Towards Abductive Functional Programming

Koko Muroya, University of Birmingham

Abstract

Abductive reasoning is a form of logical inference which seeks to uncover all possible causes of an observation. We show how abduction has a computational counterpart, like many other proof-theoretic concepts: namely the identification and modification of certain constants in a term. Abductive computation can be used to improve the behaviour of a term in some programmer-defined sense, like a typical workflow of optimisation problems including some machine-learning tasks. The emphasis of this progress report is a type system for abductive computation. It is intended to guarantee observably deterministic behaviour of programs, even though abduction may introduce a degree of computational nondeterminism.

1 Abduction in programming

In logic, abduction infers a possible cause A of a given observation B, knowing an implication $A \Rightarrow B$. Informally speaking, this is opposite to deduction, which infers a result B from a cause A—deduction goes along an implication $A \Rightarrow B$ whereas abduction goes against it. Much like induction, used carelessly abduction represents a fallacy, namely that of *affirming the consequent*. But with adequate restrictions in place we can show that abduction can be both sound and useful.

Abductive logic programming (see [3, 2] for surveys) extends logic programming by allowing a logic program to contain undefined "abducible" predicates, with applications to artificial intelligence. These predicates are candidates of possible causes of particular observations in the behaviour of a program. Inference in abductive logic programming tells us which abducible predicates of a program should hold to meet given constraints on the program. Abductive reasoning has been used in the verification of *separation logic programs*, allowing the improvement of automatic shape analysis [1]. Facebook's automatic verification tool Infer[1] is based on this technique.

We here propose a new use of abduction in functional programming, aiming at programs that modify specified constants of another program. Programs can contain normal *definitive* ground-type constants $\underline{p}$, as well as *provisional* ground-type constants $\{\underline{p}\}$ marked by curly brackets. It is these provisional constants that could be changed by abduction. The process of abduction itself detects all provisional constants in a term; it yanks them out of the abducted term, and turns the abducted term into a function that takes a collection of parameters as an argument.

[1] `http://fbinfer.com/`

2 Abductive computation

Abduction is expressed in programs as a function, written as `abd` f`@`x `->` u, which takes one argument and uses it as two bound computations in a body expression u: a *parameter collection* bound to the variable x and a *parameterised term* bound to the variable f. When applied to some argument t, the abduction `abd` f`@`x `->` u extracts all the provisional constants from it, turns them into definitive constants, and put them into the parameter collection x. Additionally it "lifts" the abducted term t into a parameterised term f, a function that will plug new arguments back at the sites where provisional constants used to be in the abducted term t. For example, abduction of a term $\underline{1} + \{\underline{0}\}$ produces a parameter collection $\underline{0}$ and a parameterised term `fun` x `->` $\underline{1} + x$.

The intended use of abduction is to recompute new values of provisional constants, and to plug them back as definitive ones into the original abducted term. This is done simply by function application, thanks to the parameterising of the abducted term, like this: `abd` f`@`x `->` $f\ (t\ f\ x)$ where the term t computes new values using the parameterised term f and the parameter collection x.

A trivial but useful example of abduction is `abd` f`@`x `->` $f\ x$, which "deprecates" a given term, i.e. transforms all its provisional constants into definitive ones. For example, a term (`abd` f`@`x `->` $f\ x$) $(\underline{1} + \{\underline{0}\})$ is evaluated to a definitive term $\underline{1}+\underline{0}$, and further to a result value $\underline{1}$. Abduction also works on functions, for example a term (`abd` f`@`x `->` $f\ x$) (`fun` w `->` $w + \{\underline{0}\}$) is deprecated and evaluated to a function value `fun` w `->` $w + \underline{0}$.

3 Types of collections

We have seen so far example terms that contain just one provisional constant. If a term t has more than one provisional constants, abduction of it produces a parameter collection x that is literally a collection of several definitive constants. Given the collection x, the parameterised term f made out of the abducted one t has to plug them back into the original t, in such a *safe* way that the application $f\ x$ yields exactly the deprecation of the abducted term t. For example a term (`abd` f`@`x `->` $f\ x$) $(\{\underline{1}\} - \{\underline{2}\})$ should be evaluated to $\underline{1} - \underline{2}$, not to $\underline{2} - \underline{1}$, which means the parameter collection x has to be *ordered*. The *size* of the collection also matters. If the parameter collection is implemented as a list, one can write a term (`abd` f`@`x `->` $f\ (\underline{1} :: x)$) $(\{\underline{1}\} + \{\underline{2}\})$ that tries to plug the extended collection $\underline{1} :: x$ back to the abducted term.

To ensure the safe restoration of a term after abduction, we use the type of a (fixed) field $\mathbb{F}$ as the ground type of value, and use "vector types" V_a over the ground type $\mathbb{F}$. Vectors V_a are thus indexed by name a, in the sense of [5].

They are a type of finite-dimensional vector spaces $\mathbb{F}^n$ over the field $\mathbb{F}$ with the standard basis. Typing judgements are indexed by sets of names, in addition to the usual sets of typed variables. The rules of the system are those of the simply-typed lambda calculus, plus a new rule for abduction:

$$\frac{\Delta, a \mid \Gamma, f : V_a \to T, x : V_a \vdash u : T'}{\Delta \mid \Gamma \vdash \texttt{abd}\ f\texttt{@}x\ \texttt{->}\ u : T \to T'}$$

where Δ is a set of names and Γ is a set of typed variables $x_i : T_i$. We require the name a in the rule to be *fresh*, in particular not to appear in Δ, T or T'.

In the rule, the two bound variables f and x share the same vector type V_a. This ensures the parameterised term f is always given a collection that has the same number of elements as the parameter collection x. The number of parameters, i.e. the dimension of the vector space V_a, is determined dynamically at runtime. For example, in a term $\texttt{let}\ y = \{\underline{2}\}\ \texttt{in}\ (\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ f\ x)\ (\{\underline{1}\} + y)$, the abducted term $\{\underline{1}\} + y$ depends directly on the provisional constant $\{\underline{1}\}$ and indirectly on $\{\underline{2}\}$ as well. It is only at runtime when the actual abducted term $\{\underline{1}\} + \{\underline{2}\}$ is determined.

The (ordered) standard basis of the vector type V_a enables the parameterised term f to know which element of a given collection fits to which place of the original abducted term. Note that this does not immediately mean a canonical order in which parameters are assigned; any order will do, as long as both the parameter collection x and the parameterised term f respect it. For example a term $(\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ f\ x)\ (\{\underline{1}\} - \{\underline{2}\})$ can be evaluated to either $(\texttt{fun}\ (p_0, p_1)\ \texttt{->}\ p_0 - p_1)\ (\underline{1}, \underline{2})$ or $(\texttt{fun}\ (p_0, p_1)\ \texttt{->}\ p_1 - p_0)\ (\underline{2}, \underline{1})$, informally.

The last remark is regarding the freshness requirement of name a in the typing rule. It ensures that two parameter collections x and x' generated by different abductive functions $\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ u$ and $\texttt{abd}\ f'\texttt{@}x'\ \texttt{->}\ u'$ always live in different vector spaces and they are never mixed together. This is crucial because these collections may not have the same number of elements, or even if they have the same dimension they may be use a different assignment of parameters to coordinates.

4 Operations on collections

As there is no canonical order in which parameters are assigned to bases/coordinates, implementation of the language can opt for either a fixed order or a non-deterministic order. In both cases the absence of canonical order leads to further restrictions on how parameter collections may be used, in other words, what operations on these collections can be allowed. The fixed order would be too intimately tied to the concrete syntax of a program, and raise difficulty in identifying terms $(\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ f\ x)\ (\{\underline{1}\} - \{\underline{2}\})$ and $(\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ f\ x)\ ((\texttt{fun}\ y\ \texttt{->}\ y - \{\underline{2}\})\ \{\underline{1}\})$, for example. The non-deterministic order would challenge the deterministic result of program execution.

The desired restriction of operations on collections appears to be the *symmetricity*, i.e. being invariant under permutation of the standard basis. In our intended form of abduction $\texttt{abd}\ f\texttt{@}x\ \texttt{->}\ f\ (t\ f\ x)$, the symmetricity applies to the term $t\ f$, meaning that $t\ f\ (1,2) = (3,4)$ implies $t\ f\ (2,1) = (4,3)$, for example. Usual operations of a vector space should be available, that is, vector additions $+_a \colon V_a \to V_a \to V_a$ and scalar multiplications $\times_a \colon \mathbb{F} \to V_a \to V_a$ for any name a. We can additionally equip each vector type V_a with inner products $\cdot_a \colon V_a \to V_a \to \mathbb{F}$.

What is desired but challenging is access to each coordinate of a collection, which naturally occurs in some optimisation algorithms. An example is numerical gradient descent for optimising the behaviour of a term relative to identified parameters:

$$\begin{aligned}
&\texttt{abd}\ f\texttt{@}x\ \texttt{->}\\
&\quad \texttt{let}\ g = \texttt{fun}\ v\ \texttt{->}\ \texttt{fun}\ e\ \texttt{->}\\
&\qquad\qquad \frac{(f\ v) - (f\ (v + (\epsilon \times e)))}{\epsilon} \times e + v\ \texttt{in} \qquad (1)\\
&\quad \texttt{let}\ y = \texttt{foldr}\ g\ x\ E_a\ \texttt{in}\ f\ y
\end{aligned}$$

where ϵ is a constant. It updates each element of the parameter collection x by folding over the list of standard basis $E_a = [\vec{e}_0; \cdots; \vec{e}_{n-1}]$.

Although the computation $\texttt{foldl}\ g\ x\ E_a$ in (1) is symmetric, i.e. invariant under permutation of x or E_a, it turns out that the arbitrary folding over E_a is not symmetric (a counterexample is the replacement of g to $\texttt{fun}\ v\ \texttt{->}\ \texttt{fun}\ e\ \texttt{->}\ (v \cdot e) \times v$). This motivates us to accommodate the following iterated vector operations, indexed by the name a:

$$\begin{aligned}
+^L_a &\colon (V_a \to V_a) \to V_a \to V_a\\
+^R_a &\colon V_a \to (V_a \to V_a) \to V_a\\
\times^L_a &\colon (V_a \to F) \to V_a \to V_a
\end{aligned}$$

which are three restricted versions of folding over the standard basis E_a.

$$\begin{aligned}
f +^L_a v_0 &:= \texttt{foldr}\ (\lambda e \lambda v.f(e) + v)\ E_a\ v_0\\
f +^R_a v_0 &:= \texttt{foldl}\ (\lambda v \lambda e.v + f(e))\ v_0\ E_a\\
f \times^L_a v_0 &:= \texttt{foldr}\ (\lambda e \lambda v.f(e) \times v)\ E_a\ v_0.
\end{aligned}$$

Our limited access to the standard basis via the iterated operations still enables us to implement some optimisation algorithms, e.g. generic combinatorial optimisations such as simulated annealing. The code (1) for numerical gradient descent is indeed covered by the first iterated operation $+^L$.

References

[1] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[2] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lect. Notes Comp. Sci.*, pages 402–436. Springer, 2002.

[3] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[4] Koko Muroya and Dan R. Ghica. The dynamic Geometry of Interaction machine: a call-by-need graph rewriter. In *CSL 2017*, 2017. To appear.

[5] Andrew M. Pitts. *Nominal sets: names and symmetry in computer science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

A Operational Semantics

Operational semantics of the language is given by graph rewriting. The use of graphs instead of terms makes it easier to handle shared intermediate evaluation results, especially shared provisional constants, that are raised by multiple occurrences of bound variables. Sharing is simply represented as connection of subgraphs, while its syntactic representation would require introducing names and stores.

Since provisional constants are subject to abduction and change, they cannot be freely copied or discarded in evaluation; in particular they should be shared instead of being copied. For instance, a term $(\texttt{fun}\ x\ \texttt{->}\ x + x)\ \{\underline{0}\}$ should not be evaluated to a term $\{\underline{0}\} + \{\underline{0}\}$, as the single provisional constant is "split" into two independent provisional constants.

The graph rewriting semantics is defined as an abstract machine that graphically evaluates terms in the strict way, i.e. function arguments are always evaluated before they are passed to a function. The machine is an adaptation of our graph-rewriting abstract machine of call-by-need λ-calculus [4], that can deterministically reduce a graphical representation of a term by passing a token on it. An on-line visualiser of the graph-rewriting abstract machine is implemented[2].

[2]Link to on-line visualiser: `http://www.cs.bham.ac.uk/~drg/goa/visualiser/index.html`