Membership Checking in Greatest Fixpoints Revisited

Martin Hofmann and Dulma Rodriguez
Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
mhofmann@ifi.lmu.de and rodrigue@ifi.lmu.de

Abstract

Pierce, in his book "Types and Programming Languages" (MIT Press, 2002), presents an efficient algorithm for computing membership in the greatest fixpoint of *invertible* operators in a goal-directed way. In this paper, we provide a new proof of correctness for it based on coinduction. Moreover, we extend the algorithm for computing membership in the gfp of arbitrary monotone operators and prove this extension correct in a very similar way. Finally, we instantiate the general algorithm to gain a subtyping algorithm for RAJA programs.

1 Introduction

We are interested in computing membership in the greatest fixpoint of a monotone operator on the powerset of some given set. Rather than computing the entire fixpoint by Knaster-Tarski iteration we want to depart from a given goal. This may be advantageous if the size of the underlying set or of the greatest fixpoint is large compared to the portion relevant for determining membership of a particular element. For a concrete example consider the operator $\mathcal{F}(X) = \{x \mid x+1 \mod 5 \in X\}$ on the powerset of $G = \{0,\ldots,2^{100}\}$. Obviously, the largest fixpoint consists of G itself; determining this by Knaster-Tarski iteration is infeasible though. If we only want to check whether a particular element, say 23 is in the gfp we can commence with the goal $23 \in \mathsf{gfp}?$. This leads to the sequence of subgoals $4 \in \mathsf{gfp}?$, $0 \in \mathsf{gfp}?$, $1 \in \mathsf{gfp}?$, $2 \in \mathsf{gfp}?$, $3 \in \mathsf{gfp}?$, $4 \in \mathsf{gfp}?$ at which point we are done because we have discovered a loop in the sequence of subgoals that have arisen.

We are specially interested in deciding subtyping for RAJA types. The RAJA system is a refinement of an extension of Featherweight Java (FJ) [6] with attribute update (FJEU), with the goal of statically analysing the heap space consumption of object-oriented programs. The system has been first described by Hofmann and Jost in [4]. Recently, the current authors analysed algorithmic typing of RAJA programs [5]. Briefly, RAJA types are FJEU classes refined with a possibly infinite set of *views*. Subtyping for RAJA types is defined as the greatest fixpoint of a monotone operator, similarly to the definitions of subtyping for other recursive types like tree types or μ-types [7, Chapter 21].

Subtyping algorithms for recursive types have been widely studied in the past. Amadio and Cardelli gave the first subtyping algorithm for recursive types [1]. Brandt and Henglein's [2] showed the underlying coinductive nature of Amadio and Cardelli's algorithm. In [7, Chapter 21] Pierce gives an overview of many algorithms for membership checking for greatest fixed points and how they can be used to decide subtyping for recursive types.

RAJA subtyping, however, is a bit more complicated than most of the other definitions of subtyping for recursive types because in RAJA methods can have many different types. Therefore, in order to check that a RAJA type C^r is a subtype of a RAJA type D^s we need to check that for a given method m for all its method types in D^s there is a method type in C^r with some properties. This causes that the *support* of a given goal is not a set of subgoals as usual but a boolean combination of subgoals.

In this paper we will extend the efficient algorithm for membership checking for greatest fixed points described in [7, Chapter 21.6] to a more general version where the support of a given goal is a positive boolean expression. Moreover, we provide a new proof of correctness for both algorithms. We found

the proof in [7, Chapter 21.6] difficult to extend and provide therefore a more abstract coinductive proof which can be easily adapted to the new algorithm.

Contents. In Section 2 we describe and prove correct an algorithm for membership checking in greatest fixed points of monotone operators closed under intersection. In Section 3 we extend the algorithm to arbitrary monotone operators. In Section 4 we instantiate the second algorithm in order to decide subtyping for the RAJA system.

2 Invertible Operators

Let G be a set. We let $\mathcal{P}(G)$ denote the powerset of G and $\mathcal{PF}(G)$ denote the set of finite subsets of G. If $\mathcal{F}:\mathcal{P}(G)\to\mathcal{P}(G)$ is a monotone operator we write $\mathsf{gfp}(\mathcal{F})$ for its greatest fixpoint. We have $\mathsf{gfp}(\mathcal{F})=\mathcal{F}(\mathsf{gfp}(\mathcal{F}))$ and whenever $X\subseteq\mathcal{F}(X)$ then $X\subseteq\mathsf{gfp}(\mathcal{F})$. The latter principle is called coinduction. We may also use the notation $\nu X.\mathcal{F}(X)$ for $\mathsf{gfp}(\mathcal{F})$.

In the following we review a goal-directed algorithm for membership checking for greatest fixed points described in [7, Chapter 21.6]. This algorithm works only for a special kind of operators, *invertible operators*, which we now characterize.

A given element $g\in G$ can be generated by a monotone operator $\mathcal{F}$ in many ways, which means that there can be more than one set $X\subseteq G$ such that $g\in\mathcal{F}(X)$. We call any such set a *generating* set for g. We focus here on the class of invertible operators, where each g has at most one minimal generating set.

Definition 2.1. *A monotone operator $\mathcal{F}$ is said to be* invertible *if, for all $g\in G$, the collection of sets*

$$G_g=\{X\subseteq G\mid g\in\mathcal{F}(X)\}$$

is either empty or contains a unique finite member that is a subset of all the others.

When F is invertible, the partial function $\mathsf{support}_{\mathcal{F}}:G\to\mathcal{PF}(G)$ is defined like this:

$$\mathsf{support}_{\mathcal{F}}(g)=\begin{cases}X & \text{if } X\in G_g \text{ and } \forall X'\in G_g.\,X\subseteq X'\\ \uparrow & \text{if } G_g=\emptyset\end{cases}$$

That is, the support of a goal g is the least generating set X for g, or undefined if g is not supported in $\mathcal{F}$.

Definition 2.2. *Let G be a set, $A\subseteq G$, $f:G\to\mathcal{PF}(G)$. A monotone operator $\mathcal{F}_{f,A}$ is defined by*

$$\begin{array}{lcl}\mathcal{F}_{f,A} & : & \mathcal{P}(G)\to\mathcal{P}(G)\\ \mathcal{F}_{f,A}(X) & = & \{g\mid g\in A\wedge f(g)\subseteq X\}\end{array}$$

Then, the support of a goal is given by the function f and it is only defined for elements $g\in A$:

$$\mathsf{support}_{\mathcal{F}_{f,A}}(g)=\begin{cases}f(g) & \text{if } g\in A\\ \uparrow & \mathsf{otherwise}\end{cases}$$

The following result seems to be folklore, well-known e.g. in the field of predicate transformers. The operators $\mathcal{F}_{f,A}$ are equivalent to invertible operators and to monotone operators closed under intersection where every goal has a finite support.

Theorem 2.3. *Let $\mathcal{F}:\mathcal{P}(G)\to\mathcal{P}(G)$ be a monotone operator. The following are equivalent:*

1. *There exists f,A such that $\mathcal{F}=\mathcal{F}_{f,A}$.*
2. *For each $g\in\mathcal{F}(G)$ there exists a finite support set $S\in\mathcal{PF}(G)$ such that $g\in\mathcal{F}(S)$ and for all $X_1,X_2\in\mathcal{PF}(G)$ one has $\mathcal{F}(X_1\cap X_2)=\mathcal{F}(X_1)\cap\mathcal{F}(X_2)$.*
3. *$\mathcal{F}$ is invertible.*

Algorithm 1.

$$
\begin{array}{lcl}
\mathsf{test} & : & G \times \mathcal{P}(G) \to \mathcal{P}(G)_\perp \\
\mathsf{test}(g,U) & = & \mathsf{if}\ g \in U\ \mathsf{then}\ U \\
 & & \mathsf{else\ if}\ g \notin A\ \mathsf{then\ fail} \\
 & & \mathsf{else} \\
 & & \quad \mathsf{let}\ \{h_1,\ldots,h_n\} = f(g)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ V_1 = \mathsf{test}(h_1, U \cup \{g\})\ \mathsf{in} \\
 & & \quad \mathsf{let}\ V_2 = \mathsf{test}(h_2, V_1)\ \mathsf{in} \\
 & & \quad \ldots \\
 & & \quad \mathsf{let}\ V_n = \mathsf{test}(h_n, V_{n-1})\ \mathsf{in} \\
 & & \quad\ V_n
\end{array}
$$

Figure 1: Algorithm for membership checking of greatest fixed points.

Membership checking

Figure 1 shows an algorithm for membership checking in the greatest fixed point of $\mathcal{F}_{f,A}$. The idea of this membership algorithm is to run $\mathcal{F}$ backwards: to check membership for an element g, we need to ask how g could have been generated by $\mathcal{F}$. The advantage of an invertible $\mathcal{F}$ is that there is at most one way to generate a given g. We have to be careful though, a goal g might be supported e.g. by the same goal g. If we do not detect these kind of loops, the algorithm will not terminate. Therefore we keep a set of assumptions U that is empty at the beginning and that will be incremented with every goal we handle. This way we are able to detect a loop if we check whether the current goal is a member of the set of assumptions, in which case we finish with a positive answer. The following algorithm takes a set of assumptions U as an argument and returns another set of assumptions as a result. This allows it to record the subtyping assumptions that have been generated during completed recursive calls and reuse them in later calls. For failure we use the convention: if an expression B fails, then $\mathsf{let}\ A = B\ \mathsf{in}\ C$ also fails.

This algorithm has been described and proved correct in [7, Chapter 21.6]. In [3], Costa Seco and Caires have used it as well for defining subtyping for a class-based object oriented language where classes are first class polymorphic values. We provide here a more abstract correctness proof based on coinduction.

Theorem 2.4.

1. *if G is a finite set the* $\mathsf{test}(g,U)$ *terminates.*
2. $\mathsf{test}(g,\emptyset) = V \iff g \in \nu X.\mathcal{F}_{f,A}(X)$.

Proof.

1. Termination of the algorithm follows using $|G \setminus U|$ as a ranking function.
2. Let $\mathcal{N}(U) := \nu X.\{h \mid h \in U \vee (h \in A \wedge f(h) \subseteq X)\}$. Note that $\mathcal{N}(U) = \nu X.U \cup \mathcal{F}_{F,A}(X)$. Consequently, $\mathcal{N}(\emptyset) = \nu X.\mathcal{F}_{f,A}(X)$. The goal follows then from the more general results:
   (a) $\mathsf{test}(g,U) = V \Rightarrow g \in \mathcal{N}(U)$ and $U \subseteq V \subseteq \mathcal{N}(U)$.
   (b) $\mathsf{test}(g,U) = \mathsf{fail} \Rightarrow g \notin \mathcal{N}(U)$.

which we prove simultaneously by induction on the runtime of the computation of $\mathsf{test}(g,U)$.

Case $g \in U$. Then $\mathsf{test}(g,U) = U$ by definition and $g \in \mathcal{N}(U)$ since $U \subseteq \mathcal{N}(U)$.

Case $g \notin U$ and $g \notin A$. Then $\mathsf{test}(g,U) = \mathsf{fail}$ and $g \notin \mathcal{N}(U)$ since $\mathcal{N}(U) \subseteq U \cup A$.

Case $g \notin U$ and $g \in A$. We consider the representative case $f(g) = \{h_1, h_2\}$.

Case $\mathsf{test}(h_1, U \cup \{g\}) = V_1$ and $\mathsf{test}(h_2, V_1) = V_2$.
Then by induction hypothesis we get $h_1 \in \mathcal{N}(U \cup \{g\})$ and $U \cup \{g\} \subseteq V_1 \subseteq \mathcal{N}(U \cup \{g\})$ and $h_2 \in \mathcal{N}(V_1)$ and $V_1 \subseteq V_2 \subseteq \mathcal{N}(V_1)$. From monotonicity of $\mathcal{N}(.)$ then follows $\mathcal{N}(V_1) \subseteq \mathcal{N}(\mathcal{N}(U \cup \{g\})) = \mathcal{N}(U \cup \{g\})$ easily [1], hence, we get $f(g) \subseteq \mathcal{N}(U \cup \{g\})$ (*).
Next we claim that $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. One direction is clear by monotonicity of $\mathcal{N}(.)$. For the other direction we use coinduction with $X_0 = \mathcal{N}(U \cup \{g\})$. To conclude $X_0 \subseteq \mathcal{N}(U)$ we thus have to prove $X_0 \subseteq U \cup \{h \mid h \in A \wedge f(g) \subseteq X_0\}$ which we now do.
Pick $h \in X_0 = \mathcal{N}(U \cup \{g\})$.
From the definition of $\mathcal{N}(.)$ we get that $h \in U$ or $h = g$ or $f(g) \subseteq X_0$. The first and third case immediately yield the desired result. In the second case ($g = h$) we get $f(g) \subseteq X_0$ from (*). So we proved $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. Then we have $f(g) \subseteq \mathcal{N}(U)$ and $g \in A$, thus, we get the desired $g \in \mathcal{N}(U)$. Moreover, we get $U \subseteq U \cup \{g\} \subseteq \mathcal{N}(U \cup \{g\}) \subseteq \mathcal{N}(U)$.

Case $\mathsf{test}(h_1, U \cup \{g\}) = \mathsf{fail}$. Then $\mathsf{test}(g,U) = \mathsf{fail}$ and by I.H. $h_1 \notin \mathcal{N}(U \cup \{g\})$, thus, $f(g) \nsubseteq \mathcal{N}(U)$ and $g \notin \mathcal{N}(U)$.

Case $\mathsf{test}(h_1, U \cup \{g\}) = V_1$ and $\mathsf{test}(h_2, V_1) = \mathsf{fail}$. Then by induction hypothesis $U \cup \{g\} \subseteq V_1 \subseteq \mathcal{N}(U \cup \{g\})$ and $h_2 \notin \mathcal{N}(V_1)$. Moreover we have by monotonicity of $\mathcal{N}(.)$ that $\mathcal{N}(U) \subseteq \mathcal{N}(U \cup \{g\}) \subseteq \mathcal{N}(V_1)$, thus $h_2 \notin \mathcal{N}(U)$ and consequently $g \notin \mathcal{N}(U)$ as desired.

□

3 Arbitrary Monotone Operators

In this section we extend the previous algorithm to an algorithm for membership checking in the greatest fixpoint of not necessarily invertible monotone operators, where the support of a given goal is the meaning of some positive boolean expression. As mentioned in the introduction, this extension is motivated by the subtyping relation of the RAJA system. In the following we describe formally positive boolean expressions and their meaning.

Definition 3.1. Positive boolean expressions *over G are defined by the grammar*

$$e ::= \mathsf{tt} \mid \mathsf{ff} \mid g \mid e_1 \wedge e_2 \mid e_1 \vee e_2$$

where g ranges over elements of G. Let $\mathsf{PBool}(G)$ *be the set of positive boolean expressions over G.*

Positive boolean expressions denote predicates on $\mathcal{P}(G)$. In particular, g denotes $\{X \mid g \in X\}$. Formally, if $X \subseteq G$ we define the *meaning* $[\![e]\!]^X : \mathsf{bool}$ as follows:

$$\begin{array}{lcl} [\![\mathsf{tt}]\!]^X & = & \mathsf{tt} \\ [\![\mathsf{ff}]\!]^X & = & \mathsf{ff} \\ [\![g]\!]^X & = & g \in X \\ [\![e_1 \wedge e_2]\!]^X & = & [\![e_1]\!]^X \wedge [\![e_2]\!]^X \\ [\![e_1 \vee e_2]\!]^X & = & [\![e_1]\!]^X \vee [\![e_2]\!]^X \end{array}$$

[1] $\mathcal{N}(\mathcal{N}(U)) = \mathcal{N}(U)$ follows by monotonicity of $\mathcal{N}(.)$ and coinduction.

Example 3.2. *Let $G = \{a,b,c,d\}$ and $e = a \wedge (b \vee c)$, then $[\![e]\!]^{\{a,b\}} = \mathsf{tt}$ and $[\![e]\!]^{\{b,c\}} = \mathsf{ff}$.*

Note that $X \subseteq Y$ implies $[\![e]\!]^X \Rightarrow [\![e]\!]^Y$.

Definition 3.3. *Let $f : G \to \mathsf{PBool}(G)$ be a boolean operator. Then we obtain a monotone operator $\mathcal{F}_f$ as follows:*

$$\begin{array}{lcl} \mathcal{F}_f & : & \mathcal{P}(G) \to \mathcal{P}(G) \\ X & \mapsto & \{g \mid [\![f(g)]\!]^X = \mathsf{tt}\} \end{array}$$

Next we prove constructively that, whenever a set G is finite, we can provide a boolean operator for any monotone operator over G. We notice though that the so constructed boolean operator might be very big, hence, applying the algorithm we are about to describe would be very inefficient.

Theorem 3.4. *If G is a finite set and $\mathcal{F} : \mathcal{P}(G) \to \mathcal{P}(G)$ then there exists $f : G \to \mathsf{PBool}(G)$ such that $\mathcal{F} = \mathcal{F}_f$.*

Proof. For each (finite) subset $X = \{g_1, \ldots, g_k\} \subseteq G$ define $\bigwedge X := g_1 \wedge \ldots \wedge g_k$. We have $[\![\bigwedge X]\!]^Y = \mathsf{tt} \iff X \subseteq Y$. Given g let $X_1 \ldots X_k$ be an enumeration of the subsets X such that $g \in \mathcal{F}(X)$. We then put $f(g) = \bigwedge X_1 \ \vee \ldots \vee \ \bigwedge X_n$. Now $g \in \mathcal{F}(X) \Rightarrow X = X_i$ for some $i \ \Rightarrow [\![\bigwedge X_i]\!]^X = \mathsf{tt} \Rightarrow [\![f(g)]\!]^X = \mathsf{tt}$. Conversely $[\![f(g)]\!]^X = \mathsf{tt} \Rightarrow X_i \subseteq X$ for some $i \ \Rightarrow g \in \mathcal{F}(X_i) \Rightarrow g \in \mathcal{F}(X)$ by monotonicity. □

For invertible operators we can provide a boolean operator directly. Given $f : G \to \mathcal{P}(G)$ as in the last section and $A \subseteq G$, define $\tilde{f}$ as follows:

$$\tilde{f}(g) = \begin{cases} \mathsf{ff} & \text{if } g \notin A \\ \bigwedge f(g) & \text{if } g \in A \end{cases}$$

Then $[\![\tilde{f}(g)]\!]^X = \mathsf{tt} \iff g \in A \wedge f(g) \subseteq X$, hence, $\mathcal{F}_{\tilde{f}}(X) = \mathcal{F}_{f,A}(X)$.

Membership checking

Figure 2 shows a new algorithm for membership in the gfp of arbitrary monotone operators whenever a boolean operator $f : G \to \mathsf{PBool}(G)$ is given. *Algorithm 2* takes a set of assumptions U as an argument and returns another set of assumptions and a boolean as a result. The difference to the first algorithm is that if the meaning of the support of a goal is tt, then the new computed set of assumptions will be returned; otherwise it will be dropped. Moreover, ff branches do not lead immediately to rejection. They can lead to a positive answer if combined by "or" with a tt branch. In the following we prove correctness and termination of the algorithm. If the basic set is finite the algorithm will terminate and the result will be correct. Otherwise, even if the basic set is infinite, if the computation of the support of a goal do not lead to an infinite chain of new goals, then the algorithm will terminate as well with a correct answer.

Theorem 3.5. *Let $f : G \to \mathsf{PBool}(G)$ and* test *defined as above. Let $\mathcal{N}(U) = \nu X . U \cup \mathcal{F}_f(X)$.*

1. *If $\mathsf{test}(e,U) = (b,V)$ then $[\![e]\!]^{\mathcal{N}(U)} = b$ and $U \subseteq V \subseteq \mathcal{N}(U)$.*
2. *If for each g there exists a finite set S such that $f(S) \subseteq \mathsf{PBool}(S)$ and $g \in S$ then $\mathsf{test}(g,\emptyset)$ terminates.*

Proof. 2. follows using $|S \setminus U|$ as a ranking function. For 1. we induct on the runtime of $\mathsf{test}(e,U)$ and – subordinately – on the structure of e. We note that for all $U \subseteq G$ we have $U \subseteq \mathcal{N}(U)$, $\mathcal{N}(U) = \mathcal{N}(\mathcal{N}(U))$.

Algorithm 2. Let $* \in \{\wedge, \vee\}$:

$$
\begin{array}{rcl}
\mathsf{test} & : & \mathsf{PBool}(G) \times \mathcal{P}(G) \to \mathsf{bool} \times \mathcal{P}(G) \\
\mathsf{test}(e_1 * e_2, U) & = & \mathsf{let}\ (b_1, V_1) = \mathsf{test}(e_1, U)\ \mathsf{in} \\
 & & \mathsf{let}\ (b_2, V_2) = \mathsf{test}(e_2, V_1)\ \mathsf{in} \\
 & & (b_1 * b_2, V_2) \\
\mathsf{test}(g, U) & = & \mathsf{if}\ g \in U\ \mathsf{then}\ (\mathsf{tt}, U) \\
 & & \mathsf{else\ let}\ (b, V) = \mathsf{test}(f(g), U \cup \{g\})\ \mathsf{in} \\
 & & \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{tt}, V)\ \mathsf{else}\ (\mathsf{ff}, U)
\end{array}
$$

Figure 2: Algorithm for membership checking in the gfp of arbitrary monotone operators.

Case $e = e_1 * e_2$. Write $(b_1, V_1) = \mathsf{test}(e_1, U)$ and $(b_2, V_2) = \mathsf{test}(e_2, V_1)$.

Inductively, we have $b_1 = [\![e_1]\!]^{\mathcal{N}(U)}$ and $U \subseteq V_1 \subseteq \mathcal{N}(U)$. Therefore, $\mathcal{N}(U) \subseteq \mathcal{N}(V_1) \subseteq \mathcal{N}(\mathcal{N}(U)) = \mathcal{N}(U)$, and thus $\mathcal{N}(V_1) = \mathcal{N}(U)$. It follows that $b_2 = [\![e_2]\!]^{\mathcal{N}(U)}$ and $U \subseteq V_1 \subseteq V_2 \subseteq \mathcal{N}(U)$. The claim then follows.

Case $e = g$.

Case $g \in U$. Then $\mathsf{test}(g, U) = (\mathsf{tt}, U)$ and obviously $[\![g]\!]^{\mathcal{N}(U)} = \mathsf{tt}$ and $U \subseteq \mathcal{N}(U)$.

Case $g \notin U$. Write $(b, V) = \mathsf{test}(f(g), U \cup \{g\})$. Inductively, we have $U \cup \{g\} \subseteq V \subseteq \mathcal{N}(U \cup \{g\})$ and $b = [\![f(g)]\!]^{\mathcal{N}(U \cup \{g\})}$.

We claim that $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. One direction is clear by monotonicity of $\mathcal{N}(.)$. For the other direction we use coinduction with $X = \mathcal{N}(U \cup \{g\})$. To conclude $X \subseteq \mathcal{N}(U)$ we have to prove $X \subseteq U \cup \{h \mid [\![f(h)]\!]^X = \mathsf{tt}\}$ which we now do.

Pick $h \in X = \mathcal{N}(U \cup \{g\})$.

From the definition of $\mathcal{N}(.)$ we get that $h \in U$ or $h = g$ or $[\![f(h)]\!]^X = \mathsf{tt}$. The first and third case immediately yield the desired result. In the second case ($g = h$) we get $[\![f(h)]\!]^X = \mathsf{tt}$ from the induction hypothesis. So we proved $\mathcal{N}(U) = \mathcal{N}(U \cup \{g\})$. The result is now direct from the definitions.

□

Corollary 3.6. $\mathsf{test}(e, \emptyset) = (b, \_)$ *iff* $[\![e]\!]^{\mathsf{gfp}(\mathcal{F}_f)} = b$.

4 Applications

In this section we consider a special application of the last algorithm. As we already mentioned we are specially interested in computing subtyping for RAJA types. In the following we give a brief and simplified introduction to the RAJA system and show how to instantiate the generic *Algorithm* 2 to gain a RAJA subtyping algorithm.

RAJA programs are annotated FJEU programs, created with the goal of statically analysing their heap space consumption. An FJEU program $\mathscr{C}$ is a partial finite map from class names to class definitions. Classes contain attributes and methods. The RAJA type system is a refinement of the FJEU type system.

A *refined (class) type* consists of a class C and a *view* r and is written C^r. The meaning of views is given by three maps $\Diamond()$, defining potentials, A, defining views of attributes, and M, defining refined method types. More precisely, $\Diamond() : \text{Class} \times \text{View} \to \mathbb{Q}^+$ assigns each class its potential according to the employed view. Next, $\mathsf{A} : \text{Class} \times \text{View} \times \text{Field} \to \text{View}$ determines the refined types of the fields. Finally, $\mathsf{M} : \text{Class} \times \text{View} \times \text{Method} \to \mathcal{P}(\text{Views of Arguments} \to \text{View of Result})$ assigns refined types to methods. We allow polymorphism in the sense that one method may have more than one (or no) refined typing. For more details and concrete examples we refer to [4, 5, 8].

Now we describe a simplified version of subtyping for RAJA types. The simplification disregards subclasses and potentials but shows the need for going beyond invertible operators. Let RT be the set of RAJA types. We define a monotone operator $\mathcal{F} : \mathcal{P}(\mathsf{RT} \times \mathsf{RT}) \to \mathcal{P}(\mathsf{RT} \times \mathsf{RT})$ as follows:

$$\begin{array}{rcll} \mathcal{F}(X) & = & \{\,(C^r, D^s) & \mid \quad \forall \text{ attributes } a \;.\; \mathsf{A}(C^r,a) = E^p,\, \mathsf{A}(D^s,a) = E^q \;.\; (E^p, E^q) \in X \\ & & & \quad\; \forall \text{ methods } \; m \;.\; \forall (E_1^{\beta_1}, \ldots, E_j^{\beta_j} \to E_0^{\beta_0}) \in \mathsf{M}(D^s,m) \;. \\ & & & \qquad \exists (E_1^{\alpha_1}, \ldots, E_j^{\alpha_j} \to E_0^{\alpha_0}) \in \mathsf{M}(C^r,m) \;. \\ & & & \qquad\quad (E_1^{\beta_1}, E_1^{\alpha_1}) \in X, \ldots, (E_j^{\beta_j}, E_j^{\alpha_j}) \in X, (E_0^{\alpha_0}, E_0^{\beta_0}) \in X \,\} \end{array}$$

Then $C^r <: D^s \iff (C^r, D^s) \in \nu X.\mathcal{F}(X)$. Now, in order to apply *Algorithm* 2, we define a function $f : \mathsf{RT} \times \mathsf{RT} \to \mathsf{PBool}(\mathsf{RT} \times \mathsf{RT})$ so that $\mathcal{F}(X) = \mathcal{F}_f(X)$:

$$\begin{array}{rcl} f(C^r, D^s) & = & \bigwedge_a (E^p, E^q) \wedge \\ & & \bigwedge_m \bigwedge_{E_1^{\beta_1}, \ldots, E_j^{\beta_j} \to E_0^{\beta_0}} \bigvee_{E_1^{\alpha_1}, \ldots, E_j^{\alpha_j} \to E_0^{\alpha_0}} (E_1^{\beta_1}, E_1^{\alpha_1}) \wedge \ldots \wedge (E_j^{\beta_j}, E_j^{\alpha_j}) \wedge (E_0^{\alpha_0}, E_0^{\beta_0}) \end{array}$$

5 Conclusions

In this paper we extended the algorithm for membership checking for greatest fixed points described in [7, Chapter 21.6] to a more general version where the support of a given goal is a positive boolean expression. For finite sets this generalization encompasses all monotone operators. Next, we provided a new coinductive correctness proof for both algorithms. Finally, we instantiated the general membership algorithm in order to compute subtyping for RAJA types in a goal-directed way.

We believe that our new algorithm can be useful for computing subtyping for other refinement systems that also provide multiple types to methods. As part of a prototype implementation of the RAJA system the algorithm has been implemented in Ocaml and we work currently in a formalization of its correctness proof in the theorem prover Coq.

Acknowledgements

We acknowledge support by the EU integrated project MOBIUS IST 15905 and by the DFG Graduiertenkolleg 1480 Programm- und Modell-Analyse (PUMA). We also thank Andreas Abel for valuable comments.

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Program. Lang. and Syst.*, 15(4):575–631, 1993.

[2] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.*, 33(4):309–338, 1998.

[3] J. Costa Seco and L. Caires. Subtyping first-class polymorphic components. In S. Sagiv, ed., *Proc. of 14th Europ. Symp. on Programming, ESOP 2005 (Edinburgh, Apr. 2005)*, v. 3444 of *Lect. Notes in Comput. Sci.*, pp. 342–356. Springer, 2005.

[4] M. Hofmann and S. Jost. Type-based amortised heap-space analysis (for an object-oriented language). In P. Sestoft, ed., *Proc. of 15th Europ. Symp. on Programming, ESOP 2006 (Vienna, March 2006)*, v. 3924 of *Lect. Notes in Comput. Sci.*, pp. 22–37. Springer, 2006.

[5] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proc. of 23rd Int. Wksh. on Computer Science Logic, CSL 2009 (Coimbra, Sept. 2009)*, *Lect. Notes in Comput. Sci.*, Springer, to appear.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. on Program. Lang. and Syst.*, 23(3):396–450, 2001.

[7] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[8] Raja. `http://raja.tcs.ifi.lmu.de`.