Spy Game: Verifying a Local Generic Solver in Iris

Paulo Emílio de Vilhena, François Pottier, Jacques-Henri Jourdan

▶ **To cite this version:**

Paulo Emílio de Vilhena, François Pottier, Jacques-Henri Jourdan. Spy Game: Verifying a Local Generic Solver in Iris. Proceedings of the ACM on Programming Languages, 2020, 4 (POPL), 10.1145/3371101 . hal-02351562

HAL Id: hal-02351562
https://hal.science/hal-02351562

Submitted on 6 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Spy Game: Verifying a Local Generic Solver in Iris*

PAULO EMÍLIO DE VILHENA, Inria, France
FRANÇOIS POTTIER, Inria, France
JACQUES-HENRI JOURDAN, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, France

We verify the partial correctness of a "local generic solver", that is, an on-demand, incremental, memoizing least fixed point computation algorithm. The verification is carried out in Iris, a modern breed of concurrent separation logic. The specification is simple: the solver computes the optimal least fixed point of a system of monotone equations. Although the solver relies on mutable internal state for memoization and for "spying", a form of dynamic dependency discovery, it is apparently pure: no side effects are mentioned in its specification. As auxiliary contributions, we provide several illustrations of the use of prophecy variables, a novel feature of Iris; we establish a restricted form of the infinitary conjunction rule; and we provide a specification and proof of Longley's *modulus* function, an archetypical example of spying.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**.

Additional Key Words and Phrases: separation logic, prophecy variables, least fixed point, program verification

ACM Reference Format:
Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy Game: Verifying a Local Generic Solver in Iris. 1, 1 (January 2020), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 The Problem

The problem of computing the least solution of a system of monotone equations often arises in the analysis of objects that have cyclic or recursive structure, such as grammars, control flow graphs, transition systems, and so on. For instance, in the analysis of a context-free grammar, determining which symbols generate an empty language, determining which symbols can derive the empty word, and computing the "first" and "follow" sets of each symbol, are four problems that can be formulated in this manner. Many other examples can be found in the area of static program analysis, where it has long been understood that most forms of static program analysis are in fact least fixed point computations. Early references include Kildall [1973], Kam and Ullman [1976], and Cousot and Cousot [1977].

The problem is traditionally stated as follows: compute the least fixed point of a monotone function $\mathcal{E}$ of type $(\mathcal{V} \to \mathcal{P}) \to (\mathcal{V} \to \mathcal{P})$, where $\mathcal{V}$ is an arbitrary type of "variables" and $\mathcal{P}$ is a type of "properties" equipped with a partial order $\leq$ and a least element $\bot$. One can think of $\mathcal{E}$ as a "system of equations" because to every variable it associates a "right-hand side" of type $(\mathcal{V} \to \mathcal{P}) \to \mathcal{P}$ — something that can be evaluated to a property, under an environment that maps variables to properties.

When the problem is stated in this way, it must be accompanied with a sufficient condition for a least fixed point to exist. Instead, we find it beneficial to rephrase it in a manner that is both simpler

*This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

Authors' addresses: Paulo Emílio de Vilhena, Inria, France, paulo-emilio.de-vilhena@inria.fr; François Pottier, Inria, France, francois.pottier@inria.fr; Jacques-Henri Jourdan, Université Paris-Saclay, CNRS, Laboratoire de recherche en informatique, 91405, Orsay, France, jacques-henri.jourdan@lri.fr.

2020. XXXX-XXXX/2020/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and more general, as follows: compute the *optimal least fixed point* of a monotone function $\mathcal{E}$ of type $(\mathcal{V} \to \mathcal{P}) \to (\mathcal{V} \to \mathcal{P})$, where $\mathcal{V}$ and $\mathcal{P}$ are as above.

The notion of an optimal least fixed point, which is defined further on (§2), can be roughly described as follows. First, it is a partial function: it is not necessarily defined on all of $\mathcal{V}$, but only on a subset of $\mathcal{V}$. Second, it is a *partial fixed point* of $\mathcal{E}$, in a sense to be made precise later on. Third, it is smaller (with respect to the partial order $\leq$) than every partial fixed point, where their domains overlap. Finally, among all partial functions that satisfy these three properties, it is *optimal*: that is, it is the one whose domain is largest.

We prove that every function $\mathcal{E}$ admits a unique optimal least fixed point. Thus, the rephrased problem does not need to be accompanied with an existence side condition. Furthermore, we prove that the optimal least fixed point coincides with the least fixed point when the existence of the latter is guaranteed either by the Knaster-Tarski theorem or by Kleene's fixed point theorem. Thus, for most practical purposes, the rephrased problem is indeed more general than the original problem.

1.2 Local Generic Solvers

Because this problem is so central and pervasive, several authors have proposed algorithms that solve it in a generic way. These algorithms, or *solvers*, are parameterized with a type $\mathcal{V}$, with a partially ordered type $\mathcal{P}$, and with a monotone function $\mathcal{E}$ of type $(\mathcal{V} \to \mathcal{P}) \to (\mathcal{V} \to \mathcal{P})$. This makes them applicable in a wide range of situations. Such solvers have been proposed by Le Charlier and Van Hentenryck [1992], Vergauwen, Wauman, and Lewi [1994], and by Fecht and Seidl [1999], who give a pleasant presentation of a solver as a higher-order function in the programming language ML. In unpublished work, Pottier [2009] describes an OCaml implementation of a solver that is closely related to Vergauwen, Wauman and Lewi's solver and to Fecht and Seidl's solver W.

The solvers cited above are of particular interest in that they are *local*. Instead of eagerly attempting to compute the value $\bar{\mu}\mathcal{E}(v)$ at every variable v, they perform no computation at all up front, and return a first-class function *get* of type $\mathcal{V} \to \mathcal{P}$, whose specification is simple: *get implements* $\bar{\mu}\mathcal{E}$. In other words, when *get* is applied to a variable v, it returns $\bar{\mu}\mathcal{E}(v)$, if the variable v is in the domain of the partial function $\bar{\mu}\mathcal{E}$; otherwise, it diverges. The function call $get(v)$ performs as little work as possible: the value $\bar{\mu}\mathcal{E}(w)$ is computed for every variable w in a certain subset W of $\mathcal{V}$ that the solver strives to keep as small as possible. In an informal sense, W is a set of variables which v directly or indirectly *depends* upon. The solvers cited above perform *dynamic dependency discovery*: instead of explicitly asking the user for dependency information, they *spy* on the user's implementation of $\mathcal{E}$ and dynamically record dependency edges $v \to w$. Furthermore, once the value $\bar{\mu}\mathcal{E}(w)$ has been computed at some variable w, it is *memoized*, so that it never needs to be recomputed. In short, a local solver is *on-demand*, *incremental*, and *memoizing*.

These solvers have imperative implementations: indeed, memoization and spying both rely on long-lived mutable state, which persists across invocations by the user of solver functions. (There are in fact two such functions, whose lifetimes are distinct, namely *get* and *request*. The latter is presented in §5, where we explain the solver in detail.) Yet, if the solver is correctly implemented, none of these side effects can be observed by the user. In fact, the specification of *get* that was proposed earlier, namely *get implements* $\bar{\mu}\mathcal{E}$, mentions no side effect: in such a case, we say that *get* is *apparently pure*. (A similar specification, not shown yet, can be given to *request*.) If one can prove, in a formal sense, that a solver satisfies these specifications, then this guarantees that it is indeed sound for the user to reason under the illusion that the solver is side-effect-free.

1.3 Our Result and its Limitations

In this paper, we formally verify that a slightly simplified version of Pottier's solver [2009; 2019] satisfies the specification sketched above. We carry out a machine-checked proof in the setting of

Iris [Jung et al. 2018], a powerful concurrent separation logic. This shows that the solver is safe (that is, combining it with valid user code cannot result in an illegal memory access or a data race) and correct (that is, it computes the optimal least fixed point of the user-supplied function $\mathcal{E}$).

Because Iris is a logic of partial correctness, our result comes with two limitations: it guarantees neither termination nor deadlock-freedom. Although, in a certain technical sense, these are two facets of the same problem,[1] let us discuss each of these limitations separately.

- A call to *get* can fail to terminate. There are two reasons why this is so. First, the optimal least fixed point $\bar{\mu}\mathcal{E}$ is a partial function, whose domain $dom(\bar{\mu}\mathcal{E})$ can be a proper subset of $\mathcal{V}$. The function call $get(v)$ *will* diverge if the variable v lies outside $dom(\bar{\mu}\mathcal{E})$. We are able to prove this fact, under its contrapositive form: our specification of *get* guarantees that, if $get(v)$ returns, then $v \in dom(\bar{\mu}\mathcal{E})$ holds. Second, even when v lies in $dom(\bar{\mu}\mathcal{E})$, the solver can still diverge; this can happen, for instance, if the partial order has infinite ascending chains.
- A call to *get* (or *request*) can in some circumstances cause a deadlock. Indeed, even though the solver is inherently sequential, it makes use of several locks, which serve as a runtime mechanism to defend against certain patterns of misuse. Quite obviously, if there are multiple user threads, then these locks forbid *concurrent* calls to the solver: in other words, they rule out data races on the solver's internal mutable state. More subtly, even if there is only one user thread, they forbid *reentrant* calls, such as a call to *get* that takes place while an earlier call to *get* is still ongoing. (Although such a scenario may seem contrived and unlikely to arise, it is technically feasible, via recursion through the store.) In such an event, the solver will attempt to acquire a lock twice, resulting in a deadlock.

The main reason why we use locks is that this allows us to equip the solver with a very simple specification, where no side effects are mentioned at all. We believe that, if desired, we could verify a variant of the solver that does not use any lock, and therefore runs no risk of a deadlock. This solver would come with a more restrictive specification, which enforces a sequential use of the solver functions by the client. This would be achieved by giving out and reclaiming unique permissions to call the solver functions.

There are two main reasons why we do not guarantee termination. One reason is that, in order to ensure termination, we would have to impose a certain requirement on $\mathcal{V}, \mathcal{P}, \mathcal{E}$, and it seems difficult to pick such a requirement without abandoning some generality. For instance, requiring $\mathcal{V}$ to be finite and requiring $\mathcal{P}$ to have finite ascending chains suffices to ensure termination, but there are reasonable use cases for the solver where $\mathcal{V}$ is infinite, so this condition seems too restrictive. Another reason is that, although there exists a variant of Iris that imposes total correctness and therefore can be used to guarantee termination, this variant currently does not support prophecy variables [Jung et al. 2020], a feature on which our proof crucially relies. We imagine that it should be possible to extend it with support for prophecy variables, but this could require nontrivial work.

1.4 Challenges and Previous Work

The solvers cited earlier (§1.2) compute a least fixed point via chaotic iteration, a well-known technique whose correctness is not difficult to establish; see, for example, Besson *et al.* [2009]. However, several challenges arise from the fact that these are *local generic solvers*, which exhibit a compact and apparently pure API, yet rely on a number of subtle internal tricks, such as memoization and spying. More specifically,

[1] Locks in Iris/HeapLang are not primitive. They are implemented in a library, and the operation of acquiring a lock spins until the lock is available. Thus, a "deadlock" manifests itself as an infinite loop. In a real-world implementation, depending on how locks are implemented by the programming language and operating system, some deadlocks may be detected and replaced with a graceful runtime failure. In the paper, we ignore this aspect and speak only of "deadlocks".

- The solver presents itself as a function *lfp* of type $((\mathcal{V} \to \mathcal{P}) \to (\mathcal{V} \to \mathcal{P})) \to \mathcal{V} \to \mathcal{P}$. This is a third-order function type, which means that there are three nested levels of interaction between the user and the solver. This interaction goes roughly as follows. Initially, the user applies *lfp* to a user function *eqs*, which represents a system of equations $\mathcal{E}$. The solver immediately returns a function *get*, which in the user's eyes represents the optimal least fixed point $\bar{\mu}\mathcal{E}$. Later on, whenever desired, the user may invoke *get*. This in turn can cause calls by the solver to the user function *eqs*. This function is passed a solver function *request*.
- The functions passed by the solver to the user, namely *get* and *request*, have internal side effects on long-lived mutable state. Yet, these effects are not mentioned in the specification. So, one must somehow prove that it is safe for the user *not* to be aware of them. Furthermore, *get* and *request* are first-class functions, which the user can store and call at any time, from a single thread or from multiple threads; one must somehow prove that this is safe.
- Long-lived mutable state is used for two distinct purposes, namely memoization and spying. Whereas memoization is a well-understood technique, whose verification in a concurrent separation logic is straightforward, spying seems much more challenging. The solver spies on the user function *eqs* by recording a list of variables at which *eqs* invokes the solver function *request*. Once *eqs* returns, the solver deduces that the set of variable-property pairs that have been exposed to the user by these calls to *request* is sufficient for *eqs* to determine its result. In other words, the result of *eqs* does not *depend* on any information beyond this set of variable-property pairs. This is a dynamic dependency discovery technique.

Spying has been studied by Longley [1999], who proposes dynamic dependency discovery as a typical application of this technique. He presents a third-order function *modulus* as an example. He describes the desired denotational semantics of this function, and argues that this function is pure. (If it wasn't, it would not admit such a semantics in the first place.) He shows how to implement *modulus* using imperative features, but does not actually prove that this imperative implementation has the desired semantics.

The problem of verifying the correctness of a local generic solver has been offered by Pottier [2009] as a challenge in an unpublished yet publicly available manuscript. He writes: "how and why encapsulation [of the solver's internal mutable state] is properly achieved is somewhat subtle, and would deserve formal verification." He offers an informal specification, but no proof.

A step towards addressing Pottier's challenge is taken by Hofmann, Karbyshev and Seidl [2010a], who use Coq to verify a simplified model of the local generic solver RLD [Seidl et al. 2012]. However, their work exhibits a number of limitations. Both the solver and its client are modeled in Coq in terms of functions, relations, and explicit state-passing. In particular, the client is modeled as a *strategy tree*, which effectively means that it must be written in monadic style, in a monad that allows querying the solver (in the same way that we allow *eqs* to invoke *request*) but does not allow any side effects. (Examples of side effects that could be useful in practice, but are thus forbidden, include logging, concurrency, or invoking another instance of the solver.) Furthermore, Hofmann *et al.*'s simplified model of the solver is not on-demand, incremental, or memoizing. In this model, the client invokes the solver just once, passing a set V of variables of interest. The solver replies with a set of value-property pairs $(v, \bar{\mu}\mathcal{E}(v))$, where v ranges over V. There is no memoization, therefore no need to reason about it, and no danger of concurrent or reentrant invocations.

1.5 Contributions

In this paper, we make the following contributions:

- We propose a specification of a local generic solver in Iris [Jung et al. 2018]. We believe that this (third-order!) specification is remarkably simple and permissive. It is apparently pure:

no side effects are mentioned. This has several consequences. On the one hand, the fact that the solver is *guaranteed* to be apparently pure means that the user need not worry about the solver's internal use of mutable state. On the other hand, the fact that the user function *eqs* is *required* to be apparently pure means that it can have internal side effects, provided one can prove in Iris that *eqs* implements a mathematical function $\mathcal{E}$.

- Instead of imposing a strong condition to guarantee the existence of a total least fixed point, our specification relies on the concept of an optimal least fixed point, which exists as soon as the partially ordered set $(\mathcal{P}, \leq)$ admits a least element $\bot$ and the user-supplied function $\mathcal{E}$ is monotone. In particular, we do not require $\mathcal{V}$ to be finite, and do not require $\mathcal{P}$ to satisfy the ascending chain condition. The theory of optimal least fixed points, a generalization of Charguéraud's theory of optimal fixed points [2010b], is a contribution of our paper.
- We make the simple yet novel remark that a local solver can use Longley's *modulus* [1999] off the shelf. We split the solver's implementation into two components, namely *modulus*, a stand-alone function, and *lfp*, which uses *modulus*. We also split the correctness proof: we verify each component independently. This requires proposing a suitable Iris specification for *modulus*, proving that *modulus* satisfies this specification, and exploiting solely this specification while reasoning about the use of *modulus* inside *lfp*. As far as we know, this is the very first proof of correctness of the imperative implementation of *modulus*.
- In the proof of *modulus*, we feel a need to exploit an *infinitary conjunction rule*, that is, a reasoning rule that states that $\forall x.\ \{P\}\ e\ \{y.\ Q\,x\,y\}$ implies $\{P\}\ e\ \{y.\ \forall x.Q\,x\,y\}$. Unfortunately, this rule is unsound; in fact, even the binary *conjunction rule*, which states that $\{P\}\ e\ \{y.\ Q_1\,y\}$ and $\{P\}\ e\ \{y.\ Q_2\,y\}$ imply $\{P\}\ e\ \{y.\ Q_1\,y \wedge Q_2\,y\}$, is unsound in Iris, due to its interaction with ghost state. We work around this issue by establishing a weak variant of the infinitary conjunction rule where the postcondition $Q\,x\,y$ is required to be a pure assertion. So far, such a conjunction rule has been missing in Iris, which is why we believe that this result and its proof are of independent interest.
- In the proof of the restricted infinitary conjunction rule (§6) and in the proofs of *modulus* (§7) and *lfp* (§8), we exploit *prophecy variables*, a recent feature of Iris [Jung et al. 2020]. Thus, these proofs offer several original applications of prophecy variables. Furthermore, we make two improvements to Iris's prophecy variable API. We extend it with a new rule for disposing of a prophecy variable, and we further extend it with support for typed prophecy variables. These extensions are defined as libraries on top of Iris's basic prophecy variable API: this shows that they are sound. We find both of them to be useful in our proofs: they allow us to work at a higher level of abstraction.
- In the end, we provide a proof that *modulus* and *lfp* satisfy the proposed Iris specifications. Our proof is machine-checked in Coq[2] and is available online [de Vilhena et al. 2020]. As far as we know, this is the first proof of a local generic solver implementation, expressed in a programming language that allows side effects, including mutable state and concurrency, both in the solver's implementation and in user code outside the solver. Thanks to the modularity of separation logic, the verified solver can be modularly combined with verified client code.

The paper is laid out as follows. We begin with background material on the theory of optimal fixed points, including optimal least fixed points (§2), followed with background material on Iris and prophecy variables, including improvements to Iris's prophecy variable API (§3). Then, we present the specification in Iris of a local generic solver (§4), followed with an overview of the

[2]We make use of two classical reasoning principles, namely the law of excluded middle and Hilbert's operator. Both are used in the theory of optimal (least) fixed points (§2). The law of the excluded middle is also used on several occasions when reasoning about the code (§6).

code of our solver (§5). We move on to our proof in Iris of the solver, which is organized in three main steps, namely the justification of a restricted infinitary conjunction rule (§6), the proof that the function *modulus* satisfies a certain specification (§7), and the proof that the solver satisfies its specification (§8). Finally, we discuss our results, the related work, and future work (§9, §10, §11).

2 A THEORY OF OPTIMAL LEAST FIXED POINTS

In this section, we define the notion of optimal least fixed point, a generalization of Charguéraud's optimal fixed point [2010b] to a setting where the type $\mathcal{P}$ is equipped with a partial order. Throughout this section, $\mathcal{V}$ is an arbitrary type, $\mathcal{P}$ is an arbitrary inhabited type, and $\mathcal{E}$ is an arbitrary function of type $(\mathcal{V} \to \mathcal{P}) \to (\mathcal{V} \to \mathcal{P})$. The material in §2.1 is taken from Charguéraud [2010b], whereas the content of §2.2 is new.

2.1 Optimal Fixed Points

Definition 2.1 (Partial function). A *partial function* $\bar{f}$ is a pair (f, D) of a total function f of type $\mathcal{V} \to \mathcal{P}$ and its domain D, a subset of $\mathcal{V}$.

Following Charguéraud, we write $dom(\bar{f})$ for the right projection of $\bar{f}$, and write just f for its left projection. We write $f =_D g$ as a short-hand for $\forall v \in D.\ f\,v = g\,v$, which means that the functions f and g agree on D. We write $\mathcal{V} \hookrightarrow \mathcal{P}$ for the type of partial functions of $\mathcal{V}$ to $\mathcal{P}$.

A partial function $\bar{f}$ *extends* a partial function $\bar{g}$, which we write $\bar{g} \sqsubseteq \bar{f}$, if $dom(\bar{g})$ is a subset of $dom(\bar{f})$ and $f =_{dom(\bar{g})} g$ holds. The partial functions $\bar{f}$ and $\bar{g}$ are *equivalent*, which we write $\bar{f} \equiv \bar{g}$, if $\bar{f}$ extends $\bar{g}$ and $\bar{g}$ extends $\bar{f}$.

Definition 2.2 (Partial fixed point). A partial function $\bar{f}$ is a *partial fixed point* if for every function g, the agreement $g =_{dom(\bar{f})} f$ implies $g =_{dom(\bar{f})} \mathcal{E}\,g$.

That is, a partial function $\bar{f}$ is a partial fixed point if every total function that extends $\bar{f}$ satisfies the fixed point equation on the domain of $\bar{f}$.

Definition 2.3 (Consistency). A partial fixed point $\bar{f}$ is *consistent* if for every partial fixed point $\bar{g}$, $f =_{dom(\bar{f}) \cap dom(\bar{g})} g$ holds.

Thus, a consistent partial fixed point is one that agrees with every other partial fixed point where their domains overlap. Charguéraud says "generally consistent"; we say "consistent" for brevity.

Definition 2.4 (Optimal fixed point). Among the consistent partial fixed points, the *optimal fixed point* is the one that is greatest with respect to the relation $\sqsubseteq$.

In other words, the optimal fixed point is a consistent partial fixed point that extends every other consistent partial fixed point. It always exists: it can be constructed as the union of all consistent partial fixed points, which is itself a consistent partial fixed point. It is unique up to equivalence $\equiv$.

This concludes our review of the theory of optimal fixed points. One key limitation of the notion of optimal fixed point is that it is meant to be used in situations where the fixed point is unequivocally defined. If two partial fixed points disagree at some point $v \in \mathcal{V}$, then v definitely *cannot* be in the domain of the optimal fixed point. Thus, this notion is of no use in situations where there exist multiple (partial) solutions to the fixed point equation and one would like to pick the *least* solution with respect to a partial order on $\mathcal{P}$. In the next subsection (§2.2), we generalize Charguéraud's theory so as to remedy this limitation.

2.2 Optimal Least Fixed Points

Throughout this subsection, we fix a partial order $\leq$ on $\mathcal{P}$. We also write $\leq$ for the pointwise partial order on functions: $f \leq g$ is a short-hand for $\forall v. f\,v \leq g\,v$. Accordingly, $f \leq_D g$ is short

for $\forall v \in D. f\, v \leq g\, v$. We adapt the notion of a consistent partial fixed point to take the partial order $\leq$ into account. The notion of an optimal least fixed point then falls out in a straightforward way. These definitions generalize those given previously: instantiating $\leq$ with equality yields the definitions of §2.1.

Definition 2.5 (Consistency up to $\leq$). A partial fixed point $\bar{f}$ is *consistent up to* $\leq$ if for every partial fixed point $\bar{g}$, we have $f \leq_{dom(\bar{f}) \cap dom(\bar{g})} g$.

Definition 2.6 (Optimal least fixed point). Among the partial fixed points that are consistent up to $\leq$, the *optimal least fixed point* of $\mathcal{E}$, written $\bar{\mu}\mathcal{E}$, is the one that is greatest with respect to the relation $\sqsubseteq$.

The optimal least fixed point always exists: it is constructed as the union of all partial fixed points that are consistent up to $\leq$, which one can prove is itself a partial fixed point and consistent up to $\leq$. It is unique up to equivalence $\equiv$.

It is worth noting that the function $\mathcal{E}$ is *not* required to be monotone: the existence and uniqueness of the optimal least fixed point can be established without this hypothesis. The solver that we verify, however, does require monotonicity; this condition appears in its specification (§4).

Although the optimal least fixed point always exists, there is a priori no guarantee about its domain $dom(\bar{\mu}\mathcal{E})$. In general, it is up to the user of the theory to establish a fact of the form $V \subseteq dom(\bar{\mu}\mathcal{E})$, which means that the partial function $\bar{\mu}\mathcal{E}$ is well-defined on a subset V of $\mathcal{V}$. In the following, we establish one such general result: if $\mathcal{E}$ admits a (total) least fixed point, then (up to a certain technical condition) the optimal least fixed point $\bar{\mu}\mathcal{E}$ is defined everywhere and coincides with the least fixed point.

Definition 2.7 (Overriding $\mathcal{E}$ with $\bar{\varphi}$). If $\bar{\varphi}$ is a partial function, then $(\mathcal{E}|_{\bar{\varphi}})$ is defined as follows:

$$(\mathcal{E}|_{\bar{\varphi}})\, f\, v \triangleq \begin{cases} \varphi\, v & \text{if } v \in dom(\bar{\varphi}) \\ \mathcal{E}\, f\, v & \text{otherwise} \end{cases}$$

Theorem 2.8 (Coincidence). *Suppose $\mathcal{E}$ admits a least fixed point μ. Suppose furthermore that, for every partial function $\bar{\varphi}$, $(\mathcal{E}|_{\bar{\varphi}})$ admits a fixed point. Then, the partial function $(\mu, \mathcal{V})$ is the optimal least fixed point of $\mathcal{E}$. In other words, the optimal least fixed point is defined everywhere and coincides with the least fixed point.*

The technical condition imposed by Theorem 2.8, namely "for every partial function $\bar{\varphi}$, $(\mathcal{E}|_{\bar{\varphi}})$ admits a fixed point", is satisfied when the least fixed point is obtained either via the Knaster-Tarski theorem (which requires $(\mathcal{P}, \leq)$ to form a complete lattice and requires $\mathcal{E}$ to be monotone) or via the Kleene fixed point theorem (which requires $(\mathcal{P}, \leq)$ to form a directed-complete partial order and requires $\mathcal{E}$ to be continuous). Indeed, if $\mathcal{E}$ is monotone or continuous, then so is $(\mathcal{E}|_{\bar{\varphi}})$. Thus, when the existence of the least fixed point is guaranteed by either of these standard theorems, the optimal least fixed point is defined everywhere and coincides with the least fixed point.

3 BACKGROUND ON IRIS

3.1 Iris and HeapLang

Iris [Jung et al. 2018] is a powerful program logic, a modern form of concurrent separation logic [O'Hearn 2007]. Both its metatheory and its user aspects (notations, tactics, etc.) are embedded in Coq. A large part of it is programming-language-independent. Nevertheless, the Iris distribution contains a sample programming language, HeapLang, so it is easy for a newcomer to quickly get started and carry out proofs of HeapLang programs in Iris.

We take advantage of this facility: the code that we verify is HeapLang code. In the paper, we take the liberty of presenting it in OCaml syntax, which is more readable and should feel more familiar to some readers. This introduces a formalization gap: our manual transcription of the HeapLang code into OCaml-like pseudocode may introduce errors. In the future, it would be desirable to have a mechanical way of bridging the gap between HeapLang and a subset of OCaml, or to instantiate Iris for a subset of OCaml.

HeapLang is a call-by-value λ-calculus equipped with structured data (unit, sums, pairs), mutable state (references), and shared-memory concurrency. A *fork* operation spawns a new thread. An atomic *CAS* operation allows implementing synchronization operations. HeapLang is untyped. It is equipped with a small-step operational semantics.

Iris defines a triple of the form $\{P\}\ e\ \{y.\ Q\}$, with a partial correctness interpretation: the meaning of such a triple is (very roughly) that the execution of expression e in an initial state that satisfies P must either diverge or terminate and return a value y such that the final state satisfies Q. The variable y may appear in the assertion Q; its type is *val*, the type of all HeapLang values. In such a triple, P and Q are Iris *assertions*, whose type is *iProp*. A triple $\{P\}\ e\ \{y.\ Q\}$ is itself an assertion. We use a few standard Iris notations: if P is a *proposition*, whose type is *Prop*, then $\ulcorner P \urcorner$ is an Iris assertion; $P * Q$ is a separating conjunction; $P \mathrel{-\!\!*} Q$ is a separating implication.

Locks are implemented in Iris/HeapLang as a library, which offers the operations *newLock*, *acquireLock*, and *releaseLock*. We also use a higher-order function *withLock* which enforces a balanced use of *acquireLock* and *releaseLock*. The reasoning rules associated with locks are standard [Gotsman et al. 2007; Hobor et al. 2008]. A lock protects an invariant, represented by an assertion I, which is chosen when the lock is allocated. Acquiring a lock transfers the invariant from the lock to the current thread; releasing a lock transfers the invariant back from the current thread to the lock.

3.2 Representation Predicates for Data and Functions

A number of simple entities, such as integers, pairs of integers, lists of integers, and so on, exist both as data in the programming language HeapLang and as mathematical entities in the meta-language (that is, in Coq). In order to write specifications for programs in a natural and uniform style, we must explicitly define this correspondence.

To do so, we introduce a universe of data types: $T ::= \mathit{val} \mid \mathit{unit} \mid T + T \mid T \times T \mid \mathit{list}\ T$. This universe includes *val*, the type of HeapLang values itself; the base type *unit*; sums, products, and lists. This universe is relatively poor, but could be extended if needed: we view it as a proof of concept, which is sufficient for the purposes of this paper.

We define an interpretation function that maps a type T in this universe to the corresponding native Coq type. In this paper, in order to avoid notational overhead, we make applications of this function implicit: thus, we allow T to be read as a Coq type.

At every type T, we define an injective encoding function $[\![\cdot]\!]_T$ of type $T \to \mathit{val}$. By doing so, we define a correspondence between certain HeapLang values and mathematical entities: if X is a mathematical value of type T, then the HeapLang value $[\![X]\!]_T$ *represents* X. We make this idea explicit by defining "x *represents* X *at type* T" as a proposition:

Definition 3.1 (Represents / data). The Coq proposition "x *represents* X *at type* T", where x is a HeapLang value and X has type T, is defined as $x = [\![X]\!]_T$.

We also define a decoding function $\mathit{decode}_T\ \cdot$ of type $\mathit{val} \to \mathit{option}\ T$, which is a partial inverse of the above encoding function: we have $\mathit{decode}_T\ [\![X]\!]_T = \mathit{Some}\ X$. The existence of this decoding function means that the encoding function is injective. It is exploited in our construction of typed prophecy variables (§3.4).

The following convenient abuse of notation is used in our proofs (§7, §8):

Definition 3.2 (Typed points-to). $m \mapsto X$ is short for $\exists x.(m \mapsto x * x \textit{ represents } X \textit{ at type } T)$.

In this paper, it is often the case that a HeapLang function f is intended to behave like a mathematical function F. By this, we mean that, if the HeapLang value x represents the mathematical value X, then the function call $f\ x$ must return a HeapLang value y that represents $F(X)$, if it returns at all (it can also diverge).

To make this idea precise, and to make it work also at higher function types and at partial function types, we proceed as follows. We introduce a universe that includes the data types T as its base types and that features total function types and partial function types: $\tau ::= T \mid \tau \to \tau \mid \tau \hookrightarrow \tau$. Then, we define the assertion "f *implements* F *at type* τ" by induction over τ. (Thus, this assertion can be viewed as a logical relation between HeapLang values and mathematical values.) The three cases of this definition are given in the next three definitions.

Definition 3.3 (Implements / data). The Iris assertion "x *implements* X *at type* T", where x is a HeapLang value and X has type T, is defined as $\lceil x \textit{ represents } X \textit{ at type } T \rceil$.

Definition 3.4 (Implements / function). The Iris assertion "f *implements* F *at type* $\tau \to \tau'$", where f is a HeapLang value and F has type $\tau \to \tau'$, is defined as the following triple:

$$\forall x.\forall X.\ \{x \textit{ implements } X \textit{ at type } \tau\}\ f\ x\ \{y.\ y \textit{ implements } F(X) \textit{ at type } \tau'\}$$

When the function f satisfies such a specification, we say that f is *apparently pure*. A pure function typically enjoys a specification of this form. However, an apparently pure function is not necessarily pure in a strong sense. Indeed, an apparently pure function can have internal side effects: its execution can involve spawning a thread, acquiring and releasing a lock, updating a piece of mutable state that is protected by a lock, and so on. As a consequence, the fact that f is apparently pure does *not* imply that the expressions $(f\ x, f\ x)$ and $\textit{let}\ y = f\ x\ \textit{in}\ (y, y)$ are observationally equivalent. It *does* guarantee a weak form of determinism: two calls to $f\ x$ must return the same result, if they both terminate. It *does* allow a user to reason about f, by applying the reasoning rules of Iris, in exactly the same way as if f was pure.

In this paper, it is sometimes the case that a function f implemented in HeapLang is intended to behave like a mathematical *partial* function $\bar{F}$. (We have explained in §2 how we encode partial functions in Coq.) There are at least two distinct ways in which this can be expressed as a triple. A *strict* way is to let the assertion $\lceil X \in \textit{dom}(\bar{F}) \rceil$ appear in the precondition, which means that the caller has the obligation to prove that the argument lies within the domain of the partial function. A *lax* way is to impose no such requirement and instead adopt the convention that the function call $f\ x$ must diverge if the argument lies outside the domain. This convention is reflected by letting the assertion $\lceil X \in \textit{dom}(\bar{F}) \rceil$ appear in the postcondition. This allows a user to reason that, if the call returns, then definitely the argument was in the domain.

The "strict" interpretation is so named because it imposes a stronger requirement *on the client* than the "lax" interpretation. However, from the point of view of the *implementor*, who wishes to prove that the code satisfies a certain specification, a "lax" triple is a stronger specification than a "strict" triple: indeed, by the rule of consequence, it is clear that a lax triple entails the corresponding strict triple.

In the following definition, we use the lax interpretation, because it is what we need in the specification of *lfp* (§4), where we wish to give the strongest possible specification to *get*.

Definition 3.5 (Implements / lax partial function). The Iris assertion "f *implements* $\bar{F}$ *at type* $\tau \hookrightarrow \tau'$", where f is a HeapLang value and $\bar{F}$ has type $\tau \hookrightarrow \tau'$, is defined as the following triple:

$$\forall x.\forall X.\ \{x \textit{ implements } X \textit{ at type } \tau\}\ f\ x\ \{y.\ y \textit{ implements } F(X) \textit{ at type } \tau' * \lceil X \in \textit{dom}(\bar{F}) \rceil\}$$

$$\textsc{Prophecy Allocation}\quad \frac{\{\mathit{true}\}\ \mathit{newProph}()}{\{p.\ \exists \mathit{xs}.\ p\ \mathit{will\ receive}\ \mathit{xs}\}}$$

$$\textsc{Prophecy Assignment}\quad \frac{\{p\ \mathit{will\ receive}\ \mathit{xs}\}\ \mathit{resolveProph}\ p\ x}{\left\{().\ \exists \mathit{xs}'.\ \begin{array}{l}\ulcorner \mathit{xs} = x :: \mathit{xs}' \urcorner \\ p\ \mathit{will\ receive}\ \mathit{xs}'\end{array}\right\}}$$

$$\textsc{Prophecy Disposal}\quad \frac{\{p\ \mathit{will\ receive}\ \mathit{xs}\}\ \mathit{disposeProph}\ p}{\{().\ \ulcorner \mathit{xs} = [] \urcorner\}}$$

Fig. 1. Rules for allocating, writing, and destroying untyped prophecy variables

$$\textsc{Typed Prophecy Allocation}\quad \frac{\{\mathit{true}\}\ \mathit{newProph}()}{\{p.\ \exists \mathit{Xs}.\ p\ \mathit{will\ receive}\ \mathit{Xs}\}}$$

$$\textsc{Typed Prophecy Assignment}\quad \frac{\left\{\begin{array}{l}\ulcorner x\ \mathit{represents}\ X\ \mathit{at\ type}\ T \urcorner \\ p\ \mathit{will\ receive}\ \mathit{Xs}\end{array}\right\}\ \mathit{resolveProph}\ p\ x}{\left\{().\ \exists \mathit{Xs}'.\ \begin{array}{l}\ulcorner \mathit{Xs} = X :: \mathit{Xs}' \urcorner \\ p\ \mathit{will\ receive}\ \mathit{Xs}'\end{array}\right\}}$$

$$\textsc{Typed Prophecy Disposal}\quad \frac{\{p\ \mathit{will\ receive}\ \mathit{Xs}\}\ \mathit{disposeProph}\ p}{\{().\ \ulcorner \mathit{Xs} = [] \urcorner\}}$$

Fig. 2. Rules for allocating, writing, and destroying typed prophecy variables

3.3 Prophecy Variables

In several places,[3] our proofs rely on *prophecy variables*, a concept originally due to Abadi and Lamport [1988], which has very recently appeared in Iris [Jung et al. 2020].[4] The purpose of prophecy variables is to remedy a shortcoming of Hoare logic that Iris inherits, namely the fact that an assertion describes just the *current* state of the computation. In other words, there is no direct way for an assertion to refer to either *past* states or *future* states. As far as the past is concerned, one typically works around this limitation by allocating ghost state in which one stores relevant history information. As far as the future is concerned, though, is there a work-around? Prophecy variables offer an answer: in short, they are a form of ghost state, and they store information about the future. Although this sentence is arguably confusing, the rules that govern prophecy variables in Iris are in fact rather easy to explain, so we review them right away.

Iris currently offers just two primitive operations on prophecy variables, namely allocation and assignment. The expression *newProph*() allocates and returns a fresh prophecy variable p. The expression *resolveProph* $p\ x$ writes x, an arbitrary HeapLang value, into the prophecy variable p. A prophecy variable can be written several times. Thus, during its lifetime, a *sequence of values* is written into it. Because our focus is on proving safety properties of programs, we can restrict our attention to *finite* runs of programs: therefore, we may consider that, during the lifetime of a prophecy variable p, a *finite sequence of values* is written into p.

This remark suffices to understand at an intuitive level the reasoning rules for *newProph* and *resolveProph*, which appear in Figure 1, left and middle. The rule Prophecy Allocation has precondition *true*, which means that it is always permitted to allocate a new prophecy variable p, and postcondition $\exists \mathit{xs}.\ p\ \mathit{will\ receive}\ \mathit{xs}$,[5] which means that *there exists a sequence of values xs that will be written to p in the future*. The key point is, even though we may not know, at this point in the code, which sequence of writes will take place in the future, we are allowed to *name* this sequence by anticipation. The assertion $p\ \mathit{will\ receive}\ \mathit{xs}$ is affine, and appears as a precondition in

[3] Prophecy variables are exploited in the proof of the restricted infinitary conjunction rule (§6), in the proof of *modulus* (§7), and in the proof of the solver (§8). We do not see how these proofs could be carried out without prophecy variables.

[4] We were fortunate to learn about prophecy variables in Iris, through a personal communication, before they were presented in a published paper.

[5] In Jung *et al.*'s paper [2020], this assertion is written $\mathit{Proph}(p, \mathit{xs})$.

the second rule, Prophecy Assignment. This means that this assertion also serves as a permission to write p. The postcondition of the rule Prophecy Assignment is natural: if initially we know that the sequence of values that will be written to p in the future is xs, then, after writing x into p, we can deduce that the sequence xs must be of the form $x :: xs'$, where x is the value that was just written and xs' is the sequence of values that remain to be written in the future.

Prophecy variables are ghost variables: they do not exist at runtime. All operations on prophecy variables are erased at compile-time. It would be nice if these operations appeared only in the proof of the program, not in the program itself: this is how ordinary ghost state is handled in Iris. Unfortunately, in the case of prophecy variables, a naïve attempt to adopt a similar treatment would lead to circularities and contradictions. This is why, at present, these operations must appear in the code, even though they have no runtime effect.

3.4 Two Improvements to the Prophecy Variable API

In this section, we present two improvements to Iris's prophecy variables, which we have implemented on top of Iris's existing prophecy variable API.

First, we remark that it seems desirable to extend the API with a third rule, Prophecy Disposal, also shown in Figure 1 (right). In this rule, the assertion p *will receive* xs appears in the precondition, but not in the postcondition, which means that it is consumed. Thus, by applying this rule, we abandon our permission to write p in the future. In return, we are assured that no more writes will take place, which implies that xs must be the empty sequence. This reasoning rule is *not* currently offered by Iris's primitive prophecy variables. Perhaps Iris could be easily extended to offer it; we have not explored this avenue. Instead, we implement the extended API in Figure 1 as an additional layer above Iris's primitive prophecy variables. The implementation is simple: in short, a write of v at the upper level is translated to a write of $inj_1\ v$ at the lower level, while a *disposeProph* operation at the upper level is translated to a write of a "mark" $inj_2\ ()$ at the lower level. The upper-level assertion p *will receive* xs is defined in terms of the lower-level assertion as follows:

$$\exists xs'. \left(\begin{array}{l} p \text{ will receive } xs' * \\ \lceil\ map\ inj_1\ xs \text{ is a prefix of } xs'\ \rceil * \\ \lceil \text{ and } xs \text{ is the longest list that enjoys this property } \rceil \end{array} \right)$$

Next, we remark that Iris's primitive prophecy variables are *untyped*. In the rules of Figure 1, x and xs respectively have types *val* and *list val*, where *val* is the type of all HeapLang values. Yet, it is often more natural and convenient to be able to work with a *typed* prophecy variable, into which one writes values of some type T, which is fixed when the prophecy variable is allocated. This often removes the annoying need, noted by [Jung et al. 2020], to explicitly deal with "spurious prophesied values" . It turns out, again, that typed prophecy variables are easy to implement as an additional layer above untyped prophecy variables. We restrict our attention to the types T in our universe of data types (§3.2), for which we have an injective encoding function $[\![\cdot]\!]_T$ of type $T \to val$. Then, we are able to offer the reasoning rules shown in Figure 2. These rules are essentially identical to those of Figure 1, except that the user works directly with an assertion of the form "p *will receive* Xs", where Xs has type *list* T. (The type T is an implicit parameter of this assertion.) The assertion p *will receive* Xs is defined in terms of a lower-level untyped-prophecy assertion as follows:

$$\exists xs. \left(\begin{array}{l} p \text{ will receive } xs * \\ \lceil map\ [\![\cdot]\!]_T\ Xs \text{ is a prefix of } xs \rceil * \\ \lceil \text{ and } Xs \text{ is the longest list that enjoys this property } \rceil \end{array} \right)$$

Each of the encoding layers described above is hidden behind an abstraction barrier, so the end user relies purely on the rules of Figure 2 and does not need to know how these rules are justified.

4 SPECIFICATION OF A SOLVER

Before presenting the particular solver that we verify, we present our specification of a "local generic solver". Although there exist several slightly different ways of implementing a solver, some of which are discussed later on (§9), we think that all of them should in principle satisfy this specification.

4.1 Assumptions about Variables and Properties

A solver is parameterized with a type of variables $\mathcal{V}$ and a type of properties $\mathcal{P}$. We assume that these types are part of the universe of data types T defined earlier (§3.2), so that the predicates "_ *represents* _ *at type* $\mathcal{V}$" and "_ *represents* _ *at type* $\mathcal{P}$" are well-defined. As a result, the predicates that characterize apparently pure functions, such as "_ *implements* _ *at type* $\mathcal{V} \to \mathcal{P}$", make sense as well.

A solver is also parameterized with a number of functions that allow manipulating variables and properties. Regarding variables, the user must provide an implementation *Var.Map* of maps whose keys are variables. Regarding properties, the user must provide a property *Prop.bottom* and an equality test *Prop.eq* on properties. (No ordering test is required.) These parameters exist at runtime: they are HeapLang values.

The specification of a solver is further parameterized with a number of objects and properties that exist at the mathematical level, but not at runtime. The type $\mathcal{P}$ must be equipped with a relation $\leq$, which must be a partial order, and with an element $\bot$, which must be the least element of this partial order. The value *Prop.bottom* must satisfy the specification "*Prop.bottom represents* $\bot$ *at type* $\mathcal{P}$". The value *Prop.eq* must satisfy the specification "*Prop.eq implements* $\lambda xy.\mathit{Is\_true}\ (x = y)$ *at type* $\mathcal{P} \to \mathcal{P} \to \mathit{bool}$". Finally, the dictionary operations in *Var.Map* must behave as expected: for the sake of brevity, we omit their specifications, which are standard. As usual in separation logic, these specifications involve a user-defined abstract predicate *isMap*, whose arguments are a HeapLang value and a mathematical finite map. Examples of the use of this predicate appear in the proof of the solver (§8). For instance, the assertion "*isMap permanent P*" grants permission to access the data structure located at address *permanent* in the heap and guarantees that this data structure currently represents the mathematical map *P*.

4.2 Specification of *lfp*

Once the stage is thus set, the remainder of the specification of a solver is simple. An Iris statement of this specification appears in Figure 3. It can be read as follows: a solver computes the optimal least fixed point of a system of monotone equations. At a slightly greater level of detail, if $\mathcal{E}$ is a monotone function, then, provided the user supplies an apparently pure function *eqs* that implements *flip* $\mathcal{E}$,[6] the function call *lfp eqs* returns an apparently pure function *get* that implements the optimal least

[6]The user-supplied function *eqs* is so named because it can be thought of as a "system of monotone equations". *flip* $\mathcal{E}$ is synonymous with $\lambda v.\lambda F.\mathcal{E}\ F\ v$. It is the function $\mathcal{E}$ whose arguments have been exchanged. We let *eqs* take first a variable, then a function (instead of the reverse) because this seems slightly more natural and convenient in practice.

$$\forall \mathit{eqs}.\forall \mathcal{E}.\ (\mathcal{E} \text{ is monotone}) \quad \Rightarrow \quad \begin{array}{c} \{\mathit{eqs}\ \mathit{implements}\ \mathit{flip}\,\mathcal{E}\ \mathit{at\ type}\ \mathcal{V} \to (\mathcal{V} \to \mathcal{P}) \to \mathcal{P}\} \\ \mathit{lfp}\ \mathit{eqs} \\ \{\mathit{get}.\ \mathit{get}\ \mathit{implements}\ \bar{\mu}\mathcal{E}\ \mathit{at\ type}\ \mathcal{V} \hookrightarrow \mathcal{P}\} \end{array}$$

Fig. 3. An Iris specification for a local generic solver

fixed point $\bar{\mu}\mathcal{E}$ of $\mathcal{E}$. Unfolding the definition of *implements* (§3.2) makes the following points apparent:

- The user can expect *eqs* to be applied to a variable v and to an apparently pure function f that implements some mathematical function ϕ of type $\mathcal{V} \to \mathcal{P}$. It must then return the property $\mathcal{E}\,\phi\,v$.
  It is worth emphasizing that the function f, which is passed by the solver to the user, is in reality likely to have a variety of internal side effects, including looking up memoized information, spying on the user (that is, dynamically recording dependencies), and scheduling variables for reexamination. By stating that f is apparently pure, this specification guarantees that it is safe for the user to be unaware of these effects. Dually, it forces the implementor of the solver to verify that these effects are properly encapsulated.
- According to Definition 3.5, the user can apply *get* to any variable v. If this function call returns, then it returns the property $\bar{\mu}\mathcal{E}(v)$, and $v \in \mathit{dom}(\bar{\mu}\mathcal{E})$ is guaranteed to hold. By contraposition, an application of *get* to a variable v *outside* $\mathit{dom}(\bar{\mu}\mathcal{E})$ is guaranteed to diverge. It may seem odd that the user is not *required* to prove $v \in \mathit{dom}(\bar{\mu}\mathcal{E})$. That would be the case if we used the *strict* interpretation of "*get implements* $\bar{\mu}\mathcal{E}$ *at type* $\mathcal{V} \hookrightarrow \mathcal{P}$" (§3.2). However, because we do not establish termination, we have no use for such a requirement. We *could* nevertheless add it to the specification so as to detect more mistakes in user code.

Expressing the specification of a solver in terms of an optimal least fixed point, as opposed to the more familiar notion of a least fixed point, is attractive on several grounds. First, this allows a solver to be used in situations where "the least fixed point is not defined everywhere", that is, where no (total) least fixed point exists. Second, this removes the need to bake a sufficient condition for the existence of a least fixed point into the specification of a solver. The literature offers several incomparable sufficient conditions; we need not choose.

Even though the monotonicity of $\mathcal{E}$ is not required for the optimal least fixed point to exist (§2), it is needed when we verify that the algorithm meets its specification (§8). Technically, it is required in order to maintain the invariant that the transient map is an under-approximation (with respect to the ordering $\leq$) of every partial fixed point.

5 PRESENTATION OF THE SOLVER

Several local fixed point computation algorithms exist in the literature [Fecht and Seidl 1999; Le Charlier and Van Hentenryck 1992; Pottier 2009; Vergauwen et al. 1994]. The code that we verify is inspired by Pottier's OCaml implementation [2019]. Our main improvement on Pottier's implementation resides in isolating "spying" from the rest of the solver. Spying is implemented as a third-order function, *modulus*, whose code appears in Figure 7. In short, the function call *modulus ff f* performs an application of *ff* to *f* and returns a pair (c, ws) of the result of this application and a list of points at which *f* was invoked by *ff* during this application. The function *modulus* has in fact been documented and studied by Longley [1999], but the remark that it can be used off the shelf in the construction of a local solver is novel. In order to facilitate our proof, we remove a few optimizations from Pottier's implementation. They are discussed later on (§9).

Our solver appears in Figure 4. We remind the reader that the code that we actually verify is expressed in HeapLang syntax inside Coq (§3.1). The pseudocode in Figure 4 is a manual transcription of this code in OCaml syntax, extended with operations on locks and prophecy variables.

The solver is parameterized with a type of variables and a type of properties, respectively named *var* and *prop* in our pseudocode (Figure 4, line 1). We assume that the module *Var.Map* offers an implementation of maps whose keys are variables. We assume that the module *Prop* provides access

```
val lfp: (var -> (var -> prop) -> prop) -> (var -> prop)
let lfp eqs =
  let permanent, transient, dependencies, todo =
    Var.Map.create(), Var.Map.create(), Var.Map.create(), ref [] in
  let master = newLock() in
  let schedule v =
    Var.Map.insert dependencies v [];
    todo := v :: !todo
  in
  let discover v =
    Var.Map.insert transient v Prop.bottom;
    schedule v
  in
  let reevaluate v =
    let p, l = newProph(), newLock() in
    let request w =
      withLock l (fun () ->
        match Var.Map.lookup permanent w with Some c -> c | None ->
        match Var.Map.lookup transient w with Some c -> c | None ->
        discover w; resolveProph p w; Prop.bottom)
    in
    let c, ws = modulus (eqs v) request in
    acquireLock l;
    disposeProph p;
    Var.Map.insert dependencies v ws;
    if not (Prop.eq c (Var.Map.lookup_exn transient v)) then begin
      Var.Map.insert transient v c;
      List.iter schedule (predecessors dependencies v)
    end
  in
  let rec loop () =
    match !todo with [] -> () | v :: vs ->
      todo := vs; reevaluate v; loop()
  in
  let get v =
    withLock master (fun () ->
      match Var.Map.lookup permanent v with
      | Some c -> c
      | None ->
        discover v; loop();
        Var.Map.transfer transient permanent;
        Var.Map.flush dependencies;
        Var.Map.lookup_exn permanent v)
  in
  get
```

Fig. 4. The function `lfp`

to the least property *Prop.bottom* and to an equality test *Prop.eq*. We omit the signatures of these modules; we have explained earlier (§4.1) which specifications they must satisfy.

The solver's main function, *lfp*, first allocates and initializes a number of internal data structures. Then, without performing any actual computation yet, it returns a function, *get* (line 45). Later on, at any time, the user may call *get* v to query the optimal least fixed point at a variable v. Each such call possibly causes a new *wave* of computation, as the solver computes the optimal least fixed point over a fragment of its domain that is as limited as possible in order to answer this query at v.

The solver maintains a number of mutable data structures. These data structures, allocated on lines 3–4, are:

- A *permanent map* of variables to properties. This map records a fragment of the optimal least fixed point $\bar{\mu}\mathcal{E}$. As its name suggests, once a variable-property pair is stored in this map, it remains there forever. The permanent map is used for memoization. It is long-lived: that is, it persists across invocations of *get*.
- A *transient* map of variables to properties. While a wave is in progress, this map stores a partial function that is less than (or equal to) the optimal least fixed point with respect to the partial order $\leq$ on properties.
- A *dependency graph*, naïvely represented as a map of each variable in the transient map to a list of its successors. A dependency edge $u \longrightarrow v$ means that u *observes* v. If the property associated with v in the transient map is updated, then u must be scheduled for reevaluation.
- A *workset*, represented as a list of variables, whose order is irrelevant.

A *master lock*, allocated on line 5, protects these data structures. Indeed, the body of *get* forms a critical section, protected by this lock (lines 36–43). This prevents two calls to *get* from taking place at the same time. A reentrant call (that is, an attempt to call *get* while a call to *get* is already in progress on the same thread) gives rise to a deadlock, a behavior that is considered safe. Two calls to *get* on distinct threads are sequentialized; thus, data races are prevented. Pottier [2009] emphasizes the need for such a lock, even in a sequential setting, and implements it as a Boolean flag, *inactive*.

The implementation of *get* can be described as follows. If the variable v that is queried by the user already appears in the permanent map, then the corresponding property is immediately returned (line 38). Otherwise (line 40), the variable v is "discovered", and the solver enters its main loop; this is the beginning of a new wave. Once this wave is over (line 41), the information stored in the transient map, which is now stable, is transferred into the permanent map, and the dependency graph is thrown away (line 42). At this point, the variable v must be in the domain of the permanent map, so the lookup at line 43 must succeed.

The function *discover* (line 10) inserts a newly discovered variable into the transient map, where it is initially associated with the property $\bot$, and schedules it for evaluation. The function *schedule* (line 6) removes v's outgoing dependency edges (if there are any) and inserts v into the workset.

During a wave, the solver repeatedly extracts a variable v out of the workset and reevaluates it (line 31). The function *reevaluate* (line 14) is in charge of this. To this end, the user-supplied function *eqs*, which implements *flip* $\mathcal{E}$, must be applied to the variable v and to a function, *request*, which implements our current under-approximation of the fixed point.

The function *request* (line 16), looks up the permanent map first, then the transient map. If both lookups fail, then it schedules its argument for later evaluation and returns $\bot$, a safe under-approximation. It too must be protected by a lock *l*. The critical section at lines 17–20 prevents concurrent calls to *request*. The final lock acquisition on line 23 blocks any invocation of *request* after the function call *eqs v request* on line 22 has returned.

Coming back to *reevaluate*, precisely on line 22, where one might expect a function call of the form $\mathit{let}\, c = \mathit{eqs}\, v\, \mathit{request}$, one finds that this call is wrapped in an invocation of the higher-order function *modulus*, like so: $\mathit{let}\, c, ws = \mathit{modulus}\,(\mathit{eqs}\, v)\, \mathit{request}$. In short, *modulus* performs the application of $\mathit{eqs}\, v$ to *request* and returns not only its result c, a property, but also a list ws of the variables at which *request* is invoked during this function application. The implementation of *modulus* is shown in Figure 7 and explained later on (§7). Coming back to *reevaluate*, still at line 22, the property c is the result of the function call $\mathit{eqs}\, v\, \mathit{request}$, and it is known that c has been computed based solely on the properties that are currently associated with the variables ws. As long as these properties do not change, there is no need to reevaluate v. Thus, we record that v depends (only) on ws (line 25).

Still inside *reevaluate*, on line 26, the property c, which should be newly associated with v, is compared with the property currently associated with v in the transient map. If they are equal, then v is stable with respect to its predecessors, and there is nothing else to do. Otherwise, the transient map is updated at v (line 27), which implies that the predecessors of v in the dependency graph must be scheduled for reevaluation (line 28).

6 A RESTRICTED INFINITARY CONJUNCTION RULE

6.1 Unsoundness of the Unrestricted Conjunction Rule

In the proof of *modulus* (§7), there seems to be a need for an *infinitary conjunction rule*, whose statement could be the following:

$$\textsc{Infinitary Conjunction (Unsound)}\quad \frac{\forall x.\ \{P\}\ e\ \{y.\ Q\, x\, y\}}{\{P\}\ e\ \{y.\ \forall x. Q\, x\, y\}}$$

The rule is written under the assumption that x does not occur in P, e, y. There is no restriction on the expression e, whose execution can involve nondeterminism and side effects. The type of the auxiliary variable x must be inhabited, but is otherwise arbitrary; in particular, it can be finite or infinite. The binary conjunction rule is obtained as the special case where the type of x is *bool*.

The rule can be read informally as follows: if for every x one can prove that the execution of the expression e, beginning in a state that satisfies P, must end in a state that satisfies $Q\, x\, y$, then one can deduce that the execution of e, beginning in a state that satisfies P, must end in a state that satisfies $Q\, x\, y$ for every x.

When read in this way, this rule seems valid. Indeed, this reading relies on the traditional "must" interpretation of Hoare triples, where $\{P\}\ e\ \{y.\ R\, y\}$ means that *every* possible final state *must* satisfy the postcondition $R\, y$. Then, the explicit universal quantification over x and the implicit universal quantification over the final state can be exchanged, which seems to justify the rule. Indeed, in traditional Hoare logic, the binary and infinitary conjunction rules are sound. The binary conjunction rule appears as Axiom 1 in Floyd's seminal paper [1967], where it is argued that it is sound with respect to the semantic interpretation of triples.

In the setting of separation logic, however, the binary and infinitary conjunction rules are unsound. O'Hearn [2007] documents a counter-example, which he attributes to Reynolds. The counter-example involves permission transfer: using the binary conjunction rule, one combines a derivation where the ownership of a certain memory cell is transferred to a lock and a derivation where ownership is not transferred. This results in a contradiction: one concludes that in the final state the local heap contains a memory cell *and* the local heap is empty.

One could say that this problem arises out of the interaction between the conjunction rule and ghost state: in the counter-example, the two derivations are incompatible because they *make*

$$\frac{\forall x.\ \{P\}\ e\ \{y.\ \lceil Q\,x\,y\rceil\}}{\{P\}\ e\ \{y.\ \lceil \forall x.Q\,x\,y\rceil\}}\ \text{(candidate rule)}$$

$$\frac{\forall x.\ \{P\}\ e\ \{y.\ \lceil Q\,x\,y\rceil\}}{\{P\}\ \left(\begin{array}{l} let\ p = newProph()\ in\\ let\ y = e\ in\\ resolveProph\ p\ y;\\ y \end{array}\right)\ \{y.\ \lceil \forall x.Q\,x\,y\rceil\}}\ \textsc{Pure Infinitary Conjunction}$$

Fig. 5. The pure infinitary conjunction rule

$$\frac{\lceil Q\rceil \mathrel{-\!\!*} \{P\}\ e\ \{y.\ R\} \qquad \{P\}\ e\ \{y.\ true\}}{\{P\}\ e\ \{y.\ \lceil Q\rceil \mathrel{-\!\!*} R\}}\ \textsc{Pure Implication}$$

Fig. 6. The pure implication rule

different decisions about who owns the memory cell. This is ghost information, not recorded at runtime in the physical state. In Iris, which has a very general notion of ghost state, the same problem exists: the binary and infinitary conjunction rules are unsound. One way to understand this is to recall that the definition of an Iris triple includes a basic update modality [Jung et al. 2018, 6.3.2, 6.3.5]. Because of this, an intuitive reading of the Iris triple $\{P\}\ e\ \{y.\ R\,y\}$ is: for every possible final physical state, *there exists* a way of updating the ghost state such that the combined physical and ghost state satisfies the postcondition $R\,y$. The presence of this existential quantifier explains why the conjunction rule is unsound: an existential quantifier and a universal quantifier cannot be exchanged.

6.2 A Sound, Restricted Infinitary Conjunction Rule

The unsoundness of the conjunction rule arises out the fact that the postcondition of an Iris triple is an arbitrary Iris assertion, which can impose a constraint on the ghost state. Thus, it seems that there is a chance that the conjunction rule might become sound if one restricts or removes the ability of the postcondition to describe the ghost state. For instance, one might require the postcondition to be pure. (In Iris, a *pure assertion* is one that is independent of the current state and step index.) The rule would then take the form shown in Figure 5 (left). Again, we assume that x does not occur in P, e, y and that the type of x is inhabited. This is a candidate rule: it is likely sound, yet we do not know how to prove it sound under this exact form. In the following, we first give an informal proof sketch for this candidate rule, then present an almost identical rule that we *are* able to prove sound.

Proof sketch. Let us suppose that $\forall x.\ \{P\}\ e\ \{y.\ \lceil Q\,x\,y\rceil\}$ holds (**1**). We wish to argue that the triple $\{P\}\ e\ \{y.\ \lceil \forall x.Q\,x\,y\rceil\}$ holds. We do so as follows. As a first step, by anticipation, let us name y the eventual value of the expression e. (We do not know whether e will terminate and what its value will be. We do know that, if e terminates, then it will return some value, which we choose to refer to as y, ahead of time.) Our next step is to apply the law of excluded middle: the proposition $\forall x.Q\,x\,y$ either holds or does not hold. The disjunction rule of Hoare logic, which happens to be (obviously) valid in Iris, then allows us to reason by cases.

- *Case 1:* $\forall x.Q\,x\,y$ holds. Then, the goal degenerates to $\{P\}\ e\ \{y.\ \mathit{true}\}$, which means that it is safe to execute e. Hypothesis (1), instantiated with an arbitrary value of x, yields $\{P\}\ e\ \{y.\ \lceil Q\,x\,y\rceil\}$, which (by the consequence rule) implies the goal.
- *Case 2:* $\forall x.Q\,x\,y$ does not hold. Then, the goal degenerates to $\{P\}\ e\ \{y.\ \mathit{false}\}$, which means that e cannot possibly return a value. Because $\forall x.Q\,x\,y$ is false, there must exist a specific value of x such that $Q\,x\,y$ is false. By instantiating Hypothesis (1) with it, we get $\{P\}\ e\ \{y.\ \lceil Q\,x\,y\rceil\}$. Because $Q\,x\,y$ is false, this is equivalent to $\{P\}\ e\ \{y.\ \mathit{false}\}$, which is the goal.

This ends the proof sketch. This proof is rather unusual in two ways. First, it exploits the law of excluded middle. Second, it exhibits a need to name a value by anticipation, that is, to assign a name to a value that is not known yet, and that will become known *once* and *if* the execution of the program reaches a certain point: here, this is the point where the expression e returns a value.

This is a typical use case for prophecy variables. To announce one's desire to name a value by anticipation, one allocates a fresh prophecy variable. To declare that the value denoted by this name has become known, one assigns a value to this prophecy variable. In Iris, as explained earlier (§3.3), these operations must be explicit in the program text. Because of this, we are unable to prove the soundness of the candidate rule shown above. Instead, in the conclusion of the rule, the expression e must be enclosed within prophecy allocation and prophecy assignment instructions. The rule that we *are* able to establish is shown in Figure 5 (right). Because *newProph* and *resolveProph* have no runtime effect, this is virtually the desired rule. It is somewhat bothersome that the code must be instrumented with these dummy instructions; hopefully, in the future, this technical requirement can be removed.

In practical use, we find it convenient to combine this restricted infinitary conjunction rule with a higher-order "function application" combinator, which we name *conjApply*. Thus, the expression $\mathit{conjApply}\ f\ x$ has the same runtime behavior as an ordinary function call $f\,x$, but allows an application of the pure infinitary conjunction rule. This combinator is exploited in the proof of *modulus* (§7).

6.3 A Restricted Implication Rule

In the proof of *modulus* (§7), we also find a need for a restricted implication rule, shown in Figure 6. This rule allows hoisting an implication, whose left-hand side is a pure assertion $\lceil Q\rceil$, out of a triple. (The variable y must not appear in Q.) It is easy to prove that this rule is valid, as follows. By the law of the excluded middle, either the proposition Q holds, or it does not. If it does, then the rule's left-hand premise and conclusion both boil down to $\{P\}\ e\ \{y.\ R\}$, so the rule is valid. If it does not, then the rule's conclusion boils down to $\{P\}\ e\ \{y.\ \mathit{true}\}$, which is exactly the rule's right-hand premise, so the rule is valid as well.

7 SPECIFICATION AND PROOF OF MODULUS

The function *modulus* is one of the most subtle components of the solver. The function call $\mathit{modulus}\ \mathit{ff}\ f$ performs an application of ff to f and returns a pair (c, ws) of the result of this application and a list of points at which f was invoked by ff during this application.

7.1 Implementation

The implementation of *modulus*, which appears in Figure 7, is simple. A reference m is allocated to keep track of the points where f is invoked (line 5). The function f is wrapped in an auxiliary function, *spy* (line 6), which takes care of keeping m up-to-date: whenever *spy* is invoked at some point x, this point is added to the list stored in m (line 8), and the call is forwarded to f (line 7).

```
1  val modulus:
2    (('a -> 'b) -> 'c            ) ->
3    (('a -> 'b) -> 'c * ('a list))
4  let modulus ff f =
5    let m, p, lk = ref [], newProph(), newLock() in
6    let spy x =
7      let y = f x in
8      withLock lk (fun () -> m := x :: !m; resolveProph p x);
9      y
10   in
11   let c = conjApply ff spy in
12   acquireLock lk;
13   disposeProph p;
14   (c, !m)
```

Fig. 7. The function `modulus`

The function *ff* is applied to *spy*, instead of f, on line 11. The combinator *conjApply*, which was introduced at the end of §6.2, behaves like function application.

A lock *lk*, allocated on line 5, protects the memory location m. The critical section on line 8 prevents data races on m and ensures that the instruction `m := x :: !m` is atomic, which is important: naïvely decomposing it into a read and a write, and allowing several threads to race on m, would lead to scenarios where some dependencies fail to be recorded. Indeed, we emphasize that, because HeapLang is a concurrent calculus, it is perfectly possible and permitted for the user-provided function *ff* to spawn several threads and perform multiple concurrent calls to f. The lock *lk* serves a second purpose, which is to prevent the user from using f after the function call *ff* f has returned. Thanks to the final lock acquisition on line 12, which is never followed with a matching *releaseLock* instruction, any subsequent attempt by the user to use f will cause a deadlock.

It seems intuitively clear that, because the user does not have access to m, she cannot tell the difference between *spy* and f. That is, the fact that we record information in m is unobservable. It also seems intuitively clear that, once the call *ff spy* returns, the set of points stored in m represents *sound* dependency information. That is, roughly speaking, the value c *cannot* depend on the value of f at any point *outside* this set. However, how to express the specification of *modulus* in a program logic, such as Iris, and how to prove that *modulus* satisfies its specification, is not so clear. To the best of our knowledge, we are the first to propose a solution to this problem.

Although Longley [1999] does study *modulus*,[7] and gives an imperative implementation of it in Standard ML, his main point is that a PCF, a *pure* λ-calculus, can be safely extended with *modulus*. The extended calculus remains pure, in the sense that every term denotes a mathematical function. Longley points out that the extended calculus can express only "sequentially realizable" functions, and can express all of them. In Longley's setting, *modulus* itself is a pure function: that is, if the equalities $ff_1 = ff_2$ and $f_1 = f_2$ hold in an *extensional* sense, then the calls *modulus* $ff_1\, f_1$ and *modulus* $ff_2\, f_2$ must yield the same outcome, that is, the same result and the same set of dependencies. In contrast, we work in the setting of HeapLang, an impure, concurrent calculus, where new phenomena appear, including nondeterminism. As a result, even though *modulus* still

[7] Our *modulus* corresponds to the function `Mod'` in Longley's paper.

$$\left\{ \begin{array}{c} f \textit{ implements } \phi \textit{ at type } T_a \to T_b \\ \textit{ff} \textit{ implements } \mathcal{F} \textit{ at type } (T_a \to T_b) \to T_c \end{array} \right\}$$
$$\textit{modulus ff } f$$
$$\left\{ y.\ \exists ws. \begin{array}{l} \lceil y \textit{ represents } (\mathcal{F}(\phi), ws) \textit{ at type } T_c \times \textit{list } T_a \rceil \\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow \mathcal{F}(\phi') = \mathcal{F}(\phi) \rceil \end{array} \right\}$$

Fig. 8. The specification of *modulus*

"works" in a certain sense, to be made precise below, it is *no longer* a pure function. Indeed, even when *ff* and f both are apparently pure functions, it is possible for two calls to *modulus ff f* to return two different sets of dependencies.[8] Thus, the question: "what does *modulus* compute?", or in other words: "what is the specification of *modulus*?" is not entirely trivial. Furthermore, as far as we understand, Longley [1999] does not prove that his imperative implementation of *modulus* is correct. That would require giving some sort of semantics to PCF with mutable state, which he does not do. In contrast, we prove that *modulus* satisfies a certain specification and, based on this specification, we verify that our use of *modulus* inside *lfp* produces sound dependency information.

7.2 Specification

Our specification of *modulus* is given in Figure 8. Its precondition just requires the user-provided functions *ff* and f to be apparently pure. That is, *ff* and f must implement two mathematical functions, named $\mathcal{F}$ and ϕ. Its postcondition states that the call *modulus ff f* must return a pair:

- whose first component represents $\mathcal{F}(\phi)$, and
- whose second component represents a list ws that satisfies $\forall \phi'.\ \phi' =_{ws} \phi \Rightarrow \mathcal{F}(\phi') = \mathcal{F}(\phi)$.

The first item means that the first pair component is the result that would be returned by the call *ff f*. (Because both *ff* and f are apparently pure, there is only one result that this call may return.) The second item states that ws is a sound set of dependencies. This is done by guaranteeing that the mathematical function $\mathcal{F}$ is insensitive to the behavior of its argument ϕ outside of ws. This statement involves a quantification over all functions ϕ' such that ϕ' and ϕ agree on the set ws. This universal quantification represents a very strong guarantee. Intuitively, the postcondition means, "*modulus* returns a pair (c, ws) where c represents not just $\mathcal{F}(\phi)$, but also $\mathcal{F}(\phi')$, for every function ϕ' that agrees with ϕ on ws".

The postcondition involves an existential quantification over ws because (as argued in §7.1) the list ws (or the underlying set) cannot be expressed as a function of $\mathcal{F}$ and ϕ. In other words, *modulus* itself is not an apparently pure function.

7.3 Proof

How can one prove that the function *modulus*, whose code appears in Figure 7, satisfies the specification in Figure 8? Before answering this question, let us first sketch a proof that *modulus* is safe. That is, let us sketch why *modulus* satisfies a weaker specification, whose postcondition is just *true*.

Why modulus is safe (proof sketch). The precondition of *modulus* lets us assume that the user-provided functions *ff* and f implement two mathematical functions $\mathcal{F}$ and ϕ. Now, the main

[8] Our specification of *modulus* allows *ff* to have internal side effects, including nondeterminism, as long as one can prove that *ff* is apparently pure. One can craft an example where *ff* is a "parallel-or" function, which invokes $f\ 0$ and $f\ 1$ on two newly spawned threads, and returns *true* as soon as one of these calls returns *true*. Then, a call *modulus ff* $(\lambda i.\, true)$ will return a pair of the form (*true*, ws), where the list ws can contain just 0, just 1, or both.

difficulty in the proof that *modulus* is safe is to argue that the function call *ff spy* on line 11 of Figure 7 is permitted. The only fact that is known about *ff* is "*ff implements* $\mathcal{F}$ *at type* $(T_a \to T_b) \to T_c$", whose meaning is given by Definition 3.4. This implies that, in order to justify the call *ff spy*, we *must* establish a fact of the form "*spy implements* ψ *at type* $T_a \to T_b$" for *some* function ψ. Because in the eye of the user the function *spy* is intended to behave exactly like f, and because f implements ϕ, it seems quite reasonable to let ψ be just ϕ. Indeed, this works: by arguing that the lock *lk* protects the memory location *m*, it is not difficult to argue that our reads and writes of *m* can remain hidden and (therefore) that *spy* implements ϕ. This allows us to conclude that the call *ff spy* is permitted and therefore that *modulus* is safe. In fact, we are even able to conclude that *c* represents $\mathcal{F}(\phi)$, which is a little stronger than what was announced at the beginning of this proof sketch. This ends the proof sketch.

The above proof sketch is a good first step, but fails to establish anything about the second component of the result of *modulus*. By strengthening the lock invariant slightly, we could keep track of the fact that, at all times, the memory location *m* stores a list of points, and conclude that the second component of the result represents *some* list *ws* at type *list* T_a. However, this still gives us no way of proving that *ws* is a sound set of dependencies, as expressed by the second assertion in the postcondition of *modulus*, namely $\forall \phi'.\ \phi' =_{ws} \phi \Rightarrow \mathcal{F}(\phi') = \mathcal{F}(\phi)$.

One idea that might come to mind is to strengthen the lock invariant so as to claim that this assertion holds at every time of the list *ws currently* stored at location *m*. However, this candidate invariant is much too strong: in particular, *m* initially holds the empty list, so, in order to establish this invariant, we would have to prove $\forall \phi'.\ \mathcal{F}(\phi') = \mathcal{F}(\phi)$, which is hopeless. Put another way, the fact that "the memory location *m* stores a sound set of dependencies" *becomes true eventually*, once the call *ff spy* on line 11 returns, but *is not true* while this call is in progress.

Before attempting to remedy this problem by imagining another candidate invariant, let us take a step back and recall why we think that *modulus* is correct. The key intuition is this: although we can justify the call *ff spy* by arguing that "*spy implements* ϕ", we could just as well argue that "*spy implements* ϕ'", for any function ϕ' of our choosing, as long as ϕ' and ϕ agree on the points that the user queries, that is, on the points in the list *ws*, where *ws* refers to the list of dependencies that is *eventually* computed by *modulus*. This intuition leads us to a second proof sketch:

Why modulus is correct (rough proof sketch). By anticipation, let *ws* refer to the list that is *eventually* stored at location *m*, just before *modulus* returns. Then, we would like to claim not only that *spy* implements ϕ, as in the previous proof sketch, but in fact that *spy* implements every function ϕ' such that $\phi' =_{ws} \phi$ holds. If we can establish this claim, then we can conclude not only that *c* represents $\mathcal{F}(\phi)$, as in the previous proof sketch, but also that *c* represents every $\mathcal{F}(\phi')$, out of which the desired result follows. Thus, assuming that an arbitrary function ϕ' is given, such that $\phi' =_{ws} \phi$ holds, there remains to argue that *spy* implements ϕ'. To do so, we must prove that, when *spy* is applied to some argument x, it returns $\phi'(x)$. However, because the value returned by *spy* is the result of the call $f\ x$, and because f implements ϕ, it is clear that *spy* in fact returns $\phi(x)$. So, there remains to prove $\phi'(x) = \phi(x)$. Because the functions ϕ' and ϕ agree on *ws*, this boils down to proving that x is a member of the list *ws*. Because *ws* is the list that is *eventually* stored at location *m*, and because *spy* has the effect of inserting x into this list, it is obvious that this is true. (Remember, this is a rough sketch, not a proof!) This ends the proof sketch.

In order to turn this proof sketch into a valid proof in Iris, a number of technical issues must be overcome. First, it appears that, at the very beginning of the proof, we must be able to refer to the list *ws* that is eventually computed by *modulus*. This reference to the future seems to call for a prophecy variable. Second, although the proof sketch claims that any value that is inserted into

the list at some point is "obviously" also a member of the list at the end, making this argument precise requires coming up with a suitable invariant. Third, although we have argued informally that "*spy* implements every function ϕ' such that $\phi' =_{ws} \phi$ holds", at the point where we wish to justify the function call *ff spy*, we must pick *one* function ψ such that *spy* implements ψ. Indeed, this is imposed on us by the specification of *ff*. So, we must focus on one function ϕ' and carry out a proof of *modulus* with respect to this specific ϕ'. Once that is done, we wish to argue that "if *modulus* is correct with respect to an arbitrary function ϕ', then it is correct with respect to all such functions at once". This is where an infinitary conjunction rule is called for. Fortunately, the postcondition of *modulus* is a pure assertion, so our restricted rule (§6) is applicable.

Why modulus is correct (detailed proof sketch). As before, the precondition of *modulus* lets us assume that the user-provided functions *ff* and *f* implement two mathematical functions $\mathcal{F}$ and ϕ. These assertions are persistent, which means that they remain available throughout the proof. At line 5 of Figure 7, the memory location m is initialized with the empty list. A prophecy variable p is allocated. It is "typed" at type T_a; see §3.4. At this point, we introduce the name *ws* to stand for the list of values that will be written to p in the future: initially, the assertion "p *will receive ws*" holds. Still on line 5, a lock *lk* is allocated. For this lock, we must choose an invariant that relates m and p. Our intuition is that, at any point in time, the list stored at location m contains the points at which *spy has been invoked in the past*, while the list held in the prophecy variable p contains the points at which *spy will be invoked in the future*. At any time, the combination of these lists forms the list *ws*, that is, the complete list that is eventually recorded. Thus, the lock invariant is as follows:

$$\exists us\ vs.\ \begin{cases} m \mapsto us * \\ p\ \textit{will receive}\ vs * \\ \lceil ws = \textit{reverse}\ us +\!\!+ vs \rceil \end{cases}$$

Instantiating *us* with the empty list and *vs* with *ws* shows that this invariant initially holds. Also, anticipating on the proof of *spy*, it is easy to check that the critical section at line 8 of Figure 7 preserves this invariant, and it is easy to see that this invariant allows establishing at line 8 that x is a member of the list *ws*.

Skipping (for now) over the definition of *spy*, the next step is to justify the application of *ff* to *spy* at line 11. We wish to establish the following triple:

$$\{\mathit{true}\}\ \mathit{conjApply}\ \mathit{ff}\ \mathit{spy}\ \{c.\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c\ \textit{represents}\ \mathcal{F}(\phi')\ \textit{at type}\ T_c \rceil\}$$

By the pure infinitary conjunction rule (§6.2) and by the pure implication rule (§6.3), this boils down to proving the following two triples:

$$\forall \phi'.\ \lceil \phi' =_{ws} \phi \rceil \mathrel{-\!\!*} \{\mathit{true}\}\ \mathit{ff}\ \mathit{spy}\ \{c.\ \lceil c\ \textit{represents}\ \mathcal{F}(\phi')\ \textit{at type}\ T_c \rceil\}$$

$$\{\mathit{true}\}\ \mathit{ff}\ \mathit{spy}\ \{c.\ \mathit{true}\}$$

The second one is unproblematic; we have already sketched why the function call *ff spy* is safe. Thus, let us focus on the first one. We assume that some function ϕ' is given, such that $\phi' =_{ws} \phi$ holds, and we must prove:

$$\{\mathit{true}\}\ \mathit{ff}\ \mathit{spy}\ \{c.\ \lceil c\ \textit{represents}\ \mathcal{F}(\phi')\ \textit{at type}\ T_c \rceil\}$$

Since *ff* implements $\mathcal{F}$, in order to establish this triple, it suffices to show that *spy* implements ϕ'. We have already explained (in the previous proof sketch) that this follows from the assumption $\phi' =_{ws} \phi$ and from the fact that at line 8 one can prove that x is a member of the list *ws*.

Thus, at line 12, the proposition "$\forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c$ *represents* $\mathcal{F}(\phi')$ *at type* T_c" holds. It might seem as though we are done; still, an important argument remains to be made, namely the fact that the final read of m on line 14 yields the list *reverse ws*. Indeed, the final lock acquisition on line 12

gives us access to the lock invariant, and the prophecy disposal instruction on the following line yields the information that the list *vs* of future writes is empty. Therefore, at line 14, we are able to deduce that the list stored at *m* is *reverse ws*. At this point, checking that the postcondition of *modulus* is satisfied is routine; the witness for the existential quantifier $\exists ws. \ldots$ in the postcondition of *modulus* is in fact *reverse ws*.

8 PROOF OF THE SOLVER

There remains to verify that the function *lfp*, whose code appears in Figure 4, satisfies the specification shown in Figure 3. We do not explain the proof in detail, but highlight its main points of interest: in particular, we give the invariants associated with the two locks and the loop invariant associated with the function *loop*.

In the following, if P is a finite map of variables to properties, we write $P^\bot$ for the total function of type $\mathcal{V} \to \mathcal{P}$ that agrees with the map P on its domain and that maps every variable outside of $dom(P)$ to $\bot$. Also, if *ws* is a list of variables, we write $ws \Mapsto x$ for the finite map that maps every variable in the list *ws* to x.

The invariant that we associate with the master lock (line 5) is the following:

$$\exists\, P.\ \left\{ \begin{array}{l} \mathit{isMap\ permanent}\ P * \mathit{isMap\ transient}\ \varnothing * \mathit{isMap\ dependencies}\ \varnothing * \mathit{todo} \mapsto [] * \\ \lceil \text{the function } P^\bot \text{ is consistent up to } \leq \rceil \end{array} \right.$$

Thus, when the master lock is available (that is, when *no* call to *get* is in progress), the permanent map stores a fragment P of the optimal least fixed point: this is stated by the proposition "the function $P^\bot$ is consistent up to $\leq$". Furthermore, the transient map, the dependency graph, and the workset are empty.

The main loop invariant (that is, the precondition of the function *loop*) is as follows:

$$\exists\, T\ D\ us.\ \left\{ \begin{array}{l} \mathit{isMap\ transient}\ T * \mathit{isMap\ dependencies}\ D * \mathit{todo} \mapsto us * \\ \lceil dom(T) = dom(D) \rceil * \lceil us \subseteq dom(T) \rceil * \lceil \forall\, v \in dom(D).\ D(v) \subseteq dom(P \cup T) \rceil * \\ \lceil \forall\, v \in dom(T).\ \ v \in us\ \vee\ \forall\, \phi.\ \ \phi =_{D(v)} (P \cup T)^\bot\ \Rightarrow\ T(v) = \mathcal{E}(\phi)(v) \rceil * \\ \lceil \forall\, \bar{f}.\ \bar{f} \text{ is a partial fixed point of } \mathcal{E} \Rightarrow T^\bot \leq_{dom(\bar{f})} \bar{f} \rceil \end{array} \right.$$

Let us say that a variable is *permanent* if it appears in the domain $dom(P)$ of the permanent map and *transient* if it appears in the domain $dom(T)$ of the transient map. Together, the propositions on the second line above state that every variable in the workset *us* is transient, that the source of every edge in the dependency graph is transient, and that the target of every such edge is either permanent or transient. The proposition on the third line above states that every transient variable v either appears in the workset ($v \in us$) or is stable with respect to its successors in the dependency graph ($\forall \phi. \ \ldots$). This expresses the intuition that it is fine to *not* reevaluate the variables that are *not* in the workset. The proposition on the last line above indicates that the partial function $T^\bot$ under-approximates every partial fixed point. Together, the last two propositions allow proving that, once the workset *us* becomes empty, the function $(P \cup T)^\bot$ is consistent up to $\leq$. Therefore, it is a fragment of the optimal least fixed point. This justifies transferring all of the information contained in the transient map into the permanent map at line 41.

Finally, the invariant associated with the auxiliary lock at line 15 is as follows:

$$\exists\, ws\ ws'.\ \left\{ \begin{array}{l} \mathit{isMap\ transient}\ (T \cup (ws \Mapsto \bot)) * \\ \mathit{isMap\ dependencies}\ (D \cup (ws \Mapsto [])) * \\ \mathit{todo} \mapsto ws +\!\!+ us\ * \\ p\ \mathit{will\ receive}\ ws' * \lceil vs = \mathit{reverse}\ ws' +\!\!+ ws \rceil \end{array} \right.$$

In this invariant, the variable *vs* occurs free. We let this name refer, by anticipation, to the list of all variables that are newly discovered (at line 20) during the execution of the function call *modulus* (*eqs v*) *request* at line 22. Thus, the above invariant expresses the fact that a mapping of every newly discovered variable to $\bot$ is added to the transient map, that every such variable is recorded in the dependency graph (with no outgoing edges), and that every such variable is inserted into the workset.

A subtle technical point, which we mention only briefly, in the interest of space, is that we must control the set *ws* returned by *modulus* on line 22. That is, we must establish an upper bound on this set: in other words, we must prove that *modulus* cannot, out of the blue, report dependencies on variables that have never been encountered before. To do so, we first argue that the function *request* (line 16) implements (in the lax sense of Definition 3.5) a partial function whose domain is $dom(P \cup T) \cup vs$, that is, the variables that are known at the beginning of *reevaluate*, plus those that are newly discovered during the execution of *reevaluate*. From this fact, we deduce that the set *ws* returned by *modulus* on line 22 is a subset of $dom(P \cup T) \cup vs$.[9] This result is exploited to prove (among other things) that adding a dependency edge from v to every member of *ws* (line 25) preserves the property that "the target of every dependency edge is either permanent or transient".

9 DISCUSSION

We have verified a "local generic solver", that is, an on-demand, incremental, memoizing least fixed point computation algorithm, inspired by Pottier's OCaml implementation [2019]. In so doing, we have also verified a version of Longley's *modulus* [1999].

The code that we verify is expressed in HeapLang, an imperative, concurrent programming language. Thus, the correctness results that we establish hold even if the client exploits mutable state or concurrency, as long as the requirements imposed by our specifications of *modulus* and *lfp* are respected: in short, the arguments that the client passes to *modulus* and *lfp* must be apparently pure functions.

We believe that these results are compelling examples of how a modern program logic, such as Iris, allows relating a subtle imperative algorithm with an elegant mathematical specification. Furthermore, these proofs provide several illustrations of the use of prophecy variables, and suggest that prophecy variables can be required in the verification of deterministic and sequential code, when this code is allowed to interact with a possibly nondeterministic and concurrent client.

We verify only the partial correctness of *modulus* and *lfp*. In other words, we establish neither termination nor deadlock-freedom. As noted at the beginning of the paper (§1.3), we believe that it should be possible to verify a variant of the solver that does not use any lock, and therefore runs no risk of a deadlock. This solver would come with a more restrictive specification, which enforces a sequential use of the solver functions by the client. This would be achieved by giving out and reclaiming unique permissions to call the solver functions. We believe that our current solver, which uses locks, could then be redefined in a modular manner, by combining the lockless solver with a wrapper that creates, acquires, and releases locks at appropriate times. Such an approach would be superior in principle, and quite interesting. We leave it to future work.

For the sake of simplicity, we have removed a couple of optimizations that exist in Pottier's implementation [2009]. First, we represent the dependency graph as a map of each vertex to the list of its successors. This gives us efficient access to the successors of a vertex, but not to its predecessors. Pottier [2009] uses a more efficient representation, which gives constant-time access

[9] This deduction step requires verifying that *modulus* satisfies a specification that is slightly stronger than the one shown in Figure 8. This strengthened specification allows the function f to implement a *partial* function $\bar{\phi}$, in the lax sense of Definition 3.5, and guarantees that the list *ws* returned by *modulus* is a subset of $dom(\bar{\phi})$.

to successors and predecessors. Although we do not anticipate any deep difficulty in verifying it, we leave this effort to future work. Second, when we find that the variable v depends on a variable w, we record this dependency unconditionally, whereas Pottier's code records it only if w is a transient variable, as opposed to a permanent variable. Indeed, there is no need to record a dependency on a variable whose value will not change in the future. That said, this optimization does not seem to be crucial. In terms of time, it should make virtually no difference. In terms of space, it might offer some savings, but only during a wave, not in the long run, because at the end of each wave the entire dependency graph is thrown away.

The solver that we verify is purely iterative: no recursive functions are involved. (The function *loop* is tail-recursive; it is a loop.) The literature documents "top-down" variants of the solver [Fecht and Seidl 1999; Le Charlier and Van Hentenryck 1992; Pottier 2009] where, instead of scheduling a newly-discovered variable w for later examination and returning the property *Prop.bottom* (line 20), one invokes *reevaluate w* before returning the property currently associated with w in the transient map. This is a true recursive call: these variants of the solver use an implicit stack. We have not yet attempted to verify such a variant. We expect its verification to be more difficult, as it mixes iterative and recursive aspects.

10 RELATED WORK

Several on-demand, incremental, memoizing fixed point computation algorithms appear in the literature [Fecht and Seidl 1999; Le Charlier and Van Hentenryck 1992; Muthukumar and Hermenegildo 1990; Pottier 2009; Vergauwen et al. 1994]. Apinis *et al.* [2016] extend such a solver with support for widening and narrowing, two standard operations in the realm of abstract interpretation. Seidl and Vogler [2018] further extend this solver so that (1) termination is guaranteed even in the absence of a monotonicity hypothesis; (2) space usage is reduced, thanks to a form of selective memoization; and (3) the construction and the resolution of the system of equations can be intertwined through so-called "side effects". Point (2) is an improvement that could probably be incorporated into our solver without requiring any change to its specification. The other extensions mentioned above seem to require altering the specification of the solver; they would constitute interesting directions for future research.

An entirely different approach to the computation of least fixed points is to propose a domain-specific language, such as Flix [Madsen et al. 2016], in which systems of monotone equations can be described and to compile this language down to efficient executable code.

Longley [1999] discusses the function *modulus* and notes that, even though its implementation is imperative, its behavior *in the setting of PCF* remains pure, so that PCF extended with *modulus* remains a pure calculus, which can express only "sequentially realizable" functions, and can express all of them. We have given earlier a more detailed comparison with Longley's work (§7.1).

Hofmann, Karbyshev and Seidl [2010a] use Coq to verify a simplified model of a local generic solver. More details appear in Karbyshev's dissertation [2013]. We have given earlier a brief comparison with their work (§1.4). In short, both the solver and its user are modeled in Coq in terms of functions, relations, and explicit state-passing. In particular, the user-provided function *eqs* is modeled as a *strategy tree*. The fact that it is safe to model the behavior of *eqs* in this way is the subject of two separate papers [Bauer et al. 2013; Hofmann et al. 2010b], where it is argued that every second-order function that is "pure" in a certain semantic sense can indeed be represented as a strategy tree. In contrast, we work in the setting of a program logic for HeapLang, a λ-calculus extended with mutable state and shared-memory concurrency, which obviates the need to construct a mathematical model of the solver and of its user and the need to argue that this model is faithful in some sense. We rely on the notion of *apparent purity* that is naturally provided by Iris: a function f is apparently pure if it satisfies a specification of the form f *implements* ϕ (Definition 3.4).

Such a specification allows the function f to have internal state, as long as this state is properly encapsulated; this is typically achieved by protecting it with a lock. The notion of apparent purity is used both ways in our specification of the solver. On the one hand, we require the user-provided function *eqs* to be apparently pure. On the other hand, we guarantee that the functions exposed by the solver to the user (namely *get* and *request*-wrapped-inside-*spy*) are apparently pure.

Continuing our comparison with Hofmann *et al.* [2010a], our result is stronger in several respects. Hofmann *et al.* require the partial order $(\mathcal{P}, \leq, \bot)$ to form a lattice, whereas we do not. They simplify the solver's API quite radically, so that *get* is called exactly once, with a set of variables of interest. This implies that there is no need for memoization and no possibility of concurrent or reentrant invocations of *get*. Finally, although they prove that the solver computes a fixed point, they do not prove that it is the least fixed point, whereas we do. In fact, by exploiting the novel notion of an optimal least fixed point, a mild generalization of Charguéraud's optimal fixed point [2010a], we avoid imposing an ad hoc sufficient condition for a least fixed point to exist everywhere, and we are able to deal with situations where the least fixed point is well-defined only on a subset of $\mathcal{V}$.

There have been several successful efforts to implement and verify abstract interpreters in Coq [Besson et al. 2009; Cachera and Pichardie 2010; Jourdan 2016; Pichardie 2008]. At their heart, these abstract interpreters contain a least fixed point computation algorithm. They are implemented in a purely functional style, which precludes the use of mutable state for the purposes of memoization or spying. As a result, they are not local (instead, they compute the least fixed point everywhere) and do not perform dynamic dependency discovery (instead, their iteration strategy is directed by the syntax of the program under analysis).

11 FUTURE WORK

There are many directions for future work. Regarding the metatheory of Iris, it would be desirable to find out whether the candidate conjunction rule in Figure 5 is sound with respect to Iris's notion of weakest precondition [Jung et al. 2018, §6.3.4]; we currently do not know. Regarding our solver, it would be desirable to replace our current naïve representation of the dependency graph with an efficient representation, such as Pottier's [2009]. It would be worthwhile to verify a "top-down" variant of the solver (§9), both as an instructive exercise and because such a variant can be more efficient. In a hypothetical variant of Iris that allows establishing total correctness and supports prophecy variables, one might wish to prove that a variant of the solver that does not use any locks always terminates. More ambitiously, in an extension of Iris with time credits [Mével et al. 2019], one might wish to bound its asymptotic complexity. Regarding the variant of the solver that does involve locks, it would be nice if one could use fewer locks, by removing or combining some of them, and if one could statically rule out some or all deadlocks. Finally, it would be desirable to verify realistic applications of the solver: program analyses and parser generators come to mind. With an eye towards applications in program analysis, it would be interesting to specify and verify the more powerful solvers proposed by Apinis *et al.* [2016] and by Seidl and Vogler [2018].

REFERENCES

Martin Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Logic in Computer Science (LICS)*. 165–175.

Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2016. Enhancing Top-Down Solving with Widening and Narrowing. In *Semantics, Logics, and Calculi – Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays (Lecture Notes in Computer Science)*, Vol. 9560. Springer, 272–288.

Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. 2013. On Monadic Parametricity of Second-Order Functionals. In *Foundations of Software Science and Computation Structures (FOSSACS) (Lecture Notes in Computer Science)*, Vol. 7794. Springer, 225–240.

Frédéric Besson, David Cachera, Thomas P. Jensen, and David Pichardie. 2009. Certified Static Analysis by Abstract Interpretation. In *Foundations of Security Analysis and Design (Lecture Notes in Computer Science)*, Vol. 5705. Springer, 223–257.

David Cachera and David Pichardie. 2010. A Certified Denotational Abstract Interpreter. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Vol. 6172. Springer, 9–24.

Arthur Charguéraud. 2010a. *Characteristic Formulae for Mechanized Program Verification.* Ph.D. Dissertation. Université Paris 7.

Arthur Charguéraud. 2010b. The Optimal Fixed Point Combinator. In *Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Vol. 6172. Springer, 195–210.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*. 238–252.

Paulo Emílio de Vilhena, Jacques-Henri Jourdan, and François Pottier. 2020. Coq proofs for "Spy game". https://gitlab.inria.fr/pdevilhe/spy-game.

Christian Fecht and Helmut Seidl. 1999. A Faster Solver for General Systems of Equations. *Science of Computer Programming* 35, 2–3 (1999), 137–162.

R. W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics)*, Vol. 19. American Mathematical Society, 19–32.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*, Vol. 4807. Springer, 19–37.

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 4960. Springer, 353–367.

Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. 2010a. Verifying a Local Generic Solver in Coq. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 6337. Springer, 340–355.

Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. 2010b. What Is a Pure Functional?. In *International Colloquium on Automata, Languages and Programming (Lecture Notes in Computer Science)*, Vol. 6199. Springer, 199–210.

Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer.* Ph.D. Dissertation. Université Paris Diderot.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proceedings of the ACM on Programming Languages* POPL (Jan. 2020).

John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM* 23, 1 (1976), 158–171.

Aleksandr Karbyshev. 2013. *Monadic Parametricity of Second-Order Functionals.* Ph.D. Dissertation. Technische Universität München.

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Principles of Programming Languages (POPL)*. 194–206.

Baudouin Le Charlier and Pascal Van Hentenryck. 1992. *A Universal Top-Down Fixpoint Algorithm.* Technical Report CS-92-25. Brown University.

John Longley. 1999. When is a Functional Program Not a Functional Program?. In *International Conference on Functional Programming (ICFP)*. 1–7.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: a declarative language for fixed points on lattices. In *Programming Language Design and Implementation (PLDI)*. 194–208.

K. Muthukumar and M. V. Hermenegildo. 1990. *Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs.* Technical Report ACT-DC-153-90. Microelectronics and Computer Technology Corporation.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Luis Caires (Ed.), Vol. 11423. Springer, 1–27.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (2007), 271–307.

David Pichardie. 2008. Building Certified Static Analysers by Modular Construction of Well-founded Lattices. *Electronic Notes in Theoretical Computer Science* 212 (2008), 225–239.

François Pottier. 2009. Lazy Least Fixed Points in ML. (2009). Unpublished.

François Pottier. 2019. Fix. https://gitlab.inria.fr/fpottier/fix.

Helmut Seidl and Ralf Vogler. 2018. Three Improvements to the Top-Down Solver. In *Principles and Practice of Declarative Programming (PPDP)*. 21:1–21:14.

Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. 2012. *Compiler Design: Analysis and Transformation*. Springer.
Bart Vergauwen, J. Wauman, and Johan Lewi. 1994. Efficient fixpoint computation. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 864. Springer, 314–328.