrotZ</td>
<td>807</td>
<td>19913</td>
<td>2760869</td>
<td>663718</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>59</td>
<td>56</td>
<td>116</td>
<td>1163</td>
</tr>
</tbody>
</table>

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, ffg, ffgg, ffgg, 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 $\times$ 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
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 explore 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
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 $\text{REWIND}$ 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 $\text{REWIND}$ where necessary.

With that binding structure available to the analysis, a further optimization is possible: instead of $\text{REWIND}$ meaning a $\text{JUMP}$ to the end of the immediate block, it could be a $\text{JUMP}$ 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 Rozenshtein for the conversations about program analysis.

References


Table 3.4. Results of the growing programs experiment. The difference between the results for the programs \( \text{sat} \times 1 \) and \( \text{eta} \times 1 \) 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 \( \text{eta} \times 32 \) in P4F are not available because the execution terminated with a StackOverflowError.

<table>
<thead>
<tr>
<th>Program</th>
<th>( k )</th>
<th>CFG nodes</th>
<th>PDR nodes</th>
<th>PDR edges</th>
<th>Time (ms)</th>
<th>( k )</th>
<th>States</th>
<th>Edges (ms)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sat} \times 1 )</td>
<td>4</td>
<td>35</td>
<td>205</td>
<td>2223</td>
<td>889</td>
<td>1</td>
<td>65</td>
<td>67</td>
<td>402</td>
</tr>
<tr>
<td>( \text{sat} \times 2 )</td>
<td>4</td>
<td>68</td>
<td>461</td>
<td>5014</td>
<td>1420</td>
<td>1</td>
<td>134</td>
<td>139</td>
<td>1332</td>
</tr>
<tr>
<td>( \text{sat} \times 4 )</td>
<td>4</td>
<td>134</td>
<td>1135</td>
<td>12322</td>
<td>4073</td>
<td>1</td>
<td>272</td>
<td>283</td>
<td>6704</td>
</tr>
<tr>
<td>( \text{sat} \times 8 )</td>
<td>4</td>
<td>266</td>
<td>3131</td>
<td>33850</td>
<td>13011</td>
<td>1</td>
<td>548</td>
<td>571</td>
<td>52949</td>
</tr>
<tr>
<td>( \text{sat} \times 16 )</td>
<td>4</td>
<td>530</td>
<td>9715</td>
<td>104554</td>
<td>47682</td>
<td>1</td>
<td>1100</td>
<td>1147</td>
<td>536471</td>
</tr>
<tr>
<td>( \text{sat} \times 32 )</td>
<td>4</td>
<td>1058</td>
<td>33251</td>
<td>356554</td>
<td>188304</td>
<td>1</td>
<td>1456</td>
<td>1512</td>
<td>1800579</td>
</tr>
<tr>
<td>( \text{eta} \times 1 )</td>
<td>1</td>
<td>18</td>
<td>103</td>
<td>775</td>
<td>69</td>
<td>1</td>
<td>26</td>
<td>25</td>
<td>55</td>
</tr>
<tr>
<td>( \text{eta} \times 2 )</td>
<td>1</td>
<td>34</td>
<td>225</td>
<td>1740</td>
<td>98</td>
<td>1</td>
<td>60</td>
<td>59</td>
<td>264</td>
</tr>
<tr>
<td>( \text{eta} \times 4 )</td>
<td>1</td>
<td>66</td>
<td>535</td>
<td>4270</td>
<td>293</td>
<td>1</td>
<td>128</td>
<td>127</td>
<td>475</td>
</tr>
<tr>
<td>( \text{eta} \times 8 )</td>
<td>1</td>
<td>130</td>
<td>1419</td>
<td>11730</td>
<td>917</td>
<td>1</td>
<td>264</td>
<td>263</td>
<td>1332</td>
</tr>
<tr>
<td>( \text{eta} \times 16 )</td>
<td>1</td>
<td>258</td>
<td>4243</td>
<td>36250</td>
<td>4130</td>
<td>1</td>
<td>536</td>
<td>535</td>
<td>6915</td>
</tr>
<tr>
<td>( \text{eta} \times 32 )</td>
<td>1</td>
<td>514</td>
<td>14115</td>
<td>123690</td>
<td>19654</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 3.4. Results of the growing programs experiment. The difference between the results for the programs \( \text{sat} \times 1 \) and \( \text{eta} \times 1 \) 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 \( \text{eta} \times 32 \) in P4F are not available because the execution terminated with a StackOverflowError.


<table>
<thead>
<tr>
<th>Program</th>
<th>CFG nodes</th>
<th>PDR nodes</th>
<th>PDR edges</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity × 1</td>
<td>0</td>
<td>10</td>
<td>14</td>
<td>110</td>
</tr>
<tr>
<td>identity × 2</td>
<td>0</td>
<td>14</td>
<td>30</td>
<td>243</td>
</tr>
<tr>
<td>identity × 4</td>
<td>0</td>
<td>22</td>
<td>66</td>
<td>565</td>
</tr>
<tr>
<td>identity × 8</td>
<td>0</td>
<td>38</td>
<td>138</td>
<td>1325</td>
</tr>
<tr>
<td>identity × 16</td>
<td>0</td>
<td>70</td>
<td>282</td>
<td>3229</td>
</tr>
<tr>
<td>identity × 32</td>
<td>0</td>
<td>134</td>
<td>570</td>
<td>8573</td>
</tr>
<tr>
<td>identity × 64</td>
<td>0</td>
<td>262</td>
<td>1146</td>
<td>25405</td>
</tr>
<tr>
<td>identity × 128</td>
<td>0</td>
<td>518</td>
<td>2298</td>
<td>83645</td>
</tr>
<tr>
<td>identity × 1</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>110</td>
</tr>
<tr>
<td>identity × 2</td>
<td>1</td>
<td>14</td>
<td>30</td>
<td>243</td>
</tr>
<tr>
<td>identity × 4</td>
<td>1</td>
<td>22</td>
<td>66</td>
<td>565</td>
</tr>
<tr>
<td>identity × 8</td>
<td>1</td>
<td>38</td>
<td>144</td>
<td>1325</td>
</tr>
<tr>
<td>identity × 16</td>
<td>1</td>
<td>70</td>
<td>296</td>
<td>3229</td>
</tr>
<tr>
<td>identity × 32</td>
<td>1</td>
<td>134</td>
<td>600</td>
<td>8573</td>
</tr>
<tr>
<td>identity × 64</td>
<td>1</td>
<td>262</td>
<td>1208</td>
<td>25405</td>
</tr>
<tr>
<td>identity × 128</td>
<td>1</td>
<td>518</td>
<td>2424</td>
<td>83645</td>
</tr>
<tr>
<td>recursive × 1</td>
<td>0</td>
<td>71</td>
<td>648</td>
<td>22694</td>
</tr>
<tr>
<td>recursive × 2</td>
<td>0</td>
<td>80</td>
<td>768</td>
<td>25726</td>
</tr>
<tr>
<td>recursive × 3</td>
<td>0</td>
<td>89</td>
<td>888</td>
<td>28720</td>
</tr>
<tr>
<td>recursive × 4</td>
<td>0</td>
<td>98</td>
<td>1008</td>
<td>31732</td>
</tr>
<tr>
<td>recursive × 5</td>
<td>0</td>
<td>107</td>
<td>1128</td>
<td>34762</td>
</tr>
<tr>
<td>recursive × 6</td>
<td>0</td>
<td>116</td>
<td>1248</td>
<td>37810</td>
</tr>
<tr>
<td>recursive × 7</td>
<td>0</td>
<td>125</td>
<td>1368</td>
<td>40876</td>
</tr>
<tr>
<td>recursive × 8</td>
<td>0</td>
<td>134</td>
<td>1488</td>
<td>43960</td>
</tr>
<tr>
<td>recursive × 9</td>
<td>0</td>
<td>143</td>
<td>1608</td>
<td>47062</td>
</tr>
<tr>
<td>recursive × 10</td>
<td>0</td>
<td>152</td>
<td>1728</td>
<td>50182</td>
</tr>
<tr>
<td>recursive × 1</td>
<td>1</td>
<td>71</td>
<td>649</td>
<td>22694</td>
</tr>
<tr>
<td>recursive × 2</td>
<td>1</td>
<td>80</td>
<td>771</td>
<td>25726</td>
</tr>
<tr>
<td>recursive × 3</td>
<td>1</td>
<td>89</td>
<td>893</td>
<td>28720</td>
</tr>
<tr>
<td>recursive × 4</td>
<td>1</td>
<td>98</td>
<td>1015</td>
<td>31732</td>
</tr>
<tr>
<td>recursive × 5</td>
<td>1</td>
<td>107</td>
<td>1137</td>
<td>34762</td>
</tr>
<tr>
<td>recursive × 6</td>
<td>1</td>
<td>116</td>
<td>1259</td>
<td>37810</td>
</tr>
<tr>
<td>recursive × 7</td>
<td>1</td>
<td>125</td>
<td>1381</td>
<td>40876</td>
</tr>
<tr>
<td>recursive × 8</td>
<td>1</td>
<td>134</td>
<td>1503</td>
<td>43960</td>
</tr>
<tr>
<td>recursive × 9</td>
<td>1</td>
<td>143</td>
<td>1625</td>
<td>47062</td>
</tr>
<tr>
<td>recursive × 10</td>
<td>1</td>
<td>152</td>
<td>1747</td>
<td>50182</td>
</tr>
</tbody>
</table>

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

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.