

Spy game—verifying a local generic solver in Iris

PAULO EMÍLIO DE VILHENA, Inria Paris

FRANÇOIS POTTIER, Inria Paris

JACQUES-HENRI JOURDAN, CNRS, LRI, Univ. Paris Sud, Université Paris Saclay

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.

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 $(\mathcal{V} \rightarrow \mathcal{P}) \rightarrow (\mathcal{V} \rightarrow \mathcal{P})$, 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 \perp . One can think of \mathcal{E} as a “system of equations” because to every variable it associates a “right-hand side” of type $(\mathcal{V} \rightarrow \mathcal{P}) \rightarrow \mathcal{P}$ – 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 and more general, as follows: compute the *optimal least fixed point* of a monotone function \mathcal{E} of type $(\mathcal{V} \rightarrow \mathcal{P}) \rightarrow (\mathcal{V} \rightarrow \mathcal{P})$, 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.

Authors’ addresses: Paulo Emílio de Vilhena, Inria Paris, paulo-emilio.de-vilhena@polytechnique.edu; François Pottier, Inria Paris, francois.pottier@inria.fr; Jacques-Henri Jourdan, CNRS, LRI, Univ. Paris Sud, Université Paris Saclay, jacques-henri.jourdan@lri.fr.

2019. 2475-1421/2019/1-ART1 \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

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 this problem 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 $(\mathcal{V} \rightarrow \mathcal{P}) \rightarrow (\mathcal{V} \rightarrow \mathcal{P})$. 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 essentially identical 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 computing 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 $\mathcal{V} \rightarrow \mathcal{P}$, whose specification is simple: *get implements $\bar{\mu}\mathcal{E}$* . In other words, when *get* is applied to a variable v , it returns $\bar{\mu}\mathcal{E}(v)$. It achieves this while performing 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 $v \rightarrow w$. 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

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 does *not* guarantee that the solver terminates. In fact, it does not terminate everywhere, for two reasons. First, in general, 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

99 $v \in \text{dom}(\bar{\mu}\mathcal{E})$ holds. Second, even when v lies in $\text{dom}(\bar{\mu}\mathcal{E})$, the solver can still diverge; this can
 100 happen, for instance, if the partial order has infinite ascending chains.

101 Even though the solver is inherently sequential, it makes use of several locks, which serve as
 102 a runtime mechanism to defend against certain patterns of misuse. Quite obviously, if there are
 103 multiple user threads, then these locks forbid *concurrent* calls to the solver: in other words, they
 104 rule out data races on the solver’s internal mutable state. More subtly, even if there is only one user
 105 thread, they forbid *reentrant* calls, such as a call to *get* that takes place while an earlier call to *get* is
 106 still ongoing. (Although such a scenario may seem contrived and unlikely to arise, it is technically
 107 feasible.) In such an event, the solver will attempt to acquire a lock twice, resulting in a deadlock.¹
 108 Our proof does *not* eliminate the possibility of deadlocks. It *does* guarantee that we have set up
 109 sufficient runtime defenses for the solver’s internal side effects to be harmless.

110 1.4 Challenges and previous work

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

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

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

144
 145 ¹In practice, depending on how locks are implemented, some deadlocks may be detected and replaced with a graceful
 146 runtime failure. In the paper, we ignore this aspect and speak only of “deadlocks”.

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

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

165 1.5 Contributions

166 In this paper, we make the following contributions:

- 167 • We propose a specification of a local generic solver in Iris [Jung et al. 2018] We believe that
 168 this (third-order!) specification is remarkably simple and permissive. It is apparently pure:
 169 no side effects are mentioned. This has several consequences. On the one hand, the fact that
 170 the solver is *guaranteed* to be apparently pure means that the user need not worry about the
 171 solver’s internal use of mutable state. On the other hand, the fact that the user function *eqs*
 172 is *required* to be apparently pure means that it can have internal side effects, provided one
 173 can prove in Iris that *eqs* implements a mathematical function \mathcal{E} .
 174 Instead of imposing a strong condition to guarantee the existence of a total least fixed point,
 175 our specification relies on the concept of an optimal least fixed point, which exists as soon as
 176 the partially ordered set (\mathcal{P}, \leq) admits a least element \perp and the user-supplied function \mathcal{E} is
 177 monotone. In particular, we do not require \mathcal{V} to be finite, and do not require \mathcal{P} to satisfy the
 178 ascending chain condition.
- 179 • We make the simple yet novel remark that a local solver can use Longley’s *modulus* [1999]
 180 off the shelf. We split the solver’s implementation into two components, namely *modulus*, a
 181 stand-alone function, and *lfp*, which uses *modulus*. We also split the correctness proof: we
 182 verify each component independently. This requires proposing a suitable Iris specification
 183 for *modulus*, proving that *modulus* satisfies this specification, and exploiting solely this
 184 specification while reasoning about the use of *modulus* inside *lfp*.
- 185 • In the proof of *modulus*, we feel a need to exploit an *infinitary conjunction rule*, that is, a
 186 reasoning rule that states that $\forall x. \{P\} e \{y. Q x y\}$ implies $\{P\} e \{y. \forall x. Q x y\}$. Unfor-
 187 tunately, this rule is unsound; in fact, even the binary *conjunction rule*, which states that
 188 $\{P\} e \{y. Q_1 y\}$ and $\{P\} e \{y. Q_2 y\}$ imply $\{P\} e \{y. Q_1 \wedge Q_2 y\}$, is unsound in Iris, due to its
 189 interaction with ghost state. We work around this issue by establishing a weak variant of the
 190 infinitary conjunction rule where the postcondition $Q x y$ is required to be a pure proposition.
 191 We believe that this result and its proof are of independent interest.
- 192 • In the proof of the restricted infinitary conjunction rule (§6) and in the proofs of the functions
 193 *modulus* (§7) and *lfp* (§8), we exploit *prophecy variables*, a recent feature of Iris [Anonymous
 194 2018, 2019]. Thus, these proofs offer several original applications of prophecy variables.

Furthermore, we make two improvements to Iris’ prophecy variable API. We extend it with a new rule for disposing of a prophecy variable. We further extend it with support for typed prophecy variables. We find both features to be useful in our proofs.

- In the end, we provide a proof that *modulus* and *lfp* satisfy the proposed Iris specifications. Our proof is machine-checked in Coq.^{2 3} 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.

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 $(\mathcal{V} \rightarrow \mathcal{P}) \rightarrow (\mathcal{V} \rightarrow \mathcal{P})$. 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 $\mathcal{V} \rightarrow \mathcal{P}$ and its domain D , a subset of \mathcal{V} .

Following Charguéraud, we write $\text{dom}(\bar{f})$ 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 $\text{dom}(\bar{g})$ is a subset of $\text{dom}(\bar{f})$ and $f =_{\text{dom}(\bar{g})} g$ 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 $g =_{\text{dom}(\bar{f})} f$ implies $g =_{\text{dom}(\bar{f})} \mathcal{E} g$.

One could say, roughly speaking, that a partial fixed point is a partial function \bar{f} that satisfies the fixed point equation *on its domain* and *regardless of how it is extended outside of its domain*.

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

²We 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).

³Our proofs are supplied in a self-contained archive that has been submitted as anonymous supplemental material.

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 $f \leq_{\text{dom}(\bar{f}) \cap \text{dom}(\bar{g})} \bar{g}$.

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 $\text{dom}(\bar{\mu}\mathcal{E})$. In general, it is up to the user of the theory to establish a fact of the form $V \subseteq \text{dom}(\bar{\mu}\mathcal{E})$, 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:

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

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 (μ, \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.

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 $[P]$ is an Iris assertion; $P * Q$ is a separating conjunction; $P \multimap Q$ 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.

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: $T ::= val \mid unit \mid T + T \mid T \times T \mid list\ T$. 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 $\llbracket \cdot \rrbracket_T$ of type $T \rightarrow val$. 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 $\llbracket X \rrbracket_T$ *represents* X . We make this idea explicit by defining “ x represents X at type T ” as a proposition:

344 *Definition 3.1 (Represents / data).* The Coq proposition “ x represents X at type T ”, where x is a
 345 HeapLang value and X has type T , is defined as $x = \llbracket X \rrbracket_T$.

346
 347 The following convenient abuse of notation is used in our proofs (§7, §8):

348
 349 *Definition 3.2 (Typed points-to).* $m \mapsto X$ is short for $\exists x. (m \mapsto x * x \text{ represents } X \text{ at type } T)$.

350
 351 In this paper, it is often the case that a HeapLang function f is intended to behave like a
 352 mathematical function F . By this, we mean that, if the HeapLang value x represents the mathematical
 353 value X , then the function call $f x$ must return a HeapLang value y that represents $F(X)$, if it returns
 354 at all (it can also diverge).

355 To make this idea precise, and to make it work also at higher function types and at partial
 356 function types, we proceed as follows. We introduce a universe that includes the data types T as its
 357 base types and that features total function types and partial function types: $\tau ::= T \mid \tau \rightarrow \tau \mid \tau \hookrightarrow \tau$.
 358 Then, we define the assertion “ f implements F at type τ ” by induction over τ . The three cases of
 359 this definition are given in the next three definitions.

360
 361 *Definition 3.3 (Implements / data).* The Iris assertion “ x implements X at type T ”, where x is a
 362 HeapLang value and X has type T , is defined as $\llbracket x \text{ represents } X \text{ at type } T \rrbracket$.

363
 364 *Definition 3.4 (Implements / function).* The Iris assertion “ f implements F at type $\tau \rightarrow \tau'$ ”, where
 365 f is a HeapLang value and F has type $\tau \rightarrow \tau'$, is defined as the following triple:

$$366 \quad \forall x. \forall X. \{x \text{ implements } X \text{ at type } \tau\} f x \{y. y \text{ implements } F(X) \text{ at type } \tau'\}$$

367
 368 When the function f satisfies such a specification, we say that f is *apparently pure*. A pure
 369 function typically enjoys a specification of this form. However, an apparently pure function is not
 370 necessarily pure in a strong sense. Indeed, an apparently pure function can have internal side effects:
 371 its execution can involve spawning a thread, acquiring and releasing a lock, updating a piece of
 372 mutable state that is protected by a lock, and so on. As a consequence, the fact that f is apparently
 373 pure does *not* imply that the expressions $(f x, f x)$ and $\text{let } y = f x \text{ in } (y, y)$ are observationally
 374 equivalent. It *does* guarantee a weak form of determinism: two calls to $f x$ must return the same
 375 result, if they both terminate. It *does* allow a user to reason about f , by applying the reasoning
 376 rules of Iris, in exactly the same way as if f was pure.

377 In this paper, it is sometimes the case that a function f implemented in HeapLang is intended to
 378 behave like a mathematical *partial* function \bar{F} . (We have explained in §2 how we encode partial
 379 functions in Coq.) There are at least two distinct ways in which this can be expressed as a triple.
 380 A *strict* way is to let the assertion $\llbracket X \in \text{dom}(\bar{F}) \rrbracket$ appear in the precondition, which means that the
 381 caller has the obligation to prove that the argument lies within the domain of the partial function.
 382 A *lax* way is to impose no such requirement and instead adopt the convention that the function call
 383 $f x$ must diverge if the argument lies outside the domain. This convention is reflected by letting
 384 the assertion $\llbracket X \in \text{dom}(\bar{F}) \rrbracket$ appear in the postcondition. This allows a user to reason that, if the
 385 call returns, then definitely the argument was in the domain. The lax interpretation is used in the
 386 following definition, because it happens to be what we need in the specification of lfp (§4).

387
 388 *Definition 3.5 (Implements / lax partial function).* The Iris assertion “ f implements \bar{F} at type $\tau \hookrightarrow \tau'$ ”,
 389 where f is a HeapLang value and \bar{F} has type $\tau \hookrightarrow \tau'$, is defined as the following triple:

$$391 \quad \forall x. \forall X. \{x \text{ implements } X \text{ at type } \tau\} f x \{y. y \text{ implements } F(X) \text{ at type } \tau' * \llbracket X \in \text{dom}(\bar{F}) \rrbracket\}$$

393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441

PROPHECY ALLOCATION $\{true\}$ $newProph()$ $\{p. \exists xs. p \text{ will receive } xs\}$	PROPHECY ASSIGNMENT $\{p \text{ will receive } xs\}$ $resolveProph \ p \ x$ $\left\{ \begin{array}{l} () . \exists xs'. \ [xs = x :: xs'] \\ p \text{ will receive } xs' \end{array} \right\}$	PROPHECY DISPOSAL $\{p \text{ will receive } xs\}$ $disposeProph \ p$ $\{(). \ [xs = []]\}$
--	---	--

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

TYPED PROPHECY ALLOCATION $\{true\}$ $newProph()$ $\{p. \exists Xs. p \text{ will receive } Xs\}$	TYPED PROPHECY ASSIGNMENT $\left\{ \begin{array}{l} [x \text{ represents } X \text{ at type } T] \\ p \text{ will receive } Xs \\ resolveProph \ p \ x \end{array} \right\}$ $\left\{ \begin{array}{l} (). \exists Xs'. \ [Xs = X :: Xs'] \\ p \text{ will receive } Xs' \end{array} \right\}$	TYPED PROPHECY DISPOSAL $\{p \text{ will receive } Xs\}$ $disposeProph \ p$ $\{(). \ [Xs = []]\}$
--	--	--

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

3.3 Prophecy variables

In several places,⁴ our proofs rely on *prophecy variables*, a concept originally due to Abadi and Lamport [1988], which has recently appeared in Iris⁵ [Anonymous 2018, 2019]. 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 $\exists xs. p \text{ will receive } xs$, 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

⁴Prophecy 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).

⁵To the anonymous reviewers: We have learned that a paper on prophecy variables in Iris has been concurrently submitted to POPL 2020. We intend to cite it once it is publicly available. In this section, we present our understanding of Iris' primitive prophecy variables, so as to make our paper self-contained. Furthermore, we present two improvements to Iris' prophecy variables.

sequence by anticipation. The assertion p will receive xs is affine, and appears as a precondition in 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

We remark that it seems natural (and it is useful in practice) to extend the API of prophecy variables 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' 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' primitive prophecy variables. The implementation is not difficult: 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:

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

Next, we remark that Iris' primitive prophecy variables are *untyped*. In the rules of Figure 1, x and xs respectively have types val and $list \text{ } val$, 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. 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 $\llbracket \cdot \rrbracket_T$ of type $T \rightarrow val$. 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 $list \text{ } T$. The assertion p will receive Xs is defined in terms of a lower-level untyped-prophecy assertion as follows:

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

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

$$\begin{array}{l}
491 \\
492 \\
493 \\
494 \\
495 \\
496 \\
497 \\
498 \\
499 \\
500 \\
501 \\
502 \\
503 \\
504 \\
505 \\
506 \\
507 \\
508 \\
509 \\
510 \\
511 \\
512 \\
513 \\
514 \\
515 \\
516 \\
517 \\
518 \\
519 \\
520 \\
521 \\
522 \\
523 \\
524 \\
525 \\
526 \\
527 \\
528 \\
529 \\
530 \\
531 \\
532 \\
533 \\
534 \\
535 \\
536 \\
537 \\
538 \\
539
\end{array}
\quad
\begin{array}{l}
\forall eqs. \forall \mathcal{E}. (\mathcal{E} \text{ is monotone}) \quad \Rightarrow \\
\{eqs \text{ implements } flip \mathcal{E} \text{ at type } \mathcal{V} \rightarrow (\mathcal{V} \rightarrow \mathcal{P}) \rightarrow \mathcal{P}\} \\
\quad \quad \quad lfp \text{ eqs} \\
\{get. get \text{ implements } \bar{\mu}\mathcal{E} \text{ at type } \mathcal{V} \hookrightarrow \mathcal{P}\}
\end{array}$$

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

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 $\mathcal{V} \rightarrow \mathcal{P}$ ”, 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. These parameters exist at runtime: they are *HeapLang* values.

The specification of a solver is further parameterized with a number of objects or properties that exist at the mathematical level, but do not exist at runtime. The type \mathcal{P} must be equipped with a relation \leq , which must be a partial order, and with an element \perp , which must be the least element of this partial order. The value *Prop.bottom* must satisfy the specification “*Prop.bottom* represents \perp at type \mathcal{P} ”. The value *Prop.eq* must satisfy the specification “*Prop.eq* implements (=) at type $\mathcal{P} \rightarrow \mathcal{P} \rightarrow bool$ ”. 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 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 F of type $\mathcal{V} \rightarrow \mathcal{P}$. It must then return the property $\mathcal{E} F v$.

⁶*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.

```

540 1 let modulus ff f =
541 2   let m, p, lk = ref [], newProph(), newLock() in
542 3   let spy x =
543 4     let y = f x in
544 5     withLock lk (fun () -> m := x :: !m; resolveProph p x);
545 6     y
546 7   in
547 8   let c = conjApply ff spy in
548 9   acquireLock lk;
549 10  disposeProph p;
550 11  (c, !m)

```

Fig. 4. The function modulus

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 solver implementor 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 $v \in \text{dom}(\bar{\mu}\mathcal{E})$ is guaranteed to hold. By contraposition, an application of *get* to a variable v outside $\text{dom}(\bar{\mu}\mathcal{E})$ is guaranteed to diverge. It may seem odd that the user is not *required* to prove $v \in \text{dom}(\bar{\mu}\mathcal{E})$. 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], with a few changes. First, we note that the technique of “spying” is of general interest and can be implemented independently as a third-order function, *modulus*, whose code appears in Figure 4. This function has in fact been documented and studied by Longley [1999], but the remark that *modulus* can be used off the shelf in the construction of a local solver is novel. Second, for the sake of simplicity, we represent the dependency graph (an internal data structure) in a somewhat naïve way, which gives us efficient access to the successors

```

589 1 let lfp eqs =
590 2   let permanent, transient, dependencies, todo =
591 3     Var.Map.create(), Var.Map.create(), Var.Map.create(), ref [] in
592 4   let master = newLock() in
593 5   let schedule v =
594 6     Var.Map.insert dependencies v [];
595 7     todo := v :: !todo
596 8   in
597 9   let discover v =
598 10    Var.Map.insert transient v Prop.bottom;
599 11    schedule v
600 12  in
601 13  let reevaluate v =
602 14    let p, l = newProph(), newLock() in
603 15    let request w =
604 16      withLock l (fun () ->
605 17        match Var.Map.lookup permanent w with Some c -> c | None ->
606 18        match Var.Map.lookup transient w with Some c -> c | None ->
607 19        discover w; resolveProph p w; Prop.bottom)
608 20    in
609 21    let c, ws = modulus (eqs v) request in
610 22    acquireLock l;
611 23    disposeProph p;
612 24    Var.Map.insert dependencies v ws;
613 25    if not (Prop.eq c (Var.Map.lookup_exn transient v)) then begin
614 26      Var.Map.insert transient v c;
615 27      List.iter schedule (predecessors dependencies v)
616 28    end
617 29  in
618 30  let rec loop () =
619 31    match !todo with [] -> () | v :: vs ->
620 32      todo := vs; reevaluate v; loop()
621 33  in
622 34  let get v =
623 35    withLock master (fun () ->
624 36      match Var.Map.lookup permanent v with
625 37      | Some c -> c
626 38      | None ->
627 39        discover v; loop();
628 40        Var.Map.transfer transient permanent;
629 41        Var.Map.flush dependencies;
630 42        Var.Map.lookup_exn permanent v)
631 43  in
632 44  get
633
634
635
636
637

```

Fig. 5. The function lfp

of a vertex, but not to its predecessors. Pottier [2009] uses a more efficient representation, which gives constant-time access to successors and predecessors. Although we do not anticipate any deep difficulty in verifying it, we leave this effort to future work.

Our solver appears in Figure 5. We remind the reader that the code that we actually verify is expressed in HeapLang syntax inside Coq (§3.1). The pseudocode in Figure 5 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. We assume that the module *Var.Map* offers an implementation of maps whose keys are variables. We assume that the module *Prop* provides access 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 44). 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, which are long-lived, that is, which persist across invocations of *get*. These data structures, allocated on lines 2–3, 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.
- 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 $u \rightarrow v$ 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 4, protects these data structures. Indeed, the body of *get* forms a critical section, protected by this lock (lines 35–42). 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 37). Otherwise (line 39), 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 40), 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 41). At this point, the variable *v* must be in the domain of the permanent map, so the lookup at line 42 must succeed.

The function *discover* (line 9) inserts a newly discovered variable into the transient map, where it is initially associated with the property \perp , and schedules it for evaluation. The function *schedule* (line 5) removes *v*’s outgoing dependency edges (if there are any) and inserts *v* into the workset.

687 During a wave, the solver repeatedly extracts a variable v out of the workset and reevaluates
 688 it (line 30). The function *reevaluate* (line 13) is in charge of this. To this end, the user-supplied
 689 function *eqs*, which implements *flip E*, should be applied to the variable v and to a function, *request*,
 690 which implements our current under-approximation of the fixed point.

691 The function *request* (line 15), looks up the permanent map first, then the transient map. If
 692 both lookups fail, then it schedules its argument for later evaluation and returns \perp , a safe under-
 693 approximation. It too must be protected by a lock l . The critical section at lines 16–19 prevents
 694 concurrent calls to *request*. The final lock acquisition on line 22 blocks any invocation of *request*
 695 after the function call *eqs v request* on line 21 has returned.

696 Coming back to *reevaluate*, precisely on line 21, where one might expect a function call of
 697 the form *let c = eqs v request*, one finds that this call is wrapped in an invocation of the higher-
 698 order function *modulus*, like so: *let c, ws = modulus (eqs v) request*. In short, *modulus* performs the
 699 application of *eqs v* to *request* and returns not only its result c , a property, but also a list ws of the
 700 variables at which *request* is invoked during this function application. This is achieved by wrapping
 701 *request* into a spy function. The implementation of *modulus* is shown in Figure 4 and explained
 702 later on (§7). Coming back to *reevaluate*, still at line 21, the property c is the result of the function
 703 call *eqs v request*, and it is known that c has been computed based solely on the properties that are
 704 currently associated with the variables ws . As long as the latter properties do not change, there is
 705 no need to reevaluate v . Thus, we record that v depends (only) on ws (line 24).

706 Still inside *reevaluate*, on line 25, the property c , which should be newly associated with v , is
 707 compared with the property currently associated with v in the transient map. If they are equal,
 708 then v is stable with respect to its predecessors, and there is nothing else to do. Otherwise, the
 709 transient map is updated at v (line 26), which implies that the predecessors of v in the dependency
 710 graph must be scheduled for reevaluation (line 27).

711

712 6 A RESTRICTED INFINITARY CONJUNCTION RULE

713 6.1 Unsoundness of the unrestricted conjunction rule

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

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

INFINITARY CONJUNCTION (UN SOUND)

$$\frac{\forall x. \{P\} e \{y. Q x y\}}{\{P\} e \{y. \forall x. Q x y\}}$$

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

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

730 When read in this way, this rule seems valid. Indeed, this reading relies on the traditional “must”
 731 interpretation of Hoare triples, where $\{P\} e \{y. R y\}$ means that *every* possible final state *must*
 732 satisfy the postcondition $R y$. Then, the explicit universal quantification over x and the implicit
 733 universal quantification over the final state can be exchanged, which seems to justify the rule.
 734 Indeed, in traditional Hoare logic, the binary and infinitary conjunction rules are sound. The binary
 735

$$\begin{array}{c}
736 \\
737 \\
738 \\
739 \\
740 \\
741 \\
742 \\
743
\end{array}
\begin{array}{l}
\text{(CANDIDATE RULE)} \\
\frac{\forall x. \{P\} e \{y. [Q x y]\}}{\{P\} e \{y. [\forall x. Q x y]\}}
\end{array}
\begin{array}{c}
\text{PURE INFINITARY CONJUNCTION} \\
\frac{\forall x. \{P\} e \{y. [Q x y]\}}{\left(\begin{array}{l} \text{let } p = \text{newProph}() \text{ in} \\ \text{let } y = e \text{ in} \\ \text{resolveProph } p \ y; \\ y \end{array} \right) \{y. [\forall x. Q x y]\}}
\end{array}$$

Fig. 6. The pure infinitary conjunction rule

$$\begin{array}{c}
744 \\
745 \\
746 \\
747 \\
748 \\
749 \\
750 \\
751 \\
752
\end{array}
\begin{array}{l}
\text{PURE IMPLICATION} \\
\frac{[Q] \multimap \{P\} e \{y. R\} \quad \{P\} e \{y. \text{true}\}}{\{P\} e \{y. [Q] \multimap R\}}
\end{array}$$

Fig. 7. The pure implication rule

753 conjunction rule appears as Axiom 1 in Floyd’s seminal paper [1967], where it is argued that it is
754 sound with respect to the semantic interpretation of triples.

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

761 One could say that this problem arises out of the interaction between the conjunction rule
762 and ghost state: in the counter-example, the two derivations are incompatible because they *make*
763 *different decisions* about who owns the memory cell. This is ghost information, not recorded at
764 runtime in the physical state. In Iris, which has a very general notion of ghost state, the same
765 problem exists: the binary and infinitary conjunction rules are unsound. One way to understand
766 this is to recall that the definition of an Iris triple includes a basic update modality [Jung et al.
767 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
768 possible final physical state, *there exists* a way of updating the ghost state such that the combined
769 physical and ghost state satisfies the postcondition $R y$. The presence of this existential quantifier
770 explains why the conjunction rule is unsound: an existential quantifier and a universal quantifier
771 cannot be exchanged.

772 6.2 A sound, restricted infinitary conjunction rule

773 The unsoundness of the conjunction rule arises out the fact that the postcondition of an Iris triple
774 is an arbitrary Iris assertion, therefore can impose a constraint on the ghost state. Thus, it seems
775 that there is a chance that the conjunction rule might become sound if one restricts or removes
776 the ability of the postcondition to describe the ghost state. For instance, one might require the
777 postcondition to be pure. (In Iris, a *pure assertion* is one that is independent of the current state
778 and step index.) The rule would then take the form shown in Figure 6 (left). Again, we assume
779 that x does not occur in P, e, y and that the type of x is inhabited. This is a candidate rule: it is
780 likely sound, yet we do not know how to prove it sound under this exact form. In the following, we
781 first give an informal proof sketch for this candidate rule, then present an almost identical rule that
782 we *are* able to prove sound.

Proof sketch. Let us suppose that $\forall x. \{P\} e \{y. [Q x y]\}$ holds (1). We wish to argue that the triple $\{P\} e \{y. [\forall x. Q x y]\}$ 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 $\{P\} e \{y. true\}$, which means that it is safe to execute e . Hypothesis (1), instantiated with an arbitrary value of x , yields $\{P\} e \{y. [Q x y]\}$, which (by the consequence rule) implies the goal.
- *Case 2:* $\forall x. Q x y$ does not hold. Then, the goal degenerates to $\{P\} e \{y. false\}$, 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 $\{P\} e \{y. [Q x y]\}$. Because $Q x y$ is false, this is equivalent to $\{P\} e \{y. false\}$, 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 6 (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 *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 7. This rule allows hoisting an implication, whose left-hand side is a pure assertion $[Q]$, 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 $\{P\} e \{y. true\}$, 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. As explained earlier (§5), the function call *modulus* $ff f$ is supposed to perform an application of ff to f and to return a

pair (c, ws) of the result of this application and a list of points at which f was invoked during this application.

7.1 Implementation

The implementation of *modulus*, which appears in Figure 4, is simple. A reference m is allocated to keep track of the points where f is invoked (line 2). The function f is wrapped in an auxiliary function, *spy* (line 3), 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 5), and the call is forwarded to f (line 4). The function *ff* is applied to *spy*, instead of f , on line 8. The combinator *conjApply*, which was introduced at the end of §6.2, behaves like function application.

A lock lk , allocated on line 2, protects the memory location m . The critical section on line 5 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 9, 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 ff₁ f₁* and *modulus ff₂ f₂* 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 “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 is it that *modulus* computes?” 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.

⁷Our *modulus* corresponds to the function *Mod'* in Longley’s paper.

⁸Our 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.

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

Fig. 8. The specification of *modulus*

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 ϕ and \mathcal{F} . 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 $\forall \phi'. \phi' =_{\text{ws}} \phi \Rightarrow \mathcal{F}(\phi') = \mathcal{F}(\phi)$.

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*”.

It is worth noting that *ws* is existentially quantified in the postcondition. There is no guarantee that *ws* can be expressed as a function of *ff* and *f*. In other words, there is no guarantee that *modulus* itself is not an apparently pure function, and indeed, as argued earlier, it is not.

7.3 Proof

How can one prove that the function *modulus*, whose code appears in Figure 4, 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 ϕ and \mathcal{F} . Now, the main difficulty in the proof that *modulus* is safe is to argue that the function call *ff spy* on line 8 of Figure 4 is permitted. The only fact that is known about *ff* is “*ff implements F at type (T_a → T_b) → 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 → 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.

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

938 One idea that might come to mind is to strengthen the lock invariant so as to claim that this
 939 assertion holds at every time of the list ws *currently* stored at location m . However, this candidate
 940 invariant is much too strong: in particular, m initially holds the empty list, so, in order to establish
 941 this invariant, we would have to prove $\forall \phi'. \mathcal{F}(\phi') = \mathcal{F}(\phi)$, which is hopeless. Put another way, the
 942 fact that “the memory location m stores a sound set of dependencies” *becomes true eventually*, once
 943 the call *ff spy* on line 8 returns, but *is not true* while this call is in progress.

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

950
 951 *Why modulus is correct (rough proof sketch).* By anticipation, let ws refer to the list that is
 952 *eventually* stored at location m , just before *modulus* returns. Then, we would like to claim not
 953 only that *spy* implements ϕ , as in the previous proof sketch, but in fact that *spy* implements every
 954 function ϕ' such that $\phi' =_{ws} \phi$ holds. If we can establish this claim, then we can conclude not only
 955 that c represents $\mathcal{F}(\phi)$, as in the previous proof sketch, but also that c represents every $\mathcal{F}(\phi')$, out
 956 of which the desired result follows. Thus, assuming that an arbitrary function ϕ' is given, such
 957 that $\phi' =_{ws} \phi$ holds, there remains to argue that *spy* implements ϕ' . To do so, we must prove that,
 958 when *spy* is applied to some argument x , it returns $\phi'(x)$. However, because the value returned by
 959 *spy* is the result of the call $f x$, and because f implements ϕ , it is clear that *spy* in fact returns $\phi(x)$.
 960 So, there remains to prove $\phi'(x) = \phi(x)$. Because the functions ϕ' and ϕ agree on ws , this boils
 961 down to proving that x is a member of the list ws . Because ws is the list that is *eventually* stored at
 962 location m , and because *spy* has the effect of inserting x into this list, it is obvious that this is true.
 963 (Remember, this is a rough sketch, not a proof!) This ends the proof sketch.

964 In order to turn this proof sketch into a valid proof in Iris, a number of technical issues must
 965 be overcome. First, it appears that, at the very beginning of the proof, we must be able to refer to
 966 the list ws that is eventually computed by *modulus*. This reference to the future seems to call for a
 967 prophecy variable. Second, although the proof sketch claims that any value that is inserted into
 968 the list at some point is “obviously” also a member of the list at the end, making this argument
 969 precise requires coming up with a suitable invariant. Third, although we have argued informally
 970 that “*spy* implements every function ϕ' such that $\phi' =_{ws} \phi$ holds”, at the point where we wish to
 971 justify the function call *ff spy*, we must pick *one* function ψ such that *spy* implements ψ . Indeed,
 972 this is imposed on us by the specification of *ff*. So, we must focus on one function ϕ' and carry
 973 out a proof of *modulus* with respect to this specific ϕ' . Once that is done, we wish to argue that
 974 “if *modulus* is correct with respect to an arbitrary function ϕ' , then it is correct with respect to all
 975 such functions at once”. This is where an infinitary conjunction rule is called for. Fortunately, the
 976 postcondition of *modulus* is a pure assertion, so our restricted rule (§6) is applicable.

977
 978 *Why modulus is correct (detailed proof sketch).* As before, the precondition of *modulus* lets us
 979 assume that the user-provided functions *ff* and f implement two mathematical functions ϕ and \mathcal{F} .

981 These assertions are persistent, which means that they remain available throughout the proof. At
 982 line 2 of Figure 4, the memory location m is initialized with the empty list. A prophecy variable p
 983 is allocated. It is “typed” at type T_a ; see §3.4. At this point, we introduce the name ws to stand for the
 984 list of values that will be written to p in the future: initially, the assertion “ p will receive ws ” holds.
 985 Still on line 2, a lock lk is allocated. For this lock, we must choose an invariant that relates m and p .
 986 Our intuition is that, at any point in time, the list stored at location m contains the points at which
 987 *spy* has been invoked in the past, while the list held in the prophecy variable p contains the points at
 988 which *spy* will be invoked in the future. At any time, the combination of these lists forms the list ws ,
 989 that is, the complete list that is eventually recorded. Thus, the lock invariant is as follows:

$$990 \quad \exists us \text{ vs. } \left\{ \begin{array}{l} m \mapsto us * \\ p \text{ will receive } vs * \\ [ws = \text{reverse } us ++ vs] \end{array} \right.$$

994 Instantiating us with the empty list and vs with ws shows that this invariant initially holds. Also,
 995 anticipating on the proof of *spy*, it is easy to check that the critical section at line 5 of Figure 4
 996 preserves this invariant, and it is easy to see that this invariant allows establishing at line 5 that
 997 x is a member of the list ws .

998 Skipping (for now) over the definition of *spy*, the next step is to justify the application of ff
 999 to *spy* at line 8. We wish to establish the following triple:

$$1000 \quad \{true\} \text{ conjApply } ff \text{ spy } \{c. [\forall \phi'. \phi' =_{ws} \phi \Rightarrow c \text{ represents } \mathcal{F}(\phi') \text{ at type } T_c]\}$$

1002 By the pure infinitary conjunction rule (§6.2) and by the pure implication rule (§6.3), this boils
 1003 down to proving the following two triples:

$$1004 \quad \forall \phi'. [\phi' =_{ws} \phi] * \{true\} ff \text{ spy } \{c. [c \text{ represents } \mathcal{F}(\phi') \text{ at type } T_c]\}$$

$$1005 \quad \{true\} ff \text{ spy } \{c. true\}$$

1007 The second one is unproblematic; we have already sketched why the function call $ff \text{ spy}$ is safe.
 1008 Thus, let us focus on the first one. We assume that some function ϕ' is given, such that $\phi' =_{ws} \phi$
 1009 holds, and we must prove:

$$1010 \quad \{true\} ff \text{ spy } \{c. [c \text{ represents } \mathcal{F}(\phi') \text{ at type } T_c]\}$$

1012 Since ff implements \mathcal{F} , in order to establish this triple, it suffices to show that *spy* implements ϕ' .
 1013 We have already explained (in the previous proof sketch) that this follows from the assumption
 1014 $\phi' =_{ws} \phi$ and from the fact that at line 5 one can prove that x is a member of the list ws .

1015 Thus, at line 9, the proposition “ $\forall \phi'. \phi' =_{ws} \phi \Rightarrow c \text{ represents } \mathcal{F}(\phi') \text{ at type } T_c$ ” holds. It might
 1016 seem as though we are done; still, an important argument remains to be made, namely the fact that
 1017 the final read of m on line 11 yields the list *reverse ws*. Indeed, the final lock acquisition on line 9
 1018 gives us access to the lock invariant, and the prophecy disposal instruction on the following line
 1019 yields the information that the list vs of future writes is empty. Therefore, at line 11, we are able
 1020 to deduce that the list stored at m is *reverse ws*. At this point, checking that the postcondition of
 1021 *modulus* is satisfied is routine; the witness for the existential quantifier $\exists ws. \dots$ in the postcondition
 1022 of *modulus* is in fact *reverse ws*.

1024 8 PROOF OF THE SOLVER

1025 There remains to verify that the function *lfp*, whose code appears in Figure 5, satisfies the specifi-
 1026 cation shown in Figure 3. We do not explain the proof in detail, but highlight its main points of
 1027 interest: in particular, we give the invariants associated with the two locks and the loop invariant
 1028 associated with the function *loop*.

1030 In the following, if P is a finite map of variables to properties, we write P^\perp for the total function
 1031 of type $\mathcal{V} \rightarrow \mathcal{P}$ that agrees with the map P on its domain and that maps every variable outside of
 1032 $\text{dom}(P)$ to \perp . Also, if ws is a list of variables, we write $ws \mapsto x$ for the finite map that maps every
 1033 variable in the list ws to x .

1034 The invariant that we associate with the master lock (line 4) is the following:

$$1035 \quad \exists P. \left\{ \begin{array}{l} \text{isMap permanent } P * \text{isMap transient } \emptyset * \text{isMap dependencies } \emptyset * \text{todo} \mapsto [] * \\ \text{the function } P^\perp \text{ is consistent up to } \leq \end{array} \right.$$

1036 Thus, when the master lock is available (that is, when *no* call to *get* is in progress), the permanent
 1037 map stores a fragment P of the optimal least fixed point: this is stated by the proposition “the
 1038 function P^\perp is consistent up to \leq ”. Furthermore, the transient map, the dependency graph, and the
 1039 workset are empty.

1040 The main loop invariant (that is, the precondition of the function *loop*) is as follows:

$$1041 \quad \exists T D us. \left\{ \begin{array}{l} \text{isMap transient } T * \text{isMap dependencies } D * \text{todo} \mapsto us * \\ \text{dom}(T) = \text{dom}(D) * [us \subseteq \text{dom}(T)] * [\forall v \in \text{dom}(D). D(v) \subseteq \text{dom}(P \cup T)] * \\ \forall v \in \text{dom}(T). v \in us \vee \forall \phi. \phi =_{D(v)} (P \cup T)^\perp \Rightarrow T(v) = \mathcal{E}(\phi)(v) * \\ \forall \tilde{f}. \tilde{f} \text{ is a partial fixed point of } \mathcal{E} \Rightarrow T^\perp \leq_{\text{dom}(\tilde{f})} \tilde{f} \end{array} \right.$$

1042 Let us say that a variable is *permanent* if it appears in the domain $\text{dom}(P)$ of the permanent map
 1043 and *transient* if it appears in the domain $\text{dom}(T)$ of the transient map. Together, the propositions
 1044 on the second line above state that every variable in the workset us is transient, that the source of
 1045 every edge in the dependency graph is transient, and that the target of every such edge is either
 1046 permanent or transient. The proposition on the third line above states that every transient variable v
 1047 either appears in the workset ($v \in us$) or is stable with respect to its successors in the dependency
 1048 graph ($\forall \phi. \dots$). This expresses the intuition that it is fine to *not* reevaluate the variables that are
 1049 *not* in the workset. The proposition on the last line above indicates that the partial function T^\perp
 1050 under-approximates every partial fixed point. Together, the last two propositions allow proving
 1051 that, once the workset us becomes empty, the function $(P \cup T)^\perp$ is consistent up to \leq , therefore is a
 1052 fragment of the optimal least fixed point. This justifies transferring all of the information contained
 1053 in the transient map into the permanent map at line 40.

1054 Finally, the invariant associated with the auxiliary lock at line 14 is as follows:

$$1055 \quad \exists ws ws'. \left\{ \begin{array}{l} \text{isMap transient } (T \cup (ws \mapsto \perp)) * \\ \text{isMap dependencies } (D \cup (ws \mapsto [])) * \\ \text{todo} \mapsto ws ++ us * \\ p \text{ will receive } ws' * [vs = \text{reverse } ws' ++ ws] \end{array} \right.$$

1056 In this invariant, the variable vs occurs free. We let this name refer, by anticipation, to the list
 1057 of all variables that are newly discovered (at line 19) during the execution of the function call
 1058 *modulus*(*eqs* v) *request* at line 21. Thus, the above invariant expresses the fact that a mapping of
 1059 every newly discovered variable to \perp is added to the transient map, that every such variable is
 1060 recorded in the dependency graph (with no outgoing edges), and that every such variable is inserted
 1061 into the workset.

1062 A subtle technical point, which we mention only briefly, in the interest of space, is that we must
 1063 control the set ws returned by *modulus* on line 21. That is, we must establish an upper bound on
 1064 this set: in other words, we must prove that *modulus* cannot, out of the blue, report dependencies
 1065 on variables that have never been encountered before. To do so, we first argue that the function
 1066 *request* (line 15) implements (in the lax sense of Definition 3.5) a partial function whose domain is
 1067 $\text{dom}(P \cup T) \cup vs$, that is, the variables that are known at the beginning of *reevaluate*, plus those
 1068

that are newly discovered during the execution of *reevaluate*. From this fact, we deduce that the set ws returned by *modulus* on line 21 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 24) 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]. We have also verified a version of Longley’s *modulus* [1999]. We verify this code in the setting of HeapLang, an imperative, concurrent programming language. Thus, the correctness results that we establish hold even if the user exploits mutable state or concurrency, as long as the user respects the requirements imposed by our specifications of *modulus* and *lfp*: in short, the arguments that the user passes to *modulus* and *lfp* must be apparently pure functions.

Our code involves several locks, whose presence seems required in order to rule out data races, as HeapLang is a concurrent language. Iris’ reasoning rules for locks do not rule out the possibility of a deadlock, and indeed it is possible in principle for a deadlock to occur in certain scenarios. Informally, one can argue that the use of a lock inside *modulus* cannot cause a deadlock, because the critical section governed by this lock does not contain any call to a user function. One would like to argue similarly concerning the lock inside *reevaluate*. However, the critical section governed by this lock (lines 16–19 in Figure 5) contains calls to the user-provided functions *Var.Map.lookup* and *Var.Map.insert*. So, in order to prove the impossibility of a deadlock, we would have to somehow forbid these functions from calling *request*.¹⁰ This could perhaps be achieved by moving to a more expressive program logic, which imposes the existence of a locking order, and where the precondition of a function or expression indicates the level of the locks that it is allowed to acquire. Chalice [Leino and Müller 2009] is an example. The master lock, which governs a critical section in the body of *get* (lines 35–42), will also cause a deadlock in the event of a reentrant call to *get*.

It might be possible to remove all locks altogether and instead expose the existence of a unique permission to invoke each of the solver functions (*get* and *request*). This would remove all risk of a deadlock, but would impose restrictions on the use of these functions, as they would no longer be apparently pure. For instance, using the function *get* produced by one instance of *lfp* in the construction of the argument *eqs* of another instance of *lfp*, which is perfectly legitimate according to our current specification of *lfp*, would be forbidden if *get* was not apparently pure.

We verify only the partial correctness of *modulus* and *lfp*. We do prove that if the call *get v* terminates then v must be in $dom(\bar{\mu}\mathcal{E})$, the domain of the optimal least fixed point. In a logic of total correctness, one might hope to prove that the converse is true as well, that is, if v is in $dom(\bar{\mu}\mathcal{E})$, then the call *get v* terminates normally. However, moving to total correctness would imply proving the absence of deadlocks, which is nontrivial, as discussed above.

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 19), 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

⁹This 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 $dom(\bar{\phi})$.

¹⁰It would be very bizarre if they did call *request*. Yet, this is technically not ruled out by our specification: the user could in theory leak the function *request* by storing its address in a reference to which *Var.Map.lookup* has access.

1128 yet attempted to verify such a variant. We expect its verification to be more difficult, as it mixes
1129 iterative and recursive aspects.

1130

1131

10 RELATED WORK

1132 Several on-demand, incremental, memoizing fixed point computation algorithms appear in the liter-
1133 ature [Fecht and Seidl 1999; Le Charlier and Van Hentenryck 1992; Muthukumar and Hermenegildo
1134 1990; Pottier 2009; Vergauwen et al. 1994]. Apinis et al. [2016] extend such a solver with support for
1135 widening and narrowing, two standard operations in the realm of abstract interpretation. Seidl and
1136 Vogler [2018] further extend this solver so that (1) termination is guaranteed even in the absence of
1137 a monotonicity hypothesis; (2) space usage is reduced, thanks to a form of selective memoization;
1138 and (3) the construction and the resolution of the system of equations can be intertwined through
1139 so-called “side effects”. Point (2) is an improvement that could probably be incorporated into our
1140 solver without requiring any change to its specification. The other extensions mentioned above
1141 seem to require altering the specification of the solver; they would constitute interesting directions
1142 for future research.

1143 An entirely different approach to the computation of least fixed points is to propose a domain-
1144 specific language, such as FLIX [Madsen et al. 2016], in which systems of monotone equations can
1145 be described and to compile this language down to efficient executable code.

1146 Longley [1999] discusses the function *modulus* and notes that, even though its implementation
1147 is imperative, its behavior *in the setting of PCF* remains pure, so that PCF extended with *modulus*
1148 remains a pure calculus, which can express only “sequentially realizable” functions, and can express
1149 all of them. We have given earlier a more detailed comparison with Longley’s work (§7.1).

1150 Hofmann, Karbyshev and Seidl [2010a] use Coq to verify a simplified model of a local generic
1151 solver. More details appear in Karbyshev’s dissertation [2013]. We have given earlier a brief
1152 comparison with their work (§1.4). In short, both the solver and its user are modeled in Coq in
1153 terms of functions, relations, and explicit state-passing. In particular, the user-provided function *eqs*
1154 is modeled as a *strategy tree*. The fact that it is safe to model the behavior of *eqs* in this way is the
1155 subject of two separate papers [Bauer et al. 2013; Hofmann et al. 2010b], where it is argued that
1156 every second-order function that is “pure” in a certain semantic sense can indeed be represented as
1157 a strategy tree. In contrast, we work in the setting of a program logic for HeapLang, a λ -calculus
1158 extended with mutable state and shared-memory concurrency, which obviates the need to construct
1159 a mathematical model of the solver and of its user and the need to argue that this model is faithful
1160 in some sense. We rely on the notion of *apparent purity* that is naturally provided by Iris: a
1161 function f is apparently pure if it satisfies a specification of the form f implements ϕ (Definition 3.4).
1162 Such a specification allows the function f to have internal state, as long as this state is properly
1163 encapsulated; this is typically achieved by protecting it with a lock. The notion of apparent purity
1164 is used both ways in our specification of the solver. On the one hand, we require the user-provided
1165 function *eqs* to be apparently pure. On the other hand, we guarantee that the functions exposed by
1166 the solver to the user (namely *get* and *request-wrapped-inside-spy*) are apparently pure.

1167 Continuing our comparison with Hofmann et al. [2010a], our result is stronger in several respects.
1168 Hofmann et al. require the partial order $(\mathcal{P}, \leq, \perp)$ to form a lattice, whereas we do not. They simplify
1169 the solver’s API quite radically, so that *get* is called exactly once, with a set of variables of interest.
1170 This implies that there is no need for memoization and no possibility of concurrent or reentrant
1171 invocations of *get*. Finally, although they prove that the solver computes a fixed point, they do not
1172 prove that it is the least fixed point, whereas we do. In fact, by exploiting the novel notion of an
1173 optimal least fixed point, a mild generalization of Charguéraud’s optimal fixed point [2010a], we
1174 avoid imposing an ad hoc sufficient condition for a least fixed point to exist everywhere, and we
1175 are able to deal with situations where the least fixed point is well-defined only on a subset of \mathcal{V} .

1176

1177 There have been several successful efforts to implement and verify abstract interpreters in
1178 Coq [Besson et al. 2009; Cachera and Pichardie 2010; Jourdan 2016; Pichardie 2008]. At their
1179 heart, these abstract interpreters contain a least fixed point computation algorithm. They are
1180 implemented in a purely functional style, which precludes the use of mutable state for the purposes
1181 of memoization or spying. As a result, they are not local (instead, they compute the least fixed point
1182 everywhere) and do not perform dynamic dependency discovery (instead, their iteration strategy
1183 is directed by the syntax of the program under analysis).

1184

1185 11 CONCLUSION AND FUTURE WORK

1186 We have verified the partial correctness of a “local generic solver”, that is, an on-demand, incremental,
1187 memoizing least fixed point computation algorithm. Although the solver relies on mutable internal
1188 state, it is apparently pure: its specification merely states that it computes the optimal least fixed
1189 point of a system of monotone equations. We believe that this is a compelling illustration of how
1190 a modern program logic, such as Iris, allows relating a particularly subtle imperative algorithm
1191 with a particularly elegant mathematical specification. Furthermore, this proof provides several
1192 illustrations of the use of prophecy variables, and suggests that prophecy variables can be required
1193 in the verification of a sequential and deterministic piece of code, such as our solver.

1194 There are many directions for future work. It would be desirable to replace our current naïve
1195 representation of the dependency graph with an efficient representation, such as Pottier’s [2009].
1196 It would be worthwhile to verify a “top-down” variant of the solver (§9), both as an instructive
1197 exercise and because such a variant can be more efficient. It would be nice if one could use fewer
1198 locks, by removing or combining some of them, and if one could statically rule out all deadlocks. In
1199 a variant of Iris that allows establishing total correctness, one might wish to prove that the solver
1200 always terminates. More ambitiously, in an extension of Iris with time credits [Mével et al. 2019],
1201 one might wish to bound its asymptotic complexity. Finally, it would be desirable to verify realistic
1202 applications of the solver: program analyses and parser generators come to mind. With an eye
1203 towards applications in program analysis, it would be interesting to specify and verify the more
1204 powerful solvers proposed by Apinis *et al.* [2016] and by Seidl and Vogler [2018].

1205

1206 REFERENCES

- 1207 Martin Abadi and Leslie Lamport. 1988. [The Existence of Refinement Mappings](#). In *Logic in Computer Science (LICS)*.
1208 165–175.
- 1209 Anonymous. 2018. [Prophecy variables](#). Gitlab merge request #173 – link is not anonymized!
- 1210 Anonymous. 2019. [Prophecy variables with lists](#). Gitlab merge request #225 – link is not anonymized!
- 1211 Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2016. [Enhancing Top-Down Solving with Widening and Narrowing](#). In
1212 *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.
- 1213 Andrej Bauer, Martin Hofmann, and Aleksandr Karbyshev. 2013. [On Monadic Parametricity of Second-Order Functionals](#).
1214 In *Foundations of Software Science and Computation Structures (FOSSACS) (Lecture Notes in Computer Science)*, Vol. 7794.
1215 Springer, 225–240.
- 1216 Frédéric Besson, David Cachera, Thomas P. Jensen, and David Pichardie. 2009. [Certified Static Analysis by Abstract](#)
1217 [Interpretation](#). In *Foundations of Security Analysis and Design (Lecture Notes in Computer Science)*, Vol. 5705. Springer,
1218 223–257.
- 1219 David Cachera and David Pichardie. 2010. [A Certified Denotational Abstract Interpreter](#). In *Interactive Theorem Proving*
1219 *(ITP) (Lecture Notes in Computer Science)*, Vol. 6172. Springer, 9–24.
- 1220 Arthur Charguéraud. 2010a. [Characteristic Formulae for Mechanized Program Verification](#). Ph.D. Dissertation. Université
1221 Paris 7.
- 1222 Arthur Charguéraud. 2010b. [The Optimal Fixed Point Combinator](#). In *Interactive Theorem Proving (ITP) (Lecture Notes in*
Computer Science), Vol. 6172. Springer, 195–210.
- 1223 Patrick Cousot and Radhia Cousot. 1977. [Abstract interpretation: a unified lattice model for static analysis of programs by](#)
1224 [construction or approximation of fixpoints](#). In *Principles of Programming Languages (POPL)*. 238–252.

1225

- 1226 Christian Fecht and Helmut Seidl. 1999. [A Faster Solver for General Systems of Equations](#). *Science of Computer Programming*
1227 35, 2–3 (1999), 137–162.
- 1228 R. W. Floyd. 1967. [Assigning meanings to programs](#). In *Mathematical Aspects of Computer Science (Proceedings of Symposia*
1229 *in Applied Mathematics)*, Vol. 19. American Mathematical Society, 19–32.
- 1230 Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzkly, and Mooly Sagiv. 2007. [Local Reasoning for Storable Locks](#)
1231 [and Threads](#). In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science)*,
1232 Vol. 4807. Springer, 19–37.
- 1233 Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. [Oracle Semantics for Concurrent Separation Logic](#).
1234 In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 4960. Springer, 353–367.
- 1235 Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. 2010a. [Verifying a Local Generic Solver in Coq](#). In *Static Analysis*
1236 *Symposium (SAS) (Lecture Notes in Computer Science)*, Vol. 6337. Springer, 340–355.
- 1237 Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. 2010b. [What Is a Pure Functional?](#). In *International Colloquium*
1238 *on Automata, Languages and Programming (Lecture Notes in Computer Science)*, Vol. 6199. Springer, 199–210.
- 1239 Jacques-Henri Jourdan. 2016. [Verasco: a Formally Verified C Static Analyzer](#). Ph.D. Dissertation. Université Paris Diderot.
- 1240 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. [Iris from the ground](#)
1241 [up: A modular foundation for higher-order concurrent separation logic](#). *Journal of Functional Programming* 28 (2018),
1242 e20.
- 1243 John B. Kam and Jeffrey D. Ullman. 1976. [Global Data Flow Analysis and Iterative Algorithms](#). *Journal of the ACM* 23, 1
1244 (1976), 158–171.
- 1245 Aleksandr Karbyshev. 2013. [Monadic Parametricity of Second-Order Functionals](#). Ph.D. Dissertation. Technische Universität
1246 München.
- 1247 Gary A. Kildall. 1973. [A unified approach to global program optimization](#). In *Principles of Programming Languages (POPL)*,
1248 194–206.
- 1249 Baudouin Le Charlier and Pascal Van Hentenryck. 1992. [A Universal Top-Down Fixpoint Algorithm](#). Technical Report
1250 CS-92-25. Brown University.
- 1251 K. Rustan M. Leino and Peter Müller. 2009. [A Basis for Verifying Multi-threaded Programs](#). In *European Symposium on*
1252 *Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 5502. Springer, 378–393.
- 1253 John Longley. 1999. [When is a Functional Program Not a Functional Program?](#). In *International Conference on Functional*
1254 *Programming (ICFP)*, 1–7.
- 1255 Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. [From Datalog to FLIX: a declarative language for fixed points on](#)
1256 [lattices](#). In *Programming Language Design and Implementation (PLDI)*, 194–208.
- 1257 K. Muthukumar and M. V. Hermenegildo. 1990. [Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpre-](#)
1258 [tation of Logic Programs](#). Technical Report ACT-DC-153-90. Microelectronics and Computer Technology Corporation.
- 1259 Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. [Time credits and time receipts in Iris](#). In *European Symposium*
1260 *on Programming (ESOP) (Lecture Notes in Computer Science)*, Luis Caires (Ed.), Vol. 11423. Springer, 1–27.
- 1261 Peter W. O’Hearn. 2007. [Resources, Concurrency and Local Reasoning](#). *Theoretical Computer Science* 375, 1–3 (2007),
1262 271–307.
- 1263 David Pichardie. 2008. [Building Certified Static Analysers by Modular Construction of Well-founded Lattices](#). *Electronic*
1264 *Notes in Theoretical Computer Science* 212 (2008), 225–239.
- 1265 François Pottier. 2009. [Lazy Least Fixed Points in ML](#). (2009). Unpublished.
- 1266 François Pottier. 2019. [Fix](#). <https://gitlab.inria.fr/fpottier/fix>.
- 1267 Helmut Seidl and Ralf Vogler. 2018. [Three Improvements to the Top-Down Solver](#). In *Principles and Practice of Declarative*
1268 *Programming (PPDP)*, 21:1–21:14.
- 1269 Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. 2012. [Compiler Design: Analysis and Transformation](#). Springer.
1270 <https://doi.org/10.1007/978-3-642-17548-0>
- 1271 Bart Vergauwen, J. Wauman, and Johan Lewi. 1994. [Efficient fixpoint computation](#). In *Static Analysis Symposium (SAS)*
1272 *(Lecture Notes in Computer Science)*, Vol. 864. Springer, 314–328.
- 1273
- 1274