WFLP 2014

23rd International Workshop on Functional and (Constraint) Logic Programming

September 15-17, 2014, Wittenberg

Proceedings
Preface

This volume contains the papers presented at WFLP 2014: 23rd International Workshop on Functional and (Constraint) Logic Programming held on September 15-17, 2014 in Wittenberg, at the Leu-corea Conference Center of Halle-Wittenberg University.

The WFLP workshops series is running since 1992 and brings together researchers interested in functional programming, logic programming, as well as their integration.

WFLP 2014 was jointly organized and located with WLP 2014: 28th Workshop on (Constraint) Logic Programming. The WLP workshops series serves as the scientific forum of the annual meeting of the Society of Logic Programming (GLP e.V.) and brings together researchers interested in logic programming, constraint programming, and related areas like databases, artificial intelligence, and operations research.

The technical program of the workshop included an invited tutorial by Michael Hanus on Declarative Multi-paradigm Programming, an invited talk by Sven Thiele on Answer Set Programming for Systems Biology, and presentations of refereed papers.

I thank the invited speakers, authors of papers, programme committee members, external reviewers, as well as the local organizers Stefan Brass, Ramona Vahrenhold and Heike Stephan, for creating a very interesting and enjoyable workshop.

September 4, 2014

Johannes Waldmann
Leipzig
Program Committee

Elvira Albert  Complutense University of Madrid
Sergio Antoy  Portland State University
Mauricio Ayala-Rincon  University of Brasilia
William Byrd  University of Utah
Michael Hanus  Universität Kiel
Herbert Kuchen  University of Münster
Carlos Olarte  Pontificia Universidad Javeriana Cali
Peter J. Stuckey  University of Melbourne
René Thiemann  University of Innsbruck
Janis Voigtländer  University of Bonn
Johannes Waldmann  HTWK Leipzig (chair)

Additional Reviewers

Maria Christakis
Moreno Falaschi
Keller, Gabriele
Daniele Nantes-Sobrinho
Ana Cristina Rocha Oliveira,
German Vidal
# Table of Contents

- **Declarative Multi-paradigm Programming**  
  *Michael Hanus*  
  5

- **Interpreting XPath by Iterative Pattern Matching with Paisley**  
  *Baltasar Trancón Y Widemann and Markus Lepper*  
  8

- **Exploring Non-Determinism in Graph Algorithms**  
  *Nikita Danilenko*  
  25

- **Curry without Success**  
  *Sergio Antoy and Michael Hanus*  
  40

- **A Partial Evaluator for Curry**  
  *Michael Hanus and Björn Peemöller*  
  55

- **Automatic Testing of Operation Invariance**  
  *Tobias Gödderz and Janis Voigtländer*  
  72

- **Partitioning 0-CFA for the GPU**  
  *Thomas Gilray, James King and Matthew Might*  
  87
Declarative Multi-paradigm Programming

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.

mh@informatik.uni-kiel.de

Abstract. This tutorial provides an overview and introduction to declarative programming exploiting multiple paradigms, in particular, functional, logic, and constraint programming. To demonstrate the possibility to support these paradigms within a single programming model, we survey the features of the declarative multi-paradigm language Curry.

1 Overview

Compared to traditional imperative languages, declarative programming languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. However, there is no distinct “declarative programming” paradigm. Instead, there are various programming paradigms and related languages based on different methods to structure declarative knowledge. Functional programming is based on the lambda calculus and provide functions as computational entities. Logic programming is based on first-order predicate logic and uses predicates as basic programming entities. Constraint programming offers constraint solvers to reason about models described with the help of various constraint structures. Although the motivation to exploit high-level programming is similar in all paradigms, the concrete languages associated to them are quite different. Thus, it is a natural idea to combine these worlds of programming into a single paradigm, and attempts for doing so have a long history. However, the interactions between functional and logic programming features are complex in detail so that the concrete design of such declarative multi-paradigm languages is a non-trivial task. This is demonstrated by a lot of research work on the semantics, operational principles, and implementation of functional logic languages since more than two decades. Fortunately, recent advances in the foundation and implementation of functional logic languages have shown reasonable principles that lead to the design of practically applicable programming languages.

This tutorial provides an overview on the principles of integrated functional logic languages. As a concrete programming language, we survey the declarative multi-paradigm language Curry\(^1\) [13, 20]. It is developed by an international initiative of researchers in this area and intended to provide a common platform for research, teaching, and application of integrated functional logic languages.

\(^1\) http://www.curry-language.org
Details about functional logic programming and Curry can be found in recent surveys [5, 18] and in the language report [20].

The integration of functional and logic programming has various advantages. Beyond the fact that one can use the best features of declarative languages in a single language, like strong typing, higher-order functions, optimal (lazy) evaluation from functional programming, or non-determinism, computing with partial information, and constraint solving from logic programming, there are also clear advantages compared to the individual paradigms. For instance, the combination of lazy evaluation and non-determinism leads to a demand-driven exploration of the search space which is sometimes more efficient and optimal for particular classes of programs [2]. Moreover, non-declarative features, which are regularly used in practical logic programs, can be avoided in functional logic languages, e.g., by functional notation or declarative I/O [22].

The combined features offered by functional logic languages led to new design patterns [3, 6], better abstractions for application programming (e.g., programming with databases [7, 11], GUI programming [14], web programming [15, 16, 19], string parsing [10]), and new techniques to implement programming tools, like partial evaluators [1] or test case generators [12, 21]. In particular, functional patterns, as proposed in [4], exploit non-determinism from logic programming and demand-driven pattern matching from functional programming in order to achieve a powerful executable specification method. For instance, functional patterns have been used for XML processing [17] where it has been shown that specialized logic programming approaches [8, 9] can be implemented with a few lines of code in Curry. Some of these techniques are reviewed in this tutorial.

References

Interpreting XPath by Iterative Pattern Matching with Paisley

Baltasar Trancón y Widemann\textsuperscript{1} and Markus Lepper\textsuperscript{2}

\textsuperscript{1} Ilmenau University of Technology
\textsuperscript{2} <semantics/> GmbH

Abstract. The Paisley architecture is a light-weight EDSL for non-deterministic pattern matching. It automates the querying of arbitrary object-oriented data models in a general-purpose programming language, using API, libraries and simple programming patterns in a portable and non-invasive way. The core of Paisley has been applied to real-world applications. Here we discuss the extension of Paisley by pattern iteration, which adds a Kleene algebra of pattern function composition to the unrestricted use of the imperative host language, thus forming a hybrid object-oriented–functional–logic framework. We subject it to a classical practical problem and established benchmarks: the node-set fragment of the XPath language for querying W3C XML document object models.

1 Introduction

The Paisley embedded domain-specific language (EDSL) and library adds the more declarative style of pattern matching to the object-oriented context of Java programming [9,10]. Paisley offers a combination of features that is, by our knowledge, not offered by any other existing pattern matching solution for a main-stream language: it is strictly typed, fully compositional, non-invasive and supports non-determinism, full reification and persistence.

The non-deterministic aspects of Paisley have been demonstrated to provide a fairly powerful basis for embedded logic programming in the object-oriented host environment. In [11] we have discussed how to solve cryptarithmetic puzzles with the backtracking facilities of Paisley, and how to obtain non-trivial efficient search plans by object-oriented meta-programming with Paisley pattern objects.

Here we describe and evaluate recent extensions to Paisley that greatly increase its capabilities as an embedded functional-logic programming system, supporting more complex control and data flow than the combinatorial generate&test tactics of [11]. In particular, the novel contributions are:

1. in the core layer, a calculus of functions on patterns and its Kleene algebra, thus providing regular expressions of nested patterns;
2. in the application layer, a library extension that demonstrates their use by matching W3C XML Document Object Models with XPath [3] expressions;
3. a practical case study that evaluates the approach in comparison with standard on-board facilities of the host language Java.
Variants: 1.//a[@href] 2.//p//a[@href] 3.//p[.//a[@href]]

```java
void collect (int variant, Node node, Collection<Element> results,
             boolean sideways) {
    if (node instanceof Element) {
        Element elem = (Element)node;
        if (elem.getTagName().equals(variant == 1 ? "a" : "p")) {
            if (variant > 1) {
                Collection<Element> sub = variant == 2 ? results : new ArrayList<>();
                collect(1, elem, sub, false);
            }
            if (variant == 1 && !elem.getAttribute("href").equals("") ||
                variant == 3 && !sub.isEmpty())
                results.add(elem);  // solution
        }
    }
    if (node.getFirstChild() != null)
        collect(variant, node.getFirstChild(), results, true);  // depth
    if (sideways && node.getNextSibling() != null)
        collect(variant, node.getNextSibling(), results, true);  // breadth
}
```

Fig. 1. Three related XML queries. Top: XPath expressions; bottom: Java code.

2 Motivation

The basic motivation for a pattern matching EDSL is that patterns as language constructs make data-gathering code more declarative, and thus more readable, more writable and more maintainable. If done properly, this additional expressive power can be used to factor code compositionally in ways that are not straightforwardly possible in a conventional object-oriented approach, where the logic and data flow of a complex query often appear as cross-cutting concerns.

As an example that highlights the issues of logical compositionality, consider a family of three related XPath expressions, depicted in Fig. 1, that select nodes from XHTML documents. The first selects, from the context node and its descendants, all <a> elements (hyperlink anchors) that have a href (target) attribute specified. The second selects only those <a> elements with href attributes that are nested within <p> elements (paragraphs). The third selects all <p> elements that have some <a> element with href attribute nested within.

The task is to implement the specified search procedures, as object-oriented queries of the standard W3C X(HT)ML Document Object Model. Evidently, one wishes to implement most of the operational code generically, with reuse and minimal adaptation for each of the three variants. Additionally, the second and third variant should be able to use the first recursively. To emphasize the role of reuse in this example, Fig. 1 gives a unified implementation, where the identifier variant is grayed out to emphasize that all its occurrences serve only the static choice between the variants. The common algorithm is to traverse nodes
by depth-first search, and add all valid matches, possibly repeatedly, to a results collection supplied by the caller. Two further improvements of the solution are suggested as exercises to the reader:

1. Of course, code that works for these three, but no other XPath expressions is hardly generic. Abstract to a large, preferably unbounded, set of useful XPath expressions. Refactor the code to separate commonalities and individual degrees of freedom, as cleanly as possible.

2. The use of a Collection for matches implies eager evaluation. This is not efficient for the recursive calls from variant 3 to variant 1, where searching can be aborted after the first successful match (which falsifies sub.isEmpty()). Refactor the code to allow for lazy evaluation, as transparently as possible.

See section 5 and Fig. 4 below for our proposed solution. We do not claim that these refactoring steps are infeasible in any particular previous framework. But we shall demonstrate that the Paisley approach naturally gives both pragmatic design guidelines, and a concrete notation for the adequate, abstract, modular and efficient expression of the underlying algorithm and its instantiations.

3 Basic Matching with Paisley

The design principles, semantics and APIs of Paisley patterns have been discussed in detail in [9,10,11]. The EDSL is extremely lightweight: It requires no language or compiler extension; API calls and programming idioms are sufficient for its full expressive power. The core API is summarized in Fig. 2.

The root of the pattern hierarchy is the abstract base class Pattern⟨A⟩ of patterns that can process objects of some arbitrary type A. A pattern Pattern⟨A⟩ p is applied to some target data x of type A by calling p.match(x), returning a boolean value indicating whether the match was successful.

Complex patterns are composed from application-layer building blocks that implement classifications and projections, the reification of instance tests and getter methods, respectively. These can be defined independently for arbitrary (public interfaces of) data models, requiring no intrusion into legacy code. Paisley comes with predefined bindings for common Java data models, such as the Collection framework; more can be added by the user. Each projection x from type A to B induces by contravariant lifting a construction of type Pattern⟨A⟩ from Pattern⟨B⟩, conveniently implemented as a method Pattern⟨A⟩ getX(Pattern⟨B⟩ p).

Information is extracted from a successfully matched pattern via embedded pattern variables. A pattern Variable⟨A⟩ v matches any value A x, and stores a reference to x that can be retrieved by invoking v.getValue(). Variables behave like imperative rather than logical variables; subsequent matches merely overwrite the stored value, no unification is implied.\(^3\)

\(^3\) This task is inherently much more difficult in conventional programming languages than in logic and lazy functional approaches; cf. [5].

\(^4\) This design choice enables the use of Paisley for general, object-oriented data models, where stable notions of equality, let alone an induction principle, cannot be taken for granted. See section 4 for the handy implications of the imperative perspective.
abstract class Pattern⟨A⟩ {
    public abstract boolean match(A target);
    public boolean matchAgain();
    public static ⟨A⟩ Pattern⟨A⟩
        both(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);
    public static ⟨A⟩ Pattern⟨A⟩
        either(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);
    public Pattern⟨A⟩ someMatch();
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
    public A getValue();
    public ⟨B⟩ List⟨A⟩ eagerBindings(Pattern⟨? super B⟩ root, B target);
    public ⟨B⟩ Iterable⟨A⟩ lazyBindings(Pattern⟨? super B⟩ root, B target);
    public ⟨B⟩ Pattern⟨B⟩ bind(Pattern⟨? super B⟩ root, Pattern⟨? super A⟩ sub);
    public Pattern⟨A⟩ star(Pattern⟨? super A⟩ root);
    public Pattern⟨A⟩ plus(Pattern⟨? super A⟩ root);
}

Fig. 2. Interface synopsis (core).

Patterns can be composed conjunctively and disjunctively by the binary combinators both and either, or their n-ary variants all and some (not shown), respectively. As in traditional logic programming, disjunction is realized as backtracking: After each successful match for a pattern p, p.matchAgain() can be invoked to backtrack and find another solution. Solutions can be exhausted, in an imperative style of encapsulated search, by the following programming idiom:

\[
\text{if } (p.\text{match}(x)) \text{ do doSomething(); while } (p.\text{matchAgain}());
\]

When not using an exhaustive loop, the computation of alternative solutions can be deferred indefinitely; the required state for choice points is stored residually in the instances of the logical combinators themselves.

The search construct can be abbreviated further to a functional style if the desired result of each match is the value stored in a single pattern variable v. Invoking v.eagerBindings(p, x) or v.lazyBindings(p, x) returns objects that give the value of v for all successive matches, collected beforehand in an implicit loop, or computed on iterator-driven demand, respectively. The latter can also deal with infinite sequences of solutions.

For cases where only the satisfiability of a pattern p, but not the actual sequence of solutions, is of concern, one can invoke p.someMatch(), yielding a wrapper that cuts all alternatives after the first solution. Thus, laziness is exploited for efficiency and, when used in conjunction with other patterns with multiple relevant solutions, spurious multiplication of solution sets is avoided.
4 Advanced Pattern Calculus

4.1 Pattern Substitution and Iteration

As discussed in [10], Paisley patterns and their variables obey an algebraic calculus where most of the expected mathematical laws hold, despite the low-level imperative nature of their implementation.

A variable $v$ known to occur in a pattern $p$ can be substituted by a subpattern $q$, by invoking $v$.bind$(p, q)$, mnemonically read as $(\lambda v. p)q$. The implementation delegates to $p$ and $q$ sequentially, and hence does not require access to the internals of either operand.

Substitution can also be given a recursive twist; a pattern $p$ with a “hole” $v$ can be nested. The patterns $v$.star$(p)$ and $v$.plus$(p)$ correspond to the $\ast$ and $+$-closures, respectively, of the search path relation between $p$ and $v$, in the sense that the usual recursive relations hold,

\[
\begin{align*}
v\.star(p) & \equiv \text{either}(v, v\.plus(p)) \\
v\.plus(p) & \equiv v\.bind(p, v\.star(p))
\end{align*}
\]

which is already almost the complete, effective implementation, up to lazy duplication of $v\.plus(p)$ for each iteration level that is actually reached. Note that iteration depth is conceptually unbounded, and solutions are explored in depth-first pre-order, due to the way $\text{either}$ is used. Thus patterns form a Kleene algebra up to equivalence of solution sets but not sequences.

Using these iteration operators, complex nondeterministic pattern constructors can be defined concisely. For instance, the contravariant lifting of the multi-valued, transitive descendant projection in XML document trees, applied to a pattern $p$, becomes simply

\[
v\.bind(v\.plus(child(v)), p)
\]

where Variable\langle Node\rangle $v$ is fresh, and the multi-valued projection Pattern\langle Node\rangle child(Pattern\langle Node\rangle ) is implemented in terms of the org.w3c.dom.Node getter methods $\text{getFirstChild()}$ and $\text{getNextSibling()}$. Note how the two sources of nondeterminism, regarding horizontal (child) and vertical (plus) position in the document tree, respectively (cf. the distinct recursive calls in Fig. 1), combine completely transparently.

4.2 Pattern Function Abstraction

Functions taking patterns to patterns have so far featured in two important roles. In operational form, implemented as lifting methods, they are the basis for contravariant representation of projection patterns. In algebraic form, as bind-based abstractions from a variable, they enable pattern composition and iteration. Being so ubiquitous in the Paisley approach, they deserve their own

\[5\] The actual condition is definite assignment rather than occurrence, for technical reasons.
abstract class Motif⟨A, B⟩ {
    public abstract Pattern⟨B⟩ apply(Pattern ⟨? super A⟩);
    public static ⟨A⟩ Motif⟨A, A⟩ id();
    public ⟨C⟩ Motif⟨C, B⟩ on(Motif⟨C, ? super A⟩ other);
    public static ⟨A⟩ Motif⟨A, A⟩ star(Motif⟨A, ? super A⟩ f);
    public static ⟨A⟩ Motif⟨A, A⟩ plus(Motif⟨A, ? super A⟩ f);
    public Iterable⟨A⟩ lazyBindings (B target);
    public List⟨A⟩ eagerBindings(B target);
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
    // ...
    public ⟨B⟩ Motif⟨A, B⟩ lambda(Pattern ⟨B⟩ root);
}

Fig. 3. First-class pattern function (motif) interface.

reification. Fig. 3 shows the relevant API. An instance of class Motif⟨A, B⟩ is
the reified form of a function taking patterns over A to patterns over B, or by
contravariance, the pattern representation of an access operation that takes data
do type B to data of type A, by its apply method.

The Motif class provides the algebra of the category of patterns, namely the
identical motif id() and the type-safe composition of motifs by the on(Motif) in-
stance method. The variable-related operations star/plus and lazy-/eagerBindings
each have a point-free counterpart in Motif, which create anonymous variables to
hold intermediate results on the fly. For instance, the XML descendant pattern
defined above can be expressed less redundantly in point-free form as

    plus(child).apply(p)

given a child access reification Motif⟨Node,Node⟩ child.

Conversely, motif abstraction is defined for pattern variables, which obeys
the beta reduction rule:

    v.lambda(p).apply(q) ≡ v.bind(p, q)

There is often a trade-off in convenience between functions in operational
and reified form, that is as either pattern lifting methods or motifs. Since there
is no automatic conversion between methods and function objects in Java prior
to version 8, our strategy in the Paisley library is to provide both redundantly.

5 Motivation Revisited

Figure 4 repeats the XPath examples expressions, now contrasting the domain-
specific XPath code with the general-purpose Java/Paisley code. The latter has
been underlined for easy reference. The solution makes full use of the shared
common structure of the three variants, and achieves complete separation of concerns:

– The search procedure implied by the operator // is not spread out as recursive control flow, but reified and encapsulated as the descendantOrSelf pattern lifting, predefined in the XPathPatterns static factory class of Paisley, discussed in detail in the following section.

– XPath subexpressions are denoted concisely and orthogonally.
  • The generic form .//tag... that reoccurs in all variants is abstracted as the extensible pattern construction dslash.
  • The particular inner and outer forms, .//a[@href] and .//p, respectively, are defined once and for all, independently of each other.

– The semantics of XPath queries are effected concisely and orthogonally.
  • The general principle of encapsulated search is expressed naturally by a call to lazyBindings. The third variant concisely prunes the recursively encapsulated search after the first solution, via the pattern modifier someMatch().
  • The particular configuration of subexpressions for all variants boils down to a simple choice of the bound variable and pattern-algebraic composition. Note that both are reified and hence first-class citizens of Java. Thus, each variant boils down to a one-line expression, where the points of variability are ordinary subexpressions that can be chosen statically (as shown) or dynamically, with arbitrarily complex meta-programming.

Fig. 4. Three related XML queries revisited (from Figure 1). Top: XPath expressions; middle: Java code template; bottom: template instantiations.
enum Axis {
    public Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Test {
    public abstract Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Predicate {
    public abstract boolean accepts(NodeSet context, int position, Node candidate);
    public Motif⟨? extends Node, Node⟩ apply(Motif⟨? extends Node, Node⟩ base);
}

class NodeSet {
    private Collection⟨? extends Node⟩ elems;
    public NodeSet(Iterable⟨? extends Node⟩ elems);
    public int size();
    public Iterable⟨Node⟩ filter(Predicate pred);
}

class Path {
    Path(Motif⟨? extends Node, Node⟩ motif);
    public Motif⟨? extends Node, Node⟩ getMotif();
    public Path step(Axis axis, Test test, Predicate... predicates);
}

Fig. 5. XPath base classes in Paisley.

6 The Paisley XPath Interpreter

As a case study of real-world data models and queries, we have implemented the navigational (proper path) fragment of the XPath 1.0 language as a Paisley pattern combinator library. The complete implementation consists of the factory classes XMLPatterns for generic DOM access and XPathPatterns for XPath specifics, with currently 433 and 282 lines of code, respectively. Given an XPath parser, it can be extended to a full-fledged interpreter, and hence a non-embedded DSL, by a straightforward mapping of abstract syntax nodes to pattern operations.

6.1 Language Fragment

The XPath 1.0 language comes with many datatypes and functions that do not contribute directly to its main goal, namely the addressing of nodes in an XML document. For simplicity, we restrict our treatment of the language to a fragment streamlined for that purpose. Typical uses of XPath within XSLT or XQuery [2] conform to this fragment. We conjecture that the missing features can be added without worsening essential operational complexity and runtime performance.
We omit external variables and functions, and all operations on the datatypes of strings, numbers and booleans, as well as intangible document nodes such as comments and processing instructions. The supported sublanguage is reflected one-to-one by API operations based in the class hierarchy depicted in Fig. 5. (Operation signatures are given in Fig. 11 in the appendix.) Its focus is on so-called path expressions of the general syntactic form

\[
\text{path ::= abs\_path} \mid \text{rel\_path} \quad \text{abs\_path ::= } /\text{rel\_path}^2 \\
\text{step ::= axis : test ([ predicate ])*} \quad \text{rel\_path ::= (rel\_path / )}^2\text{step}
\]

Here \textit{axis} is one of the twelve XPath document axes, omitting the deprecated namespace declaration axis. The nonterminal \textit{test} is the tautological test \texttt{node()}, the text node test \texttt{text()} or an explicit node name test. For \textit{predicate}, used to filter selected nodes, we accept not the full XPath expression language, but only

\[
\text{predicate ::= path} \mid \text{integer} \mid \text{not predicate} \mid (\text{predicate (and | or) predicate})
\]

where a \texttt{path} predicate holds if and only if it selects at least one node (existential quantification). Positive and nonpositive \texttt{integer} predicates \([\pm i]\) select the \(i\)-th node from the candidate sequence, and the \((n - i)\)-th node, respectively, from the sequence of \(n\) candidates in total. The former is a valid abbreviation for \([\text{position()} == i]\) in standard XPath; we add the latter as an analogous abbreviation for \([\text{position()} == \text{last()} - i]\). Logical connectives on predicates are defined as usual.

Various abbreviation rules apply; for instance, the shorthand \texttt{.//a[@href]} expands to the verbose form \texttt{self::node()/descendantOrSelf::node()/child::a[attribute::href]}. This translates to the following semantic object:

\[
\text{relative().step(Axis.self, node())} \\
.\text{step(Axis.descendantOrSelf, node())} \\
.\text{step(Axis.child, name("a"),} \\
.\text{exists(relative().step(Axis.attribute, "href")))}
\]

The relation between XPath predicates and candidate sequences, misleadingly called “node-sets” in the standard, is rather idiosyncratic (they can not be modeled adequately as plain sets) and the major challenge in this case study. A node-set is implicitly endowed with either of two predefined orders, namely \textit{forward} or \textit{reverse document order}. These orders are loosely specified by a preorder traversal of nodes, up to attribute permutation. A predicate filters the nodes in a node-set not purely by point-wise application, but may depend on some context, namely the position of the node in the node-set, starting from one, and the total number of members. This information is available in XPath via the “functions” \texttt{position()} and \texttt{last()}, respectively. It is realized in the API by the parameter \texttt{position} of method \texttt{Predicate.accepts} and the method \texttt{size()} of class \texttt{NodeSet}, respectively, to be supplied from the method \texttt{filter} of class \texttt{NodeSet}.

### 6.2 Pattern-Based Interpreter Design

The node-extracting semantics of the XPath language can be rendered naturally in the \texttt{Paisley} framework by contravariant lifting. An XPath expression can be
ancestor ≡ plus(parent) descendant ≡ plus(child)
ancestorOrSelf ≡ star(parent) descendantOrSelf ≡ star(child)
followingSibling ≡ plus(nextSibling) precedingSibling ≡ plus(previousSibling)
following ≡ ancestorOrSelf.on(followingSibling).on(descendantOrSelf)
preceding ≡ ancestorOrSelf.on(precedingSibling).on(descendantOrSelf)
self ≡ id

Fig. 6. Non-primitive XPath axes.

class Path {
    // ...
    public Path step(Axis axis, Test test, Predicate... predicates) {
        Motif ⟨? extends Node, Node⟩ r = axis.getMotif().on(test.getMotif());
        for (Predicate p : predicates)
            r = p.apply(r);
        return new Path(getMotif().on(r));
    }
}

Fig. 7. Implementation of composite path expressions.

applied to any node of an XML document, here implemented as DOM, and extracts some other nodes, possibly of a more special type, such as elements or attributes. This gives rise to a semantic type Motif ⟨? extends Node, Node⟩.

Except for the context-sensitivity of predicates, all language constructs could simply been given motif semantics and lumped together by composition.

XPath axes are all defined concisely in terms of motifs. Primitive DOM access operations from factory class XMLPatterns directly define the attribute, child and parent axes. Given the additional primitives next-/previousSibling, which are only implicit in the XPath standard, all other axes are definable in terms of elementary motif algebra; see Figure 6. Node tests are straightforward applications of test pattern lifting.

Atomic path expressions are realized by the document root access motif and the identity motif, for absolute and relative paths, respectively. Composite path expressions conceptually compose their constituents left to right. The exception are predicates, which are implemented as motif transforms in order to deal with context sensitivity. These are applied in order to the step basis, that is the composition of axis and test, for local filtering, and the result is then composed with the front of the path expression; see Fig. 7.

The real challenge is the implementation of node-set predicates: On the one hand, context information about relative element position and total node-set size must be provided, which transcends the context-free realm of pattern and motif composition. On the other hand, elements should be enumerated lazily, in
class Predicate {
    // . . .
    public Motif⟨Node, Node⟩ apply(final Motif⟨? extends Node, Node⟩ base) {
        return new Motif⟨? extends Node, Node⟩();
    }
    public Pattern⟨Node⟩ apply(Pattern⟨? super Node⟩ p) {
        return new MultiTransform⟨Node, Node⟩(p);
    }
    protected Iterable⟨Node⟩ apply(Node n) {
        return new NodeSet(base.lazyBindings(n)).filter(Predicate.this); // [*]
    }
};
}

class NodeSet {
    // . . .
    public NodeSet(Iterable⟨? extends Node⟩ elems) {
        this.elems = cache(elems);
    }
    public Iterable⟨Node⟩ filter(final Predicate pred) {
        return new Iterable⟨Node⟩() {
            public Iterator⟨Node⟩ iterator() {
                return new FilterIterator⟨Node⟩(elems.iterator()) {
                    int i = 0;
                    protected boolean accepts(Node candidate) {
                        return pred.accepts(NodeSet.this, ++i, candidate); // [*]
                    }
                };
            }
        }
    }
    public static Predicate exists(final Path cond) {
        return new Predicate() {
            public boolean accepts(NodeSet context, int position, Node node) {
                return cond.getMotif().apply(any()).match(node); // [*]
            }
        };
    }
    public static Predicate index(final int i) {
        return new Predicate() {
            public boolean accepts(NodeSet context, int position, Node node) {
                return position == (i > 0 ? i : context.size() - i); // [*]
            }
        };
    }
}

Fig. 8. Implementation of context-sensitive predicate filtering. See text for underlining and [*].
order to make tests for non-emptiness efficient and avoid scanning for unneeded solutions. Obviously, evaluation must switch transparently from lazy to eager strategy if the size of the node-set is observed. And lastly, for elegance reasons, we strive for an implementation that is as declarative as possible, with very limited amounts of specific imperative coding.

The solution is depicted in Figure 8. Generic Paisley API method calls are underlined for easy reference. The action of a predicate on a base motif requires control over the solution node-set. Hence it needs to intercept both the pattern parameter at the motif level and the target node parameter at the pattern level, by means of two nested anonymous classes. Then a lazy disjunction is spliced in, which enumerates the solutions of the base motif, wraps them in a local node-set and filters them context-sensitively.

The counterpart on the node-set side of the implementation works as follows: It wraps the lazy enumeration of candidate nodes is a collection, via the auxiliary method cache (definition not shown). This collection caches enumerated items for repeated access, and forces eager evaluation if its size() method is called.

The actual filtering operation yields a lazy enumeration that intercepts iterator creation, again by means of two nested anonymous classes. The iterator of the cached candidate collection is overwritten by an instance of the auxiliary abstract class FilterIterator⟨A⟩ (definition not shown) that in-/excludes elements ad-hoc, determined by the result of its method boolean accepts(A). This acceptance test is then routed back to the context-sensitive acceptance test of the given predicate, by addition of a single minuscule piece of explicit imperative (stateful) programming, namely a counter for the relative position. Note that, Java formal noise apart, the actual problem-specific code consists of five single-line statements, marked with [*].

Existential predicates forget their results, hence the invocation of the catch-all pattern any(). They prune the search after the first hit, as witnessed by the absence of a call to matchAgain() on the freshly created pattern. Eager evaluation is only ever triggered by index predicates via a call to context.size(). Logical predicate connectives are defined point-wise (not shown).

7 Experimental Evaluation

We have tested the performance and scalability of our implementation using material from XMark [8], a benchmark suite for XML-based large databases. The homepage of XMark [7] offers a downloadable tool to generate pseudo-random well-typed XML files according to a published DTD, with a linear size parameter. We have used the tools to produce test data files for size parameter values from 0.04 to 0.40 in increments of 0.04, where 0.01 corresponds roughly to 1 MiB of canonical XML. From the 20 published XQuery benchmark queries we have chosen as test cases the four where the entire logic is expressed in XPath rather than XQuery operations; see Fig. 9 (and Table 1 in the appendix).

For each pair of data file and query expression, we processed the document with lazyBindings of the XPath motif, and computed running times for extracting
the first solution and all solutions (in a loop), as well as the effective time (total time divided by number of solutions).

Timing values were obtained as real time with System.nanoTime(), median value of ten repetitions, with interspersed calls to System.gc(). In order to compensate deferred computation costs in the DOM implementation, we reused the same document instances for all successive queries. All experiments were performed on an Intel Core i5-3317U quad-core running at 1.70 GHz, with 8 GiB of physical memory, under Ubuntu 14.04 LTS (64bit), and Oracle Java SE 1.8.0_05-b13 with HotSpot 64bit Server VM 25.5-b02 and 800 MiB heap limit.

Our first quantitative goal, beyond highlighting the elegance and effectiveness of Paisley-style pattern specification, is to demonstrate the efficiency payoff of lazy pattern execution. Our second quantitative goal comes back to methodological arguments from the motivation section of this paper. Pattern matching has been hailed as a declarative tool that brings expressiveness for data querying extremely close to the hosting programming language environment. Tools for non-embedded domain-specific languages may be more powerful in many respects, but the burden of proof is on them that this power is worth the trouble incurred by the impedance mismatch with ordinary host code. Nevertheless, we should verify that the benefits of tight embedding produced by our methodological approach are not squandered by the implementation.

To that end, we repeated our experiments with the next-closest tool at hand, namely the Java on-board XPath implementation accessible as javax.xml.xpath.*. We compared both preparation and running times of Paisley XPath patterns with XPathExpression objects obtained via XPath.compile(String). The first solution and all solutions were obtained by calling XPathExpression.evaluate with the type parameter values NODE and NODESET, respectively.

The Paisley approach fares well, even as a non-embedded DSL. The Paisley preparation process (a simple recursive descent parser for full XPath generated with ANTLR7, followed by direct translation of abstract syntax nodes to pattern operations) is consistently faster than on-board compilation to XPathExpression objects, although the task may have been sped up marginally by considering

---

6 In our Java environment, com.sun.org.apache.xpath.internal.jaxp.XPathImpl.
7 http://www.antlr.org/
only a subset of the language after parsing. (See Table 2 in the appendix for details.) When patterns are constructed in embedded DSL style, using the API rather than textual input, static safety is improved, and even the small overhead eliminated, at the same time.

The differences in running times are so drastic that they can only be visualized meaningfully in logarithmic scale. Proportions range from on-board facilities being 13% faster for all solutions of Q06 at size 0.04, to being over 26,000 times slower for the first solution of Q06 at size 0.40. With the exception of the exhaustion of "brute-force" case Q06, Paisley is 1–2 (all solutions) or 2–4 (first solution) orders of magnitude faster, respectively; see Fig. 10. Accessing only the first solution with the on-board tools is particularly disappointing, being only marginally faster than exhausting all solutions. It appears that our motivation is confirmed, and conventionally developed tools are ill-suited to scalable lazy evaluation, where external demand governs internal control. (More detailed results are given in Fig. 12 the appendix.)

8 Conclusion

The experimental figures for the on-board tools have been obtained without any tweaking of features; hence there is large uncertainty in the amount of possible improvements. Therefore the comparison should not be taken too literally. But we feel that it is fair in a certain sense nevertheless: Our Paisley implementation has been obtained in the straightest possible manner, also without any tweaking. Its advantage lies thus chiefly in the fact that we have chosen the application domain, namely the specified XPath fragment, and tailored the tool design to avoid any complexity inessential to the task at hand. It is this light-weight flexibility by effective manual programming we wish to leverage with the Paisley
approach – the capabilities of existing, more heavy-weight tools with respect to automated, adaptive specialization are generally no match.

8.1 Related Work

Related work with regard to language design and implementation, and to object-oriented-logic programming, has been discussed in [9,10] and [11], respectively.

Purely declarative accounts of XPath, although theoretically interesting, have little technical impact on our main concern, the concrete embedding in a mainstream programming language. A few random examples: In [6], XPathLog is presented, adding variable binding capabilities and sound Herbrandt semantics to XPath, for a full-fledged logic programming language. In [4], an algorithmic analysis of XPath in terms of modal logic is given. The language fragment and experimental approach used there is a model predecessor for our own work, including the particular benchmark. In the recent paper [1], the implementation of XPath in the Haskell-like functional-logic programming language TOY is discussed. A combinatorial approach to XPath constructs very similar to Paisley is taken, which appears to corroborate our claims of a natural design.

References

## A Appendix: Supplementary Material

### Table 1. Size of input files and solution spaces for XPath expression test cases from the XMark benchmark.

<table>
<thead>
<tr>
<th>Parameter File (kiB)</th>
<th># Nodes</th>
<th># Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Q01</td>
</tr>
<tr>
<td>0.04</td>
<td>4 648</td>
<td>191 083</td>
</tr>
<tr>
<td>0.08</td>
<td>9 212</td>
<td>379 719</td>
</tr>
<tr>
<td>0.12</td>
<td>13 696</td>
<td>569 434</td>
</tr>
<tr>
<td>0.16</td>
<td>18 100</td>
<td>748 766</td>
</tr>
<tr>
<td>0.20</td>
<td>22 964</td>
<td>946 554</td>
</tr>
<tr>
<td>0.24</td>
<td>27 396</td>
<td>1 129 553</td>
</tr>
<tr>
<td>0.28</td>
<td>31 976</td>
<td>1 315 902</td>
</tr>
<tr>
<td>0.32</td>
<td>36 616</td>
<td>1 502 611</td>
</tr>
<tr>
<td>0.36</td>
<td>41 076</td>
<td>1 690 597</td>
</tr>
<tr>
<td>0.40</td>
<td>45 600</td>
<td>1 877 979</td>
</tr>
</tbody>
</table>

### Table 2. Preparation times for Paisley and Java on-board XPath implementations.

<table>
<thead>
<tr>
<th></th>
<th>Q01</th>
<th>Q06</th>
<th>Q15</th>
<th>Q16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paisley</td>
<td>56.40</td>
<td>101.73</td>
<td>89.26</td>
<td>154.32</td>
</tr>
<tr>
<td>On-Board</td>
<td>233.44</td>
<td>204.73</td>
<td>162.61</td>
<td>215.92</td>
</tr>
<tr>
<td>Ratio</td>
<td>4.14</td>
<td>2.01</td>
<td>1.82</td>
<td>1.40</td>
</tr>
</tbody>
</table>
enum Axis {
    ancestor, ancestorOrSelf, attribute, child, descendant, descendantOrSelf,
    following, followingSibling, parent, preceding, precedingSibling, self
    // . . .
}

public static Test node ();
public static Test text ();
public static Test name (String tagName);
public static Predicate exists (Path cond);
public static Predicate index (int i);
public static Predicate not (Predicate p);
public static Predicate and (Predicate p, Predicate q);
public static Predicate or (Predicate p, Predicate q);
public static Path absolute ();
public static Path relative ();

Fig. 11. XPath operations as combinators in the Paisley application-layer library.

Fig. 12. Running times for Paisley and Java on-board XPath implementations. Logarithmic scale; lower end of scale arbitrary.
Exploring Non-Determinism in Graph Algorithms

Nikita Danilenko
Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel
nda@informatik.uni-kiel.de

Abstract. Graph algorithms that are based on the computation of one or more paths are often written in an implicitly non-deterministic way, which suggests that the result of the algorithm does not depend on a particular path, but any path that satisfies a given property. Such algorithms provide an additional challenge in typical implementations, because one needs to replace the non-determinism with an actual implementation. In this paper we explore the effects of using non-determinism explicitly in the functional logic programming language Curry. To that end we consider three algorithms and implement them in a prototypically non-deterministic fashion.

1 Introduction

Consider a graph \( G = (V, E) \), two vertices \( s, t \in V \) and the question, whether there is a cycle in \( G \) that contains both vertices (i.e. both are contained in the same strong connected component). It is easy to come up with a solution for this task – simply check whether there are paths from \( s \) to \( t \) and from \( t \) to \( s \) in \( G \), if so, the answer is “yes”, otherwise it is “no”. Now all one needs to do is to come up with how to specify the existence of paths, which is just as straight-forward. Next, one could ask for an actual cycle in case such a cycle exists. For every path \( p \) from \( s \) to \( t \) and every path \( q \) from \( t \) to \( s \), the composition of \( p \) and \( q \) is such a cycle, i.e. the existence of a cycle does not depend on a particular choice of a path and there may be several different cycles.

When the above problems are considered in light of logic programming, one can use non-determinism to express the independence of choice and to compute one or more results from the specification alone. Aside from the usual benefit of the declarative approach of what to compute and not how, one can also observe that the different intermediate results (cycles or paths) yield the same overall answer (yes or no). While this is somewhat trivial in the above example, there are more sophisticated graph algorithms that rely on the computation of \textit{some} value with a special property and it may not be obvious at all that different choices of such values yield the same results in the end. This invariance can be considered as a certain kind of confluence and the possibility to observe this invariance is a useful tool, particularly in teaching.
In this paper we consider graph algorithms that are based on the computation of paths. We begin with a simple observation relating the path search strategy to the solution search strategies and proceed to provide solutions to two non-trivial graph problems, namely the maximum matching problem and the maximal flow problem and provide examples of matchings and flows. Our main focus is on experimentation and applications in teaching, where theoretical results are likely to be followed by examples. Observing implicit non-determinism explicitly is of great assistance in this context since one can witness the invariance of the solution with respect to different choices that lead to it.

All code in this paper is written in Curry [12] and compiled with KiCS2 [5], which can be obtained at http://www-ps.informatik.uni-kiel.de/kics2/. Throughout the paper we refer to several functions and modules, all of which can be found using the search engine Currygle https://www-ps.informatik.uni-kiel.de/kics2/currygle/. A polished version of the presented code is available at GitHub under https://github.com/nikitaDanilenko/ndga.

2 Preliminaries

A graph is a pair $G = (V,E)$, where $V$ is a non-empty finite set and $E \subseteq V \times V$. This is to say that we consider all graphs to be directed and realise graphs in which the direction does not matter as symmetric directed graphs. For any $v \in V$ we set $N_\to(v) := \{ w \in V \mid (v, w) \in E \}$ and $N_\leftarrow(v) := \{ w \in V \mid (w, v) \in E \}$; elements of $N_\to(v)$ are called successors and those of $N_\leftarrow(v)$ predecessors of $v$.

A path in a graph is an injective sequence of vertices, i.e. a sequence that does not contain multiple occurrences of the same vertex. Since $V$ is finite, every path traverses at most $|V|$ vertices. We represent vertices by integers, edges by pairs of vertices and paths by lists of distinct vertices, where the distinctness is required implicitly. We discuss this design decision in Section 3.

```
type Vertex = Int
$type Edge = (Vertex, Vertex)
type Path = [Vertex]
```

Graphs are represented by an adjacency list model, where the adjacency lists are sorted in ascending natural order of the integers. These adjacency lists are stored in a finite map, one implementation of which is provided in the standard KiCS2 library FiniteMap as the data structure FM key value.

```
data Graph = Graph (FM Vertex [Vertex])
```

Additionally, we use sets of vertices explicitly for the maintenance of visited vertices. We use finite maps provided by KiCS2 for a representation of sets of vertices, where the functions emptyFM, addToFM, elemFM, delFromFM are provided, too. The function emptyFM is parametrised over an irreflexive order predicate which in our case is ($<$) :: Int →Int →Bool.

```
type VertexSet = FM Vertex ()
```
empty :: VertexSet
empty = emptyFM (<)
insert :: Vertex → VertexSet → VertexSet
insert i m = addToFM m i ()
inSet :: Vertex → VertexSet → Bool
inSet = elemFM
remove :: Vertex → VertexSet → VertexSet
remove = flip delFromFM
vertexListToSet :: [Vertex] → VertexSet
vertexListToSet = foldr insert empty

For the purpose of demonstration we use the two example graphs from the Figures 1, 2 in the following two sections and assume that they are implemented in the values graph1 and graph2 respectively.

![Fig. 1. Example graph $G_1$](image1)

![Fig. 2. Example graph $G_2$](image2)

3 Reachability and Paths

One standard example of logic programming in graph theory is the check whether a vertex set $ts$ is reachable from a given vertex $s$. This is the case iff $s \in ts$ or if there is a successor $i \in V$ of $s$, such that $ts$ is reachable from $i$. Since successors can lead back to already visited vertices, one additionally has to check, if the successor has been visited or not\(^1\). The implementation of the above may be as follows.

reachable :: Graph → Vertex → VertexSet → Success
reachable g from ts = find empty from where
find vis s | s ‘inSet’ ts = success
| isEdge g s i ∧ ¬ (i ‘inSet’ vis) = find (insert s vis) i where i free

This implementation contains a high level of abstraction and is very close to the original specification. Also, instead of defining how to access the successors of

\(^1\) Avoiding this check can easily result in non-termination: consider the graph \(\{(0, 1, 2), \{(0, 1), (1, 0), (0, 2)\}\}\) with DFS. Then checking whether 2 is reachable from 0 diverges, because the loop (0, 1, 0) is entered before checking the other successors of 0. A similar example shows the non-termination in case of using BFS.
s, this implementation relies only on a test whether a certain edge is contained in the graph and the use of a free variable to find a fitting edge. The technique of translating existential quantifications into free variables is common in logic programming.

Instead of just checking for the existence of a path, one can ask for an actual path between two vertices. It is easy to implement such a search by modifying the above function.

\[
\text{path} :: \text{Graph} \rightarrow \text{Vertex} \rightarrow \text{VertexSet} \rightarrow \text{Path}
\]

\[
\text{path } g \text{ from } ts = \text{find empty from where find vis s | s \text{ inSet } ts} = [s] | \text{isEdge } g \text{ s i \&\& (i \text{ inSet vis})} = s : \text{find (insert s vis) i where i free}
\]

\[
\text{pathToSingle} :: \text{Graph} \rightarrow \text{Vertex} \rightarrow \text{Vertex} \rightarrow \text{Path}
\]

\[
\text{pathToSingle } g \text{ from } t = \text{path } g \text{ from (vertexListToSet [t])}
\]

Again, the function is non-deterministic in its choice of the successor. There are two interesting consequences of this implementation. First of all, using the interactive mode of KiCS2 one can find all paths between two given vertices without any additional work. Alternatively, when all paths are required in a program, one can use set functions \[2\] to encapsulate the search and collect all results. Set functions are provided in the KiCS2 library SetFunctions, which provides functions setK (for \(K \in \{0, \ldots, 7\}\))

\[
\text{setK} :: (a_0 \rightarrow \ldots \rightarrow a_{K-1} \rightarrow b) \rightarrow a_0 \rightarrow \ldots \rightarrow a_{K-1} \rightarrow \text{Values } b
\]

which turn a function of a given arity into a set function, where Values b is the list of all results. Assuming that the graph from Figure 2 is implemented as graph2 we can test the following results.

\[
\text{kics2}\gg\text{set3 path graph2 0 (vertexListToSet [5, 8])}
\]

\[
(\text{Values } [[0, 1, 2, 5], [0, 3, 6, 7, 4, 5], [0, 3, 6, 7, 8]])
\]

\[
\text{kics2}\gg\text{set3 pathToSingle graph2 0 8}
\]

\[
(\text{Values } [[0, 1, 2, 5, 4, 7, 8], [0, 1, 2, 5, 8], [0, 3, 6, 7, 4, 5, 8], [0, 3, 6, 7, 8]])
\]

The second consequence is that the path search is based upon the order in which the free variable \(i\) is instantiated, which in turn depends on the search strategy. The search strategy can be chosen in KiCS2 either interactively (by using :set STRATEGY) or directly in the code. The latter can be accomplished by explicitly representing the search space as a SearchTree from the homonymous Curry module and using strategy dependent operations on the tree to obtain actual results. For example, we get:

\[
\text{kics2}\gg\text{someValueWith bfsStrategy (pathToSingle graph1 2 5)}
\]

\[
[2, 1, 4, 5]
\]

\[
\text{kics2}\gg\text{someValueWith dfsStrategy (pathToSingle graph1 2 5)}
\]

\[
[2, 1, 0, 3, 4, 5]
\]

Note that these two results correspond precisely to what the respective graph-theoretic strategies yield. In fact, in the above implementation one can decide
between choosing a successor and descending the recursive call (DFS) or checking other successors first and descending afterwards (BFS). However, both strategies will find all paths eventually, so that the above observation holds for the first result, but not necessarily for subsequent results (this can be observed using set functions, but not with someValueWith, because the latter is deterministic).

```kics2> set3With bfsStrategy pathToSingle graph1 2 5
(Values [[2,1,4,5],[2,1,0,3,4,5]])
```

```kics2> set3With dfsStrategy pathToSingle graph1 2 5
(Values [[2,1,0,3,4,5],[2,1,4,5]])
```

Since we check whether a vertex has been visited before by hand, the condition that the vertices of every resulting Path are distinct is maintained. Using an implementation based upon the constrained constructor pattern one can disallow the creation of invalid paths. However, the search itself requires access to previously visited vertices to avoid (infinite) repetition and thus its implementation would remain essentially the same as above. We omit the smart constructor approach for the sake of simplicity.

4 Maximum Matchings

A more sophisticated application of non-deterministic path search is the algorithm for finding maximum matchings. A matching in a symmetric graph is a set \( M \subseteq E \) that is symmetric and functional\(^2\). In other words a matching consists of edges that do not share common vertices. A matching is called maximum matching iff there is no matching with a strictly greater cardinality. Clearly, maximum matchings are not necessarily unique, since they are maximal elements with respect to cardinality. Figure 3 shows some examples of matchings in the example graph from Figure 2, where the bold lines show matching edges and the dashed lines are non-matching edges.

![Examples of matchings](image)

Fig. 3. Examples of matchings.

\(^2\) I.e. \( \forall x, y, z \in V : (x, y) \in M \land (x, z) \in M \Rightarrow y = z. \)
There are several natural applications for maximum matchings, like assigning jobs to people, processes to machines or, more classically, spouses to one another. In all cases one may wish to make as many one-to-one assignments as possible, which translates directly into the search for a maximum matching. All of these examples can be modelled with so-called bipartite graphs, which are graphs that allow a partition of vertices into two sets, such that all edges of the graph connect only vertices from two different sets. For bipartite graphs there is a simple imperative algorithm that computes maximum matchings, while the algorithms for the non-bipartite case is significantly more sophisticated [9].

The Berge theorem [3] characterises maximum matchings and provides an algorithm for finding such matchings. We state it exactly as in [6].

**Theorem 1 (Characterisation of maximum matchings, Berge).**

Let $M \subseteq E$ be a matching. Let $\oplus$ denote the symmetric difference. For a path $p$ we denote the set of the edges along $p$ by $E(p)$. A path is called $M$-augmenting iff it starts and ends in a vertex that is not contained in some edge of $M$ and alternates between edges of $E \setminus M$ and those of $M$. Then we have

1. If there are no $M$-augmenting paths in $G$, then $M$ is a maximum matching.
2. If there is an $M$-augmenting path $p$ in $G$, then $M \oplus E(p)$ is a larger matching.

This theorem can be used for an actual computation of a maximum matching. Starting with the empty matching one searches for an augmenting path and if such a path exists the current matching is expanded. If on the other hand no such path exists, the current matching is already a maximum matching.

For the remainder of this section we assume all graphs to be symmetric, in particular this concerns all graph arguments in functions. This condition can (and should!) be checked in a proper application, but to avoid additional code that is not necessary for the presentation, we assume this check to have been performed beforehand implicitly.

Let us first deal with the matching augmentation. Given a matching $M$ and an augmenting path $p$ Berge’s theorem states that $M \oplus E(p)$ is a larger matching. Once we have computed this matching, we will check whether there is an augmenting path, using $M \oplus E(p)$ instead of $M$, which requires the computation of $E \setminus (M \oplus E(p))$. As we have stated in [6] it is simple to see that

$$E \setminus (M \oplus E(p)) = (E \setminus M) \oplus E(p),$$

so that both $M$ and $E \setminus M$ are updated in the same fashion. Suppose that

$$xorBiEdges :: \text{Graph} \to \text{[Edge]} \to \text{Graph}$$

traverses a list of edges and for every edge it adds its undirected version (i.e. both directions) to the graph if the edge is not already present in the graph and removes both directions otherwise. Then the augmentation update can be implemented as follows, where we assume that $m$ is the current matching and $notM$ is its complement in $E$ (i.e. $notM = E \setminus m$).
augmentBy :: Path → Graph → Graph → (Graph, Graph)

augmentBy path m notM = (m 'xorBiEdges' es, notM 'xorBiEdges' es) where
es = zip path (tail path)

Without logic means checking the existence of an augmenting path and finding such a path in the positive case is quite technical\(^3\). Using logic means however, we can specify an augmenting path by first dealing with alternating paths and then adding the conditions for the first and last vertex. Suppose we have a target set of vertices \(ts :: \text{VertexSet}\), a starting vertex \(s\) and a list of graphs \((g: gs)::[\text{Graph}]\) that we want to traverse in sequence. Then an alternating path from \(s\) to \(ts\) through \(g: gs\) exists iff \(s \in ts\) holds or there is a successor \(i\) of \(s\) in \(g\) and there exists an alternating path from \(i\) to \(ts\) through \(gs + [g]\). The intention is that we will use the list \([notM, m]\) in our application, where \(m\) is the current matching and \(notM\) is its complement in \(E\). Except for the traversal of multiple graphs in a cyclic order\(^4\), the above specification is very similar to the one that specified the existence of a path. Similarly, the corresponding function is easy to modify to return an alternating path as well.

\[
\text{alternatingPath} :: [\text{Graph}] → \text{Vertex} → \text{VertexSet} → \text{Path}
\]

\[
\text{alternatingPath}\ g\ s\ ts = \text{find empty from}\ g\ s\ ts\ where
\]

\[
\text{find vis} 's' (g: gs) = \cases{s \inSet ts = [s] \mid \text{isEdge}\ g\ s\ i\ \&\ \neg(i \inSet vis) = s : \text{find} (\text{insert} s\ vis)\ i\ (gs + [g])}\ where\ i\ \text{free}
\]

With this function we can find augmenting paths, too. Let us assume that we have a function at hand that computes those vertices of a symmetric graph which do not have any neighbours (i.e. successors, due to symmetry).

\[
\text{noSuccessors} :: \text{Graph} → \text{VertexSet}
\]

By definition an augmenting path starts and ends in a vertex that is not covered by the matching and since it is a path the start vertex should also be different from the end vertex, which is not guaranteed by the \text{alternatingPath} function from above. Fortunately, this is easily corrected by simply excluding the start vertex from the target vertex set. Again, we assume that \(m\) is a matching and \(notM\) is its complement in \(E\).

\[
\text{augmentingPath} :: \text{Graph} → \text{Graph} → \text{Path}
\]

\[
\text{augmentingPath}\ m\ notM\ |\ s \inSet unc = \text{alternatingPath}\ [notM, m]\ s\ (s \\text{‘remove’}\ unc)
\]

\[
\text{where}\ unc = \text{noSuccessors}\ m\ s\ \text{free}
\]

This function is prototypical by design – guessing a possible start vertex and trying to find a path is obviously not the most efficient way of finding an augmenting path. Instead, one could either modify the search function in a fashion

\(^3\) The usual procedure is to implement a modified breadth-first search, which explicitly alternates between two graphs.

\(^4\) We have implemented the cyclic list traversal inefficiently by adding the first element to the end of the list for demonstration purposes only. It is simple to replace this implementation by either functional lists or queues, both of which allow adding an element to the end in constant time.
that searches from a set of vertices instead of a single one or to use an explicitly parallel search strategy. These improvements lead to a less declarative look-and-feel, which is why we use the above version.

If there is no augmenting path in the graph then this function does not return any value. We use negation as failure with set functions to obtain a function that repeats the search for an augmenting path until there is no such path left.

maximumMatching :: Graph → Graph
maximumMatching g = go (emptyGraph (graphSize g), g) where
  go (m, notM) | isEmpty ps = m
               | otherwise = go (augmentBy (chooseValue ps) m notM)
  where ps = set2With augmentingPath m notM

The function graphSize returns the number of vertices in the graph, emptyGraph creates an empty graph of a given size, isEmpty :: Values a → Bool checks whether the list of values is empty or not and chooseValue :: Values a → a nondeterministically chooses a value from the list of all values. Additionally, we can parametrise the maximumMatching function by a search strategy that is passed to set2With which is a version of set2 that is parametrised by a search strategy.

We test the above function in the interactive mode of KiCS2 on the example graph \( G_2 \) from Figure 2. The function graphEdges :: Graph → [Edge] computes the edges of a graph and we use it for an uncluttered output.

kics2> graphEdges (maximumMatching graph2)
[(0,1),(1,0),(2,5),(3,6),(4,7),(5,2),(6,3),(7,4)]
More values? [Y(es)/n(o)/a(ll)] Y
<< four more times the same result >>
More values? [Y(es)/n(o)/a(ll)] Y
[(0,1),(1,0),(2,5),(3,6),(5,2),(6,3),(7,8),(8,7)]
More values? [Y(es)/n(o)/a(ll)] n

Observe that the first results are the first maximum matching from Figure 3.

As we have mentioned before, one typically distinguishes the cases of bipartite and non-bipartite graphs, because the search for an augmenting path is simpler in the former case. The above implementation does not rely on the graph being bipartite and will in fact work for non-bipartite graphs too, even those where usual algorithms will fail. The essence of this failure is known to be that every search is guided by some vertex ordering and in the non-bipartite case one can always create examples where a vertex is marked as visited prematurely, thus excluding this vertex from possible further searches. This problem cannot occur in the bipartite case – if there is an augmenting path, it can be found with every vertex ordering. The above implementation however, uses all possible vertex orderings and thus always finds a path if it exists. Clearly, this comes at the price of not being efficient (polynomial), but it is still interesting to observe that the function itself still yields the correct results, but only its complexity changes.

Another interesting observation is that such an implementation is very well suited for presentation, particularly in teaching. We have stated before that maximum matchings are not necessarily unique and the above function can be
used interactively in KiCS2 to find different maximum matchings. Similarly, for any given matching there might be several augmenting paths and one can again check that the choice of a particular path does not matter for the maximality of the result. In graphs with unique maximum matchings the confluence of the algorithm is observable by considering all solutions and removing duplicates. In the graph from Figure 1 there is exactly one maximum matching, but it can be found with the above function in 184 ways\(^5\). This demonstration of confluence is again interesting in teaching to indicate that even a possibly non-deterministic algorithm can yield a deterministic result.

Finally, we point out that there are more efficient algorithms that compute maximum matchings, namely the Hopcroft-Karp algorithm [13] and the Mucha-Sankowski [15] algorithm. The latter is based upon Gaussian elimination and not directly related to a search for paths. The former algorithm, however, computes in every augmentation step not just a single augmenting path, but a set of shortest, pairwise disjoint augmenting paths that is maximal with respect to inclusion. A function that computes such a set can be implemented as a combination of two searches (first a breadth-first search, followed by a modified depth-first search, cf. [7]). Interestingly, the algorithm still contains a very similar non-deterministic component as above, since there can be several sets that are all maximal and again, the correctness of the output (and sometimes even the output itself) does not depend on a particular choice of such a set. The actual implementation is not particularly difficult, but it explicitly relies on the fact that the BFS returns a graph that is the union of all shortest paths from the source set to the target set, rather than the property that some path has been found using BFS implicitly.

5 Maximal Flows in Networks

A problem that is conceptually related to maximum matchings is that of a maximal flow through a network. A network is a quadruple \(N = ((V, E), s, t, c)\) that consists of an asymmetric graph\(^6\) \((V, E)\), two designated vertices \(s, t \in V\) (where \(s\) is called source and \(t\) is called sink) such that \(s \neq t\) and a capacity function\(^7\) \(c : E \rightarrow \mathbb{N}\). To avoid unnecessary brackets whenever \(X\) is a set and \(h : E \rightarrow X\), we write \(h(x, y)\) instead of \(h((x, y))\). For any function \(g : E \rightarrow \mathbb{N}\) let

\[
\partial g : V \rightarrow \mathbb{Z}, \quad v \mapsto \left( \sum_{w \in N_\rightarrow(v)} g(v, w) \right) - \left( \sum_{w \in N_\leftarrow(v)} g(w, v) \right).
\]

The function \(\partial g\) measures for every vertex the difference between the amount of all outgoing values and the incoming values. A flow is a function \(f : E \rightarrow \mathbb{N}\) that

\(^5\) Using \(\text{liftIO} \circ \text{length} \circ \text{values2list} \circ \text{set1} \ \text{maximumMatching}\) instead of just the \text{maximumMatching} function yields the number of successfully found matchings.

\(^6\) That is that for all \(v, w \in V\) such that \((v, w) \in E\), we have \((w, v) \notin E\).

\(^7\) Typically, one chooses \(c : E \rightarrow \mathbb{Q}_{\geq 0}\), but since only finite graphs are considered, it is possible to multiply \(c\) by the least common multiple of its image values and, if necessary, to divide them later.
satisfies \( f \leq c \) (pointwise) and for all \( v \in V \setminus \{s, t\} \) we have \( \partial f(v) = 0 \). This is known as the Kirchhoff’s law “what goes in, must come out”. The value of a flow \( f \) is defined as \( |f| := \partial f(s) \) and a maximal flow is a flow that has a maximal flow value. In typical applications there are no edges leading into the source, which then allows the intuition that the flow value is the amount of goods that is sent through the network from the source.

Fig. 4. Examples of a network and flows.

Flow problems are related (among many others) to routing problems, where one wishes to send a certain amount of goods through different distribution lines that have a limited capacity only (e.g. traffic or electrical current). The Kirchhoff law then simply states that there is no loss of goods along the way. There has been extensive research on finding maximal flows (cf. [11, 10, 7] for overviews and results) and efficient algorithms are known. We consider the original algorithm and variations thereof. The original algorithm for finding maximal flows is due to a theorem by Ford and Fulkerson, which is based upon [14].

**Theorem 2 (Characterisation of maximal flows, Ford & Fulkerson).**

Let \( N = ((V, E), s, t, c) \) be a network and \( f : E \to \mathbb{N} \) a flow. Let

\[
    c_f : E \cup E^{-1} \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} 
        c(v, w) - f(v, w) & : (v, w) \in E \\
        f(w, v) & : \text{otherwise}
    \end{cases}
\]

and \( E_f := \{ e \in E \cup E^{-1} \mid c_f(e) > 0 \} \). We call \( c_f \) the residual capacity w.r.t. \( f \).

Then the following hold:

1. If there is no path from \( s \) to \( t \) in \( (V, E_f) \), then \( f \) is a maximal flow.
2. If \( p \) is a path from \( s \) to \( t \) in \( (V, E_f) \), let \( \varepsilon := \min \{ c_f(e) \mid e \in E(p) \} \) and

\[
    f_p : E \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} 
        f(v, w) + \varepsilon & : (v, w) \in E(p) \cap E \\
        f(v, w) - \varepsilon & : (w, v) \in E(p) \cap E \\
        f(v, w) & : \text{otherwise.}
    \end{cases}
\]

Then \( f_p \) is a flow and \( |f_p| = |f| + \varepsilon \) (\( p \) is called a flow-augmenting path).
This theorem is very similar to the Berge theorem from the previous section. In fact, it is known that the problem of finding maximum matchings in bipartite graphs can be solved using a particular instance of the flow problem. However, this technique works only for bipartite graphs in which an explicit bipartition is known, while the presented strategy does not require an explicit bipartition.

The theorem of Ford and Fulkerson provides an algorithm for finding maximum flows, which checks for the existence of a path and improves the flow in the positive case. When searching for any path, the algorithm is known as the Ford-Fulkerson algorithm, which is not necessarily polynomial in the graph size. When searching for shortest paths, this algorithm is known as the Edmonds-Karp algorithm [8] and has a complexity that is polynomial in graph size. Assuming a deterministic choice of the first element of a list of non-deterministically found augmenting paths, this fact can be reflected in Curry by specifying an explicit strategy for the path search. An additional variation of the algorithm is finding a set of paths from \( s \) to \( t \) that are disjoint up to \( s \) and \( t \), such that the set is maximal with respect to inclusion and use all paths from this set for an improvement. This is a version of the Dinitz [7] algorithm and is more efficient than the Edmonds-Karp algorithm, since it is possible to implement the search for such a set in a fashion that is just as complex as finding a single path.

For an implementation we consider capacities and flows to be functions from \( V \times V \) to \( \mathbb{N} \), which yield zeroes on \( (V \times V) \setminus E \). For any \( g, h : V \times V \to \mathbb{Z} \) set

\[
\text{swap}(g) : V \times V \to \mathbb{Z}, \quad (x, y) \mapsto g(y, x)
\]

\[
g \sqcap h : V \times V \to \mathbb{Z}, \quad e \mapsto \begin{cases} h(e) : g(e) \neq 0 \\ 0 : \text{otherwise.} \end{cases}
\]

Then “swap” is an uncurried version of the function \( \text{flip} \) and \( \sqcap \) is the left-forgetful intersection of functions. Let \( \oplus, \ominus \) be the pointwise addition and subtraction of functions respectively and \( \bullet \) the multiplication of a function with a constant.

From now on we assume that \( c \) is a capacity and \( f \) is a flow. The residual capacity \( c_f \) as in the Ford-Fulkerson theorem can be computed as

\[
c_f := c \ominus f \oplus \text{swap}(f).
\]

Indeed, for every \( v, w \in V \) we find that due to the asymmetry at most one of the two values \( f(v, w) \) and \( f(w, v) \) can be non-zero, which yields

\[
(c \ominus f \oplus \text{swap}(f))(v, w) = c(v, w) - f(v, w) + f(w, v)
\]

\[
= \begin{cases} c(v, w) - f(v, w) : (v, w) \in E \\ f(w, v) : (v, w) \in E^{-1} = c_f(v, w), \\ 0 : \text{otherwise} \end{cases}
\]

\[\text{However, we use a non-deterministic choice of an augmenting path to be able to observe the different choices.}\]
where the last step is only true up to the extension of $c_f$ to $V \times V$. Now let $p$ be a path from $s$ to $t$ in $(V, E_f)$ and let $\varepsilon := \min \{ c_f(e) \mid e \in E(p) \}$. Then set

$$\sigma_p : V \times V \rightarrow \mathbb{Z}, \quad e \mapsto \begin{cases} 1 & : e \in E(p) \\ 0 & : \text{otherwise,} \end{cases}$$

i.e. $\sigma_p$ is the characteristic function of $E(p)$ in $V \times V$, and

$$u_p := \varepsilon \bullet (c \cap (\sigma_p \ominus \text{swap}(\sigma_p))).$$

The value $u_p$ is a “point-free” version of the flow update indicated by the Ford-Fulkerson theorem: the term $\sigma_p \ominus \text{swap}(\sigma_p)$ produces a function that yields 1 along the edges on $p$ and $-1$ along all the reversed edges along $p$. The intersection produces a function that is 1 along the edges along $p$ which are contained in $E$ and $-1$ on those that are contained in $E^{-1}$. One easily verifies that $f_p = f \oplus u_p$. With this we can compute as follows, where all of the arithmetic rules below follow immediately from their pointwise counterparts.

$$c_{fp} = c \ominus f_p \oplus \text{swap}(f_p) = c \ominus (f \oplus u_p) \oplus \text{swap}(f \oplus u_p)$$

$$= c \ominus f \ominus u_p \oplus \text{swap}(f) \oplus \text{swap}(u_p) = (c \ominus f \ominus \text{swap}(f)) \ominus u_p \ominus \text{swap}(u_p)$$

$$= c_f \ominus u_p \ominus \text{swap}(u_p).$$

Thus we can update the flow and the residual capacity using only the changes provided by the path, which reduces the number of necessary computations. To implement paths with values along traversed edges, we use the data type

```
data Path a = Final Vertex | From Vertex a (Path a)
```

and assume a function $\text{toEdges} :: \text{Path } a \rightarrow [(\text{Edge}, a)]$ to be at hand that collects all edges along the path with their corresponding values. We then model capacities and flows using finite maps $\text{FM}$ with $(\text{Vertex}, \text{Vertex})$ keys and $\text{Int}$ values.$^9$

```
type EdgeMap = FM (Vertex, Vertex) Int
```

The functions $(\oplus), (\ominus), (\cap) :: \text{EdgeMap} \rightarrow \text{EdgeMap} \rightarrow \text{EdgeMap}, (\bullet) :: \text{Int} \rightarrow \text{EdgeMap} \rightarrow \text{EdgeMap}$ and $\text{swap} :: \text{EdgeMap} \rightarrow \text{EdgeMap}$ are rather simple to define using standard operations on $\text{FiniteMaps}$ (e.g. $(\oplus) = \text{plusFM}_C (\oplus)$). We can then implement the flow augmentation as follows, where the first argument is an augmenting path, the second one denotes the original capacity, the third one is the current residual capacity and the fourth one is the current flow.

```
augmentBy :: Path Int \rightarrow \text{EdgeMap} \rightarrow \text{EdgeMap} \rightarrow \text{EdgeMap} \rightarrow (\text{EdgeMap}, \text{EdgeMap})
augmentBy p c c f = ((c_f \ominus u_p) \ominus \text{swap } u_p, f \oplus u_p) \text{ where}
```

$^9$ We could define a data type for natural numbers as well, but then manual conversion between naturals and integers requires some additional overhead; since this implementation is for demonstration, only, we assume the correct usage.
\[ u_p = \varepsilon \cdot (c \cap (\sigma_p \ominus \text{swap } \sigma_p)) \quad \text{-- the update} \]
\[ \varepsilon = \text{minlist } (\text{map } \text{snd } \text{edges}) \quad \text{-- the minimum along the path} \]
\[ \sigma_p = \text{fromList } (\text{map } (\lambda(e, _) \rightarrow (e, 1)) \text{edges}) \quad \text{-- the characteristic function} \]
\[ \text{edges} = \text{toEdges } p \]

Note that the actual result is an exact copy of the computations from above. The maximisation function can then be realised in a fashion very similar to the one we used for matchings, but explicitly parametrised over a search strategy. This implementation adds an additional non-deterministic component, namely the choice of the actual augmenting path. The strategy for this choice is given by the top-level search strategy.

\[
\text{data } \text{Network} = \text{Network } \text{Graph } \text{Vertex } \text{Vertex } \text{EdgeMap}
\]

\[
\text{maximalFlowWith } :: \text{Strategy } (\text{Path } \text{Int}) \rightarrow \text{Network } \rightarrow \text{EdgeMap}
\]
\[
\text{maximalFlowWith } \text{str } (\text{Network } \_ \_ \_ \_ \text{t }) = \text{go } (\_ , \text{empty}) \text{ where}
\]
\[
\text{go } (\text{cf}, f) \mid \text{isEmpty } \text{ps } = f
\]
\[
| \text{otherwise } = \text{go } (\text{augmentBy } (\text{chooseValue } \text{ps}) \text{ c } \text{cf } \text{f})
\]
\[
\text{where } \text{ps } = \text{set1With } \text{str } \text{findAugmenting } \text{cf}
\]
\[
\text{findAugmenting } = \text{augmenting } \text{s } \text{t}
\]

All that remains is the \text{augmenting} function. In essence, it is another path search, but this time we use the capacity map to check for existing edges, because edges in the residual graph exist iff their capacity is positive.

\[
\text{augmenting } :: \text{Vertex } \rightarrow \text{Vertex } \rightarrow \text{EdgeMap } \rightarrow \text{Path } \text{Int}
\]
\[
\text{augmenting } \text{s } \text{t } \text{capacity } = \text{go } (\text{emptyFM } (<)) \text{ s where}
\]
\[
\text{go } \text{vis } \text{from } = \text{Final from}
\]
\[
\quad | \text{cf } > 0 \wedge \neg (\text{from } ' \text{inSet } \text{vis }) = \text{From from } \text{cf } (\text{go } (\text{insert from } \text{vis } \_))
\]
\[
\text{where } \text{i free}
\]
\[
\text{cf } = \text{capacity } ! (\text{from }, \text{i})
\]

The non-determinism is again enclosed in the path search. Just as was the case with matchings, maximal flows are usually not unique and the presented implementation can be used to find all possibilities. Still, every maximal flow has the same flow value and this fact can be observed by defining the \( \partial \) function.

Again, we test our implementation with the example network from Figure 4 and wrap the call in the function \text{showEdgeMap} :: \text{EdgeMap } \rightarrow \text{String} that pretty-prints key-value pairs as \text{key } \rightarrow \text{value}.

\[
\text{kics2> showEdgeMap } (\text{maximalFlowWith } \text{bfsStrategy}) \text{ network1}
\]
\[
(2, 5) \rightarrow 5, (0, 2) \rightarrow 5, (0, 1) \rightarrow 7, (1, 4) \rightarrow 3, (0, 3) \rightarrow 5, (1, 5) \rightarrow 4, (5, 6) \rightarrow 5,
\]
\[
(4, 7) \rightarrow 7, (3, 6) \rightarrow 5, (5, 4) \rightarrow 4, (6, 7) \rightarrow 10
\]
\[
\text{More values? } [Y(es)/n(o)/a(ll)] \text{ y}
\]
\[
(2, 5) \rightarrow 3, (0, 2) \rightarrow 3, (0, 1) \rightarrow 7, (1, 4) \rightarrow 3, (0, 3) \rightarrow 7, (1, 5) \rightarrow 4, (5, 6) \rightarrow 5,
\]
\[
(4, 7) \rightarrow 7, (3, 6) \rightarrow 5, (3, 5) \rightarrow 2, (5, 4) \rightarrow 4, (6, 7) \rightarrow 10
\]
\[
\text{More values? } [Y(es)/n(o)/a(ll)] \text{ n}
\]

The first flow is exactly the maximal flow from Figure 4 and the second one demonstrates that maximal flows can variate in their edges as well as the flow values along the edges. Clearly, both flows have a flow value of 17.
6 Discussion

We have demonstrated how to apply non-deterministic path computations to compute the solution to some selected graph problems. The presented functions are not the most efficient ones by design, but intended as prototypes for demonstration. This prototypical approach has the additional advantage of being simple and declarative. Clearly, several parts of our implementations can be improved or described in a more declarative or more efficient fashion, but our focus is on the non-deterministic path computations, which are the essence of all the described algorithms.

The overall strategy of all computations in this paper can be considered as the computation of the preimages of a given function which needs to be maximised. For instance in case of flows this function is $|\cdot| : F \to \mathbb{N}, \ f \mapsto |f|$, where $F$ is the set of all flows in a given network. Similarly, every improvement step is a preimage computation for the function $\text{improve} : P \to F, \ p \mapsto f_p$ where $P$ is the set of all flow-augmenting paths with respect to a given flow $f$. It is interesting to note that every preimage choice is a branching point in the overall computation and that every new choice can allow different branches that will still lead to the same final result. In the case of maximum matchings two different maximum matchings are distinct only in one or more edges, for flows we can observe significantly more variation, since not only the edges that have non-zero flow can be different, but even different non-zero flow values for the same edge are possible.

Being able to observe such differences is interesting in its own right, but can be of particular interest in teaching. In case of the maximum matching function there is no difference between the search strategies. The maximal flow function on the other hand behaves differently, as we have stated above, and the choice of a depth-first strategy combined with an “unlucky” vertex ordering yields a non-polynomial complexity, while the very same concrete program with a breadth-first search strategy is in a completely different complexity class.

In our applications, results may be computed repeatedly through different branches. Since set functions are implemented using lists, removing duplicates is possible but costly, since a straight-forward graph comparison takes $O(|E|)$ operations. For matchings this is slightly better, since every matching has only $O(|V|)$ edges, which makes the naive duplicate removal less inefficient. Still, with focus on experimentation and teaching, an inefficient duplicate removal is still rather simple, because the Curry function $\text{nub} :: [\alpha] \to [\alpha]$ removes duplicates from a given list (of ground terms).

The presented implementation and the ideas behind it are not exclusive to Curry, but passing search strategies to a set function is already a built-in feature of Curry and particularly KiCS2. However, KiCS2 translates Curry programs to Haskell and using non-deteminism monads (see [4]) and replacing logic variables by overlapping rules (as in [1]) one can obtain a purely functional implementation. Such an implementation should be portable to every other language that supports higher-order functions. It should not be too difficult to translate the above functions into a relational setting, e.g. in Prolog, after removing the en-
capsulated non-determinism. Additionally, negations need to be handled with care in general, but in our case we used negations only to check for “being not contained in the visited vertices”, which can be inlined and implemented by hand without explicit negation. However, Prolog uses a built-in DFS, which disallows the parametrisation over the search strategy. It is difficult to estimate how declarative and structurally complex the resulting program will be in another language. While we omitted some auxiliary functions, we still consider our code to be rather simple and straightforward.

Acknowledgements: I thank Rudolf Berghammer for encouraging this work, Frank Huch for sparking its idea and the highly appreciated feedback of the reviewers.

References

Curry without Success

Sergio Antoy\textsuperscript{1}  Michael Hanus\textsuperscript{2}

\textsuperscript{1} Computer Science Dept., Portland State University, Oregon, U.S.A.  
antoy@cs.pdx.edu

\textsuperscript{2} Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.  
mh@informatik.uni-kiel.de

Abstract. Curry is a successful, general-purpose, functional logic programming language that predefines a singleton type \textit{Success} explicitly to support its logic component. We take the likely-controversial position that without \textit{Success} Curry would be as much logic or more. We draw a short history and motivation for the existence of this type and justify why its elimination could be advantageous. Furthermore, we propose a new interpretation of rule application which is convenient for programming and increases the similarity between the functional component of Curry and functional programming as in Haskell. We outline some related theoretical (semantics) and practical (implementation) consequences of our proposal.

1 Motivation

Recently, we coded a small Curry \cite{16} module to encode and pretty-print JSON formatted documents \cite{14}. The JSON format encodes floating point numbers with a syntax that makes the decimal point optional. Our Curry System prints floating numbers with a decimal point. Thus, integers, which were converted to floats for encoding, were printed as floats, e.g., the integer value 2 was printed as “2.0”. We found all those point-zeros annoying and distracting and decided to get rid of them. To avoid messing with the internal representation of numbers, and risking losing information, our algorithm would look for “.0” at the end of the string representation of a number in the JSON document and remove it. In the \textit{List} library, we found a function, \textit{isSuffixOf}, that tells us whether to drop the last two characters, but we did not find a function to drop the last 2 characters. How could we do that?

In the library we found the usual \textit{drop} and \textit{take} functions that work at the beginning of a string \textit{s}. Hence, we could reverse \textit{s}, drop 2 characters, and reverse again. We were not thrilled. Or we could take from \textit{s} the first \(n - 2\) characters, where \(n\) is the length of \textit{s}. We were not thrilled either. In both cases, conceptually the string is traversed 3 times (probably in practice too) and extraneous functions are invoked. Not a big deal, but there must be a better way. Although the computation is totally \textit{functional}, we started to think \textit{logic}.

Curry has this fantastic feature called \textit{functional patterns} \cite{4}. With it, we could code the following:

\begin{equation}
\text{fix\_int } (x \mathbin{+ +} ".0") = x
\end{equation}
Now we were thrilled! This is compact, simple and obviously correct. Of course, we would need a rule for cases in which the string representation of a number does not end in “.0”, i.e.:

\[
\begin{align*}
\text{fix\_int} \ (x ++ \ "\cdot0\") &= x \\
\text{fix\_int} \ x &= x
\end{align*}
\]

(2)

Without the last rule \text{fix\_int} would fail on a string such as “2.1”. With the last rule the program would be incorrect because both rules would be applied for a number that ends in “.0”. The latter is a consequence of the design decision that established that the order of the rules in a program is irrelevant—a major departure of Curry from popular functional languages. One of the reasons of this design decision is \textit{Success}.

2 History

Putting it crudely, a functional logic language is a functional language extended with logic variables. The only complication of this extension is what to do when some function \( f \) is applied to some unbound logic variable \( u \). There are two options, either to residuate on \( u \) or to narrow \( u \). Residuation suspends the application of \( f \), and computes elsewhere in the program in hopes that this computation will narrow \( u \) so that the suspended application of \( f \) can continue. Narrowing instantiates \( u \) to values that sustain the computation. For example, given the usual concatenation of lists:

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x : (xs ++ ys)
\end{align*}
\]

(3)

Narrowing \( u ++ t \), where \( u \) is unbound and \( t \) is any expression, instantiates \( u \) to \( [] \) and \( u' : us \) and continues these computations either one at the time or concurrently depending on the control strategy.

In early functional logic languages [1, 17], in the tradition of logic programming, only predicates (as opposed to any function) are allowed to instantiate a logic variable. In the early days of Curry, we were not brave enough. Indiscriminate narrowing, such as that required for (1), which is based on (3), was uncharted territory and we decided that all functions would residuate except a small selected group called \textit{constraints}. These functions are characterized by returning a singleton type called \textit{Success}.

Narrowing has the remarkable property of solving equations [23]. Indeed, the rule in (1) works by solving an equation by narrowing. An application \textit{fix\_int}(s), where \( s \) is a string, attempts to solve \( s = x++\cdot0\). A solution, “the” if any exists, gives the desired result \( x \). Returning \textit{Success} rather than \textit{Boolean}, as the constrained equality does, had the desirable consequence that we would not “solve” an equation by deriving it to \textit{False}, but had the drawback of introducing a new variant of equality, implemented in Curry by the operation “:=”\textit{= “}, and the undesirable consequence that “some expressions were more equal than others” [19].

As a consequence of our hesitation, narrowing was limited to the arguments of constraints—a successful model well-established by Prolog. However, this model is at odds with a language with a functional component with normal (lazy) order of evaluation. Without functional nesting, there is no easy way to tell whether or not some argument of some constraint should be evaluated. Consider a program to solve the 8-queens puzzle:
permute \ x \ y = \ldots \text{ succeed if } y \text{ is a permutation of } x
\text{safe } y = \ldots \text{ succeed if } y \text{ is a safe placement of queens} \tag{4}

A solution of the puzzle is obtained by \text{permute} [1..8] \ y & \& \text{safe } y, \text{ where } y \text{ is likely a free variable. The constraint } \text{permute} \text{ fully evaluates } y \text{ upon returning even if } \text{safe} \text{ may only look at the first two elements and determine that } y \text{ is not safe.}

This prompted the invention of “non-deterministic functions”, i.e., a function-like mechanism, that may return more than one value for the same combination of arguments, but is used as an ordinary function. With this idea, Example (4) is coded as:

\text{permute } x = \ldots \text{ return any permutation of } x
\text{safe } y = \ldots \text{ as before} \tag{5}

In this case, a solution of the puzzle is obtained by \text{safe } y \text{ where } y = \text{permute} [1..8]. Since \( y \) is nested inside \text{safe}, \text{permute} [1..8] \text{ can be evaluated only to the extent needed by its context. A plausible encoding of } \text{permute} \text{ is:}

\begin{align*}
\text{permute } [] &= [] \\
\text{permute } (x : xs) &= \text{nd_insert} x (\text{permute} \ xs) \\
\text{nd_insert} x \ ys &= x : ys \\
\text{nd_insert} x (y : ys) &= y : \text{nd_insert} x \ ys \tag{6}
\end{align*}

The evaluation of \text{permute} [1..8] \text{ produces any permutation of [1..8] only if } \text{both rules defining } \text{nd_insert} \text{ are applied when the second argument is a non-empty list. Thus, the well-established convention of functional languages that the first rule that matches an expression is the only one being fired had to be changed in the design of Curry.}

### 3 Proposed Adjustments

Our modest proposal is to strip the \textit{Success} type of any special meaning. Since \textit{Success} is isomorphic to the \textit{Unit} type, which is already defined in the \textit{Prelude}, probably it becomes redundant. Future versions of the language could keep it for backward compatibility, but deprecate it.

The first consequence of this change puts in question the usefulness of “\(=:=\)”, the constrained equality. Equations can be solved using the Boolean equality “\(==\)” bringing Curry more in line with functional languages. To solve an equation by narrowing, we simply evaluate it using the standard rules defining Boolean equality. For example, below we show these rules for a polymorphic type \textit{List}:

\begin{align*}
[] &= [ ] = \text{True} \\
(x:xs) &= (y:ys) = x==y & xs==ys \\
[] &= (_,_) = \text{False} \\
(_,_) &= [ ] = \text{False} \tag{7}
\end{align*}

However, we certainly want to avoid binding variables with instantiations that derive an equation to \textit{False} since these bindings are not solutions. Avoiding these bindings is achieved with the following operation:

\text{solve} \text{ True} = \text{True} \tag{8}

and wrapping an equation with \text{solve}, i.e., to solve \( x = y \), we code \text{solve} (x == y). Nostalgic programmers could redefine “\(=:=\)” as:
When an equation occurs in the condition of a rule, the intended behavior is implied, i.e., the rule is fired only when the condition is (evaluates to) True.

In a short paragraph above, we find the symbols “=” “==” and “:=:”. The first one is the (mathematical) equality. The other two are (computational) approximations of it with subtle differences. Our proposal simplifies this situation by having only “==” as the implementation of “=”, as in functional languages, without sacrificing any of Curry’s logic aspects.

The second consequence of our proposal is to review the rule selection strategy, i.e., the order, or more precisely its lack thereof, in which rules are fired. We have already hinted at this issue discussing example (2). Every rule that matches the arguments and satisfies the condition of a call is non-deterministically fired. A motivation for the independence of rule order was discussed in example (6). The ability of making non-deterministic choices is essential to functional logic programming, and it must be preserved, but it can be achieved in a different way.

The predefined operation “?” non-deterministically returns either of its arguments. This operation allows us to express non-determinism in a way different from rules with overlapping left-hand sides [3]. For instance, the non-determinism of the operation \texttt{nd_insert} in (6) can be moved from the left-hand sides of its defining rules to the right-hand sides as in the following definition:

\begin{equation}
\begin{align*}
\texttt{nd_insert} \ x \ ys &= (x : ys) \ ? \ nd\_insert2 \ x \ ys \\
\texttt{nd_insert2} \ x \ (y : ys) &= y : \texttt{nd_insert} \ x \ ys
\end{align*}
\end{equation}

Indeed, some Curry compilers, like KiCS2 [8], implement this transformation.

The definition of Curry at the time of this writing [16] establishes that the order of the rules defining an operation is irrelevant. The same holds true for the conditions of a rule, except in the case in which the condition type is Boolean, and for flexible case expressions. Our next proposal is to change this design decision of Curry. Although this is somehow independent of our first proposal to remove the \texttt{Success} type, it is reasonable to consider both proposals at once since both simplify the use of Curry.

We propose to change the current definition of rule application in Curry as follows. To determine which rule(s) to fire for an application $t = f(t_1, \ldots, t_n)$, where $f$ is an operation and $t_1, \ldots, t_n$ are expressions, use the following strategy:

1. Scan the rules of $f$ in textual order. An unconditional rule is considered as a conditional rule with condition \texttt{True}.
2. Fire the first rule whose left-hand side matches the application $t$ and whose condition is satisfied. Ignore any remaining rule.
3. If no rule can be applied, the computation fails.
4. If a combination of arguments is non-deterministic, the previous points are executed independently for each non-deterministic choice of the combination of arguments.

In particular, if an argument is a free variable, it is non-deterministically instantiated to all its possible values.

As usual in a non-strict language like Curry, arguments of an operation application are evaluated as they are demanded by the operation’s pattern matching and condition.
However, any non-determinism or failure during argument evaluation is not passed inside the condition evaluation. A precise definition of “inside” is in [6, Def. 3]. This is quite similar to the behavior of set functions to encapsulate internal non-determinism [6]. Apropos, we discuss in Section 5 how to exploit set functions to implement this concept.

Before discussing the advantages and implementation of this concept, we explain and motivate the various design decisions taken in our proposal. First, it should be noted that this concept distinguishes non-determinism outside and inside a rule application. If the condition of a rule has several solutions, this rule is applied if it is the first one with a true condition. Second, the computation proceeds non-deterministically with all the solutions of the condition. For instance, consider an operation to look up values for keys in an association list:

\[
\text{lookup } \text{key assoc} \\
\mid \text{assoc} == (\_ ++ [(\text{key, val})] ++ \_) \\
= \text{Just val} \\
\text{where val free} \\
\text{lookup } \_ \_ = \text{Nothing} 
\]

(11)

If we evaluate \(\text{lookup } 2 \ [(2, 14), (3, 17), (2, 18)]\), the condition of the first rule is solvable. Thus, we ignore the remaining rules and apply only the first rule to evaluate this expression. Since the condition has the two solutions \(\{\text{val} \mapsto 14\}\) and \(\{\text{val} \mapsto 18\}\), we yield the values \(\text{Just } 14\) and \(\text{Just } 18\) for this expression. Note that this is in contrast to Prolog’s if-then-else construct which checks the condition only once and proceeds just with the first solution of the condition. If we evaluate \(\text{lookup } 2 \ [(3, 17)]\), the condition of the first rule is not solvable but the second rule is applicable so that we obtain the result \(\text{Nothing}\).

On the other hand, non-deterministic arguments might trigger different rules to be applied. Consider the expression \(\text{lookup } (2?3) \ [(3, 17)]\). Since the non-determinism in the arguments leads to independent rule applications (see item 4), this expression leads to independent evaluations of \(\text{lookup } 2 \ [(3, 17)]\) and \(\text{lookup } 3 \ [(3, 17)]\). The first one yields \(\text{Nothing}\), whereas the second one yields \(\text{Just } 17\).

Similarly, free variables as arguments might lead to independent results since free variables are equivalent to non-deterministic values [5]. For instance, the expression \(\text{lookup } 2 \ \text{xs}\) yields the value \(\text{Just } v\) with the binding \(\{\text{xs} \mapsto (2, v):\_\}\), but also the value \(\text{Nothing}\) with the binding \(\{\text{xs} \mapsto [\] \}\) (as well as many other solutions). Again, this behavior is different from Prolog’s if-then-else construct which performs bindings for free variables inside the condition independently of its source. In contrast to Prolog, our design supports completeness in logic-oriented computations even in the presence of if-then-else.

The latter desirable property has also implications for the handling of failures occurring when arguments are evaluated. For instance, consider the expression “\(\text{lookup } 2 \ \text{failed}\)” (where \text{failed} is a predefined operation which always fails whenever it is evaluated). Because the evaluation of the condition of the first rule fails, the entire expression evaluation fails instead of returning the value \(\text{Nothing}\). This is motivated by the fact that we need the value of the association list in order to check the satisfiability of the condition, but this value is not available.
To see the consequences of an alternative design decision, consider the following contrived definition of an operation that checks whether its argument is the unit value () (which is the only value of the unit type):

\[
\text{isUnit } x \mid x == () = \text{True} \\
\text{isUnit } x = \text{False}
\]  

(12)

In our proposal, the evaluation of \text{isUnit failed} fails. In an alternative design (like Prolog’s if-then-else construct), one might skip any failure during condition checking and proceed with the next rule. In this case, we would return the value \text{False} for the expression \text{isUnit failed}. This is quite disturbing since the (deterministic!) operation \text{isUnit}, which has only one possible input value, could return two values: \text{True} for the call \text{isUnit } () and \text{False} for the call \text{isUnit failed}. Moreover, if we call this operation with a free variable, like \text{isUnit } x, we obtain the single binding \{x \mapsto ()\} and value \text{True} (since free variables are never bound to failures). Thus, either our semantics would be incomplete for logic computations or we compute too many values. In order to get a consistent behavior, we require that failures of arguments demanded for condition checking lead to failures of evaluations.

Changing the meaning of rule selection from an order-independent semantics to a sequential interpretation is an important change in the design of Curry. However, this change is relevant only for a relatively small amount of existing programs. First, most of the operations in a functional logic program are inductively sequential [2], i.e., they are defined by rules where the left-hand sides do not overlap. Hence, the order of the rules does not affect the definition of such operations. Second, rules defined with traditional Boolean guards residuate if they are applied to unknown arguments, i.e., it is usually not intended to apply alternative conditions to a given call. This fits to a sequential interpretation of conditions. Moreover, our proposal supports the use of conditional rules in a logic programming manner with unknown arguments, since this “outside” non-determinism does not influence the sequential condition checking.

Nevertheless, there are also cases where a sequential interpretation of rules is not intended, e.g., in a rule-oriented programming style, which is often used in knowledge-based or constraint programming. Although we argued that one can always translate overlapping patterns into rules with non-overlapping patterns by using the choice operator “?”, the resulting code might be less readable. Finally, we have to admit that in a declarative language ignoring the order of the rules is more elegant though not always as convenient. Hence, a good compromise would be a compiler pragma that allows to choose between a sequential or an unordered interpretation of overlapping rules.

4 Advantages

In this section we justify through exemplary problems the advantages of the proposed changes.

**Example 1.** With the proposed semantics, (2) is a simple and obviously correct solution of the problem, discussed in the introduction, of “fixing” the representation of integers in a JSON document.

**Example 2.** As in the previous example, our proposed semantics is compatible with functional patterns. Hence, (11) can be more conveniently coded as:
Example 3. Consider a read-eval-print loop of a functional logic language such as Curry. A top-level expression may contain free variables that are declared by a free clause such as in the following example:

\[
\text{x ++ y == [1,2,3,4] where x, y free}
\]

(14)

Of course, the free clause is absent if there are no free variables in the top-level expression. The free variables, when present, are easily extracted with a “deep” pattern as follows:

\[
\text{breakFree (exp++" where "++wf++" free")}
\]

\[
= (\text{exp}, \text{wf})
\]

\[
\text{breakFree exp}
\]

\[
= (\text{exp}, "")
\]

(15)

For this code to work, the rules of breakFree must be tried in order and the second one must be fired only if the first one fails.

Example 4. Suppose that World Cup soccer scores are represented in either of the following forms:

\[
\text{GER -:- USA}
\]

\[
\text{GER 1:0 USA}
\]

(16)

where the first line represents a game not yet played and the second one a game in which the digits are the goals scored by the adjacent team (a single digit suffices in practice). The following operation parses scores:

\[
\text{parse (team1++" -:- "++team2) = (team1,team2,Nothing)}
\]

\[
\text{parse (team1++[\ ' ',x,\ ' :\ ',y,\ ' ']++team2)}
\]

\[
| \text{isdigit x && isdigit y}
\]

\[
= (\text{team1,team2, Just (toInt x,toInt y)})
\]

\[
\text{parse _ = error "Wrong format!"}
\]

(17)

Example 5. The Dutch National Flag problem [13] has been proposed in a simple form to discuss the termination of rewriting [12]. A formulation in Curry of this simple form is equally simple:

\[
\text{dnf (x++[White,Red]++y) = dnf (x++[Red,White]++y)}
\]

\[
\text{dnf (x++[Blue,Red]++y) = dnf (x++[Red,Blue]++y)}
\]

\[
\text{dnf (x++[Blue,White]++y) = dnf (x++[White,Blue]++y)}
\]

(18)

However, (18) needs a termination condition to avoid failure. With our proposed semantics, this condition is simply:

\[
\text{dnf x = x}
\]

(19)

With the standard semantics, a much more complicated condition is needed.

5 Implementation

A good implementation of the proposed changes in the semantics of rule selection requires new compilation schemes for Curry. However, an implementation can also be
obtained by a transformation over source programs when existing advanced features of Curry are exploited. This approach provides a reference semantics that avoids explicitly specifying all the details of our proposal, in particular, the subtle interplay between condition solving and non-determinism and failures in arguments. Hence, we define in this section a program transformation that implements our proposed changes within existing Curry systems.

Initially, we discuss the implementation of a single rule with a sequence of conditions, i.e., a program rule of the form

\[
\begin{align*}
  l & \mid c_1 = e_1 \\
  \vdots \\
  & \mid c_k = e_k 
\end{align*}
\]  

(20)

According to our proposal, if the left-hand side \( l \) matches a call, the conditions \( c_1, \ldots, c_k \) are sequentially evaluated. If \( c_i \) is the first condition that evaluates to \( \text{True} \), all other conditions are ignored so that (20) becomes equivalent to

\[
\begin{align*}
  l & \mid c_i = e_i 
\end{align*}
\]

Note that the subsequent conditions are ignored even if the condition \( c_i \) also evaluates to \( \text{False} \). Thus, the standard translation of rules with multiple guards, as defined in the current report of Curry \cite{curry}, i.e., replacing multiple guards by nested if-then-else constructs, would yield a non-intended semantics. Moreover, non-determinism and failures in the evaluation of actual arguments must be distinguished from similar outcomes caused by the evaluation of the condition, as discussed in Section 3.

All these requirements call for the encapsulation of condition checking where “inside” and “outside” non-determinism are distinguished and handled differently. Fortunately, recent developments for encapsulated search in functional logic programming \cite{encapsulated,logic} provide an appropriate solution of this problem. For instance, \cite{logic} proposes an encapsulation primitive \texttt{allValues} so that the expression \( \texttt{(allValues e)} \) evaluates to the set of values of \( e \) where only internal non-determinism inside \( e \) is considered. Thus, we can use the following expression to check a condition \( c \) with our intended meaning:\footnote{\cite{logic} defines only an operation \texttt{isEmpty}. Hence we assume that \texttt{notEmpty} is defined by the rule \texttt{notEmpty x = not (isEmpty x)}.}

\[
\begin{align*}
  \text{if notEmpty (allValues (solve c)) then } e_1 \text{ else } e_2 
\end{align*}
\]  

(21)

According to \cite{logic}, the meaning of this expression is as follows:

1. Test whether there is some evaluation of \( c \) to \( \text{True} \).
2. If the test is positive, evaluate \( e_1 \).
3. If there is no evaluation of \( c \) to \( \text{True} \), evaluate \( e_2 \).

The semantics of \texttt{allValues} ensures that non-determinism and failures caused by expressions not defined inside \( c \), in particular, parameters of the left-hand side \( l \) of the operation, are not encapsulated. The Curry implementations PAKCS \cite{pakcs} and KiCS2 \cite{kics} provide set functions \cite{encapsulated} instead of \texttt{allValues} which allows the implementation of this conditional in a similar way.
Our expected semantics demands that a rule with a solvable condition be applied for each true condition, in particular, with a possible different binding computed by evaluating the condition. To implement this behavior, we assume an auxiliary operation \textit{ifTrue} that combines a condition and an expression. This operation is simply defined by

\[
\text{ifTrue True } x = x
\]  
(22)

Then we define the meaning of (20) by the following transformation:

\[
l = \text{if notEmpty (allValues (solve } c_1)\text{)} \\
\quad \text{then } (\text{ifTrue } c_1 e_1) \text{ else} \\
\quad \vdots \\
\quad \text{if notEmpty (allValues (solve } c_k)\text{)} \\
\quad \text{then } (\text{ifTrue } c_k e_k) \text{ else failed}
\]  
(23)

There are obvious simplifications of this general scheme. For instance, if \(c_k = \text{True}\), as frequently is the case, the last line of (23) becomes \(e_k\).

This transformation scheme is mainly intended as the semantics of sequential condition checking rather than as the final implementation (similarly to the specification of the meaning of guards in Haskell [20]). A sophisticated implementation could improve the actual code. For instance, each condition \(c_i\) is duplicated in our scheme. Moreover, it seems that conditions are always evaluated twice. However, this is not the case if a lazy implementation of encapsulated search via \textit{allValues} or set functions is used, as in the Curry implementation KiCS2 [10]. If \(c_i\) is the first solvable condition, the emptiness test for \(\text{allValues } c_i\) can be decided after computing a first solution. In this case, this solution is computed again (and now also all other solutions) in the \textit{then}-part in order to pass its computed bindings to \(e_i\). Of course, a more primitive implementation might avoid this duplicated evaluation.

Next we consider the transformation of a sequence of rules

\[
l_1 \quad r_1 \\
\vdots \\
l_k \quad r_k
\]  
(24)

where each left-hand side \(l_i\) is a pattern \(f \ p_{i1} \ldots p_{in_i}\) for the same function \(f\) and each \(r_i\) is a sequence of condition/expression pairs of the form “\(c = e\)” as shown in (20).\(^4\)

We assume that the pattern arguments \(p_{ij}\) contain only constructors and variables. In particular, functional patterns have been eliminated by moving them into the condition using the function pattern unification operator “\(= :<=\)” (as shown in [4]). For instance, rule (1) is transformed into

\[
\text{fix_int xs } | \ (x ++ \".0\") =:\= xs = x
\]  
(25)

Finally, we assume that subsequent rules with the same pattern (up to variable renaming) are joined into a single rule with multiple guards. For instance, the rules (2) can be joined (after eliminating the functional pattern) into the single rule

\(^4\) In order to handle all rules in a unique manner, we consider an unconditional rule “\(l_i = e_i\)” as an abbreviation for the conditional rule “\(l_i | \ \text{True } = e_i\)”.

48
\[
\text{fix} \_\text{int} \ xe \hspace{1cm} | \ (x \ ++ \ "\ .0\") =:<= \ xe = x \\
| \ True = xe
\]

(26)

Now we distinguish the following cases:

- The patterns in the left-hand sides \( l_1, \ldots, l_k \) are inductively sequential [2], i.e., the patterns can be organized in a tree structure such that there is always a discriminating (inductive) argument: since there are no overlapping left-hand sides in this case, the order of the rules is not important for the computed results. Therefore, no further transformation is necessary in this case. Note that most functions in typical functional logic programs are defined by inductively sequential rules.

- Otherwise, there might be overlapping left-hand sides so that it is necessary to check all rules in a sequential manner. For this purpose, we put the pattern matching into the condition so that the patterns and conditions are checked together. Thus, a rule like

\[
f \ p_1 \ldots p_n \ | \ c = e
\]

is transformed into

\[
f \ x_1 \ldots x_n \ | \ (\ p_1 \ldots p_n \ \rightarrow \ c) \ x_1 \ldots x_n \\
= (\ p_1 \ldots p_n \ \rightarrow \ \text{ifTrue} \ c \ e) \ x_1 \ldots x_n
\]

where \( x_1, \ldots, x_n \) are fresh variables (the extension to rules with multiple conditions is straightforward). Using this transformation, we obtain a list of rules with identical left-hand sides which can be joined into a single rule with multiple guards, as described above.

For instance, the definition of \text{fix} \_\text{int} (26) is transformed into

\[
\text{fix} \_\text{int} \ xe = \\
\quad \text{if notEmpty} \ (\text{allValues} \ (\text{solve} \ (x++\".0\" =:<= xe))) \\
\quad \text{then} \ (\text{ifTrue} \ (x++\".0\" =:<= xe) \ xe) \\
\quad \text{else} \ xe
\]

(27)

For an example of transforming rules with overlapping patterns, consider an operation that reverses a two-element list and leaves all other lists unchanged:

\[
\text{rev} \_2 \ [x,y] = [y,x] \\
\text{rev} \_2 \ xe = xe
\]

(28)

According to our transformation, this definition is mapped into (after some straightforward simplifications):

\[
\text{rev} \_2 \ xe = \\
\quad \text{if notEmpty} \ (\text{allValues} \ (\text{\{[x,y] \rightarrow True\}} \ xe) \\
\quad \text{then} \ (\text{\{[x,y] \rightarrow [y,x]\}} \ xe \\
\quad \text{else} \ xe
\]

(29)

Thanks to the logic features of Curry, one can also use this definition to generate appropriate argument values for \text{rev} \_2. For instance, if we evaluate the expression \text{rev} \_2 \ xe
(where \( xs \) is a free variable), the Curry implementation KiCS2 [8] has a finite search space and computes the following bindings and values:

\[
\begin{align*}
\{xs = []\} & \{\} \\
\{xs = [x1]\} & \{x1\} \\
\{xs = [x1,x2]\} & \{x2,x1\} \\
\{xs = (x1:x2:x3:x4)\} & (x1:x2:x3:x4)
\end{align*}
\]

As mentioned above, the transformation presented in this section is intended to serve as a reference semantics for our proposed changes and to provide a prototypical implementation. There are various possibilities to improve this implementation. For instance, if the right-hand side expressions following each condition are always evaluable to a value, i.e., to a finite expression without defined operations, the duplication of the code of the condition as well as the potential double evaluation of the first solvable condition can be easily avoided. As an example, consider the following operation that checks whether a string contains a non-negative float number (without an exponent):

\[
isNNFloat (f1 ++ "." ++ f2) \\
| all isDigit f1 && all isDigit f2 = True
\]

(30)

If \( c \) denotes the condition

\[
(f1 ++ "." ++ f2) =:<= s && all isDigit f1 && all isDigit f2
\]

(31)

by functional pattern elimination [4], program (30) is equivalent to

\[
isNNFloat s | c = True \\
isNNFloat _ = False
\]

(32)

Applying our transformation, we obtain the following code with the duplicated condition \( c \):

\[
isNNFloat s = \\
\quad \text{if notEmpty (allValues (solve c))} \\
\quad \quad \text{then (ifTrue c True)} \\
\quad \quad \text{else False}
\]

(33)

Since the expressions on the right-hand side are always values (\( True \) or \( False \)), we can put these expressions into the sets computed by \( allValues \). Then the check for a solvable condition becomes equivalent to check the non-emptiness of these value sets so that we return non-deterministically some value of this set.\(^5\) This idea can be implemented by the following scheme which does not duplicate the condition and evaluates it only once (the actual code can be simplified but we want to show the general scheme):

\[
isNNFloat s = \\
\quad \text{if notEmpty s1 then chooseValue s1 else False}
\]

(34)

where

\[
s1 = allValues (ifTrue c True)
\]

Note that this optimization is not applicable if it is not ensured that the right-hand side expressions are always evaluable to values. For instance, consider definition (28) of

\(^5\) The predefined operation \( chooseValue \) non-deterministically returns some value of a set.
rev2 and the expression \( \text{head} \left( \text{rev2} \left[ \text{nv}, 0 \right] \right) \), where \( \text{nv} \) is an expression without a value (e.g., failure or non-termination). With our current transformation (29), we compute the value 0 for this expression. However, the computation of the set of all values of \( \left( \text{rev2} \left[ \text{nv}, 0 \right] \right) \) w.r.t. the first rule defining rev2 does not yield any set since the right-hand side \([0, \text{nv}]\) has no value. This explains our transformation scheme (23) which might look complicated at a first glance.

However, there is another transformation to implement overlapping rules like (28) with our intended semantics. If the rules are unconditional, one can “complete” the missing constructor patterns in order to obtain an inductively sequential definition. For the operation rev2, we obtain the following definition:

\[
\begin{align*}
\text{rev2} \ [x, y] &= [y, x] \\
\text{rev2} \ [ ] &= [] \\
\text{rev2} \ [x] &= [x] \\
\text{rev2} \ (x:y:z:xs) &= x:y:z:xs
\end{align*}
\]  

(35)

Since a case can be more efficiently executed than an encapsulated computation, this alternative transformation might lead to larger but more efficient target code.

6 Related Work

Declarative programming languages support the construction of readable and reliable programs by partitioning complex procedures into smaller units—mainly using case distinction by pattern matching and conditional rules. Since we propose a new interpretation of case distinctions for functional logic programs, we compare our proposal with existing ones with similar objectives.

The functional programming language Haskell [20] provides, similarly to Curry, also pattern matching and guarded rules for case distinctions. Our proposal for a new sequential interpretation of patterns increases the similarities between Curry and Haskell. Although Curry provides more features due to the built-in support to deal with non-deterministic and failing computations, our proposal is a conservative extension of Haskell’s guarded rules, i.e., it has the same behavior as Haskell when non-determinism and failures do not occur. To see this, consider a program rule with multiple conditions:

\[
\begin{align*}
\mathit{l} &\mid c_1 = e_1 \\
\vdots &\mid c_k = e_k
\end{align*}
\]  

(36)

Since non-deterministic computations do not exist in Haskell and failures lead to exceptions \( e \), we assume that, if this rule is applied in Haskell to an expression \( e \), there is one condition \( c_i \) which evaluates to \( \text{True} \) and all previous conditions \( c_1, \ldots, c_{i-1} \) evaluate to \( \text{False} \). If we consider the same rule translated with the transformation scheme (23), obviously each condition \( \text{notEmpty} \left( \text{allValues} \left( \text{solve} \left( c_j \right) \right) \right) \) reduces to \( \text{False} \) for \( j = 1, \ldots, i - 1 \) and to \( \text{True} \) for \( j = i \). Thus, the application of this rule reduces \( e \) to \( (\text{ifTrue} \ c_i \ e_i) \) and, subsequently, to \( e_i \), as in Haskell.

The logic programming language Prolog [11] also supports pattern matching and, for sequential conditions, an if-then-else construct of the form “\( c \to e_1 ; e_2 \)”. Although Prolog can deal, similarly to Curry, with non-deterministic and failing compu-
tions, the if-then-else construct usually restricts the completeness of the search space due to cutting the choice points created by \( c \) before executing \( e_1 \). Hence, only the first solution of \( c \) is used to evaluate \( e_1 \). Furthermore, inside and outside non-determinism is not distinguished so that variables outside the condition \( c \) might be bound during its evaluation. This has the effect that predicates where if-then-else is used are often restricted to a particular mode. For instance, consider the re-definition of \( \text{rev2} \) (28) as a predicate in Prolog using if-then-else:

\[
\text{rev2}(Xs,Ys) :- Xs=[X,Y] \rightarrow Ys=[Y,X] ; Ys=Xs. \tag{37}
\]

If we try to solve the goal \( \text{rev2}(Xs,Ys) \), Prolog yields the single answer \( Xs = [A,B], Ys = [B,A] \). Thus, in contrast to our approach, all other answers are lost.

Various encapsulation operators have been proposed for functional logic programs [7] to encapsulate non-deterministic computations in some data structure. Set functions [6] have been proposed as a strategy-independent notion of encapsulating non-determinism to deal with the interactions of laziness and encapsulation (see [7] for details). We can also use set functions to distinguish successful and non-successful computations, similarly to negation-as-failure in logic programming, exploiting the possibility to check result sets for emptiness. When encapsulated computations are nested and performed lazily, it turns out that one has to track the encapsulation level in order to obtain intended results, as discussed in [10]. Thus, it is not surprising that set functions and related operators fit quite well to our proposal.

Computations with failures for the implementation of an if-then-else construct and default rules in functional logic programs have been also explored in [18, 22]. In these works, an operator, \texttt{fails}, is introduced to check whether every reduction of an expression to a head-normal form is not successful. The authors show that this operator can be used to define a single default rule, but not the more general sequential rule checking of our approach. Moreover, nested computations with failures are not considered by these works. As a consequence, the operator \texttt{fails} might yield unintended results if it is used in nested expressions. For instance, if we use \texttt{fails} instead of \texttt{allValues} to implement the operation \( \text{isUnit} \) defined in (12), the evaluation of \( \text{isUnit failed} \) yields the value \( \text{False} \) in contrast to our intended semantics.

### 7 Conclusions

We proposed two changes to the current design of Curry. The first one concerns the removal of the type \texttt{Success} and the related constraint equality “\( \_ =:= \)” . This simplifies the language since it relieves the programmer from choosing the appropriate equality operator. The second one concerns a strict order in which rules and conditions are tried to reduce an expression. This makes the language design more similar to functional languages like Haskell so that functional programmers will be more comfortable with Curry. Nevertheless, the logic programming features, like non-determinism and evaluating functions with unknown arguments, are still applicable with our new semantics. This distinguishes our approach from similar concepts in logic programming which simply cuts alternatives.

However, our proposal comes also with some drawbacks. We already mentioned that in knowledge-based or constraint programming applications, a sequential ordering
of rules is not intended. Hence, a compiler pragma could allow the programmer to choose between a sequential or an unordered interpretation of overlapping rules.

A further drawback of our approach concerns the run-time efficiency. We argued that solving "==" equations by narrowing with standard equational rules can replace the constraint equality ":=". Although this is true from a semantic point of view, the constraint equality operator ":=" is more efficient from an operational point of view. If \( x \) and \( y \) are free variables, the equational constraint \( x:y \) is deterministically solved by binding \( x \) to \( y \) (or vice versa), whereas the Boolean equality \( x==y \) is solved by non-deterministically instantiating \( x \) and \( y \) to identical values. The efficiency improvement of performing bindings is well known, e.g., it is benchmarked in [9] for the Curry implementation KiCS2. On the other hand, the Boolean equality \( x==y \) is more powerful since it can also solve negated conditions, i.e., evaluate \( x==y \) to \text{False} by binding \( x \) and \( y \) to different values.

Hence, for future work it is interesting to find a compromise, e.g., performing variable bindings when \( x==y \) should be reduced to \text{True} without any surrounding negations. A program analysis could be useful to detect such situations at compile time.

Finally, the concurrency features of Curry must be revised. Currently, concurrency is introduced by the concurrent conjunction operator \( & \) on constraints. If the constraint type \text{Success} is removed, other forms of concurrent evaluations might be introduced, e.g., in operators with more than one demanded argument ("="", "+",\ldots\), explicit concurrent Boolean conjunctions, or only in the I/O monad similarly to Concurrent Haskell [21].

Despite all the drawbacks, our proposal is a reasonable approach to simplify the design of Curry and make it more convenient for the programmer.

8 Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249.

References


Abstract. We present a partial evaluator for functional logic programs written in Curry. In contrast to previous approaches to the partial evaluation of functional logic programs, we take into account the features used in contemporary Curry programs, in particular, non-deterministic operations and recursive let expressions. For this purpose, we base our partial evaluator on FlatCurry, an intermediate language for the representation of Curry programs. We sketch our approach and present initial benchmarks of our implementation.

1 Introduction

Partial evaluation of programs is a technique to anticipate the evaluation of computations once at compile time instead of performing them (possibly several times) at run time. This is possible if some part of the input data, also called static data, is known at compile time. In this case, some parts of the program are evaluated so that a residual program, i.e., a specialized version of the original one, is returned. Since some computations have been performed at compile time, the run time of the specialized program could be considerably decreased. The static data does not need to be some user input, but can also be subexpressions in the original program. Offline partial evaluators obtain information about static data from a separate static analysis phase (binding-time analysis), whereas online partial evaluators obtain this information on the fly and propagate it during the partial evaluation process. In this work we follow the online partial evaluation approach.

Partial evaluation has already been studied for different kinds of programming languages, like functional languages, logic languages, as well as for combined functional logic languages. An interesting aspect of the partial evaluation of functional logic programs is the fact that the effects of supercompilation [23] can be obtained by applying the operational semantics of the source language (narrowing) at partial evaluation time [5]: if a function is called with unknown arguments, narrowing instantiates these arguments such that the rules defining this function can be applied. Hence, one mainly needs to control the partial evaluator, i.e., avoiding infinite unfoldings and instantiations of logic variables, in order to obtain residual programs.

Thanks to this insight, partial evaluators for functional logic languages can be constructed with techniques similarly to the implementation of these languages. For instance, Albert et al. [3] proposed a partial evaluator for Curry [17] based on the intermediate language FlatCurry. Since FlatCurry makes the evaluation strategy of Curry programs explicit [1], the use of FlatCurry led to a partial evaluator able to optimize practical Curry programs. Unfortunately, when this partial evaluator was constructed,
the use of non-deterministic operations, although proposed some years ago [14], was not well established. Therefore, the partial evaluation scheme was based on term rewriting and restricted to confluent programs, i.e., all operations were required to be deterministic, and recursive let expressions were also not taken into account. Thus, if this partial evaluator is applied to programs containing non-deterministic operations, which is a useful programming pattern in contemporary functional logic programs [6,8], the resulting programs are not semantically equivalent to the source programs.

In order to deal with realistic Curry programs, it is crucial for a partial evaluator to cover the full source language, including both logic features such as non-determinism and functional features such as recursive let expressions. Therefore, we extend in this work the partial evaluator of Albert et al. [3] to cover the full language of FlatCurry. In contrast to [3], we base our partial evaluator on an operational semantics [1] which is adequate for contemporary Curry programs.

We start with an introduction to the functional logic language Curry in Sect. 2 before we sketch the structure of the partial evaluator in Sect. 3. The partial evaluation scheme is presented in Sect. 4, whereas control issues are discussed in Sect. 5. We evaluate our implementation with some benchmarks in Sect. 6 before we conclude in Sect. 7.

2 Curry

We briefly review the basic concepts of the functional logic language Curry. More details can be found in recent surveys on functional logic programming [7,15] and in the language report [17].

The syntax of Curry [17] is close to Haskell [21], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of an operation \( f \) to an expression \( e \) is denoted by juxtaposition ("\( f \ e \)"). In addition to Haskell, Curry allows free (logic) variables in rules and initial expressions. If the data type of Booleans and a negation operation are defined by

\[
\text{data } \text{Bool} = \text{False} \mid \text{True}
\]

\[
\text{not } \text{True} = \text{False}
\]

\[
\text{not } \text{False} = \text{True}
\]

the expression "not \( x \) where \( x \) free" non-deterministically reduces to False with the binding \( x = \text{True} \), and to True with the binding \( x = \text{False} \). A further kind of non-determinism is supported in Curry by the choice operator "\( ? \)", which can be considered as predefined by the overlapping rules

\[
x \ ? \ _ = x \\
_ \ ? \ y = y
\]

Thus, we can define a non-deterministic operation \( \text{coin} \) yielding the values 0 and 1 by

\[
\text{coin} = 0 \ ? \ 1
\]

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operation

\[
\text{double } x = x + x
\]

and the expression “\( \text{double } \text{coin} \)”. If we evaluated this expression by term rewriting, we could have the reduction
leading to the unintended result 1. Note that this result cannot be obtained with a strict reduction strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, Curry is based on the rewriting logic CRWL, proposed by González-Moreno et al. [14] as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the call-time choice semantics [18] where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of \texttt{coin} are shared so that all of them consistently reduce to either 0 or 1.

3 Overview of the Partial Evaluator

Before describing the details of the partial evaluation process, we provide an overview of the partial evaluator and its usage. Since our partial evaluator is an extension of the first partial evaluator for Curry described in [3], our representation is oriented towards the original description.

Our partial evaluator is intended to specialize some parts of a given input program in order to create an optimized, residual program. In order to support the specification of expressions to be optimized, we assume that these expressions are annotated with \texttt{PEVAL}. For example, we assume a program which contains the function definition

\begin{verbatim}
main xs = map (twice square) xs
\end{verbatim}

We can then annotate the main expression (or parts of it) as follows:

\begin{verbatim}
main xs = PEVAL (map (twice square) xs)
\end{verbatim}

Actually, \texttt{PEVAL} is the identity function, i.e., it has the type \texttt{a \rightarrow a}. As a consequence, annotations with \texttt{PEVAL} do not change the semantics of the original program. After annotating the program, the process of partial evaluation is fully automatic. The process itself consists of the following phases (depicted in Fig. 1):

1. The partial evaluator is called for a given program, containing annotated expressions as described above. This source program is converted into the standard intermediate representation for Curry programs, called FlatCurry (see Sect. 4.1).
2. The process continues by extracting the set of annotated expressions and creating a copy of the original program without annotations.
3. Both form the input for the partial evaluation phase, which is later described.
4. The output of the partial evaluation is a set of semantically equivalent, potentially more efficient expressions. These expressions are converted to new function definitions to allow reuse, a process called renaming.
5. The evaluation process tends to produce some “intermediate” functions which only pass their parameters to another function. Therefore, the process is finished by a compression phase which removes such intermediate functions by inlining and simplifies expressions to produce a more efficient and legible result.
6. Finally, the annotated expressions of the form (\texttt{PEVAL e}) are replaced with their (hopefully more efficient) equivalents \texttt{e'}, where \texttt{e'} is the renaming of \texttt{e}. This optimized program is then stored as a FlatCurry program.
For instance, with the usual definitions of `map`, `twice`, and `square`, the example above is transformed into

\[
\begin{align*}
\text{main } x_0 &= \text{map0 } x_0 \\
\text{map0 } x_0 &= \text{case } x_0 \text{ of } \[
\rightarrow \[
\text{let } z = (y \cdot y) \text{ in } (z \cdot z) : \text{map0 } y_0
\end{align*}
\]

so that the overhead of the higher-order operations `map` and `twice` is eliminated.

The fact that the partial evaluator internally operates on the FlatCurry format is no restriction, since this format is used by current Curry compilers anyway, e.g., PAKCS [16] or KiCS2 [11]. Hence, the partial evaluator can easily be incorporated into a compilation chain.

## 4 The Partial Evaluation Scheme

As already mentioned, the partial evaluator described in [3] lacks support for two language features, namely non-deterministic operations and `let` expressions. For instance, consider the definition

\[
\text{main } = \text{PEVAL } (\text{double } \text{coin})
\]

w.r.t. the definitions of `coin` and `double` shown in Sect. 2. The partial evaluator [3] unfolds the call to `double` in the body of `main` to \((0 \oplus 1) + (0 \oplus 1)\), so that the residual program yields the values 0, 1, 1, and 2 for `main`. However, according to the call-time choice semantics of Curry [14,18], the correct result would be 0 or 2 but not 1. This problem arises from the residual semantics of the original partial evaluator, which is based on term-rewriting so that non-determinism in shared subexpressions is duplicated in the residual programs.

The second missing feature are (mutually recursive) `let` expressions, i.e., bindings where the variables to be bound might occur in the right-hand side of the bindings. For example, it is not possible to partially evaluate the program

\[
\begin{align*}
\text{ones } &= \text{let } \text{ones } = 1 : \text{ones } \text{in } \text{ones} \\
\text{main } &= \text{PEVAL } (\text{take } 2 \text{ ones})
\end{align*}
\]

One might encounter that this does not impose a real restriction because recursive let-bindings could be interpreted by recursive function definitions (at the cost of some overhead). While this is possible for the example above, it is not whenever a non-deterministic value should be shared. For instance, consider the following program:

\[
\begin{align*}
\text{digits } &= \text{let } \text{digits } = (0 \oplus 1) : \text{digits } \text{in } \text{digits} \\
\text{main } &= \text{PEVAL } (\text{take } 2 \text{ digits})
\end{align*}
\]
Because of the let binding, the decision to bind digit to either 0 or 1 is shared, and, in consequence, the expression main evaluates to either [0, 0] or [1, 1]. If we replaced the definition of digits by a top-level operation, as in
digits = (0 ? 1) : digits
main = PEVAL (take 2 digits)
the expression main would produce the additional results [0, 1] and [1, 0]. Thus, recursive let expressions cannot be transformed into operations but must be explicitly considered by a partial evaluator.

The usage of both features in contemporary Curry programs is the motivation for us to develop a new partial evaluator. In contrast to [3], we do not use a semantics based on term rewriting. Instead, we base our work on the natural semantics for FlatCurry proposed in [1] which is intended to specify the call-time choice semantics of non-deterministic operations by modeling a heap structure to express sharing. A similar semantics has been used in [13] in a partial evaluator for first-order functional programs. In contrast to our approach, non-determinism, which is essential for Curry, has not been considered there.

4.1 FlatCurry

FlatCurry is a simple intermediate language used by Curry compilers [11, 16]. Moreover, it is also the basis of precise descriptions of the semantics of Curry [3] and semantics-based tools for Curry (e.g., [2, 3, 4]). The syntax of this representation is depicted in Fig. 2, where we denote a sequence of objects $o_1, \ldots, o_n$ by $o_n$. A FlatCurry program $P$ consists of a sequence of function definitions $D$ such that each function must be defined by a single rule with a linear left-hand side, i.e., the variables $\overline{x_n}$ must be pairwise different. The right-hand side of a function definition is an expression $e$ composed of variables ($x, y, z, \ldots$), constructors ($A, B, C, \ldots$), and function calls ($f, g, h, \ldots$). In the following, we denote by $\phi$ a constructor $c$ or a function $f$. For the sake of simplicity, we assume that literals occurring in the source program, like numbers or characters, are represented as nullary constructors. Additionally, we allow local (mutually recursive) bindings of variables, the introduction of free (logic) variables, disjunctions (to represent overlapping left-hand sides in the source language), and pattern matching. The patterns $p_i$ in case expressions are required to be pairwise different
and only consist of constructors applied to variables. In consequence, nested patterns in
the source language are represented by nested case expressions. For example, the list
concatenation \( \text{conc} \) is represented in FlatCurry as

\[
\text{conc}(xs, ys) = \text{case } xs \text{ of } \{ \text{[]} \rightarrow ys ; z:zs \rightarrow z : \text{conc}(zs, ys) \}
\]

Note that, in contrast to [1], we do not distinguish between flexible and rigid case
expressions. Although they behave differently on free (logic) variables [17], this differ-
ence is not relevant for partial evaluation [3]. Furthermore, we omit the representation
of external functions like arithmetics, which are implemented in the partial evaluator
but do not play a significant role in the evaluation scheme. Finally, we do not consider
higher-order applications in the syntax of FlatCurry since they can be represented by an
operation apply where partial applications are interpreted as constructor calls [3].

4.2 Natural Semantics

We base our partial evaluator on a variant of the operational semantics of FlatCurry
[1], also referred to as the natural semantics of FlatCurry. The semantics uses a heap
structure to specify sharing of expressions and computes the (flat) value of an expression
which is either a logic variable (w.r.t. the associated heap) or a constructor applied to
variables.

\[
\text{Heap} = \mathcal{V} \rightarrow \{ \text{free, } \text{■} \} \uplus \mathcal{E}xp \quad \text{Value} ::= x \mid c(\overline{x}_n)
\]

A heap is a partial mapping from a set of variables \( \mathcal{V} \) to either an expression (\( \mathcal{E}xp \) is
the set of expressions according to the syntax of FlatCurry), a special symbol “free”
to represent a free variable,\(^1\) or a symbol “■” representing a black hole.\(^2\) We denote
the empty heap by [], and the value associated to a variable \( x \) in a heap \( \Gamma \) by \( \Gamma[x] \).
\( \Gamma[x \mapsto e] \) denotes a heap \( \Gamma' \) with \( \Gamma'[x] = e \) and \( \Gamma'[y] = \Gamma[y] \) for all \( y \neq x \).

We use judgements of the form \( \Gamma : e \downarrow \Delta : v \) which express the fact that “the
expression \( e \) under the heap \( \Gamma \) evaluates to the value \( v \) and the (possibly modified) heap
\( \Delta \)”. The basic inference rules of the natural semantics are depicted in Fig. 3. We briefly
describe these rules and explain the differences to the original version of [1].

\textbf{(Value)} Evaluation of a value directly returns the value without modifying the heap.

\textbf{(VarExp)} This rule implements sharing of subexpressions. If a variable to be evaluated
is bound to an expression, the expression is evaluated and its value is returned.
In addition, the heap is updated with the value. During evaluation of the expres-
sion, the binding is replaced by ■, in contrast to [1]. This allows the detection of
black holes and is necessary for the correctness of the semantics [10] (see also
Appendix A for a detailed explanation).

\textbf{(Flatten)} To correctly implement sharing, arguments of function or constructor calls
must be represented in the heap. This is usually achieved by a preprocessing step
called flattening or normalization [1,19], but it can also be performed on demand.

---

\(^1\) [1] represents free variables by circular \texttt{let} bindings of the form \texttt{let } \{ x = x \} \texttt{ in } e, but
this prohibits the correct representation of such bindings occurring in the source code.

\(^2\) We use a special symbol for black holes instead of simply removing the binding for a variable
(as in [19]) in order to distinguish black holes from unbound variables.
\( \Gamma : v \downarrow \Gamma : v \) where \( v = c(\overline{x_n}) \) or \( v \in \mathcal{V} \) with \( \Gamma[v] = \text{free} \)

\[ \frac{\Gamma[x \mapsto e] : e \downarrow \Delta : v}{\Gamma[x \mapsto \mathbf{■}] : e \downarrow \Delta : v} \quad \text{where } e \notin \{\text{free, } \mathbf{■}\} \]

\[ \frac{\Gamma[y \mapsto e_i] : \phi(x_1, \ldots, x_{i-1}, y, e_{i+1}, \ldots, e_k) \downarrow \Delta : v}{\Gamma : \phi(x_1, \ldots, x_{i-1}, e_{i+1}, \ldots, e_k) \downarrow \Delta : v} \quad \text{where } e_i \notin \mathcal{V}, y \text{ fresh} \]

\[ \frac{\Gamma : \sigma(e) \downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \downarrow \Delta : v} \quad \text{where } f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n} \mapsto \overline{y_n}\} \]

\[ \frac{\Gamma[y_k \mapsto \sigma(e_k)] : \sigma(e) \downarrow \Delta : v}{\Gamma : \text{let } \{\overline{x_k} = e_k\} \text{ in } e \downarrow \Delta : v} \quad \text{where } \sigma = \{\overline{x_k} \mapsto y_k\}, y_k \text{ fresh} \]

\[ \frac{\Gamma : e_1 \downarrow \Delta : v}{\Gamma : e_2 \downarrow \Delta : v} \quad \text{where } i \in \{1, 2\} \]

\[ \frac{\Gamma[y_n \mapsto \text{free}]}{\Gamma : \text{let } \overline{x_n} \text{ free in } e \downarrow \Delta : v} \quad \text{where } \sigma = \{\overline{x_n} \mapsto y_n\}, y_n \text{ fresh} \]

\[ \frac{\Gamma : e \downarrow \Delta : c(\overline{y_n})}{\Gamma : \text{case } e \text{ of } \{\overline{p_k} \mapsto e_k\} \downarrow \Theta : v} \quad \text{where } p_i = c(\overline{x_n}), \sigma = \{\overline{x_n} \mapsto \overline{y_n}\} \]

\[ \frac{\Gamma : e \downarrow \Delta[x \mapsto \text{free}]}{\Gamma : e \downarrow \Delta[x \mapsto \sigma(p_i), \overline{y_n} \mapsto \text{free}]} \quad \text{where } i \in \{1, \ldots, k\}, p_i = c(\overline{x_n}), \sigma = \{\overline{x_n} \mapsto \overline{y_n}\}, y_n \text{ fresh} \]

**Fig. 3.** Natural semantics

(Fun) This rule unfolds a function call, where the result is obtained by evaluation of the function’s right-hand side. We assume that the program \( P \) is a global parameter of the calculus. Generally, whenever new variables are introduced by the program, we apply a renaming substitution \( \sigma \) to prohibit name clashes.

(Let) The bindings of a let construct are added to the heap after all variables have been renamed to fresh variable names.

(Or) This rule non-deterministically chooses one of the arguments to be further evaluated. In consequence, this rule introduces non-determinism into the calculus itself.

(Free) Like variables bound to expressions, logic variables are renamed and afterwards bound in the heap.

(Select) For case expressions whose argument evaluates to a constructor-rooted term the right-hand side of the corresponding alternative is selected and evaluated.

(Guess) For case expressions whose argument evaluates to a logic variable, one of the alternatives is non-deterministically chosen to be evaluated. The variable is then bound to the corresponding pattern where the variables inside the pattern are also bound as logic variables.

### 4.3 Ensuring Termination

Following the general idea of partial evaluation of functional logic programs [5] as well as logic programs [20], we evaluate an annotated expression \( e \) with a (possibly
incomplete) standard derivation $\varepsilon \Downarrow \Delta : e'$. In order to ensure the termination of the partial evaluation process, we defer the evaluation of some expressions. For example, consider the program

$$\text{loop } xs = \text{loop } xs$$
$$\text{main } xs = \text{PEVAL} (\text{loop } xs)$$

The evaluation of the expression $\text{main}$ does not terminate due to the recursive function call to $\text{loop}$. To achieve termination of the partial evaluation process, we modify the natural semantics as follows:

1. The evaluation of an expression can be deferred to avoid non-termination.
2. An operation $\text{proceed}$ is used to decide whether a function call should be unfolded or deferred.

This residualizing natural semantics is similar to [3,4] but more complex due to the use of a heap for sharing instead of term rewriting. Regarding the first modification, we extend the representation of values with a new symbol $\mathtt{\langle \cdot \rangle}$ which encloses expressions whose evaluation should be deferred.

This annotation directly corresponds to the $\text{PEVAL}$ annotation in source programs. Second, we extend the inference system with the operation $\text{proceed}$, deciding whether a function call should be unfolded, and replace the rule $\text{Fun}$ with:

$$\text{(FunEval)} \quad \frac{\Gamma : \sigma(e) \Downarrow \Delta : v \quad \text{where } f(x_n) = e \in P, \sigma = \{ x_n \mapsto y_n \}, \quad \text{proceed}(\Gamma, f(y_n)) = \text{true}}{\Gamma : f(y_n) \Downarrow \Delta : v}$$

$$\text{(FunDefer)} \quad \frac{\Gamma : f(y_n) \Downarrow \Gamma : \mathtt{\langle f(y_n) \rangle} \quad \text{where } f(x_n) = e \in P, \sigma = \{ x_n \mapsto y_n \}, \quad \text{proceed}(\Gamma, f(y_n)) = \text{false}}{\Gamma : f(y_n) \Downarrow \Gamma : \mathtt{\langle f(y_n) \rangle}}$$

Approaches for the concrete definition of $\text{proceed}$ will be discussed in Sect. 5.1. Furthermore, we extend the rule $\text{Value}$ to also return deferred expressions unchanged and constrain the rule $\text{VarExp}$ in that the value must not be a deferred expression. Finally, we add two more rules for deferred expressions where the annotation is lifted upwards:

$$\text{(VarDefer)} \quad \frac{\Gamma[x \mapsto \mathtt{\langle \cdot \rangle}] : e \Downarrow \Delta : \mathtt{\langle e' \rangle} \quad \text{where } e \notin \{ \text{free, } \mathtt{\langle \cdot \rangle} \}}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto e'] : \mathtt{\langle x \rangle}}$$

$$\text{(CaseDefer)} \quad \frac{\Gamma : e \Downarrow \Delta : \mathtt{\langle e' \rangle} \quad \text{where } e \notin \{ \text{free, } \mathtt{\langle \cdot \rangle} \}}{\Gamma : \text{case } e \text{ of } \{ p_k \mapsto e_k \} \Downarrow \Delta : \mathtt{\langle \text{case } e' \text{ of } \{ p_k \mapsto e_k \} \rangle}}$$

### 4.4 Dealing with Partial Information

In contrast to the evaluation performed in a standard interpreter, the partial evaluation process has to deal with partial knowledge in the form of unbound variables. For instance, if the right-hand side of a function declaration like $f \, x = \text{PEVAL} (g \, x)$ should be evaluated, there is no binding information for the parameter variable $x$. 

$^3$ Actually, the operation $\text{proceed}$ also takes into account the context of reductions already performed, but we omit them here for the sake of simplicity.
A possible solution is to handle such unbound variables as logic variables, as done in [5], so that they are bound to appropriate values by the partial evaluator. Since it has been shown in [2] that the back-propagation of these bindings can lead to incorrect residual programs, [3] uses a residualizing semantics which represents such bindings by \texttt{case} expressions in the residual program. However, this is only necessary for unbound variables. Explicitly introduced logic variables are known to be free during the actual evaluation so that they can be bound during partial evaluation time. For instance, consider the expression

\[
\text{let } x \text{ free in case } x \text{ of } \{ \text{True } \rightarrow 1 \}
\]

Here we can bind \( x \) to \( \text{True} \), since this binding is not visible outside the scope of this expression, select the (single) branch as the value of the \texttt{case} expression, and continue by evaluating its right-hand side. In consequence, our implementation evaluates this expression to 1, while the partial evaluator described in [3] cannot evaluate the expression any further.

Hence, we distinguish unbound variables from logic variables by \textit{not binding} them in the heap. Furthermore, we assume that rule \texttt{Value} is also applicable to variables not bound in the heap so that unknown variables reduce to themselves. Thus, only the rules for \texttt{case} expressions have to be changed, where it is now also possible that the scrutinized value is an unknown variable. Following the idea of [3], we generate \textit{residual case expressions} to defer the inspection of the variable to the run time of the specialized program. Therefore, we extend the definition of values to

\[
\text{Value ::= } \ldots \mid \text{case } x \text{ of } \{ p_k \rightarrow e_k \} \quad \text{(residual case expression)}
\]

where the variable \( x \) inspected in the \texttt{case} expression is not bound in the corresponding heap. Because \texttt{case} expressions are now contained in the set of values, we also have to consider them as the value of a variable or an expression examined by another \texttt{case} expression. Hence, we add the following rules:

\[
\begin{align*}
\text{(CaseUnbound) } & \quad \Gamma : e \Downarrow \Delta : x \\
\text{(CaseCase) } & \quad \Gamma : \text{case } e \text{ of } \{ p_k \rightarrow e_k \} \Downarrow \Delta : \text{case } x \text{ of } \{ p_k \rightarrow \llbracket e_k \rrbracket \} \\
\text{(VarCase) } & \quad \Gamma[x \mapsto \mathbf{\_}]: e \Downarrow \Delta : \text{case } y \text{ of } \{ p_k \rightarrow \llbracket e_k \rrbracket \} \\
\text{(CaseVarCase) } & \quad \Gamma : \text{case } e \text{ of } \{ p_k \rightarrow e_k \} \Downarrow \Delta : \text{case } y \text{ of } \{ p_k \rightarrow \llbracket e_k \rrbracket \} \\
\end{align*}
\]

The general idea is to lift \texttt{case} expressions inspecting an unbound variable upwards and to defer the evaluation of the alternatives. Such deferred expressions are not further evaluated in the residual semantics but later extracted by the global iterative process.
as the initial expressions of a new specialization run. Because the alternatives are then evaluated independently, it will be possible to take the binding information of the case expression into account. For instance, if we consider the expression

\[
\text{case } x \text{ of } \{ \text{True} \to \neg x \}
\]
a subsequent evaluation of the right-hand side \(\neg x\) may respect the binding of \(x\) to \(\text{True}\) and, thus, directly evaluate to \(\text{False}\).

### 4.5 Dereferencing the Heap

After evaluating an expression to a residual value, this value might contain variables which are either free or bound to expressions in the corresponding heap. To be able to replace parts of the input program with residual values, these bindings have to be added to the values to form valid expressions, a process we call *dereferencing the heap*. Conceptually, for a given configuration \(\Gamma : e\), we retrieve the set of variables transitively reachable from \(e\) and bound in \(\Gamma\) and add the corresponding bindings to the expression. For residual case expressions, we also respect the bindings represented by the case expression. The bindings are divided into logic variables \((fv)\) and variables bound to expressions \((bv)\) and added to the original expression:

\[
drf(\Gamma, e) = \begin{cases} 
\text{case } x \text{ of } \{ p_k \to drf(\Gamma[x \mapsto p_k], e_k) \} & \text{if } e = \text{case } x \text{ of } \{ p_k \to e_k \} \\
\langle \{ \text{let } fv(\Gamma, e) \text{ free in let } bv(\Gamma, e) \text{ in } e \} \rangle & \text{otherwise}
\end{cases}
\]

For instance, if we consider the configuration

\[
[y \mapsto \text{free}]: \text{case } x \text{ of } \{ \text{True} \to x; \text{False} \to y\}
\]

then dereferencing will produce the expression

\[
\text{case } x \text{ of } \{ \text{True} \to \langle \{ \text{let } x = \text{True} \text{ in } x \} \rangle; \text{False} \to \langle \{ \text{let } y \text{ free in } y \} \rangle \}
\]

### 5 Control

Our partial evaluation algorithm follows the general procedure of Alpuente et. al. [5], which is parametric w.r.t. an unfolding rule used to construct a finite derivation for an expression and an abstraction operator used to guarantee that only finitely many expressions are evaluated. The basic algorithm is depicted in Fig. 4 and works as follows. Given an input program \(P\) and a set of annotated expressions \(E\), the algorithm starts by applying an unfolding rule which evaluates each expression according to the residual semantics presented in the previous section and extracts the results by \(drf\). If there is more than one derivation in the residual semantics due to the non-deterministic inference rules Or and Guess, the different extracted results of the derivation are combined by the choice operator “?:”. If there is no derivation at all, the result is represented by the predefined operation failed. In the next step, an abstraction operator is applied to this set, adding the new expressions to the set of already evaluated expressions. This phase yields a new set which may need further evaluation, hence, this process is iteratively repeated until no more expressions are added to the set. This iteration is necessary for the correctness of partial deduction [20] in order to achieve a “closed” set of expressions.
that covers all expressions possibly occurring in the residual program. To generate the resulting program, the same unfolding rule has to be applied to the resulting set of expressions to generate the corresponding resultants, i.e., the rules of the residual program (in our implementation, this step is integrated into the algorithm). Finally, the set of generated resultants are compressed to eliminate intermediate and redundant functions (see [5] for details).

This procedure distinguishes two levels of control, namely the local level, managed by the unfolding rule to avoid infinite evaluations, and the global level, managed by the abstraction operator to avoid infinitely repetitions of the partial evaluation algorithm. To ensure termination of the whole process, both local and global termination is required.

5.1 Local Control

Termination of the unfolding rule directly corresponds to termination of the residual semantics presented in Sect. 4. For this purpose, the semantics has already been extended by an oracle proceed(Γ,e) responsible for the decision whether a function call should be unfolded or not. There exist several well-known techniques in the literature to come to this decision, e.g., depth-bounds, loop-checks [9], well-founded orderings [12], or well-quasi orderings [22]. Our implementation currently supports the following simple strategies:

None No unfolding is performed for user-defined functions.
One Only one function call is unfolded for each evaluation.
Each At most one call is unfolded for each user-defined function, subsequent calls are deferred.
All All function calls are unfolded, which corresponds to the original inference system.

This does not guarantee termination but may be useful if the user is sure that the process terminates.

Note that, regardless of the chosen strategy, built-in functions (such as arithmetics) are evaluated in any case, since they are known to terminate.

Expressions that have been deferred during evaluation will be extracted and eventually added to the set of expressions to be evaluated, depending on the operation abstract (see Sect. 5.2 for details). Generally, a strategy that allows more evaluation steps in one derivation than another strategy might seem superior. If an evaluation is split into multiple derivations with deferred subexpressions, each of these subexpressions has to
be evaluated anew and leads to a new residual function to be generated. In contrast, longer derivations will produce less deferred subexpressions and, hence, less residual functions. Nevertheless, although a simpler strategy may produce more intermediate expressions, there are better chances that some of these expressions have already been encountered before, reducing the overall number of expressions to be evaluated. Furthermore, the final compression phase will eliminate intermediate functions so that even the simple strategies perform very well in practice.

5.2 Global Control

The local control is parametric w.r.t. the decision whether to stop or to proceed with the evaluation, since it is safe to terminate the evaluation at any point. This flexibility does not apply to the global control because we cannot stop the iterative extension of the set of expressions until all function calls in this set are “closed” w.r.t. the set of expressions. An expression \( e \) is closed w.r.t. a set of expressions if it is an instance of an expression in the set and all expressions in the matching substitution are recursively closed (see [5] for details). This condition is necessary to ensure the correctness of the partial evaluator so that the specialized program computes the same solutions as the original program. In order to avoid the construction of infinite sets of expressions, expressions in this set are generalized to ensure termination of this process.

Hence, the operation \( \text{abstract} \) returns a safe approximation of \( E_i \cup E' \) so that each expression in the set of \( E_i \cup E' \) is closed w.r.t. the result of \( \text{abstract}(E_i, E') \). More precisely, an expression \( e' \in E' \) is added to the set \( E_i \) according to the following rules (note that the result of unfolding is either a variable, a deferred expression, a constructor application, a case expression, or a choice of these results):

1. If \( e' \) is a variable, it is discarded.
2. If \( e' \) has the form \( \langle e \rangle \), one of the following options is considered:
   - (a) add \( e \) to the set \( E_i \),
   - (b) discard the expression \( e \), or
   - (c) compute the most specific generalization of \( e \) and some expression \( e' \in E' \), say \( \hat{e} \), and try to add both \( \hat{e} \) and the expressions in the corresponding substitutions \( \sigma \) and \( \theta \), where \( e = \sigma(\hat{e}) \) and \( e' = \theta(\hat{e}) \).
3. For all other cases (constructor calls, case expressions, choices), the corresponding subexpressions are considered.

Like for the unfolding rule, the abstraction can be parameterized by a criterion to decide the option taken in (2). Our implementation currently supports abstractions using a well-founded ordering or an embedding ordering to distinguish between (2a) and (2c), i.e., smaller expressions are added but larger expressions are generalized.

To achieve a good level of specialization, it is crucial to recognize different variants of one expression as equivalent in order to discard them in (2b). This is more complex in our framework compared to [3], since we take let expressions into account. For example, consider the equivalent expressions “map(square,xs)” and “let \( f = \text{square} \) in map(f,xs)”. If we do not recognize them as variants, they might be generalized to map(f,xs) which could not further be specialized. Therefore, we normalize expressions by applying \( \alpha \)-conversion and flattening [19] before computing their abstractions.
In this section we evaluate the implementation of our partial evaluator by some benchmarks. We compile both the partial evaluator and the benchmarks with the PAKCS Curry compiler (version 1.11.3, based on SICStus Prolog 4.2.3). All benchmarks were executed on a Linux machine (Debian Wheezy) with an Intel Core i5-750 (2.66GHz) processor and 4GiB of memory. The timings were performed using the profiling operation `profileTimeNF` of PAKCS and denote the time required for computing the normal form of the respective result in milliseconds (the arguments passed to the various functions were evaluated before to bring out the speedup obtained by partial evaluation). The benchmark examples have been specialized with one unfolding per evaluation and without any abstraction, since all examples terminated. Experiments with both a well-founded ordering or a well-quasi ordering resulted in the same or worse performance. Table 1 presents the time required for the partial evaluation process itself, for executing the original and the specialized program, and the gained speedup.

In the first group of benchmarks, we consider some typical examples of partial deduction and functional program transformations. These are simple functions working on lists or trees as (intermediate) data structures: `allOnes` computes the length of its input list, represented as Peano numbers, and constructs a new list of the same length with `1` as all elements, `doubleApp` is the concatenation of three lists, `doubleFlip` flips a tree structure twice, returning the same tree, `lengthApp` computes the length of the concatenation of two lists, and `kmp` implements a generic string pattern matcher. The first four functions were specialized without static input data, while the `kmp` example was specialized w.r.t. a fixed pattern of length 4, explaining both the time needed for partial evaluation and the gained speedup.

In the second group, we benchmark some examples with higher-order functions: the computation of the sum of list elements using `foldr`, the sum of squared numbers, the concatenation of a list of lists, and repeatedly applying a function to a list. All functions are applied to an input list `xs` containing 200,000 elements. The speedup is generally achieved because of the removal of intermediate data structures. For instance, the Curry
expression “foldr (+) 0 (map square xs)” is specialized to the following residual FlatCurry definition:

\[
\text{sumSquare}(xs) = \text{case}\ \text{xs}\ \text{of}\ \{\ [
\] \rightarrow 0 \\
; \ y:ys \rightarrow (y*y) + \text{sumSquare}(ys) \}
\]

Finally, we evaluate two (complicated) variants of the function choose, which nondeterministically chooses one element of a given list ys containing 10,000 elements:

\[
\text{choose}\ (x:xs) = x \ ? \ \text{choose}\ xs
\]

Our partial evaluator computes this simple implementation of choose for the first example. The result for the second example only differs from choose in the order in which the two non-deterministic alternatives are taken, which stems from the implementation of perm. The huge speedup is achieved because of the omission of the non-deterministic intermediate list structure.

To summarize, our partial evaluator shows promising results and is capable of performing optimizations such as deforestation [24] and transformation of higher-order functions to first-order ones. In addition, non-deterministic operations are correctly specialized in contrast to [3], and the results for deterministic operations are almost identical.

7 Conclusions and Future Work

We have presented a new partial evaluation scheme for the functional logic language Curry based on its intermediate representation FlatCurry. The partial evaluator is based on an adaptation of the natural semantics of FlatCurry, extending the semantics to deal with the requirements of partial evaluation such as ensuring termination. In contrast to the original partial evaluator [3], which is based on term rewriting without sharing, the new implementation correctly handles both recursive let expressions and non-deterministic operations and, thus, supports full (Flat)Curry. As our benchmarks demonstrate, the implementation is capable of powerful optimizations both to deterministic and non-deterministic programs.

For future work, we intend to formally prove the correctness of the partial evaluation scheme, which should be manageable due to the similarity of the original and residual semantics. Another aspect for further investigations is the improvement of the abstraction operator. While the abstraction is necessary to ensure termination, a too general abstraction reduces the quality of the specialization. Thus, more sophisticated abstraction operators might be beneficial.

References


A  Black Hole Detection

As mentioned in Sect. 4.2, rule VarExp of the natural semantics shown in Fig. 3 replaces the variable binding \( x \mapsto e \) by \( x \mapsto \mathbf{e} \) in the heap when evaluating the associated expression \( e \). This allows the detection of black holes (a self-dependent infinite loop) [19], as done in some implementations of functional (logic) languages. For instance, an attempt to evaluate the expression “let \( \{ x = x \} \) in \( x \)” would result in a finite but incomplete derivation tree, whereas it would trigger the construction of an infinite derivation tree if the binding \( x \mapsto e \) was kept.

The detection of black holes included in the semantics seems to be an optimization that could be omitted for deterministic programs [19]. However, it is crucial in combination with non-determinism in order to prevent the binding of a variable to different values in the same derivation, as shown in [10]. For example, consider the expression “let \( \{ x = \mathbf{T} \ ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} \} \) in \( x \)”. If we do not replace the variable binding in rule VarExp, the following derivation would be possible:

\[
\begin{align*}
\Gamma : \mathbf{T} & \Downarrow \Gamma : \mathbf{T} \\
\Gamma : \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} & \Downarrow \Gamma : \mathbf{T} \\
\Gamma : x & \Downarrow [x \mapsto \mathbf{T}] : \mathbf{T} & [x \mapsto \mathbf{F}] : \mathbf{T} \\
\Gamma : \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} & \Downarrow [x \mapsto \mathbf{T}] : \mathbf{F} \\
\Gamma : \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} & \Downarrow [x \mapsto \mathbf{T}] : \mathbf{F} \\
\Gamma : x & \Downarrow [x \mapsto \mathbf{F}] : \mathbf{F} \\
[\] : \text{let } \{ x = \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} \} \ 	ext{in } x & \Downarrow [x \mapsto \mathbf{F}] : \mathbf{F}
\end{align*}
\]

where \( \Gamma = [x \mapsto \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \}] \).

In this derivation, the variable \( x \) is looked up in the heap twice, where at first the right (non-deterministic) branch is chosen and afterwards the left branch. Hence, \( x \) is bound to \( \mathbf{T} \) as well as \( \mathbf{F} \), which violates the single assignment property of call-time choice.

With our semantics, there is one successful and one failing derivation but no derivation where \( x \) is bound to \( \mathbf{T} \) as well as \( \mathbf{F} \):

\[
\begin{align*}
[x \mapsto \mathbf{e}] : \mathbf{T} & \Downarrow [x \mapsto \mathbf{e}] : \mathbf{T} \\
[x \mapsto \mathbf{e}] : \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} & \Downarrow [x \mapsto \mathbf{e}] : \mathbf{T} \\
[x \mapsto \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \}] : x & \Downarrow [x \mapsto \mathbf{T}] : \mathbf{T} \\
[\] : \text{let } \{ x = \mathbf{T} ? \ \text{case } x \ \text{of } \{ \mathbf{T} \rightarrow \mathbf{F} \} \} \ 	ext{in } x & \Downarrow [x \mapsto \mathbf{T}] : \mathbf{T}
\end{align*}
\]

B  Residualizing Semantics

Since the various rules of the residualizing semantics used in our partial evaluator are distributed over the paper and some of them were only informally sketched, we summarize in the following the complete set of rules of our residualizing semantics.
Automatic Testing of Operation Invariance

Tobias Gödderz and Janis Voigtländer

University of Bonn, Germany, {goedderz,jv}@cs.uni-bonn.de

Abstract. We present an approach to automatically generating operation invariance tests for use with Haskell’s random testing framework QuickCheck. The motivation stems from a paper by Holdermans [8] which showed how to address certain shortcomings of straightforward testing of implementations of an abstract datatype. While effective, his solution requires extra generation work from the test engineer. Also, it may not even be doable if the person responsible for testing has no knowledge about, and program-level access to, the internals of the concrete datatype implementation under test. We propose and realize a refinement to Holdermans’ solution that improves on both aspects: Required operation invariance tests can be formulated even in ignorance of implementation internals, and can be automatically generated using Template Haskell.

1 Introduction

It is good software engineering practice to test one’s, and possibly other’s, code. In declarative languages, QuickCheck [4] and related tools [1, 3, 10] are an attractive option, combining convenient ways of generating test input data and a DSL for expressing properties to be tested.

One recurring situation where property-based testing is desirable is in connection with implementations of an abstract datatype (ADT) specification. The scenario is that some interface (API) is given as a collection of type signatures of operations along with a collection of equational axioms that are expected to hold of those operations. Then there is an, or possibly several, implementations of that specification. The user of such an implementation should not need to be concerned with its internals, but would still like to be convinced of its correctness. It seems very natural to simply turn the given axioms into QuickCheck-like properties, and to accept an implementation as correct if it passes all those axioms-become-tests. However, it has been known for a while that such an approach is not enough to uncover all possible errors [6, 9]. Subtle bugs can remain hidden, due to an unfortunate interplay of buggy implementation and programmed equality (while actual semantic equality is impossible to test in general). Recently, Holdermans [8] presented a solution that works by adding operation invariance tests, to ensure that the assumed notion of equality is not just an equivalence relation, but actually a congruence relation. His solution has its own problems though. Specifically, it requires hand-writing a certain kind of additional test input data generator. But not only is that something which may depend on implementation internals (so is not necessarily something that a user having access only to the datatype’s external API could do); it is also extra work and an additional source of potential errors: get that generator wrong by not covering enough ground, and even the additional tests will not be enough to really establish overall correctness. We set out to improve on these aspects.
2 The Need for Operation Invariance Tests

This section is largely a recap of material from Holdermans’ paper. Specifically, we also use his very compelling example of how the approach of simply (and only) testing the axioms from the specification of an ADT may lead to a false sense of security.

Assume a set of people is interested in integer FIFO queues, and in particular in separately specifying, implementing, and using them. The specification is just a mathematical entity, consisting of a listing of signatures of desirable operations:

\[
\begin{align*}
\text{empty} & :\text{Queue} \\
\text{enqueue} & :\text{Int} \rightarrow \text{Queue} \rightarrow \text{Queue} \\
\text{isEmpty} & :\text{Queue} \rightarrow \text{Bool} \\
\text{dequeue} & :\text{Queue} \rightarrow \text{Queue} \\
\text{front} & :\text{Queue} \rightarrow \text{Int}
\end{align*}
\]

and of axioms expected to hold:

\[
\begin{align*}
Q_1: \text{isEmpty empty} &= \text{True} \\
Q_2: \text{isEmpty (enqueue x q)} &= \text{False} \\
Q_3: \text{front (enqueue x empty)} &= x \\
Q_4: \text{front (enqueue x q)} &= \text{front q} & \text{if isEmpty q} &= \text{False} \\
Q_5: \text{dequeue (enqueue x empty)} &= \text{empty} \\
Q_6: \text{dequeue (enqueue x q)} &= \text{enqueue x (dequeue q)} & \text{if isEmpty q} &= \text{False}
\end{align*}
\]

An implementation in Haskell would be a module

\[
\text{module} \quad \text{Queue (Queue, empty, enqueue, isEmpty, front, dequeue) where}
\]

\[
\ldots
\]

that contains some definition for Queue as well as definitions for the operations adhering to the type signatures from the specification. Importantly, the type Queue is exported without any data constructors, so from the outside of the module, Queue values can only be created, manipulated, and inspected using the provided operations. A prospective user of the implementation can import the above module and call those operations in application code. Now, the user would like to have some certainty that the implementation is correct. The appropriate notion of correctness is adherence to the axioms from the mathematical specification, on which user and implementer should have agreed beforehand. One way for the implementer to convince the user of the correctness is to do extensive property-based testing to establish validity of the specification’s axioms for the provided implementation. Using QuickCheck, the test suite would consist of the following properties:

\[
\begin{align*}
q_1 &= \text{property (isEmpty empty == True)} \\
q_2 &= \text{property (\lambda x q \rightarrow isEmpty (enqueue x q) == False)} \\
q_3 &= \text{property (\lambda x \rightarrow front (enqueue x empty) == x)}
\end{align*}
\]
\[ q_4 = \text{property } (\lambda x \ q \to \text{isEmpty} \ q == \text{False} \implies \text{front} (\text{enqueue} x \ q) == \text{front} q) \]
\[ q_5 = \text{property } (\lambda x \to \text{dequeue} (\text{enqueue} x \text{ empty}) == \text{empty}) \]
\[ q_6 = \text{property } (\lambda x \ q \to \text{isEmpty} \ q == \text{False} \implies \text{dequeue} (\text{enqueue} x \ q) == \text{enqueue} x (\text{dequeue} q)) \]

These could even be written down by the user person in ignorance of any internals of the implementation. However, the use of \(==\) on Queue values requires availability of an appropriate instance of the Eq type class (which is how Haskell organizes overloading of \(==\)). Let us assume the implementer provides such an instance. Moreover, we have to assume that the implementer also provides an appropriate random generator for Queue values (the \(q\) values quantified via the lambda-abstractions above – whereas for the \text{Int} values quantified as \(x\) we can take for granted that a random generator already exists), because otherwise the properties cannot be tested. In general, the implementer may even have to provide several random Queue generators for different purposes, for example since otherwise certain preconditions in axioms might be too seldom fulfilled, thus preventing effective testing.\(^1\) But in the example here this is not an issue, since both empty and nonempty queues will appear with reasonable likelihood among the generated test data.

So if a Queue implementation passes all the above tests, it should be correct, right? Unfortunately not. As Holdermans [8] demonstrates, it can happen, even under all the assumptions above, that an implementation passes all the tests but is still harmfully incorrect. Let us repeat that faulty implementation in full as well (also since we will later want to refer to it in checking the adequacy of our own testing solution to the overall problem). Here it is:

\begin{verbatim}
data Queue = BQ [Int] [Int]
bq :: [Int] -> [Int] -> Queue
bq [] r = BQ (reverse r) []
bq f r = BQ f r
empty :: Queue
empty = bq [] []
enqueue :: Int -> Queue -> Queue
enqueue x (BQ f r) = bq f (x:r)
isEmpty :: Queue -> Bool
isEmpty (BQ f _) = null f
\end{verbatim}

\(^1\) QuickCheck’s restricting conditional operator \(\implies\), written as \(==>\), does not count “\(A\)” being false as evidence for “\(A\) implies \(B\)” being true. Thus, in order to reach a certain number of positive test cases for “\(A\) implies \(B\)”, that many cases with both “\(A\)” and “\(B\)” being true must be encountered (and, of course, not a single case with “\(A\)” true but “\(B\)” false). Consequently, QuickCheck gives up testing if not enough cases with “\(A\)” being true are encountered.
front :: Queue → Int
front (BQ f _) = last f

dequeue :: Queue → Queue
dequeue (BQ f r) = bq (tail f) r

This implementation uses the “smart constructor” bq to preserve the invariant that, for every BQ f r, it holds that null f implies null r, which makes front simpler to implement. Ironically, in the implementation above the error nevertheless lies in the definition of front, which should have been head f rather than last f.

As already mentioned, in order to test properties q₁–q₆, we need a definition of == for Queue and a random generator for Queue values. The implementer is kind enough to provide both (and both are perfectly sane) within the Queue module:

```haskell
instance Eq Queue where
  q == q' = toList q == toList q'
toList :: Queue → [Int]
toList (BQ f r) = f ++ reverse r

instance Arbitrary Queue where
  arbitrary = do
    f ← arbitrary
    r ← arbitrary
    return (bq f r)
```

Now we can run tests quickCheck q₁ through quickCheck q₆, and find that all are satisfied. And this is not just happenstance, by incidentally not hitting a counterexample during the random generation of test input data. No, it is a mathematical fact that the implementation above satisfies the ==-equations in properties q₁–q₆. And yet the implementation is incorrect. To see this, consider:

```haskell
> let q = dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty)))
> let q′ = enqueue 3 (dequeue (enqueue 2 (enqueue 1 empty)))

front (dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty)))) = 2
```

Clearly, a correct implementation of a FIFO queue should have output 2 here. In fact, Q₂–Q₆ can be used to prove, by equational reasoning, that

\[
\text{front (dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty))))} = 2
\]

The key to what went wrong here (the implementation being incorrect and yet passing all the tests derived from the specification axioms) is that properties q₁–q₆ do not, and in fact cannot, test for semantic equivalence. They can only be formulated using the programmed equivalence ==. That would be fine if == were not just any equivalence relation that satisfies q₁–q₆ (which we know it does), but actually a congruence relation that does so. For otherwise, == cannot correspond to the semantic equivalence = intuitively used in Q₁–Q₆. Being a congruence relation means to, in addition to being an equivalence relation, be compatible with all operations from the ADT specification. And that is not the case for the implementation given above. For example,

```haskell
> let q = dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty)))
> let q′ = enqueue 3 (dequeue (enqueue 2 (enqueue 1 empty)))

75
```
That does not necessarily mean that, morally, the implementation of \texttt{==} given for \texttt{Queue} (and used in the tests) is wrong. The real bug here resides in the implementation of \texttt{front}, but it cannot be detected by just checking \texttt{q1\textasciitilde q6}; one additionally needs appropriate compatibility axioms.\footnote{In general, one should also convince oneself in the first place that \texttt{==} is indeed an equivalence relation as well (reflexive, symmetric, transitive), but often this will be a triviality. For example, any definition of the form \texttt{x \textasciitilde x' = f x \textasciitilde f x'} for some function \texttt{f}, like in the \texttt{Eq} instance for \texttt{Queue}, is already guaranteed to give an equivalence relation if \texttt{==} on the target type of \texttt{f} is known to be an equivalence relation.}

So Holdermans [8] observes that one should check additional axioms like

\[
q_7 = \text{property } (\lambda q q' \rightarrow q == q' \Rightarrow \text{front } q == \text{front } q')
\]

Actually, he wisely adds a non-emptiness check to the precondition to prevent both \texttt{front q} and \texttt{front q'} from leading to a runtime error. But even then, the new axiom (or any of the other added compatibility axioms) does not – except in very lucky attempts – actually uncover the bug in the implementation. The problem is that it is very unlikely to hit a case with \texttt{q == q'} for random \texttt{q} and \texttt{q'}. And even if QuickCheck hits one such case every now and then, it is not very likely that it is also a counterexample to \texttt{front q == front q'}. So in the vast majority of test runs QuickCheck simply gives up, and we are none the wiser about whether the compatibility axioms do hold for the given implementation or do not.

Holdermans’ solution to this problem is to invest manual effort into randomly generating pairs of \texttt{q} and \texttt{q'} that ought to be considered equivalent queues. Concretely,

\begin{verbatim}
data Equiv a = a :\equiv: a

instance Arbitrary (Equiv Queue) where
  arbitrary = do
    z <- arbitrary
    x <- from z
    y <- from z
    return (x :\equiv: y)

  where
    from xs = do
      i <- choose (0, length xs - 1)
      let (xs1, xs2) = splitAt i xs
      return (bq xs1 (reverse xs2))
\end{verbatim}

Given some \texttt{Show} instances for \texttt{Queue} and \texttt{Equiv Queue}, testing the newly formulated property

\[
q_7 = \text{property } (\lambda (q :\equiv: q') \rightarrow \text{not } (\text{isEmpty } q) \Rightarrow \text{front } q == \text{front } q')
\]

would yield, for example:
That, finally, is useful information which can be used to figure out the bug in the implementation of \textit{front}.

But this success depends on the implementer having provided a good definition for the Arbitrary (Equiv Queue) instance above, specifically, a good \textit{from}-function for “perturbing” a randomly generated queue in different ways. Note, this is not something the user could do themselves, or exercise any control over, since implementing that instance crucially depends on having access to the internals of the queue implementation module. If the implementer makes a mistake in writing that function/instance, then the bug in the original implementation of \textit{front} will possibly remain unnoticed. For example, in an extreme case, the implementer may accidentally write the Arbitrary (Equiv Queue) instance in such a way that \textit{from} is semantically equivalent to \textit{return}, in which case \textit{q7} and all other compatibility axioms will be tested positively, despite the implementation of \textit{front} still being incorrect. Holdermans notes that one might test the chosen perturbation functionality itself to ensure not having introduced an error there. But this, which would correspond to a test \textit{quickCheck} (\lambda (q \equiv: q') \rightarrow q == q') does not help with judging the suitability of the perturbation in terms of having “enough randomness”; clearly \textit{from} \textit{xs} = \textit{return} \textit{xs} would make this test succeed as well. And the inverse property, that any pair of equivalent queues \textit{q} == \textit{q'} has indeed a chance of also being generated as \textit{q} :\equiv: \textit{q'}, is unfortunately not expressible as a QuickCheck test.

Our purpose now is to avoid the need of defining an Arbitrary (Equiv Queue) instance. Thus, we will empower the user to detect operation invariance violations in an ADT implementation without having access to internals or relying on the implementer to generate the required pairs of equivalent values for tests.

### 3 Thoughts on Random Terms

A simple idea would be to work purely symbolically as long as possible. For example, the values

\[
q = \text{dequeue} \left( \text{enqueue} \ 3 \left( \text{enqueue} \ 2 \left( \text{enqueue} \ 1 \text{ empty} \right) \right) \right)
\]

and

\[
q' = \text{enqueue} \ 3 \left( \text{dequeue} \left( \text{enqueue} \ 2 \left( \text{enqueue} \ 1 \text{ empty} \right) \right) \right)
\]

that we first mentioned as evidence for a case with \textit{q} == \textit{q'} but \textit{front} \textit{q} /= \textit{front} \textit{q} \textsuperscript{3}, are semantically equivalent according to Q\textsubscript{1}–Q\textsubscript{6}. Naturally, any pair of conceptually equivalent queues can be shown (mathematically) to be equivalent by using those original axioms. After all, that is the whole point of having the ADT specification comprising of those axioms in the first place.

\textsuperscript{3} We have /= as the negation of programmed equivalence ==.
So a suitable strategy for generating terms \( q \) and \( q' \) that would be eligible as \( q \equiv q' \) pairs might be to start from the given signatures of operations, randomly generate a well-typed term \( q \) from those operations, then randomly apply a random selection of the axioms \( Q_1-Q_6 \) at random places in the term \( q \) to obtain another term \( q' \). Using pairs of such \( q \) and \( q' \) for tests like \( \text{front } q == \text{front } q' \) (now actually computing the result, no longer working symbolically) would indeed, in principle, have the power to detect the bugs that violate operation invariance. In fact, this is exactly the strategy we initially pursued.

However, it does not work out in practice. Already generating random terms is known to be non-trivial, though solutions exist [5]. Moreover applying random axioms at random places is difficult in general to get right in the sense of ending up with a probability distribution that is effective in terms of hitting cases that uncover bugs. For example, given an integer queue, only the root node can be of type \( \text{Bool} \), which will lead to a low probability of axiom \( Q_2 \) to be applicable at a randomly selected node and an even lower probability of it to be selected for application, especially for large terms. Conversely first deciding on an axiom to apply and then randomly generating a (sub)term at which it is indeed applicable leads to similar uncertainties about coverage.

Additionally, there is a problem with “infeasible” terms. For integer queues, that is every term where \( \text{dequeue} \) or \( \text{front} \) is applied to a subterm \( t \) with \( \text{isEmpty} \ t \) being \( \text{True} \). Assume for now that \( x \) in each term \( \text{enqueue } x \ q \) might only be an integer (and not a more complex subterm \( \text{front } q' \)).\(^4\) Then the tree of each term is a path. Its leaf will be \( \text{empty} \), its root node one of \( \text{enqueue}, \text{isEmpty}, \text{front}, \text{or} \text{dequeue} \). All inner nodes can only be one of \( \text{enqueue} \) or \( \text{dequeue} \). If at any node in the path more of its descendants are of type \( \text{dequeue} \) than \( \text{enqueue} \), then the term is not valid. The probability that a term of length \( n \) is feasible is \( \frac{1}{2} \) for \( n = 1 \), \( \frac{252}{1024} \) for \( n = 10 \), and about \( 0.125 \) for \( n = 40 \). This might be much worse for ADTs with more complex terms.

Without prior knowledge about which operations are problematic and which axioms contribute to situations of equivalent values (queues, in the example) whose equivalence is not preserved by applying an operation, there is not much hope with the naive approach described above. But of course we precisely want to avoid having to invest any such knowledge about a specific ADT or implementation, since our goal is a generic solution to the problem. Also, working purely symbolically would not be practical for another reason: some axioms, like \( Q_4 \) and \( Q_6 \), have preconditions that need to be established before application. So in the hypothetical rewriting of \( q \) into \( q' \) by random applications of axioms described above, we would either have to symbolically prove such preconditions along the way (arbitrarily hard in general), or resort to actually computing values of subterms of \( q \) to establish whether some axiom is applicable there or not.

### 4 Our Solution

So what can we do instead? We want to avoid having to apply arbitrarily many rewrites at random places deep in some terms. Toward a solution, let us discuss the case of a binary operation \( f \) with two argument positions to be filled by a value of the abstract

\(^4\) This could be seen as having an operation \( \text{enqueue}_x : \text{Queue} \rightarrow \text{Queue} \) for every \( x : \text{Int} \).
The required operation invariance test would be that \( t_1 \equiv t'_1 \) and \( t_2 \equiv t'_2 \) imply \( f t_1 t_2 \equiv f t'_1 t'_2 \), where \( t_i \equiv t'_i \) means that \( t_i \) and \( t'_i \) are convertible, i.e., there is a sequence of axiom applications that leads from term \( t_i \) to term \( t'_i \). We claim that it would be enough to focus on the two argument positions of \( f \) separately, and to consider only single axiom applications. Indeed, assume the above test leads to a counterexample, i.e., \( f t_1 t_2 \neq f t'_1 t'_2 \). Then there are already \( t, t' \), and \( t'' \) with \( f t t' \equiv f t t'' \) or \( f t' t \equiv f t' t'' \) and where \( t' \) and \( t'' \) are exactly one axiom application apart. Why? By the given assumptions, we can form a sequence \( f t_1 t_2 = \ldots = f t'_1 t'_2 = \ldots = f t''_1 t''_2 \) of single axiom applications, where in the first part of that sequence axioms are only applied inside the first argument position of \( f \), later only inside the second one. Now if \( t, t' \), and \( t'' \) with the claimed properties would not exist, then it would have to be the case that \( f t_1 t_2 = \ldots = f t'_1 t'_2 = \ldots = f t''_1 t''_2 \) and thus \( f t_1 t_2 \equiv f t'_1 t'_2 \) by transitivity of \( \equiv \). But this is a contradiction to \( f t_1 t_2 \neq f t'_1 t'_2 \).

So, generalizing from the case of a binary operation, we have that in order to establish operation invariance overall, it suffices to test that for every operation \( f \) it holds \( f \ldots t' \ldots \equiv f \ldots t'' \ldots \) for every argument position and all pairs \( t' \) and \( t'' \) of terms related by exactly one axiom application and where the other argument positions are appropriately filled with random input (but with the same on left and right; this is what the “…” indicate in \( f \ldots t' \ldots \equiv f \ldots t'' \ldots \)). Still, the axiom application relating \( t' \) and \( t'' \) could be somewhere deeply nested, i.e., \( t' \) could be \( f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots \text{lhs} \ldots) \ldots) \ldots) \ldots \) and \( t'' \) be \( f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots \text{rhs} \ldots) \ldots) \ldots) \ldots \), where \( \text{lhs} \) and \( \text{rhs} \) are the two sides of an instantiated axiom, while all the other parts (“…” in the two nestings agree. We could generate tests arising from this, an important observation being that for the “…” parts, since they are equal on both sides (not only equivalent), we can simply use random values – no random symbolic terms are necessary, which is an important gain. Note that, as Haskell is purely functional, it is not necessary to model dependencies between arguments to obtain completeness. Actually, we restrict to a subset of all tests, namely ones where the axiom application in fact is at the root of \( t' \) and \( t'' \). That is, we only employ tests \( f \ldots \text{lhs} \ldots \equiv f \ldots \text{rhs} \ldots \), not e.g. \( f \ldots (f_1 \ldots \text{lhs} \ldots) \ldots \equiv f \ldots (f_1 \ldots \text{rhs} \ldots) \ldots \) – a pragmatic decision, but also one that has turned out well so far. We have not encountered a situation where this narrowing of test cases has missed some bug, except in very artificial examples manufactured just for that purpose. The intuitive reason seems to be that by doing all tests \( f \ldots \text{lhs} \ldots \equiv f \ldots \text{rhs} \ldots \), which of course also includes the tests \( f_1 \ldots \text{lhs} \ldots \equiv f_1 \ldots \text{rhs} \ldots \), for varying \( \text{lhs/rhs} \) as obtained from all axioms, one casts a fine enough net – situations where these all go through, along with all standard instantiations \( \text{lhs} \equiv \text{rhs} \) obtained from the axioms, but where \( f \ldots (f_1 \ldots \text{lhs} \ldots) \ldots \equiv f \ldots (f_1 \ldots \text{rhs} \ldots) \ldots \) would fail, appear to be extremely rare.

To summarize, we propose to test operation invariance via properties of the following form, for each combination of one operation, one argument position, and one axiom:

\[
f x_1 \ldots (\text{axiom}_{\text{lhs}} y_1 \ldots y_m) \ldots x_n \equiv f x_1 \ldots (\text{axiom}_{\text{rhs}} y_1 \ldots y_m) \ldots x_n
\]

5 The consideration of binary operations here is without loss of generality. Dealing with unary operations is simpler, and dealing with operations of higher arity than two is analogous to dealing with binary operations, just requires more index fiddling in the discussion.

6 Note that while the operation invariance property of \( \equiv \) is still under test, we were willing to simply take for granted that \( \equiv \) is at least an equivalence relation. See also Footnote 2.
where like the $x_i$ (alluded to as the “other” values above), the $y_j$ – which are meant to fill any variable positions in a symbolic axiom $lhs = rhs$ – can be simply values. No need for symbolic terms, since neither in the $x_i$ nor in the $y_j$ positions we need to assume any further possible axiom rewrites. Also, no need to generate terms/values in an implementation-dependent way, since the signature and axiom information from the ADT specification suffices. The discussion in the paragraphs preceding this one implies that the proposed approach is sound (if a counterexample is found, then there is a genuine problem with the ADT implementation under test), but not in general complete (even if the axioms and our subset of operation invariance properties are tested exhaustively, some bug in the ADT implementation might be missed – due to our pragmatic decision not to consider deeper nested rewrites like $f x_1 \ldots (f_1 y_1 \ldots (axiom_{lhs} z_1 \ldots z_l) \ldots y_m) \ldots x_n == f x_1 \ldots (f_1 y_1 \ldots (axiom_{rhs} z_1 \ldots z_l) \ldots y_m) \ldots x_n$).

The full set of tests of the proposed form for integer queues follows. An index $enqueue_1$ or $enqueue_2$ indicates that the first or second argument position of $enqueue$ is filled by the particular axiom, respectively. Parameters with a prime (i.e. $q'$ and $x'$) fill the other arguments of the tested operation as opposed to being variables of the relevant axiom.

\[
\begin{align*}
\text{enqueue}_1 q_3 &= \text{property} \ (\lambda x \ q' \rightarrow \text{let} \ \text{lhs} = \text{front} \ (\text{enqueue} \ x \ \text{empty}) \\
& \quad \text{rhs} = x \\
& \quad \text{in} \ \text{enqueue} \ \text{lhs} \ q' == \text{enqueue} \ \text{rhs} \ q') \\
\text{enqueue}_1 q_4 &= \text{property} \ (\lambda x \ q' \rightarrow \text{isEmpty} \ q == \text{False} \\
& \quad \text{let} \ \text{lhs} = \text{front} \ (\text{enqueue} \ x \ q) \\
& \quad \text{rhs} = \text{front} \ q \\
& \quad \text{in} \ \text{enqueue} \ \text{lhs} \ q' == \text{enqueue} \ \text{rhs} \ q') \\
\text{enqueue}_2 q_5 &= \text{property} \ (\lambda x \ x' \rightarrow \text{let} \ \text{lhs} = \text{dequeue} \ (\text{enqueue} \ x \ \text{empty}) \\
& \quad \text{rhs} = \text{empty} \\
& \quad \text{in} \ \text{enqueue} \ x' \ \text{lhs} == \text{enqueue} \ x' \ \text{rhs}) \\
\text{isEmpty}_5 &= \text{property} \ (\lambda x \rightarrow \text{let} \ \text{lhs} = \text{dequeue} \ (\text{enqueue} \ x \ \text{empty}) \\
& \quad \text{rhs} = \text{empty} \\
& \quad \text{in} \ \text{isEmpty} \ \text{lhs} == \text{isEmpty} \ \text{rhs}) \\
\text{enqueue}_2 q_6 &= \text{property} \ (\lambda x \ q' \rightarrow \text{isEmpty} \ q == \text{False} \\
& \quad \text{let} \ \text{lhs} = \text{dequeue} \ (\text{enqueue} \ x \ q) \\
& \quad \text{rhs} = \text{enqueue} \ x \ (\text{dequeue} \ q) \\
& \quad \text{in} \ \text{enqueue} \ x' \ \text{lhs} == \text{enqueue} \ x' \ \text{rhs}) \\
\text{isEmpty}_6 &= \text{property} \ (\lambda x \ q \rightarrow \text{isEmpty} \ q == \text{False} \\
& \quad \text{let} \ \text{lhs} = \text{dequeue} \ (\text{enqueue} \ x \ q) \\
& \quad \text{rhs} = \text{enqueue} \ x \ (\text{dequeue} \ q) \\
& \quad \text{in} \ \text{isEmpty} \ \text{lhs} == \text{isEmpty} \ \text{rhs}) \\
\text{dequeue}_6 &= \text{property} \ (\lambda x \ q \rightarrow \text{isEmpty} \ q == \text{False} \\
& \quad \text{let} \ \text{lhs} = \text{dequeue} \ (\text{enqueue} \ x \ q) \\
& \quad \text{rhs} = \text{enqueue} \ x \ (\text{dequeue} \ q) \\
& \quad \text{in} \ \text{isEmpty} \ \text{lhs} == \text{isEmpty} \ \text{rhs})
\end{align*}
\]
\[
\text{let } lhs = \text{dequeue (enqueue x q)} \\
\quad rhs = \text{enqueue x (dequeue q)} \\
\text{in not (isEmpty lhs) } \implies \\
\quad \text{dequeue lhs == dequeue rhs)
\]

\[
\text{front\_q}_6 = \text{property } (\lambda x q \rightarrow \text{isEmpty q \implies False)} \\
\quad \implies \\
\quad \text{let } lhs = \text{dequeue (enqueue x q)} \\
\quad \quad rhs = \text{enqueue x (dequeue q)} \\
\quad \text{in not (isEmpty lhs) } \implies \\
\quad \quad \text{front lhs == front rhs)
\]

There are two additional tests that syntactically belong to the others, but do not really add anything:

\[
\text{dequeue\_q}_5 = \text{property } (\lambda x \rightarrow \text{let } lhs = \text{dequeue (enqueue x empty)} \\
\quad rhs = \text{empty} \\
\quad \text{in not (isEmpty lhs) } \implies \\
\quad \quad \text{dequeue lhs == dequeue rhs)
\]

\[
\text{front\_q}_5 = \text{property } (\lambda x \rightarrow \text{let } lhs = \text{dequeue (enqueue x empty)} \\
\quad rhs = \text{empty} \\
\quad \text{in not (isEmpty lhs) } \implies \\
\quad \quad \text{front lhs == front rhs)
\]

This is because both sides of \(Q_5\) are empty queues, but neither dequeue nor front work on those.

5 A Practical Implementation

The tests shown in the previous section can (almost) strictly syntactically be derived from the specification. For every operation, every argument the operation has, and every axiom, one test can be obtained – by applying the operation at the selected argument to the axiom if the types allow it. In addition, constraints may have to be added per operation to prevent their application to invalid values, like the constraints not (isEmpty lhs) in dequeue\_q_6 and front\_q_6 (and in dequeue\_q_5 and front\_q_5).

A tool or library that generates these tests automatically needs type information about both operations and axioms. Details about the implementation of the datatype and operations, or the specific terms in the axioms, are not necessary. As relatively arbitrarily typed code must be generated, it seems to be at least very tricky to do this with plain Haskell and without a lot of manual help. Thus, to automate our solution as much as possible, we used Template Haskell \[11\]. As a result, none of the shown tests need to be hand-written by the user.

As a case study, we demonstrate here different ways how our tool/library (in a new module Test.OITestGenerator) can be used to generate appropriate QuickCheck tests. First, the axioms have to be specified. For this, OITestGenerator exports a datatype AxiomResult a, its constructor =!=, and a restricting conditional operator \(\Rightarrow\) that works

\[7\] which is written as \(\implies\)
akin to \(\Rightarrow\) as known from QuickCheck. Axioms with variables are written as functions with corresponding arguments, returning an \texttt{AxiomResult} \(a\) where \(a\) is the codomain of the axiom’s left- and right-hand side.

\[
q_1 :: \texttt{AxiomResult Bool} \\
q_1 = \texttt{isEmpty} \; \texttt{empty} =!\; \texttt{True}
\]

\[
q_2 :: \texttt{Int} \rightarrow \texttt{Queue} \rightarrow \texttt{AxiomResult Bool} \\
q_2 = \lambda x q \rightarrow \texttt{isEmpty} \; (\texttt{enqueue} \; x \; q) =!\; \texttt{False}
\]

\[
q_3 :: \texttt{Int} \rightarrow \texttt{Queue} \rightarrow \texttt{AxiomResult Int} \\
q_3 = \lambda x \rightarrow \texttt{front} \; (\texttt{enqueue} \; x \; \texttt{empty}) =!\; x
\]

\[
q_4 :: \texttt{Int} \rightarrow \texttt{Queue} \rightarrow \texttt{AxiomResult Int} \\
q_4 = \lambda x q \rightarrow \texttt{not} \; (\texttt{isEmpty} \; q) \; \Rightarrow \; \texttt{front} \; (\texttt{enqueue} \; x \; q) =!\; \texttt{front} \; q
\]

\[
q_5 :: \texttt{Int} \rightarrow \texttt{Queue} \rightarrow \texttt{AxiomResult Queue} \\
q_5 = \lambda x \rightarrow \texttt{dequeue} \; (\texttt{enqueue} \; x \; \texttt{empty}) =!\; \texttt{empty}
\]

\[
q_6 :: \texttt{Int} \rightarrow \texttt{Queue} \rightarrow \texttt{AxiomResult Queue} \\
q_6 = \lambda x q \rightarrow \texttt{not} \; (\texttt{isEmpty} \; q) \; \Rightarrow \; \texttt{dequeue} \; (\texttt{enqueue} \; x \; q) =!\; \texttt{enqueue} \; x \; (\texttt{dequeue} \; q)
\]

This is already enough preparation to generate the basic tests, i.e., direct translations of the axioms, with the provided function \texttt{generate \_ basic \_ tests}. It gets one argument, a list of Axioms. \texttt{Axiom} is a container holding a) the name of an axiom, which (the axiom) must be a function returning an \texttt{AxiomResult} \(a\), and b) possibly custom generators for the function’s arguments. It has one constructor function \texttt{axiom}, which takes a \texttt{Name} – custom generators can be assigned to an \texttt{Axiom} via \texttt{withGens}, which is explained later. Then \texttt{generate \_ basic \_ tests} returns an expression of type \texttt{[Property]}.

Property is the type of QuickCheck tests, as returned by the function \texttt{property} in the earlier given versions of \(q_1\)–\(q_6\).

In using \texttt{generate \_ basic \_ tests} to generate the basic tests from the above functions returning \texttt{AxiomResult}\_s, two of Template Haskell’s syntactic constructs will be needed: The first are \texttt{splices}. A splice is written \$\{(\ldots)\}, where \ldots is an expression. A splice may occur instead of an expression. The splice will be evaluated at compile time and the syntax tree returned by it will be inserted in its place. The second construct used is ‘\ldots, where \ldots is a name of a function variable or data constructor. Then ‘\ldots is of type \texttt{Name} and its value represents the name of the \ldots that was quoted. Using these constructs, we can write:

\[
\texttt{adt \_ basic \_ tests :: [Property]} \\
\texttt{adt \_ basic \_ tests = \$ (let axs = \texttt{map \ axiom} [’q_1’, ’q_2’, ’q_3’, ’q_4’, ’q_5’, ’q_6’] \\
\texttt{in \ generate \_ basic \_ tests axs)}
\]

Now, \texttt{adt \_ basic \_ tests} can be executed via \texttt{mapM \_ quickCheck adt \_ basic \_ tests}.

Before the, for our purposes more interesting, operation invariance tests can be generated, constraints for \texttt{dequeue} and \texttt{front} must be specified. Such a constraint function has the purpose of deciding whether a set of arguments is valid for the respective operation. Thus it takes the same arguments, but returns a \texttt{Bool} independently of the operation’s return type.
may dequeue :: Queue → Bool
may dequeue = not ◦ isEmpty

may front :: Queue → Bool
may front = not ◦ isEmpty

Given these, the operation invariance tests can be generated by the provided function
\texttt{generate\_oi\_tests}. For operations there exists a datatype \texttt{Op} similar to \texttt{Axiom}, with one
constructor function \texttt{op}. The function \texttt{withConstraint} may be used to add a constraint to
an operation.\footnote{As opposed to constraints for axioms, which are specified using ⇛ in the function whose name
is passed to \texttt{axiom}.} It may also be used multiple times on the same operation, in which case
the constraints are connected with a logical “and”.

\[
\texttt{adt\_oi\_tests} :: \text{[Property]}
\]

\[
\texttt{adt\_oi\_tests} = \$ (\texttt{let \_ops = [op 'empty
, \_op 'enqueue
, \_op 'isEmpty
, \_withConstraint \(\_op 'dequeue\) \_may\_dequeue
, \_withConstraint \(\_op 'front\) \_may\_front\]
\_axs = \text{map axiom ['q1, 'q2, 'q3, 'q4, 'q5, 'q6]}
\text{in generate\_oi\_tests \_axs \_ops})
\]

Note that the repeated local definition of \texttt{axs} (in both \texttt{adt\_basic\_tests} and \texttt{adt\_oi\_tests})
is necessary due to Template Haskell’s stage restrictions. It is not possible to refer to a
top-level declaration in the same file, because it is still being compiled when the splices
are executed. Note also that \texttt{empty} could be omitted here from the list of operations as it
takes no arguments.

Running the tests automatically generated above is enough to detect the buggy
implementation of \texttt{front}!

+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q3).
+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q4).
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q5).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q5).
*** Gave up! Passed only 0 tests.
*** Gave up! Passed only 0 tests.
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q6).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q6).
+++ OK, passed 100 tests (100% Queue.dequeue@1/Main.q6).
*** Failed! Falsifiable (after 5 tests):
3
BQ [4] [4, −3]

The test that fails here is \texttt{front\_q6} (which would have appeared in the output as
\texttt{Queue.front@1/Main.q6}). Note that two tests were generated that are correctly typed
but have no valid input (as already observed further above when writing down properties by hand); namely \texttt{dequeue\_q5} and \texttt{front\_q5} alias \texttt{Queue.dequeue@1/Main.q5} and
Queue.front@1/Main.q5. They could be avoided by using other functions exported by OITestGenerator and explained below.

Generators can be passed to an operation via withGens in a list, with one generator name for each argument, as in withGens (op `enqueue) [‘arbitrary,’arbitrary]. It is not allowed to omit generators when using withGens; instead arbitrary must be passed if no special generator should be used for some position. The function withGens can be intermingled with withConstraint and may also be used on Axioms.

Also, there is a convenience function generate axiom’s tests which takes only one Axiom and a list of Ops. It is useful when certain combinations of axioms and operations should be excluded. It can also be used when only specific argument positions of an operation should be excluded for an axiom. The function but :: Op → Arg → Op, when called as o ‘but’ i, excludes the ith argument from o when generating tests. Arg has a single constructor function arg :: Int → Arg. The function but may be called multiple times on the same operation to exclude multiple arguments. To supplement it, there also is only :: Op → Arg → Op to include only one argument. For illustration:

\[
\begin{align*}
all.q5 :: [Property] \\
all.q5 = \$\{&\text{let ops} = [\op\ `\text{empty} \\
&\quad \op\ `\text{enqueue} \\
&\quad \op\ `\text{isEmpty} \\
&\quad \op\ `\text{dequeue} \ ‘\text{but}‘ \ arg \ 1 \\
&\quad \op\ `\text{front} \ ‘\text{but}‘ \ arg \ 1 \} \\
\quad \text{in generate axiom's tests (axiom 'q5) ops}
\end{align*}
\]

Of course, in this case, dequeue and front could simply be omitted completely as they do only have one argument.

Another convenience function is generate single test, which again works similarly to generate all tests, but takes only one Axiom and one Op instead of lists, and generates only a single test. It may be used when more control is needed.

\[
\begin{align*}
enqueue_{1}.q3 &= \$\{\text{generate single test (axiom 'q3) (op 'enqueue 'only' 1)}\} \\
enqueue_{1}.q4 &= \$\{\text{generate single test (axiom 'q4) (op 'enqueue 'only' 1)}\} \\
enqueue_{2}.q5 &= \$\{\text{generate single test (axiom 'q5) (op 'enqueue 'only' 2)}\} \\
isEmpty_{1}.q5 &= \$\{\text{generate single test (axiom 'q5) (op 'isEmpty 'only' 1)}\} \\
enqueue_{2}.q6 &= \$\{\text{generate single test (axiom 'q6) (op 'enqueue 'only' 2)}\} \\
isEmpty_{1}.q6 &= \$\{\text{generate single test (axiom 'q6) (op 'isEmpty 'only' 1)}\} \\
dequeue_{1}.q6 &= \$\{\text{generate single test (axiom 'q6) (op 'dequeue 'only' 1)}\} \\
front_{1}.q6 &= \$\{\text{generate single test (axiom 'q6) (op 'front 'only' 1)}\}
\end{align*}
\]

No constraints are passed here because the superfluous tests dequeue_{1}.q5 and front_{1}.q5 were purposefully omitted, and because no axiom but Q_5 can result in an empty queue.

As writing such a list of tests is cumbersome, there is a function show all tests :: Maybe (String → Int → String → String) → [Name] → [Name] → ExpQ which takes an optional formatting function, a list of axiom names, and a list of operation names, and produces a String-typed expression whose content is exactly the code above (plus dequeue_{1}.q5 and front_{1}.q5, which were removed manually from the output).
single_test_str = 
(\let ops = \[
\text{\textquote{"empty","enqueue","isEmpty","dequeue","front"}}
\]
\text{\textquote{\textquote{\textquote{q1,q2,q3,q4,q5,q6}}}}
\text{\textquote{(in show_all_tests Nothing axs ops)}}
)

If constraints have to be added to the generated code, they must be added manually. On
the other hand, all ‘only’ n could be omitted in the case above, since there is no operation
with two arguments of the same type.\footnote{The function generate_single_test throws a compile time error unless there is exactly one way
to combine the given Axiom and Op.} Instead of Nothing above, a custom formatting
function can be passed that generates the names of the properties.

The implementation is available as a Cabal package at http://hackage.haskell.
org/package/qc-oi-testgenerator. Insights into the implementation, as well as
more discussion of formal aspects of the overall approach, can be found in the first
author’s diploma thesis [7].

6 Conclusion and Discussion

We have presented and implemented an approach for automatically checking operation in-
variance for Haskell implementations of abstract datatypes. The user writes down axioms
by closely following standard QuickCheck syntax, and both the ordinary QuickCheck
tests as well as additional operation invariance tests are derived from that.

It might have been more desirable to obtain the necessary information about axioms
directly from existing QuickCheck tests, freeing the user from possibly having to rewrite
them. For this it would be necessary, in the implementation, to obtain the syntax tree of
a given Haskell declaration and find the axiom by looking for a call to ==. Then, type
information for the left- and right-hand side would be needed. As of today, neither is
possible with Template Haskell: The Info data structure returned by reify has a field of
type Maybe Dec to hold the syntax tree of the right-hand side of a declaration. However,
according to the documentation of Template Haskell’s most current version 2.9.0.0,
there is no implementation for this and the field always contains Nothing. Another way
to obtain the syntax tree would be to write the axioms in expression quotations. In
both cases, though, Template Haskell offers no way of obtaining the necessary type
information, as those can only be retrieved for Names using reify, but not for arbitrary
syntax trees. Also due to Template Haskell not offering a way of calling the type checker
or other convenient ways to help generate correctly typed code yet, the implementation
does not currently support polymorphic types. A workaround making it possible to test
polymorphic abstract datatypes is to construct a (suitably chosen, cf. [2]) monomorphic
instance and rename all participating functions. That way, reify returns the monomorphic
types.

On the conceptual side, it would be attractive to gain more insight into how effective
the kind of tests we generate are in finding bugs in general. This might be achieved by a
formalization of our approach and/or collecting experimental evidence from more case
studies. Here we discuss only a small variation on the running example, which illustrates
an interesting aspect concerning the implementation of equality:
The error in the shown version of \textit{front} is hidden from the basic tests only because the implemented equality compares two values by how the implementer thinks they \textit{should} behave. An observational equality like the following one, which even does not touch the internals, would not so be fooled.

\[
q == q' \mid \text{isEmpty } q /= \text{isEmpty } q' = \text{False} \\
\text{isEmpty } q = \text{True} \\
\text{otherwise} = \text{front } q == \text{front } q' \land \text{dequeue } q == \text{dequeue } q'
\]

Still, the basic tests will not suffice in general. As a very artificial example, consider the queue implementation used so far, but now without the error in \textit{front}, and replace the following two functions:

\[
bq f r = \text{BQ} (f ++ \text{reverse } r) []
\]
\[
enqueue x q@(\text{BQ } f r) \mid \text{isEmpty } q = bq f (r ++ [x]) \\
\text{otherwise} = \text{BQ} f (r ++ [x])
\]

This error will never be found with the basic tests and using the above observational equality. So using operation invariance tests is still a good idea.

\textbf{Acknowledgments}

We thank the anonymous reviewers for their comments and suggestions.

\textbf{References}

Partitioning 0-CFA for the GPU

Thomas Gilray, James King, Matthew Might

University of Utah
{tgilray, jsking2, might}@cs.utah.edu

Abstract. We generalize EigenCFA, a linear formulation of 0-CFA, to features needed for real-world intermediate languages and apply our approach to an analysis of Scheme. EigenCFA lacks the full precision of 0-CFA and is restricted to programs in the subset of pure λ-calculus conforming to binary continuation-passing-style. This constraint is borne of the need to encode a monolithic transfer function as a GPU kernel free of thread-divergence. We show the soundness and precision of partitioning this transfer function to obtain an encoding with both the fine-grained parallelism of matrix operations as well as coarse-grained parallelism between transfer functions. Our approach supports real-world languages with a diverse array of forms and has the full precision of a traditional 0-CFA. An implementation effort is in progress and preliminary results are promising. Both CPU and GPU versions of our encoding have been tested for correctness against a standard worklist implementation.

1 Introduction

The aim of static analysis is to make guarantees about the behavior of a program before runtime. For example, if we wanted to know the possible types for a variable in a dynamic language (type-recovery), or if we wanted to know where sensitive data might exit a program (taint-propagation), we are asking for a data-flow analysis. If we wanted to remove code which can never be run (dead-code elimination) or determine which code is likely to be run most frequently (static profiling), we are asking for a control-flow analysis of our program. The goal of these analyses is to give a computable approximation of the propagation of data or control through a program which is precise enough to answer useful questions for the purposes of compiler optimization, bug-finding, or security, among other applications.

A flow-analysis allows us to answer a question like: “What are the possible return values of the following Scheme snippet?”

\[
\text{(let ([add5 (lambda (a) (+ a 5))])}
\text{(add5 10))}
\]

In this particular case, the code is guaranteed to halt with a specific answer, and so it may simply be executed. In the general case however, static analysis brings us up against the halting problem and fundamental limitations on computability. Rice’s theorem shows that proving any non-trivial program property
is incomputable in the general case. The challenge of designing a static analysis is therefore the unavoidable trade off between precision and scalability. Effectively reformulating a flow-analysis so that parallelism may be easily exploited allows solutions to be found using throughput-oriented hardware and benefits the scalability of an analysis at no cost to precision.

EigenCFA is a preliminary attempt at doing just that [13]. It derives a rather clever encoding of 0-CFA (a specific control-flow analysis) as linear-algebraic operations which are efficient to compute on SIMD architectures like the GPU. Unfortunately, it is restricted by its need to encode the entire analysis as a single GPU kernel. This restriction has made it impossible to encode an analysis of simple language features like primitive operations and conditions. The * operation in our code snippet places even such a simple example outside the ability of EigenCFA to handle efficiently. Our approach solves this problem with a technique we call transfer function partitioning, allowing the essential encoding to be extended to real-world analyses. As 0-CFA is nearly cubic in complexity and impractical to compute for large programs, such parallelism is of vital importance in bringing sophisticated program-analysis to bear on everyday problems.

In addition, EigenCFA is less precise than it should be as it uses the trivially sound approximation for control-flow behavior. Consider the following sample taken from a larger program where f, g, and h are unreachable functions.

```scheme
(define (f x) (h x 0))
(define (g) (h 0 0))
```

Proving that these are in-fact unreachable requires a precise control-flow analysis. EigenCFA models only data-flow and assumes the reachability of all expressions in a program. Because of this, the callsites (h 0 0) and (h x 0) will be examined and their data-flows propagated. As x is unbound, even after handling propagations for (h x 0), the corresponding formal parameter for h will remain unbound. An unbound variable is an implicit indication that a function (in this case h) is in-fact unreachable. In this case, an approximation of data-flow has resulted in a bound on control-flow, but this cannot be relied on to give the same precision as a traditional 0-CFA which models control explicitly. Because the callsite in g applies h on two constant values, once the analysis is complete, it will appear that h may be reachable as possible values will have been found for both its parameters. Our linear encoding solves this problem by explicitly modeling both the control-flow and data-flow aspects of a program.

In the following sections, we will give the necessary background on static analysis by abstract interpretation, introduce a concrete and abstract semantics for a Scheme intermediate-representation, and derive a linear encoding for 0-CFA. Our encoding has been implemented both as a single-threaded CPU version and as single-stream and multi-stream GPU versions. All these have been tested for correctness against a standard worklist implementation and produce identical results for a suite of Scheme benchmarks. While the optimization effort is ongoing, preliminary results are promising, showing potential speedups of 20x or better over a single-threaded version of the encoding.
2 Background

Abstract interpretation is a very general framework for static analysis which allows us to perform an approximate evaluation of a program [2] [3]. The result is a single, time-bounded execution which represents the set of all possible exact executions. The approximation used, formally known as a Galois-connection, is a precisely defined relationship between a concrete semantics and an abstract semantics. In the code snippet on the front page for example, approximating concrete integers as \( \text{INT} \) could instantiate the framework to perform a type-recovery and we would determine the result to be at least some integer. If instead they were approximated as one of \( \text{POS, ZERO, or NEG} \), we could perform a sign-analysis and discover that the result can only be a positive integer more specifically.

While there are a number of common formulations, our presentation uses a small-step operational semantics defined over the configurations of an abstract-machine. Such a semantics is determined by a series of inference rules which define, given a machine configuration, what configurations may immediately succeed it. A static analysis in this style proceeds by evaluation of an abstract abstract-machine starting with an initial configuration and exploring all abstract states reachable by the semantic rules. The simulation may be called sound if the defined Galois-connection strictly bounds the concrete executions represented by an abstract execution (for example, over-approximation or under-approximation). This property is proved inductively by showing that soundness before state-transition implies soundness after state-transition in all cases [10].

In Shivers’ seminal work on control-flow analysis (CFA) of higher-order languages, he introduced a hierarchy of increasingly precise analyses for Scheme known as \( k \)-CFA [14]. The most basic of these, 0-CFA, has been extended and improved in a variety of ways since then and may be considered the foundational analysis of functional languages by abstract interpretation. For \( k > 0 \) the analysis is known to be EXPTIME-complete [5] and therefore intractable, but 0-CFA and many of its related algorithms are merely of a large polynomial complexity. Specifically, 0-CFA is in \( O\left(\frac{n^3}{\log(n)}\right) \), but remains stubbornly difficult to compute in practice.

2.1 GPU programming

GPUs have become increasingly popular in recent years for solving computationally intense problems outside of graphics processing [11]. The FLOP throughput on GPUs has continued to increase exponentially, significantly outpacing CPUs. Many high performance computers now rely on the GPU as a work-horse for the computationally demanding linear-algebra which is often needed in large-scale scientific and engineering applications. GPUs achieve this performance due to the use of a streaming SIMD (single-instruction/multiple-data) architecture. This allows a group of small lightweight cores to perform the same operation on a vector of data and is ideal for certain tasks such as graphics and linear-algebra. With the addition of soldered memory, GPUs are also able to attain a significantly higher memory bandwidth than that of traditional CPUs.
Modern GPUs have thousands of cores and can achieve parallelism on multiple levels. The first is fine-grained SIMD parallelism which relies on many threads operating in lock-step. The second is parallelism from concurrent program execution. Thread-division occurs when a branch instruction separates a group of threads into different control-flow paths. This causes threads on one path to continue while others wait to run sequentially and is disastrous for performance.

2.2 EigenCFA

EigenCFA is a linear encoding of 0-CFA [13] and the primary inspiration for our approach. This formulation of 0-CFA allows the flow-analysis to be computed on a GPU; however, it is constrained to a very simple language without support for fundamental and important features like mutation and conditions. This extreme simplicity is required so the entire analysis can be implemented as a single GPU kernel free of thread-divergence. Prior to analysis, programs must be reduced to a subset of the pure λ-calculus conforming to binary continuation-passing-style (binary CPS). This restricts programs to values ranging over closures that accept precisely two arguments and never return. CPS itself is a very practical intermediate-representation but currying all functions may not be desirable, and church-encoding basic values is problematic as it obfuscates the data under analysis. To make matters worse, this encoding requires a variety of standard language forms like letrec, set!, and if, to be desugared into direct lambda application in such a way that the program’s behavior according to 0-CFA is unaffected, even if its concrete execution is no longer sound. This technique, termed abstract church encoding, is another highly destructive transformation which makes it difficult to map the results of an analysis back onto the original program. A final shortcoming is a lack of precision in modeling control-flow behavior. 0-CFA necessarily models both control-flow and data-flow as the nature of higher-order programming languages entangles these two concerns. EigenCFA however, seems ironically named as it assumes the reachability of all callsites in the program, giving the trivially sound result for its control-flow approximation.

2.3 Points-to analysis

The other recent attempt to bring flow-analysis to the GPU implements an inclusion-based points-to analysis [9]. This formulation operates on the adjacency matrix of a points-to graph and is manageable for some real-world language features, but lacks generality. Like EigenCFA, the strategy is constrained to monovariance and supports a restricted language. Unlike EigenCFA, this approach is not readily extensible to analysis of higher-order functions and their environments, or to more general program analyses with richer abstract domains.

3 Concrete Semantics for CPS Scheme

The target language for our analysis is a desugared Scheme in continuation-passing-style. CPS constrains function calls never to return; instead, a caller
must explicitly pass a continuation forward to be invoked on the result [12]. CPS is an excellent and widely used language for compiler optimization and program analysis [1]. If the transformation to CPS makes note of which lambdas correspond to continuations, the simplified program may again, along with any optimizations and analysis results, be precisely reconstituted in its direct-style form. This means the advantages of CPS can be utilized without compromise or loss of information [8].

The grammar for this language structurally distinguishes between atomic-expressions \( ae \) and complex-expressions \( e \).

\[
\begin{align*}
e & \in E ::= (ae \, ae \, \ldots)^l \\
& \quad | \ (set! \, x \, ae \, ae)^l \\
& \quad | \ (prim \, op \, ae \, \ldots)^l \\
& \quad | \ (if \, ae \, e \, e)^l \\
& \quad | \ (halt)^l
\end{align*}
\]

\[
\begin{align*}
ae & \in AE ::= c \mid x \mid lam
\\
lam & \in Lam ::= (\lambda \ (x \ldots) \ e)
\\
c & \in Const ::= \#t \mid \#f \mid \langle \text{number} \rangle \mid \ldots
\\
x & \in Var ::= \langle \text{set of program variables} \rangle
\\
op & \in OP ::= \langle \text{set of primitive operations} \rangle
\\
l & \in Label ::= \langle \text{set of unique labels} \rangle
\end{align*}
\]

To specify the behavior of this language, we define a small-step operational semantics for an abstract-machine. A transition relation \( \Rightarrow \) is needed which defines at most one successor for any valid machine state \( \varsigma \). We use a CES-style machine with control-expression \( e \), binding-environment \( \rho \), value-store \( \sigma \), and timestamp \( t \) (execution context) components [4]. The binding-environment \( \rho \) maps variables in scope to an address. The value-store \( \sigma \) maps addresses \( a \) to values \( v \). Timestamps are unbounded lists of labels representing a complete history of program execution.

\[
\varsigma \in \Sigma = E \times Env \times Store \times Time
\]
\[
\rho \in Env = Var \rightarrow Addr
\]
\[
\sigma \in Store = Addr \rightarrow Value
\]
\[
t \in Time = Label^*
\]
\[
a \in Addr = Var \times Time
\]
\[
v \in Value = Lam \times Env + Const
\]

For evaluating atomic-expressions we define an auxiliary function \( A \) which maps a syntactic \( ae \) in the context of a current state to a semantic program value. In the case of a variable, \( A \) uses \( \rho \) and \( \sigma \) to lookup the variable’s current binding
In the store. In the case of a lambda, a value is produced by closure-conversion:
apairing the lambda with its current environment.

\[ \mathcal{A}: \text{AE} \times \Sigma \rightarrow \text{Value} \]
\[ \mathcal{A}(x, (e, \rho, \sigma, t)) = \sigma(\rho(x)) \]
\[ \mathcal{A}(\text{lam}, (e, \rho, \sigma, t)) = (\text{lam}, \rho) \]
\[ \mathcal{A}(c, \varsigma) = c \]

In addition, we need a primitive-operation evaluator \( \delta \) which maps an operation \( op \) and list of values \( v \) to a result.

\[ \delta: \text{OP} \times \text{Value}^* \rightarrow \text{Value} \]

We may now define the small-step transition relation \( \Rightarrow \) by pattern-matching against the four complex-expressions that need to be handled: conditionals, mutation, primitive operations, and callsites. The (halt) expression does not need to be handled as it has no successors.

An inference rule like the one below asserts that the conclusion below the line is true whenever the premise above is true. When a callsite is reached, control moves inside the body \( e \) of the invoked closure.

\[
\begin{align*}
(\lambda (x_1 \ldots x_j) e), \rho_\lambda &\Rightarrow \mathcal{A}(ae_f, \varsigma) \\
((ae_f ae_1 \ldots ae_j)^l, \rho, \sigma, t) &\Rightarrow (e, \rho', \sigma', t') \\
\text{where} &\quad \rho' = \rho_\lambda[x_i \mapsto (x_i, t')] \\
&\quad \sigma' = \sigma[(x_i, t') \mapsto \mathcal{A}(ae_i, \varsigma)] \\
&\quad t' = l: t
\end{align*}
\]

A new environment \( \rho' \) is produced from the closure’s \( \rho_\lambda \) augmented with bindings for each formal parameter \( x_i \). An updated store is produced by mapping these addresses for each \( x_i \) to the value indicated by atomic-evaluation of \( ae_i \). As the timestamp \( t' \) is always a complete and unique history of program execution, all transitions use a fresh set of addresses.

Mutation is handled similarly as it implies the invocation of a continuation. The continuation indicated for \( ae_k \) receives \text{VOID}, the return value of a \text{set!} expression in Scheme. In addition, we update the current address in scope for \( x \) to be the current value for \( ae_v \).

\[
\begin{align*}
(\lambda (x_k) e), \rho_\lambda &\Rightarrow \mathcal{A}(ae_k, \varsigma) \\
(set! x ae_v ae_k)^l, \rho, \sigma, t &\Rightarrow (e, \rho', \sigma', t') \\
\text{where} &\quad \rho' = \rho_\lambda[x_k \mapsto (x_k, t')] \\
&\quad \sigma' = \sigma[(x_k, t') \mapsto \text{VOID}] \\
&\quad [\rho(x) \mapsto \mathcal{A}(ae_v, \varsigma)] \\
&\quad t' = l: t
\end{align*}
\]
Primitive operations use $\delta$ to obtain a return value and propagate this $v_k$ to the prim-op’s continuation.

$$
((\lambda (x_k) e), \rho_{\lambda}) = A(ae_k, \varsigma) \\
((\text{prim op } ae_1 \ldots ae_j ae_k)^t, \rho, \sigma, t) \Rightarrow (e, \rho', \sigma', t')
$$

where $\rho' = \rho_{\lambda}[x_k \mapsto (x_k, t')]$

$\sigma' = \sigma[(x_k, t') \mapsto v_k]$

$v_k = \delta(\text{op}, (A(ae_1, \varsigma) \ldots A(ae_j, \varsigma)))$

$t' = l; t$

Conditionals are probably the simplest case, with control moving inside the true branch or false branch as appropriate.

$$
A(ae, \varsigma) = \text{FALSE} \\
((\text{if } ae \ e_t \ e_f), \rho, \sigma, t) \Rightarrow (e_t, \rho, \sigma, t)
$$

$$
A(ae, \varsigma) \neq \text{FALSE} \\
((\text{if } ae \ e_t \ e_f), \rho, \sigma, t) \Rightarrow (e_f, \rho, \sigma, t)
$$

### 3.1 Evaluating a program

To evaluate a program $e$ with these concrete semantics we produce a starting configuration $\varsigma_0 = I(e)$ using a concrete state-space injection function $I : E \rightarrow \Sigma$:

$$
I(e) = (e, \bot, \bot, ())
$$

We can then compute the transitive closure of $(\Rightarrow)$ starting from $\varsigma_0$. As our state-space is unbounded, and the interpretation may continue to produce new states indefinitely, concrete executions are incomputable in the general case.

### 4 Abstract Semantics for 0-CFA

We perform a structural abstraction bounding the machine’s address-space to obtain a computable approximation of our concrete semantics [6][7]. Notice that our abstract semantics contains several fundamental changes from its concrete counterpart. 0-CFA bounds the address-space to include exactly one address for each variable (monovariance). All values bound to a variable $x$ in any context therefore must be represented by a single address. This introduces merging between values in our store and non-determinism in the transition relation.

To define an abstract operational semantics, we again need an abstract machine and a transition relation ($\Rightarrow$) which matches up successors and predecessors within the machine’s configuration-space. As we are effectively re-using an
empty timestamp for every allocation, expressions will uniquely identify an environment mapping free-variables to themselves and the store may directly map variables \( \hat{x} \) to sets of abstract values \( \hat{v} \). Such a flow-set may indicate a range of possible concrete values for an address. Closures are now just lambdas.

Program constants map to their corresponding basic values. When performing a concrete interpretation, these values are precise. When performing our abstract interpretation, there should only be a finite number of abstract basic values so they can be enumerated in our forthcoming encoding. For constant propagation, a set of program locations may be used. We use the notation \( \alpha \) below to informally indicate the abstraction function a fully defined Galois-connection would employ to map a concrete machine component to its most precise abstract representative. For example, \( \alpha(-3) \) could yield \texttt{NEG}.

The abstract atomic-expression evaluator returns flow-sets \( \hat{v} \).

\[
\hat{A}: \mathcal{AE} \times \hat{\Sigma} \rightarrow \hat{Values}
\]

\[
\hat{A}(x, (e, \hat{\sigma})) = \hat{\sigma}(x)
\]

\[
\hat{A}(\text{lam}, (e, \hat{\sigma})) = \{\text{lam}\}
\]

\[
\hat{A}(c, \hat{\zeta}) = \{\alpha(c)\}
\]

We also need an abstract prim-op evaluator \( \hat{\delta} \) which maps a primitive operation \( op \) and list of flow-sets to a sound result. For example, \( \hat{\delta}(+, \{\text{POS}\}, \{\text{POS}\}) = \{\text{POS}\} \).

\[
\hat{\delta}: \mathcal{OP} \times \hat{Values}^* \rightarrow \hat{Values}
\]

A callsite has one successor for each closure that accepts a matching number of arguments indicated by the flow-set for \( ae_f \) (the atomic-expression in call position).

\[
(\lambda (x_1 \ldots x_j) e) \in \hat{A}(ae_f, \hat{\zeta})
\]

\[
((ae_f ae_1 \ldots ae_j), \hat{\sigma}) \Rightarrow (e, \hat{\sigma}')
\]

where \( \hat{\sigma}' = \hat{\sigma} \sqcup [x_i \mapsto \hat{A}(ae_i, \hat{\zeta})] \)

Control moves inside the body of all invoked closures \( e \). The updated store is now conservatively approximated by finding the least-upper-bound of the current store and each new binding. Stores are ordered point-wise by inclusion, i.e.

\[
(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(\hat{a}) = \hat{\sigma}_1(\hat{a}) \cup \hat{\sigma}_2(\hat{a}).
\]
Mutation is succeeded by a state for each possible continuation.

\[ (\lambda (x_k) \; e) \in \hat{A}(ae_k, \hat{\varsigma}) \]

\[ ((\text{set! } x \; aev \; ae_k), \hat{\sigma}) \approx (e, \hat{\sigma}') \]

where \( \hat{\sigma}' = \hat{\sigma} \sqcup [x_k \mapsto \{\text{VOID}\}] \sqcup [x \mapsto \hat{A}(ae_v, \hat{\varsigma})] \)

In addition, to conservatively simulate mutation of the variable \( x \), all flows indicated for \( ae_v \) are included along with all previous values.

\[ (\lambda (x_k) \; e) \in \hat{A}(ae_k, \hat{\varsigma}) \]

\[ ((\text{prim op } ae_1 \ldots ae_j \; ae_k), \hat{\sigma}) \approx (e, \hat{\sigma}') \]

where \( \hat{\sigma}' = \hat{\sigma} \sqcup [x_k \mapsto \hat{v}_k] \)

\[ \hat{v}_k = \delta(op, (\hat{A}(ae_1, \hat{\varsigma}) \ldots \hat{A}(ae_j, \hat{\varsigma}))) \]

Primitive operations use \( \hat{\delta} \) to obtain an approximation of the return value and propagate this flow-set \( \hat{v}_k \) to each continuation indicated for \( ae_k \).

\[ \hat{v} \in \hat{A}(ae, \hat{\varsigma}) \]

\[ (\text{if } ae \; et \; ef), \hat{\sigma}) \approx (et, \hat{\sigma}) \]

\[ \hat{v} \in \hat{A}(ae, \hat{\varsigma}) \]

\[ (\text{if } ae \; et \; ef), \hat{\sigma}) \approx (ef, \hat{\sigma}) \]

When a conditional is reached, both branches may be taken.

### 4.1 Naïvely computing the analysis

We first define an injection function \( \hat{I} \) which, given a program \( e \), determines an initial state \( \hat{\varsigma}_0 = \hat{I}(e) \).

\[ \hat{I} : E \rightarrow \hat{\Sigma} \]

\[ \hat{I}(e) = (e, \bot) \]

To compute our analysis, we can simply visit all states reachable from \( \hat{\varsigma}_0 \). We define a transfer function for the system-space of our program \( \hat{f} : \mathcal{P}(\hat{\Sigma}) \rightarrow \mathcal{P}(\hat{\Sigma}) \):

\[ \hat{f}(\hat{S}) = \{ \hat{\varsigma}' : \hat{\varsigma}' \in \hat{S} \text{ and } \hat{\varsigma} \approx \hat{\varsigma}' \} \cup \{ \hat{\varsigma}_0 \} \]

Unfortunately, this approach is impracticable as the total number of stores is exponential in the size of the program, even for this context-insensitive analysis.
4.2 Efficiently computing the analysis

A more efficient method uses a single store to replace the multitude of individual stores. This global store is maintained as the least-upper-bound of all stores seen so far. Global-store-widening is a sound, and in practice quite reasonable, approximation of the naive calculation \[10, 14\]. With this form of widening applied, 0-CFA is in \(O(\frac{n^3}{\log(n)})\). We factor the store out of our state-space while retaining a set of reachable expressions denoted \(\hat{r}\) as an explicit model of control-flow. EigenCFA compromises on precision by modeling only the store.

\[
\hat{r} \in \hat{\text{Reach}} = \mathcal{P}(E)
\]
\[
\hat{\xi} \in \hat{\Xi} = \hat{\text{Reach}} \times \hat{\text{Store}}
\]

Over factored system-spaces \(\hat{\Xi}\), the transfer function becomes:

\[
\hat{f} : \hat{\Xi} \rightarrow \hat{\Xi}
\]
\[
\hat{f}(\hat{r}, \hat{\sigma}) = (\hat{r} \cup \hat{r}', \hat{\sigma}')
\]

where \(\hat{S} = \{\zeta' : e \in \hat{r} \text{ and } (e, \hat{\sigma}) \approx \zeta'\}\)

\[
\hat{r}' = \{e : (e, \perp) \in \hat{S}\}
\]

\[
\hat{\sigma}' = \bigsqcup \{\hat{\sigma}'' : (\perp, \hat{\sigma}'') \in \hat{S}\}
\]

The notation \(\perp\) matches any value without binding it to a variable.

The store grows monotonically across transition, i.e. \((\perp, \hat{\sigma}) \approx (\perp, \hat{\sigma}')\) implies \(\hat{\sigma} \sqsubseteq \hat{\sigma}'\), so \(\hat{f}\) grows monotonically over \(\hat{\Xi}\). Because \(\hat{\Xi}\) is finite and \(\hat{f}\) is continuous, we know that the least-fix-point of \(\hat{f}\) is \(\hat{f}^n(\perp, \perp)\) for some finite \(n\).

5 Partitioning the transfer function

A central insight to our work is that we can partition a transfer function by reachable state under evaluation. By grouping these individual transfer functions into GPU kernels according to like control-flow we can minimize thread-divergence in a SIMD implementation. An individual transfer function \(\hat{f}_e\) handles only the propagation of flows caused directly by \(e\):

\[
\hat{f}_e : \hat{\Xi} \rightarrow \hat{\Xi}
\]
\[
\hat{f}_e(\hat{r}, \hat{\sigma}) = (\hat{r} \cup \hat{r}_e', \hat{\sigma}_e')
\]

where \(\hat{S}_e = \{\zeta' : e \in \hat{r} \text{ and } (e, \hat{\sigma}) \approx \zeta'\}\)

\[
\hat{r}_e' = \{e'' : (e'', \perp) \in \hat{S}_e\}
\]

\[
\hat{\sigma}_e' = \bigsqcup \{\hat{\sigma}'' : (\perp, \hat{\sigma}'') \in \hat{S}_e\}
\]

To determine the correctness and precision of this technique, we show its equivalence to an unpartitioned transfer function so we may exploit the corollary that a solution \(\hat{\xi}\) which is simultaneously a fix-point for all \(\hat{f}_e\) is guaranteed to be a fix-point for \(\hat{f}\).
Theorem 1 (Transfer Partitioning).

\[ \hat{f}(\hat{r}, \hat{\sigma}) = \bigcup_{e \in \hat{r}} \hat{f}_e(\hat{r}, \hat{\sigma}) \]

Proof. (Sketch) Follows from the observation that \( \hat{S}_e \) for all \( e \in \hat{r} \) is a collection of covering subsets for \( \hat{S} \).

\[ \hat{S} = \bigcup_{e \in \hat{r}} \hat{S}_e \]

Therefore \( \hat{r}' \) is also the least-join of all \( \hat{r}'_e \) as is \( \hat{\sigma}' \) of all \( \hat{\sigma}'_e \).

\[ \square \]

6 Linear Encoding for 0-CFA

Now that we may arbitrarily partition a transfer function to minimize thread-divergence, a linear encoding for handling a callsite can be defined separately from a linear encoding that handles a conditional, a primitive operation, or another form. The goal now is to produce an implementation for each \( f_e \) defined exclusively in terms of matrix multiplication (\( \times \)), outer product (\( \otimes \)), element-wise boolean-or (\( + \)), and dot product (\( \cdot \)).

For any finite domain, we can assign a canonical order to its contents and represent elements of its set or power-set as boolean vectors. Where vectors contain a single entry, they represent a single element in the set they encode, and where they contain more than one entry, the representation naturally extends to encoding more than one element at once. For example, as defined below, a value \( v \in V \) is a vector representing a flow-set of abstract values. In the case of \( S \), we use \( r \) in all cases to denote a set of states and \( s \) to denote a particular state (i.e. a vector with a single entry).

\[ r, s \in S = \{0, 1\}^{|E|} \]
\[ a \in A = \{0, 1\}^{|\hat{Var}|+|\hat{Value}|} \]
\[ v \in V = \{0, 1\}^{|\hat{Value}|} \]

Vectors \( a \) represent atomic-expressions, either variables or values. This is a design choice taken directly from EigenCFA which allows the various cases required for \( \hat{A} \) to be implemented as a single multiplication.

A function \( g \) over these vectors can be encoded as multiplication with a matrix, and may handle inputs which encode a set so long as the property \( g(x \cup y) = g(x) \cup g(y) \) holds for all \( x \) and \( y \). The store is such a function, one which maps variables to a flow-set of abstract values, and values to themselves:

\[ \sigma : A \rightarrow V \]

If values are ordered after variables in \( A \), the bottom of the store will always be an identity matrix. Below is an example of a lookup showing how the store is used to map a variable to its flow-set via matrix multiplication. We use a CPS version of our original snippet of Scheme code for clarity.
((lambda (add5x0)
  (add5x0 10^d3 (lambda (resultx3) (halt)^l3)^d2)^l1)^d0
(lambda (ax1 add5kx2)
  (prim + ax1 5^d3 add5kx2)^l2)^l1
(lambda (a add5k)
  (add5x0 0^d0)
  (add5x1 0^d1)
  (add5x2 0^d2)
  (add5x3 0^d3)
)

Annotations show an assignment of labels to expressions, variables to vectors in \( A \), and abstract values to vectors in \( V \). The abstract value \( d_3 \) represents \( \text{INT} \).

Example 1. \( \langle \langle a \rangle \rangle \times \sigma = \langle \langle \{\text{INT}\} \rangle \rangle \)

\[
\begin{bmatrix}
\hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\
x_0 & 0 & 1 & 0 \\
x_1 & 0 & 0 & 0 \\
x_2 & 0 & 0 & 1 \\
x_3 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
\hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
\hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

The notation \( \langle \langle \cdot \rangle \rangle \) is used informally to denote the matrix representation of a given entity. Including an identity matrix in the store composes two mappings as one matrix so that all cases in \( A \) may be handled together as a single multiplication.

A program’s syntax tree can also be encoded as a series of matrices. For example, a matrix \( \text{Body} \) maps lambdas in \( V \) to their body expression in \( S \). The same can be done for the true and false branches of a conditional form.

\[
\begin{align*}
\text{Body}: & \quad V \rightarrow S \\
\text{CondTrue}: & \quad S \rightarrow S \\
\text{CondFalse}: & \quad S \rightarrow S \\
\text{Fun}: & \quad S \rightarrow A \\
\text{Arg}_i: & \quad S \rightarrow A \\
\text{Var}_i: & \quad V \rightarrow A
\end{align*}
\]

\( \text{Fun} \) maps callsites to the atomic-expression in call-position. If this is a variable, a value in the top portion of \( A \) will result, if it’s a lambda or constant value, an entry in the lower portion of \( A \) results. \( \text{Arg}_i \) represents a similar encoding for argument \( i \) of a callsite. \( \text{Var}_i \) encodes formal parameter \( i \) of a lambda. For example, we can expect \( \langle \langle (\lambda (a \text{add5k}) \ldots) \rangle \rangle \times \langle \langle \text{Var}_2 \rangle \rangle \) to yield a value \( \langle \langle \text{add5k} \rangle \rangle \).

For a callsite \( s \), the value of its second argument can be computed as \( v_2 = s \times \text{Arg}_2 \times \sigma \) and the value of the applied lambda as \( v_f = s \times \text{Fun} \times \sigma \). The second formal parameter for \( v_f \) may then be computed as \( a_2 = v_f \times \text{Var}_2 \) and with these two values, the store can be updated with a binding to \( v_2 \) for \( a_2 \). This is accomplished by using the outer product \( a_2 \otimes v_2 \) as this will give a store-update matrix with an entry at index \( (m,n) \) whenever \( a_2 \) has an entry at position \( m \) and \( v_2 \) has one at \( n \). An update is applied to the current store using element-wise boolean-or. The example below shows the store update produced for \text{add5k}.
Example 2. \(\langle\langle \text{add5k}\rangle\rangle \otimes \langle\langle \text{(lambda (result) (halt))}\rangle\rangle\)
\[= \langle\langle \text{[add5k \rightarrow (lambda (result) (halt))]}\rangle\rangle\]

\[
\begin{bmatrix}
  x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix} \otimes \begin{bmatrix}
  \hat{d}_0 & \hat{d}_1 & \hat{d}_2 & \hat{d}_3 \\
  0 & 0 & 0 & 0
\end{bmatrix} =
\begin{bmatrix}
  d_0 & d_1 & d_2 & d_3 \\
  0 & 0 & 0 & 0
\end{bmatrix}
\]

An operation \(v_f \times \text{Body}\) finds the body for \(v_f\), and boolean-or is used to extend the vector of reachable expressions \(r\). A full encoding for \(f_e\) where \(e\) is a callsite of length \(j\) can now be defined in full using these operations:

\[
f_{\text{call}_j}(r, \sigma) = (r', \sigma')
\]
where
\[
v_f = s_{\text{call}_j} \times \text{Fun} \times \sigma
\]
\[
v_i = s_{\text{call}_j} \times \text{Arg}_i \times \sigma
\]
\[
a_i = v_f \times \text{Var}_i
\]
\[
\sigma' = \sigma + (a_1 \otimes v_1) + \ldots + (a_j \otimes v_j)
\]
\[
r' = r + (v_f \times \text{Body})
\]

To handle \(\text{set!}\) forms, we may reuse the matrix \(\text{Fun}\) for encoding the continuation, \(\text{Arg}_1\) for encoding the variable being set, and \(\text{Arg}_2\) for encoding the atomic-expression it's being assigned to. The continuation receives a value \(\langle\langle \text{VOID}\rangle\rangle\) we'll denote as \(\rightarrow\text{void}\):

\[
f_{\text{set!}}(r, \sigma) = (r', \sigma')
\]
where
\[
v_f = s_{\text{set!}} \times \text{Fun} \times \sigma
\]
\[
a_{\text{var}} = v_f \times \text{Var}_1
\]
\[
a_{\text{set}} = s_{\text{set!}} \times \text{Arg}_1
\]
\[
v_{\text{set}} = s_{\text{set!}} \times \text{Arg}_2 \times \sigma
\]
\[
\sigma' = \sigma + (a_{\text{var}} \otimes \rightarrow\text{void}) + (a_{\text{set}} \otimes v_{\text{set}})
\]
\[
r' = r + (v_f \times \text{Body})
\]

Conditionals make no changes to the store, but extend reachability to the subexpression for the true or false branches as appropriate. \(\rightarrow\text{false}\) is used to denote \(\langle\langle \text{FALSE}\rangle\rangle\) and \(\rightarrow\text{notfalse}\) to denote its inverse – which is notably not the same as \(\langle\langle \text{TRUE}\rangle\rangle\). A dot product is used to obtain a boolean value which is false
exactly when the intersection of two sets is empty:

\[ f_{s_{i,f}}(r, \sigma) = (r', \sigma) \]

where \( v_{\text{cond}} = s_{i,f} \times \text{Arg1} \times \sigma \)

\( tb = v_{\text{cond}} \cdot \text{notfalse} \)

\( fb = v_{\text{cond}} \cdot \text{false} \)

\( r' = r + tb(s_{i,f} \times \text{CondTrue}) + fb(s_{i,f} \times \text{CondFalse}) \)

### 6.1 A final algorithm

To find a solution, we can iterate to a fix-point \((r, \sigma)\) over all \(f_s\) where \(s\) is drawn from the entries of \(r\). In practice, we may exploit both the fine-grain parallelism of matrix operations and the coarse-grain parallelism of running each \(f_s\) concurrently. As each individual \(f_s\) is monotonic and continuous, our reasoning on termination and precision from section 4.2 remains applicable.

```plaintext
while \(\sigma\) or \(r\) changes do
  foreach \(s\) in \(r\) do
    \((r, \sigma) = f_s(r, \sigma)\)
  end
end
```

### 7 Conclusion

This analysis represents a faithful formulation of 0-CFA for Scheme and goes beyond the capabilities of EigenCFA in two key regards. First, including a range of reachable expressions allows the encoding to represent a more precise approximation of control-flow behavior corresponding to a traditional worklist implementation of 0-CFA. By only updating a global store, EigenCFA uses a fully imprecise control-flow approximation unnecessarily. Second, a sound and precise partitioning of the transfer function has allowed a variety of very different inference rules to be used without the need to entangle their implementation within a single GPU kernel (introducing thread-divergence). This permits extending the essential approach beyond trivial analyses of trivial languages. The work of applying our technique to 0-CFA can be further expanded to parallelize more sophisticated analyses of languages with an even greater variety of forms.

**Acknowledgments.** The authors would like to thank William E. Byrd, Leif Andersen, and Steven Lyde for helpful feedback on a draft of this work. This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.
References