Branching execution symmetry in Jeopardy by available implicit arguments analysis

Joachim Kristensen[1][0000−0002−1619−5944], Robin Kaarsgaard[2][0000−0002−7672−799X], and Michael Kirkedal Thomsen[1,3][0000−0003−0922−3609]

[1] University of Oslo, Oslo, Norway
[2] University of Edinburgh, Edinburgh, United Kingdom
[3] University of Copenhagen, Copenhagen, Denmark

Abstract. When the inverse of an algorithm is well-defined – that is, when its output can be deterministically transformed into the input producing it – we say that the algorithm is invertible. While one can describe an invertible algorithm using a general-purpose programming language, it is generally not possible to guarantee that its inverse is well-defined without additional argument. Reversible languages enforce deterministic inverse interpretation at the cost of expressibility, by restricting the building blocks from which an algorithm may be constructed.
Jeopardy is a functional programming language designed for writing invertible algorithms *without* the syntactic restrictions of reversible programming. In particular, Jeopardy allows the limited use of locally non-invertible operations, provided that they are used in a way that can be statically determined to be globally invertible. However, guaranteeing invertibility in Jeopardy is not obvious.
One of the central problems in guaranteeing invertibility is that of deciding whether a program is symmetric in the face of branching control flow. In this paper, we show how Jeopardy can solve this problem, using a program analysis called available implicit arguments analysis, to approximate branching symmetries.

Keywords: Program analysis, functional programming, invertible languages

1 Introduction

The interest in programs that can recover the inputs from a computed output have long existed: from McCarthy's generate-and-test method [13] to the numerous inversion techniques associated with the *reversible model of computation* [1, 6, 11]. Many languages have been designed to guarantee that programs describe reversible algorithms, often by restricting *which* programs are allowed. A notable such language is Janus [12, 19], a reversible imperative language which guarantees partial reversibility[4] by restricting programs to be sequences of lo-

[4] Often (as in Janus) local invertibility only guarantees invertibility of partial functions. This comes from the fact that control structures (like conditional and loops) require assertion of specific values, and because procedures may fail to terminate.

cally invertible statements. Furthermore, Theseus [8] restricts programs to be a composition of locally invertible surjective functions. Finally, RFun [16, 18] imposes constraints that enforce function invertibility by enforcing a bidirectional first-match policy on choice points (at runtime), and requiring programs to be linear in their arguments. Doing so, is sufficient to guarantee invertible algorithms while not restricting computational power beyond R-Turing completeness[5]. Both of these constraints can clearly be checked statically: for the former, we may require the programmer to unroll their program until input and output patterns are syntactically orthogonal, and the latter can be enforced by linear typing [3, 17] as has been shown for CoreFun [7], a simple typed version of RFun.

However, writing algorithms in a way that makes their reversibility evident can be difficult, as it corresponds, in a certain way, to asking the programmer to prove this property as they are writing the program. Writing programs in these reversible languages requires some experience, and can in some cases be notoriously hard. The first attempts to this approach, was McCarthy's generate-and-test algorithm [13]. As this method is often infeasible in practice, later research approached the problem using program inversion [4, 5] or even semi-inversion [9, 14]. Since these methods all build on the conventional programming model, they may fail in cases where a deterministic inverse does not exist.

The language Jeopardy [10] has been designed with inversion in mind. It can be seen as a combination of the above two approaches: restricting the syntax enough to be able to give static inversion guarantees, but relaxing the execution model enough to make programming as natural as possible. It has a syntactic resemblance to your garden variety functional programming language and exhibits the expected semantics for programs running in the conventional direction.

However, not all algorithms describe bijective functions, and the problem of deciding whether an algorithm is invertible is undecidable in general following from Rice's Theorem. This means that the static analysis needed to guarantee inversion of even simple Jeopardy programs is not straightforward. In this work, we investigate the approach of approximating global program invertibility by developing a data flow analysis that infers the information necessary to make the approximation.

To be precise, in Section 2 we outline the problem by providing an instructive program example. In Section 3 we briefly outline the syntax and semantics of the Jeopardy programming language; a more formal introduction to the language was presented at IFL 2022 [10]. In Section 4 we detail the meaning of implicit arguments to functions that are inversely interpreted. In Section 5 we provide an algorithm for performing *implicitly available arguments analysis* on Jeopardy programs. Furthermore, in Section 6 we run the algorithm on the program example, and in Section 7 we discuss the implications of the result. Finally, in Section 8 we conclude on the results.

[5] R-Turing completeness is Turing completeness restricted to programming languages (and hence Turing machines) defining only reversible programs.

A Haskell implementation of *available implicit arguments analysis* for Jeopardy can be found at:

`https://github.com/jtkristensen/Jeopardy/blob/main/src/Analysis/ImplicitArguments.hs`

2 Branching symmetries and invertibility

The extensional behavior of a program can be reasonably thought of as a function mapping inputs to outputs [2]. In this perspective, the existence of an inverse program is analogous to the existence of an inverse function. That is, a program $f : A \to B$ is invertible when a deterministic inverse program $f^{-1} : B \to A$ exists, such that the functions they describe satisfy $f^{-1} \circ f = \mathrm{id}_A$ and $f \circ f^{-1} = \mathrm{id}_B$.

At the extensional abstraction of mathematical functions, we cannot infer any more about a function's behavior than that which may be derived from the premises given by the function's provider. However, when we are presented with a program, we can perform program analysis that inspect it to gain insight into its properties.

One such analysis, called *available expressions analysis* [15], produces a set of expressions per program point that has already been computed when the point is reached at runtime. One purpose to perform this particular analysis can be to transform programs into equivalent programs that do not recompute expressions when it is not necessary.

In Jeopardy we wish to decide the set of available expressions that could have been implicitly provided as function arguments at particular call sites in a program. Providing extra arguments to functions do not necessarily make those programs run more efficiently, but it allows us to infer more precise things about branching inside function calls ahead of runtime. For instance, consider the program given in Figure 1.
The main function of the program `fibonacci` takes a natural number n as its argument, and produces a pair containing the n'th Fibonacci number together with n itself. It does so by projecting from the function `fibonacci_pair` that produces a pair containing the n'th Fibonacci number together with its successor. The pair is computed by recursively applying the `fibber` function, which transforms a pair of Fibonacci numbers into the next pair. As dictated by the definition of the Fibonacci sequence, `fibber` finds the next number in the sequence by summing the two previous numbers.

Starting from the top, the function `sum`, which computes sums of pairs of natural numbers, is *not* invertible. To be precise, the solution 4 does not have a unique corresponding problem. In particular, it can be the sum of 1 and 3, or it can be the sum of 0 and 4. In the latter case, `sum` would simply return the 4 by taking the first branch of the case statement, and in the former case, it would take the second branch and call itself recursively.

However, the output of `sum` is still uniquely determined by the input, and so, inferring which branch was taken in each call to `sum`, is sufficient for uniquely determining its input. For instance, in the function `fibber` we never call `sum`

```
data natural_number = [zero] [successor natural_number].

sum (m, n) =
  case m of
  ; [zero]        -> n
  ; [successor k] -> sum (k, [successor n]).

fibber (m, n) = (sum (m, n), m).

fibonacci_pair n =
  case n of
  ; [zero]        -> ([successor [zero]], [successor [zero]])
  ; [successor k] -> fibber (fibonacci_pair k).

fibonacci n =
  case fibbonacci_pair n of
  ; (_, nth_fibonacci_number) -> (nth_fibonacci_number, n).

main fibonacci.
```

Fig. 1. Example program, computing fibonacci numbers.

without also returning the first of its arguments as well. Thus, in the inverse interpretation of `sum`, if the first argument is 0, we are done. Otherwise, the first argument is the successor of some natural smaller number `k`, and we can inverse interpret the `sum` function recursively.

Similarly, it is not trivial that `fibonacci_pair` is an invertible algorithm, since you need to know that the second branch of the case-statement will always be syntactically orthogonal to a pair of ones. However, as the argument for `fibbonacci_pair` is directly available in the output of `fibonacci`, it is clearly invertible in the context of inverse interpreting the `fibonacci` function.

To summarize, obtaining the inverse of the entire `fibbonacci` program corresponds to writing a program that takes as argument a pair containing the n'th Fibonacci number together with n itself and merely gives back the n. The right projection is insufficient in terms of correctly determining the problem for this solution, because its interpretation in the conventional direction does not compose to an identity. However, recovering information about branching *is* sufficient for inverse interpretation. Reversible programming languages that enforce local invertibility, such as RFun and CoreFun [7,18] simply throw an error at runtime if branching does not comply with a *symmetric first match policy* for pattern matching, and Theseus [8] requires the programmer to account for branching structure syntactically.

However, as we have seen in the `fibonacci` program example, realistic programs often exhibit inter-procedural information that allows us to recover the branching structure. It remains to show how this information may be obtained systematically. In the remainder of this article, we concern ourselves with doing just that.

3 The Jeopardy Programming Language

Jeopardy is a carefully designed first order functional language aimed at expressing invertible algorithms while enabling concise program analysis and dissemination. The main features are user-definable algebraic data types and explicit function level program inversion. The full grammar can be found in Figure 2.

$x \in \mathbf{Name}$	(Well-formed variable names).
$c \in \mathbf{Name}$	(Well-formed constructor names).
$\tau \in \mathbf{Name}$	(Well-formed datatype names).
$f \in \mathbf{Name}$	(Well-formed function names).
$p ::= [c\ p_i] \mid x$	(Patterns).
$v ::= [c\ v_i]$	(Values).
$\Delta ::= f\ (p : \tau_p) : \tau_t\ =\ t\ .\ \Delta$	(Function definition).
$\mid \texttt{data}\ \tau\ =\ [c\ \tau_i]_j\ .\ \Delta$	(Data type definition).
$\mid \texttt{main}\ g\ .$	(Main function declaration).
$g ::= f \mid (\texttt{invert}\ g)$	(Inversion).
$t ::= p$	(Patterns in terms).
$\mid g\ p$	(First order function application).
$\mid \texttt{case}\ t : \tau\ \texttt{of}\ p_i \rightarrow t_i$	(Case statement).

Fig. 2. The syntax of Jeopardy.

Running a program corresponds to calling the declared `main` function on a value provided by the caller in the empty context. Similarly, running a program backwards corresponds to calling the main function's inverse on a value provided by the caller in the empty context as well. Since an application is a term, reasoning about inversion of terms *is the same as* reasoning about inversion of programs.

The syntax of terms has been designed with the goal of providing program analysis at the cost of making programs harder to read and write. In the interest of writing intuitive program examples, we have therefore equipped Jeopardy with a set of derived syntactic connectives that we have shown in Figure 3. Additionally, we may choose to omit type annotations whenever these are not necessary, and use literal syntax for natural numbers (`0`, `1`, ...) to mean their data representations (`[zero]`, `[successor [zero]]`, ...).

$$\begin{aligned}
[\![[c\ t_i]]\!]_{\Delta[\texttt{data}\ \tau=[c\tau_i]_j]} &:= \texttt{case}\ t_i : \tau_i\ \texttt{of}\ p_i \to [c\ p_i] \\
[\![(t_1, t_2)]\!]_{\Delta} &:= [\![[\texttt{pair}\ t_1\ t_2]]\!]_{\Delta} \\
[\![t_1 : t_2]\!]_{\Delta} &:= [\![[\texttt{cons}\ t_1\ t_2]]\!]_{\Delta} \\
[\![\texttt{[]}]\!]_{\Delta} &:= [\texttt{nil}] \\
[\![f\ t]\!]_{\Delta[f(\cdot:\tau):\cdot=\cdot]} &:= \texttt{case}\ t : \tau\ \texttt{of}\ p \to f\ p \\
[\![\texttt{let}\ p : \tau =\ t\ \texttt{in}\ t']\!]_{\Delta} &:= \texttt{case}\ t : \tau\ \texttt{of}\ p \to t' \\
[\![t']\!]_{\Delta[f(p_i:\tau_p):\tau_t=t_i\,.]} &:= [\![t']\!]_{\Delta[f(x:\tau_p):\tau_t=\texttt{case}\ x:\tau_p\ \texttt{of}\ p_i \to t_i]}
\end{aligned}$$

Fig. 3. Disambiguation of syntactic sugar.

4 Implicit Arguments

Recall from Section 2 that sum was not injective, and thus its inverse was not well-defined. A naïve solution to this problem is to automatically injectivise the program using, e.g., Bennett's method [1]: that is, we keep a computation history of our program, copy its result, and uncompute the history to be left with the input and a copy of the result. However, this is a very inefficient way of doing invertible computing and would constantly generate extra unwanted data. In many cases we can do better by first determining which inputs are needed for uniquely deciding branching information bidirectionally. For example, if we want to make sum invertible it suffices to copy one of its inputs to the outputs as follows:

```
sum_and_copy_first (m, n) =
  case m of
  ; [zero]        -> (m, n)
  ; [successor k] ->
    case sum_and_copy_first (k, [successor n]) of
    ; (k, k+suc_n) -> (m, k+suc_n).

sum_and_copy_second (m, n) =
  case m of
  ; [zero]        -> (n, n)
  ; [successor k] ->
    case sum_and_copy_second (k, [successor n]) of
    ; (k+suc_n, [successor n]) -> (n, k+suc_n).
```

To convince the reader that the branching symmetry is recoverable from the transformed program, in sum_and_copy_first we are matching on m, and m is embedded directly in the output. To see that branching is symmetric in sum_and_copy_second, we need to show that k+suc_n and n are different in the last branch, which we can by co-induction since the recursive call returns a pair of successors of n, or some larger structure that contains n from previous calls. Regardless of our choice of injectivisation of `sum`, we need to store information from the previous call in order to make branching symmetry decidable for the next, effectively by supplying a function (in this case the inverse to `sum`) with extra arguments.

By producing specialised functions that take extra arguments in this way, we can transform programs that contain the original functions into equivalent programs that call the specialised function whenever the extra arguments are available. For instance, we might rewrite `fibber` from Figure 1 as follows:

```
fibber_specialized_for_sum_and_copy_first (m, n) =
  case sum_and_copy_first (m, n) of
  ; (m, m+n) -> (m+n, m).
```

This transformation depends on knowing which specialised versions of a particular function it can use, and in turn, it depends heavily on knowing what terms are available, or can be made available in the program point at which the call happens. Because Jeopardy is a pure functional language, this is simply all terms that appear in all paths to the program point in a call graph with the programs main function as its entry point, and we already know how to construct such a graph [15].

In fact, we can do a little better, since we only need to require that a term appears on every path that allows us to apply specialisation; though not necessarily the same specialisation in every path. In this way, the goal of our algorithm is as follows: for every function application in a program, for every distinct path to that application from the main function, compute what terms are available to be provided as implicit (extra) arguments.

5 The Algorithm

To avoid having to deal with names, our algorithm performs an initial annotation of the input program, where each program point is assigned a unique integer label, as will be demonstrated for Figure 1 in Section 6. Furthermore, to make things more concrete, we specify what it means to be a call-configuration, as specified in Definition 1.

Definition 1. *A call-configuration is a 4-tuple* (c, f, A, I) *, containing the name "c" of the function in which the call occurred, the (possibly inverted) function "f" being called, a set "A" containing the labels of the arguments to the function, a set "I" of available implicit arguments from previous calls in which the program is running at the time of the call.*

Now, achieving the goal presented in Section 4, is equivalent to answering the question: for each call in a program, what are the possible configurations of the call?

We answer this question by solving a set of equations. Each equation, have a fixed program-of-interest Δ. We give the name $\mathcal{F}$ to the set of function names defined in Δ; The superset of $\mathcal{F}$ that includes inversions (function names occurring under the keyword `invert`), we call $\mathcal{I}$. Furthermore, we assign the name $\mathcal{L}$ to the set of labels of Δ, and finally, we give the name $\mathcal{C} \subseteq (\mathcal{F} \times \mathcal{I} \times \mathcal{L} \times \mathcal{L})$ to the set of possible call-configurations.

To produce all the possible configurations, we declare a function that computes the closure of the call-configurations that are reachable from two initial configurations[6]:

$$\text{configuration} : \Delta \to \mathcal{P}(\mathcal{C})$$

Its corresponding definition can be found in Figure 4, where Δ has been extended with a special top level function $\top$ and two special labels "input" and "output" for the arguments that should be provided by the entity that runs the program. The equation, and the computations it depends, on are all defined in this section.

$$\begin{aligned}\text{configurations}(\Delta[\mathit{main}\ g.]) = {} & \{(\top, g, \{\text{input}\}, \emptyset), (\top, (\texttt{invert}\ g), \{\text{output}\}, \emptyset)\} \\ & \cup \left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right)\end{aligned}$$

Fig. 4. The reachable call configurations from main.

In the last part of the definition of configurations, a function:

$$\text{call} : \mathcal{C} \to \mathcal{P}(\mathcal{C})$$

takes as argument, a configuration and returns all the possible configurations that are reachable by calling the (possibly inverted) function f from that configuration, as defined in figure 5

$$\text{call}((c, f, A, I))_{\Delta[f p = t.]} = \begin{cases} \text{term} \downarrow (f, (I \cup A) \setminus (\text{labels}(p, t)), t) & : \ \text{dir}(f) = \downarrow \\ \pi_1(\text{term} \uparrow (f, (I \cup A) \setminus (\text{labels}(p, t)), t)_\Delta) & : \ \text{otherwise} \end{cases}$$

Fig. 5. The reachable configurations from a given configuration.

Depending on the direction of execution, the reachable definitions are defined by a inspecting the terms that constitute the terms and patterns that define the

[6] forward and backward from main.

body and argument of a function. We give the name $\mathcal{T}$ for such terms, and further declare two functions:

$$
\begin{aligned}
&\text{term}\downarrow\colon (\mathcal{F}, \mathcal{I}, \mathcal{T}) \rightarrow \mathcal{P}(\mathcal{C}) \\
\text{and}\quad &\text{term}\uparrow\colon (\mathcal{F}, \mathcal{I}, \mathcal{T}) \rightarrow \mathcal{P}(\mathcal{C} \times \mathcal{L})
\end{aligned}
$$

that compute the reachable configurations depending on said direction $dir(f)$ in which the call is to be interpreted. We define these functions in Figure 6. They both return the set of configurations reachable from their argument term t. However, the function interpreting calls against the conventional direction returns the labels of available expressions from "the future" as a means of definitional convenience.

In the case for patterns, both functions yield an empty set of call-configurations since a pattern cannot contain an application. That is, terms in patterns are syntactic sugar that we disambiguated in Figure 3. The cases for function application yield the configurations reachable as defined by the function "call". Regarding case-statements, both functions return the collection of configurations reachable in each branch.

$$
\begin{aligned}
\text{term}\downarrow (f, I, [\![p]\!]) &= \emptyset \\
\text{term}\downarrow (f, I, [\![g\ p]\!]) &= \{(f, g, \text{labels}(p), I, \text{direction}(g))\} \\
\text{term}\downarrow (f, I, [\![\texttt{case}\ t\ \texttt{of}\ p_j \rightarrow t_j]\!]_{j\in J}) &= \text{term}\downarrow (f, I, t) \\
&\quad \cup \Big(\bigcup_{j\in J} (\text{term}\downarrow ((f, I \cup (\text{labels}(t) \cup \text{labels}(p_j)), t_i)))\Big) \\
\text{term}\uparrow (f, I, [\![p]\!])_{\Delta} &= \{(\emptyset, I \cup \text{labels}(p))\} \\
\text{term}\uparrow (f, I, [\![g\ p]\!]^{l_1})_{\Delta[gq=t^{l_0}]} &= \{(((f, g, \{l_0\}, I, \text{op}(\text{dir}(g))), \{l_1\}) \| c \in \text{call}((f, g, \} \\
\text{term}\uparrow (f, I, [\![\texttt{case}\ t\ \texttt{of}\ p_j \rightarrow t_j]\!]^{l}_{j\in J}) &= \\
\bigcup_{j\in J} \{(L_h, \{l\} L_t) \| (L_s, L_j) &\in \text{term}\uparrow (f, I, t_j) \\
(L_h, L_t) &\in \text{term}\uparrow (f, L_j \cup \text{labels}(p_j), t)\}
\end{aligned}
$$

Fig. 6. Call configurations, reachable in terms for each direction of interpretation.

The direction dir, and the opposite direction op, of a function call are defined by their corresponding functions in Figure 7, and with that we are done defining the algorithm.

Computing the set of possible reachable call configurations in a program Δ now, corresponds to calling the function `configurations` on Δ. It does so by finding the least fixed point of the function `call` from two initial configurations. And, we know that this least fixed point always exists (and thus the algorithm is well defined), from observing that configurations can be given the structure of

$$\begin{aligned}
\mathrm{dir}(g) &= \begin{cases} \mathrm{op}(\mathrm{dir}(f)) & : \ g = (\texttt{invert}\ f) \\ \downarrow & : \text{otherwise} \end{cases} \\
\mathrm{op}(\downarrow) &= \uparrow \\
\mathrm{op}(\uparrow) &= \downarrow
\end{aligned}$$

Fig. 7. Definition, the direction of a function call and its opposite direction.

a complete lattice by $(c, f, A, I) \sqsubseteq (c', f', A', I')$ iff $c = c'$, $f = f'$, $A = A'$, and $I \subseteq I'$, with joins and meets of configurations (with the same name, function, and label set) given by unions and intersections of available implicit arguments. Further, it can be shown that the functions in Figures 4, 5 and 6 are all monotone with respect to this order, so it follows by Tarski's fixed point theorem that the least fixed point we are looking for always exists. Furthermore, since a program contains only finitely many labels, it follows additionally that the analysis always terminates.

6 Instructive Example

On a less theoretical note, let us look at an example, namely that of finding the available implicit arguments at all call sites in the Jeopardy example from Figure 1. This is the same as finding a minimal fixed point for the equation for "configurations(Δ)", where

$\Delta =$

```
(data natural_number = [zero] [successor natural_number].

 sum (m^1, n^2)^0 =
   (case m^4 of
    ; [zero]^5          -> n^6
    ; [successor k^8]^7 -> (sum (k^10, [successor n^12]^11))^9)^3.

 fibber (m^14, n^15)^13 = ((sum (m^19, n^20)^18)^17, m^21)^16.

 fibonacci_pair n^22 =
   (case n^24 of
    ; [zero]^25           -> ([successor [zero]^28]^27, [successor [zero]^30]^29)^26
    ; [successor k^32]^31 -> (fibber (fibonacci_pair k^35)^34)^33)^23.

 fibonacci n^36 =
   (case (fibbonacci_pair n^39)^38 of
    ; (_^41, nth_fibonacci_number^42)^40 -> (nth_fibonacci_number^44, n^45)^43)^37.

 main fibonacci^46.)
```

As the main function is `fibbonaci` we immediately get:

$$\begin{aligned}
&\text{configurations}(\Delta[\mathit{main}\ \texttt{fibbonaci.}]) \\
&\qquad = \{(\top, \texttt{fibbonaci}, \{\text{input}\}, \emptyset), (\top, (\texttt{invert fibbonaci}), \{\text{output}\}, \emptyset)\} \\
&\qquad \cup \left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right)
\end{aligned}$$

And, if we focus on the last term, and unfold it one step, we get:

$$\left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right) = \text{call}((\top, \texttt{fibbonaci}, \{\text{input}\}, \emptyset))_\Delta \cup \left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right)$$

If we again restrict our attention to the "call" part, the result is:

$$\begin{aligned}
&\text{call}((\top,\texttt{fibbonaci}, \{\text{input}\}, \emptyset))_{\Delta[\texttt{fibbonaci n}^{36}=t]} \\
&\quad = \text{term} \downarrow (\texttt{fibbonaci}, \{\text{input}\} \setminus \{36\}, t) \\
&\quad = \text{term} \downarrow (\texttt{fibbonaci}, \{\text{input}\}, [\![\texttt{case}\ t^{38}\ \texttt{of}\ p_i^{40}\ \rightarrow\ t_i^{43}]\!]) \\
&\quad = \text{term} \downarrow (\texttt{fibbonacci}, \{\text{input}\}, t^{38}) \\
&\quad \cup \text{term} \downarrow (\texttt{fibbonacci}, \{\text{input}\} \cup \text{labels}(t^{38}) \cup \text{labels}(p_i^{40}), t_i^{43}) \\
&\quad = \{(\texttt{fibbonacci}, \texttt{fibbonacci\_pair}, \{39\}, \{\text{input}\})\} \cup \emptyset
\end{aligned}$$

Finally, we get

$$\begin{aligned}
&\text{configurations}(\Delta[\mathit{main}\ \texttt{fibbonaci.}]) \\
&\qquad = \{(\top, \texttt{fibbonaci}, \{\text{input}\}, \emptyset), \\
&\qquad\qquad (\top, (\texttt{invert fibbonaci}), \{\text{output}\}, \emptyset), \\
&\qquad\qquad (\texttt{fibbonacci}, \texttt{fibbonacci\_pair}, \{39\}, \{\text{input}\})\} \\
&\qquad\quad \cup \left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right) \\
&\qquad = \{(\top, \texttt{fibbonaci}, \{\text{input}\}, \emptyset), \\
&\qquad\qquad (\top, (\texttt{invert fibbonaci}), \{\text{output}\}, \emptyset), \\
&\qquad\qquad (\texttt{fibbonacci}, \texttt{fibbonacci\_pair}, \{39\}, \{\text{input}\}), \\
&\qquad\qquad (\texttt{fibbonacci\_pair}, \texttt{fibber\_pair}, \{35\}, \{\text{input}, 39, 22, 23, 24, 31, 32\}), \\
&\qquad\qquad (\texttt{fibbonacci\_pair}, \texttt{fibber}, \{34\}, \{\text{input}, 39, 22, 23, 24, 31, 32\})\} \\
&\qquad\quad \cup \left(\bigcup_{c \in \text{configurations}(\Delta)} \text{call}(c)_\Delta \right) \\
&\qquad = \{(\top, \texttt{fibbonaci}, \{\text{input}\}, \emptyset),
\end{aligned}$$

$$\begin{aligned}
&(\top, (\texttt{invert fibbonaci}), \{\text{output}\}, \emptyset),\\
&(\texttt{fibbonacci}, \texttt{fibbonacci\_pair}, \{39\}, \{\text{input}\}),\\
&(\texttt{fibbonacci\_pair}, \texttt{fibber\_pair}, \{35\}, \{\text{input}, 39, 22, 23, 24, 31, 32\}),\\
&(\texttt{fibbonacci\_pair}, \texttt{fibber}, \{34\}, \{\text{input}, 39, 22, 23, 24, 31, 32\}),\\
&(\texttt{fibber}, \texttt{sum}, \{18, 19, 20\}, \{\text{input}, 39, 22, 23, 24, 31, 32, 13, 14, 15, 21\}),\\
&(\texttt{sum}, \texttt{sum}, \{10, 11, 12\}, \{0, 1, 2, 4, 7, 8\})\}\\
\cup &\left(\bigcup_{c \in \text{configurations}(\mathit{\Delta})} \text{call}(c)_{\mathit{\Delta}} \right)
\end{aligned}$$

Here the additional call-configurations in the last part are derived in a similar fashion to that which we did when we focussed on the equation for "call" in the conventional direction, and in the interest of saving space in the paper, deriving the call configurations for (`invert fibbonacci`) is left as an exercise for the reader.

7 Discussion

The transformation suggested in Section 2 requires us be able to infer a particular set of call-configurations. We have designed and implemented an algorithm for inferring said configurations. The algorithm works well for finding implicitly available arguments in function calls, but is limited in scope, to the configurations that do not reach beyond a single step of recursion in its search for implicit arguments. However, a single step is not a theoretical limit. It is easy to imagine a generalized algorithm that infers implicitly available arguments up to a fixed depth, but less so for arbitrary depth recursion.

In Section 6, we have seen how to compute the implicitly available arguments at each program point in an example program that computes the Fibonacci numbers. The result of the analysis allows us to replace functions for which we cannot decide branching symmetry with specialized variations for which we can. For instance, at program point 17, `fibber` calls `sum` in a context where its first argument `m` is implicitly available in both directions (from program point 21) as witnessed, for the conventional direction, by the tuple

$$(\texttt{fibber}, \texttt{sum}, \{\dots\}, \{\dots, 21, \dots\})$$

And so, we can make `fibber` invertible by replacing the non-invertible function call to `sum` with a call to the invertible specialization `sum_and_copy_first` from Section 4, which in turn allows us to invert `fibber`, as demonstrated in the example function `fibber_specialized_for_sum_and_copy_first` also in Section 4.

The problem with regards to recursion, is that the set of implicitly available expressions' labels I in a configuration, according to Definition 1, corresponds to the terms that were "available from previous calls". So, in a circular call structure (like recursion) it is not possible to see the difference in I between a term bound in this call, or the previous. In the example program from Section 2, this is not a problem, because deciding if e.g. `sum` is symmetric with respect to branching, only relies on implicit arguments either from the previous call to `fibber` or `sum` itself. However, one could imagine a scenario, where a term actually exhibits branching symmetry even though our analysis does not find sufficient information to say so.

8 Conclusion

We have designed a program analysis for statically inferring the expressions that are available as implicit arguments in function calls in the Jeopardy programming language. The current formulation of the algorithm can be implemented in about 200 lines of code in fairly readable and maintainable Haskell code. Our implementation makes use of the nifty Reader-Writer-State monad, which turns out to be suitable for threading around the program Δ, as well as keeping track of termination conditions etc.

In the near future, we expect to develop a program transformation, that rewrites Jeopardy programs where branching symmetry is not syntactically apparent, into programs where it is. To give an intuition, we have provided an example in Section 4, of applying such a transformation to the function `fibber` from the program example in Section 2.

The reasons for wanting to design and implement such a transformation is twofold. As mentioned earlier, the main motivation is to enable static analysis for deterministic backwards branching execution inference. But such a transformation also has similar motivation to that of compiling programs rather than interpreting them. That is, a jeopardy interpreter should not perform this analysis every time a function is called, or thread around the implicit arguments at runtime, it should transform the program into an equivalent program where implicit arguments have been explicitly provided when necessary.

Static program analysis is almost always an approximation, and cyclic call structures is where you will find the limitations of the available implicit arguments analysis. However, it is not hard to imagine a definition of call-configurations, that include (finitely many) layers of cyclic references to previous calls. And, it is unclear at the time of writing, if doing so will by useful in practice.

Acknowledgments The second author is supported by DFF–International Postdoctoral Fellowship 0131-00025B.

References

1. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development **17**(6), 525–532 (1973)

2. Danielsson, N., Hughes, J., Jansson, P., Gibbons, J.: Fast and loose reasoning is morally correct. In: 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 206–217. POPL '06, ACM (2006)
3. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987)
4. Glück, R., Kawabe, M.: A method for automatic program inversion based on LR(0) parsing. Fundamenta Informaticae **66**(4), 367–395 (2005)
5. Hou, C., Quinlan, D., Jefferson, D., Fujimoto, R., Vuduc, R.: Synthesizing loops for program inversion. In: Glück, R., Yokoyama, T. (eds.) Reversible Computation, Lecture Notes in Computer Science, vol. 7581, pp. 72–84. Springer Berlin Heidelberg (2013)
6. Huffman, D.A.: Canonical forms for information-lossless finite-state logical machines. IRE Transactions on Information Theory **5**(5), 41–59 (1959)
7. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: A typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation. pp. 304–321. Springer International Publishing (2018)
8. James, R.P., Sabry, A.: Theseus: A high level language for reversible computing (2014), work in progress paper at RC 2014. Available at `www.cs.indiana.edu/~sabry/papers/theseus.pdf`
9. Kirkeby, M.H., Glück, R.: Semi-inversion of conditional constructor term rewriting systems. In: Gabbrielli, M. (ed.) Logic-Based Program Synthesis and Transformation. pp. 243–259. Springer International Publishing (2020)
10. Kristensen, J.T., Kaarsgaard, R., Thomsen, M.K.: Jeopardy: An invertible functional programming language (2022). `https://doi.org/10.48550/ARXIV.2209.02422`, work-in-progress paper presented at 34th Symposium on Implementation and Application of Functional Languages
11. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 261–269 (1961)
12. Lutz, C., Derby, H.: Janus: A time-reversible language. A letter to R. Landauer (1986), available at `http://tetsuo.jp/ref/janus.pdf`
13. McCarthy, J.: The inversion of functions defined by Turing machines. In: Shannon, C.E., McCarthy, J. (eds.) Automata studies. Annals of Mathematics Studies, Princeton University Press (1956)
14. Mogensen, T.Æ.: Semi-inversion of functional parameters. In: Proceedings of the 2008 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 21–29. PEPM '08, ACM (2008)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Berlin / Heidelberg, Berlin, Heidelberg (2015)
16. Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language. In: Symposium on the Implementation and Application of Functional Programming Languages. pp. 8:1–8:13. IFL '15, ACM (2016)
17. Wadler, P.: Linear types can change the world! In: IFIP TC 2 Working Conference on Programming Concepts and Methods. pp. 347–359. North Holland (1990)
18. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) Reversible Computation, RC '11. LNCS, vol. 7165, pp. 14–29. Springer-Verlag (2012)
19. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. PEPM '07. pp. 144–153. ACM (2007)