Karp: A Language for NP Reductions

Chenhao Zhang
Northwestern University
USA

Jason D. Hartline
Northwestern University
USA

Christos Dimoulas
Northwestern University
USA

Abstract

In CS theory courses, NP reductions are a notorious source of pain for students and instructors alike. Invariably, students use pen and paper to write down reductions that "work" in many but not all cases. When instructors observe that a student's reduction deviates from the expected one, they have to manually compute a counterexample that exposes the mistake. In other words, NP reductions are subtle yet, most of the time, unimplemented programs. And for a good reason: there exists no language tailored to NP reductions.

We introduce Karp, a language for programming and testing NP reductions. Karp combines an array of programming languages techniques: language-oriented programming and macros, solver-aided languages, property testing, higher-order contracts and gradual typing. To validate the correctness of Karp, we prove that its core is well-defined. To validate its pragmatics, we demonstrate that it is expressive and performant enough to handle a diverse set of reduction exercises from a popular algorithms textbook. Finally, we report the results from a preliminary user study with Karp.

CCS Concepts: **• Software and its engineering → Domain specific languages**; **• Social and professional topics → Computer science education**; **• Theory of computation →** *Problems, reductions and completeness.*

Keywords: domain-specific language, solver-aided programming, reduction, teaching and learning theoretical computer science

ACM Reference Format:
Chenhao Zhang, Jason D. Hartline, and Christos Dimoulas. 2022. Karp: A Language for NP Reductions. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523732

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523732

1 Introduction

A staple recipe for solving a computational problem is by *reduction*: (i) construct a translation from problem A to problem B and (ii) compose the translation with an existing algorithm for B. In other words, reducing from A to B establishes that if B is tractable so is A; modulo the translation, A is at most as hard as B.

Reductions are also useful in the other direction. Reducing from a hard problem A to a problem B proves that B is at least as hard as A. Such reductions from hard problems constitute the backbone of the study of complexity classes; the NP-complete class [6] was populated via a series of transitive reductions from Sat, the canonical NP-Complete problem.

Hence, it is no surprise that reductions feature prominently in theory courses (e.g. [2, 7, 20]). As an example, a prominent online textbook on data structures and algorithms contains a module about the so called Karp reduction from 3-Sat to Directed-Hamiltonian-Cycle. A Karp [19], or single-call, reduction is a restriction on general Turing reductions. In this case, the Karp reduction boils down to a correct translation from CNF formulas to graphs. For the translation to be correct, it must map an input satisfiable(unsatisfiable) 3-CNF formula to an output graph that has a(no) directed cycle that passes through every vertex exactly once. To ensure this property holds, in typical fashion, the translation builds its output out of repeating structural patterns, dubbed *gadgets*, that each encode some aspects of the input.

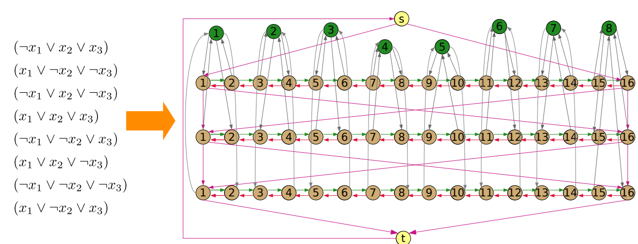

Figure 1. From a 3-CNF formula to a directed graph

Figure 1 shows how the reduction from the textbook turns a 3-CNF formula with 3 variables and 8 clauses (left) into an involved graph (right). The output graph consists of repetitions of triangles and rows of vertices. The careful crafting and stitching together of these gadgets are key for correctness: the direction of each triangle allows or blocks moving along or between the rows of vertices. Hence, arranging the gadgets appropriately makes it possible to construct a graph

with a Hamiltonian cycle if and only if the input formula is satisfiable. Unsurprisingly, even when students have the right intuition about how to arrange gadgets correctly, they often fail to get all the details right. And sometimes this is also true for instructors! In fact, the reduction from the textbook is incorrect: the 3-Sat instance on the left of Figure 1 is unsatisfiable while the graph on the right has a Hamiltonian cycle as shown in Figure 2. [1]

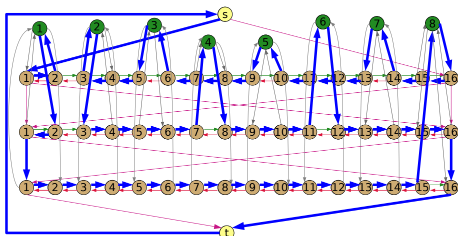

Figure 2. The Hamiltonian cycle in the constructed graph

Most such mistakes remain undetected until a reader notices a deviation from a correct reduction and painstakingly constructs by hand a counterexample; a situation that is a direct consequence of the fact that students and instructors alike treat reductions as mere pen-and-paper exercises. Put differently, they miss out on decades of experience on expressing algorithms that translate between complex data structures, which is exactly what Karp reductions are, as programs that can be run, tested and debugged.

To rectify this missed opportunity, we introduce Karp. Karp builds on Racket's language-oriented programming tradition [13] to deliver math-based vocabulary that helps students and instructors define problems and reductions between them as programs. In addition to notation tailored to reductions, Karp **automatically tests the correctness of reductions**. To that end, from each defined problem, Karp produces (i) contracts [14] that recognize problem instances and their solutions; (ii) a problem instance solver and; (iii) a problem instance generator. When programmers define a reduction from one problem to another, Karp puts together the corresponding derived contracts and solvers to construct a correctness contract for the reduction. Specifically, with the help of the solvers, the contract checks, at run time, that given (un)solvable instances of its from-problem, the reduction produces (un)solvable instances of its to-problem. As a final step, Karp combines the contract with the derived generator for the reduction's from-problem to property test [5] the correctness of the reduction.

As outlined above, the instance solvers for the from- and to- problem of a reduction are critical pieces of the correctness contract of the reduction, and hence, its property testing.

[1]Roughly, a correct reduction requires additional vertices in between the bases of triangles that restrict movement from one triangle to another along each row.

However, their manual construction is erroneous and time-consuming. Therefore, the derivation of instance solvers from problem definitions is key for Karp to be a practical language for programming and testing reductions. To address the problem of deriving instance solvers, Karp leverages the idea of solver-aided languages pioneered by Rosette [33]. In particular, Karp translates a problem definition to a Rosette function that, given a concrete problem instance, symbolically solves for a certificate. However, the construction of such instance solvers from Karp problem definitions requires care; certificates consist of data structures, such as sets and mappings, that, in general, Rosette cannot handle symbolically in a safe manner. For that, inspired by recent work on symbolic types [4], Karp problem definitions are written in a dedicated domain-specific typed language that translates to Rosette code that provably treats all its symbolic expressions in a safe manner. To enforce that code generated from problem definitions is used according to its types by the untyped code that implements and tests reductions, Karp wraps code generated from problem definitions with type-like contracts. Hence, a Karp program is in-effect a mix-typed program [23, 31, 32].

Overall, Karp strives to offer students and instructors a robust but familiar and flexible setting to experiment with reductions. Our experience with Karp so far confirms that it can express solutions to a wide variety of reduction exercises from Kleinberg and Tardos [20]'s standard algorithms textbook. Property testing these solutions has helped us discover mistakes, including the one in the reduction from 3-Sat to Directed-Hamiltonian-Cycle discussed above. Finally, the experimental use of Karp during a lab-style session of the Design & Analysis of Algorithms course at Northwestern University has offered similarly encouraging evidence.

The remainder of the paper is organized as follows. Section 2 gives an introduction to Karp reductions. Section 3 demonstrates the key design elements of Karp. Section 4 discusses the implementation of Karp and provides a formal account of Karp's problem definitions together with a proof of their safe translation to Rosette. Section 5 reports on our experience from using Karp. Section 6 places Karp in the context of prior work on software that facilitates the instruction of theoretical computer science. Section 7 closes the paper with a few concluding remarks and thoughts on future work.

2 A Crash Course on Karp Reductions

In classroom settings, Karp reduction is a common tool for proving that a decision problem is NP-Complete. As a running example for introducing key ideas and vocabulary about Karp reductions, this section examines the reduction from 3-Sat to Independent-Set, which proves that the latter is NP-Hard.

As a decision problem, 3-Sat asks a yes or no question: *Given a 3-CNF formula φ, is there a variable assignment that satisfies φ?*. This question can be cast into a generic form that is common for all NP problems: given an instance of the problem in hand, is there a certificate for the instance that is accepted by a polynomial time certificate verifier for the problem? For 3-Sat, (i) an instance a is a structure $\langle\varphi\rangle$ with one field, the 3-CNF formula φ; (ii) a certificate c^a is a mapping from the variables of φ to Booleans; and (iii) the certificate verifier $V^{\text{3-SAT}}(a, c^a)$ is a function that checks whether, for every clause $C \in \varphi$, there exists a literal $l \in Literals(C)$ that is true according to c^a. In other words, a description of instances and certificates, and the certificate verifier are sufficient to formally define 3-Sat, or any other NP problem.

For example, a formal definition of Independent-Set is: (i) an instance b has the shape $\langle G, k\rangle$ where G is an undirected graph and k is a natural number; (ii) a certificate c^b is a subset of the vertices of G; and (iii) the certificate verifier $V^{\text{I-SET}}(b, c^b)$ checks whether $|c^b| \geq k$ and for all pairs of vertices u, v that are neighbors in G, u and v are not both in c^b.

Given these definitions of 3-Sat and Independent-Set, a proof by reduction that Independent-Set is NP-Hard boils down to the construction of three algorithms:

1. A polynomial-time forward instance construction f that consumes a 3-Sat instance a and produces an Independent-Set instance b. The construction is correct if a is a yes-instance, i.e., there exists c^a such that $V^{\text{3-SAT}}(a, c^a) = true$, if and only if b is a yes-instance, i.e., there exists c^b such that $V^{\text{I-SET}}(b, c^b) = true$.
2. A forward certificate construction g^f that consumes an instance a and a certificate c^a and produces a certificate $c^b = g^f(a, c^a)$ of the instance $b = f(a)$.
3. A backward certificate construction h^f that consumes an instance a and a certificate c^b of the instance $b = f(a)$ and produces a certificate $c^a = h^f(a, c^b)$ of a.

The first algorithm, the forward instance construction, is the Karp reduction itself. The other two are the proof that the reduction is correct: it maps yes-instances to yes-instances (step 2) and no-instances to no-instances (contrapositive of step 3).

Figure 3 demonstrates with pictures how a standard forward instance construction f turns a 3-Sat instance $a = \langle\varphi\rangle$ into an Independent-Set instance $b = \langle G, k\rangle$ in three steps. The first step creates the set of vertices V of G. Specifically, $V = \{v_{l,i} \mid l \in Literals(C_i), C_i \in Cs\}$ where Cs is the set of clauses of φ and l ranges over the literals of a clause C_i. Hence, each vertex in V corresponds to a literal in φ. The second step adds an initial batch of edges E_1 to G. In particular, it adds one edge between two vertices in V that correspond to literals from φ with the same variable but opposite sign. That is, $E_1 = \{(v_{l_1,i}, v_{l_2,j}) \mid l_1 \in Literals(C_i), l_2 \in Literals(C_j), C_i \in Cs, C_j \in Cs \text{ if } l_1 \text{ is the negation of } l_2\}$. The third step creates the final set of edges E_2 that connect the vertices of G that correspond to the literals of a clause C_i in φ. Hence, $E_2 = \{(v_{l_1,i}, v_{l_2,i}) \mid l_1, l_2 \in Literals(C_i) \text{ if } l_1 \neq l_2\}$.

In essence, edges in E_1 prevent any pair of vertices that correspond to a literal and its negation from being members of an independent set for G. At the same time, edges in E_2 ensure that for every clause, there is at most one vertex in G's independent set. Consequently, the maximum independent set of G consists of as many vertices as the clauses C_s of φ. When φ has a satisfying assignment, the assignment renders true at least one literal in each clause of φ, which implies that G has an independent set of size $|C_s|$ which is equal to $|C_s|$, and vice versa. Putting all the pieces together, the resulting independent set instance is $b = \langle(V, E_1 \cup E_2), |C_s|\rangle$. Proving the correctness of this construction requires mapping certificates for 3-Sat instances to those of Independent-Set instances and vice versa.

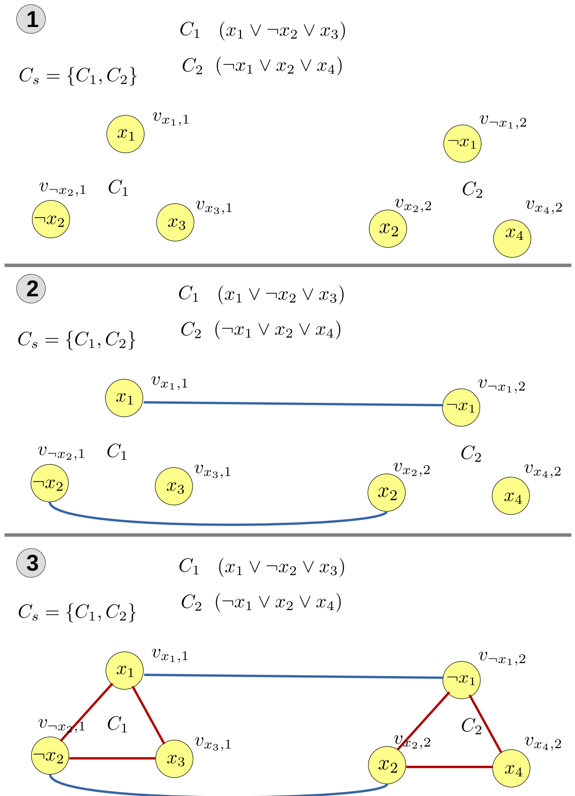

Figure 3. From a 3-Sat to an Independent-Set instance.

As foreshadowed above, a 3-Sat certificate c^a corresponds to the vertices of the Independent-Set instance $b = f(a)$ that form an independent set of size $|C_s|$. Since a satisfying assignment implies the existence of at least one true literal in each C_i and an assignment cannot render true a literal and its negation, the vertices of G that correspond to one of the true literals of each C_i form an independent set whose size is $|C_s|$.

Precisely, given a 3-Sat certificate c^a an Independent-Set certificate is $\{v_{l_{C_i},i} \mid C_i \in Cs\}$ where l_{C_i} is one of the literals of C_i such that either: (i) l_{C_i} is positive and c^a maps its underlying variable to true; or (ii) l_{C_i} is negative and c^a maps its underlying variable to false. Concretely, for the example from Figure 3, the forward certificate construction g^f can turn any certificate c^a that maps x_1 and x_4 to true to the set of vertices $v_{x_1,1}$ and $v_{x_4,2}$.

In the opposite direction, a backward certificate construction h^f constructs a satisfying assignment for a out of the literals that correspond to the vertices in c^b. If one of these literals is positive, then the assignment can map the underlying variable to true:

$$c^a(x) = \begin{cases} true & x \in T \\ false & x \in Vars(\varphi) \setminus T \end{cases}$$

where

$$T = \{x \mid x \in Vars(\varphi), v_{l_i,i} \in c^b, Positive(l_i), VarOf(l_i, x)\}.$$

As a final remark, a common mistake is the omission of all or some of the edges from step 3 of the forward instance construction from 3-Sat to Independent-Set. These edges form triangle gadgets that are important for establishing a proof of correctness of the above reduction. Figure 4 shows why. Due to the absence of the triangle gadgets, the reduction also translates the 3-Sat instance $\langle(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)\rangle$ to a graph that has an independent set of size 2. However, the proof of correctness of the reduction falls apart because of the backward certificate construction: it turns the certificate for the given Independent-Set instance into a non-certificate for the corresponding 3-Sat instance. Specifically, the resulting assignment fails to satisfy the clause C_2 of the 3-Sat instance as shown in Figure 4. In fact, this reduction is wrong and there is no backward certificate construction that translates all Independent-Set certificates back to 3-Sat certificates.

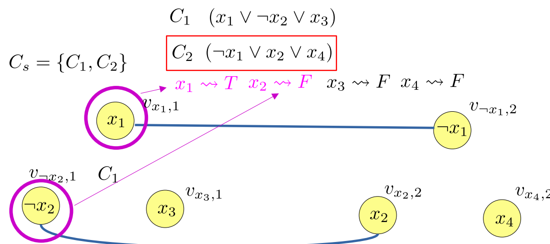

Figure 4. A wrong reduction from 3-Sat to Independent-Set.

3 Karp by Example

We demonstrate programming in Karp by revisiting the reduction from 3-Sat to Independent-Set from Section 2. Along the way, we discuss the rationale of Karp's design. Overall, the structure of the Karp version of the reduction mirrors the structure of the exposition in Section 2 to demonstrate how Karp's design and notation create a familiar linguistic setting for algorithms students and instructors.

3.1 Decision Problems in Karp

In Karp, programmers use karp/problem-definition, a small typed language, to define new problems. The type system helps Karp programmers avoid simple mistakes when defining new problems, but as mentioned in Section 1 and discussed in detail in Section 4, it also plays a significant role in the correctness of karp/problem-definition.

The central construct of karp/problem-definition is decision-problem that names a decision problem and outlines the shape of its instances and certificates. For example, in Figure 5, it (i) binds the defined 3-Sat problem to the identifier 3sat; (ii) specifies that its instance is a data structure whose single field φ is a 3-CNF formula and (iii) describes that its certificate is a finite map from the set of variables of φ to the set of booleans. When defining problems, Karp programmers have at their disposal a collection of libraries for data structures such as CNF formulas, finite maps and graphs that are staple ingredients of NP-Complete problems. In the specific case of 3-Sat, the cnf and mapping libraries contribute the relevant constructors and accessors for the corresponding data structures that appear in the body of decision-problem.

```
#lang karp/problem-definition

(require karp/lib/cnf
         karp/lib/mapping)

(decision-problem #:name 3sat
 #:instance ([φ is-a (cnf #:arity 3)])
 #:certificate (mapping
                #:from (variables-of φ)
                #:to (the-set-of boolean)))

; 3-SAT verifier definition
(define-3sat-verifier a c^a
 (∀ [c ∈ (clauses-of (φ a))]
  (∃ [l ∈ (literals-of c)]
   (or
    (and
     (positive-literal? l)
     (c^a (underlying-var l)))
    (and
     (negative-literal? l)
     (not (c^a (underlying-var l))))))))
```

Figure 5. 3sat.karp: 3-Sat problem definition

The use of decision-problem in Figure 5 produces a new specific form for defining the verifier for 3-Sat in Karp:

define-3sat-verifier. It consumes a 3sat instance a and a 3sat certificate c^a, and its body describes when c^a is indeed a certificate of a. The body of define-3sat-verifier matches exactly the description of the 3-Sat verifier from Section 2. Most importantly, the verifier description is declarative and uses math-inspired notation that mimics a pen-and-paper definition of a 3-Sat verifier. In general, verifiers in Karp are written in a subset of first-order predicate logic and, by construction, they are polynomial time with respect to the size of their arguments.

Analogous to Figure 5 and 3-Sat, Figure 6 depicts the problem definition for Independent-Set. It uses Karp's graph library and specifies an iset instance as a data structure with two fields: an undirected graph G and a natural number k. The certificate of an iset instance is a subset of the vertices of G. The verifier for iset checks two conditions: (i) the size of the certificate c^b is greater than or equal to the k field of the input instance and (ii) for all pairs u and v of neighbors in G, u and v are not both in c^b. Overall, same as for 3sat, the definition of iset matches closely the content and notation of the description of Independent-Set from Section 2.

```
#lang karp/problem-definition
 (require karp/lib/graph)

(decision-problem
 #:name iset
 #:instance ([G is-a (graph #:undirected)]
             [k is-a natural])
 #:certificate (subset-of (vertices-of G)))

; INDEPENDENT-SET verifier definition
(define-iset-verifier b c^b
 (define g (G b))
 (and
  (>= (set-size c^b) (k b))
  (∀ [u ∈ (vertices-of g)]
   (∀ [v ∈ (neighbors g u)]
     (not (and (set-∈ u c^b)
               (set-∈ v c^b)))))))
```

Figure 6. iset.karp: Independent-Set problem definition

In addition to define-X-verifier, decision-problem produces a collection of utilities that are necessary for writing and testing reductions in Karp about a defined problem X. The instance constructor create-X-instance expects arguments that comply with the shape of the pieces of an instance and produces instances that are recognized by the instance contract X-instance/c. The contract combinator X-certificate/c consumes an X instance and produces a contract that recognizes certificates of the appropriate shape. For instance, given a 3sat instance, 3sat-certificate/c returns a contract that recognizes mappings from the variables of the instance to the booleans. The instance generator generate-X-instance is derived from the shape of the definition of X and hand-rolled generators for the data structures from the libraries that come with Karp. Specifically, generate-iset-instance generates a graph using the generator from graph library and packages it together with a natural k in an iset instance using create-iset-instance.

The define-X-verifier construct contributes two more important utilities: X-verifier and X-solver. The first is the expected verifier for problem X. The second is an instance solver that given an instance x of X returns a certificate for x. To produce the second function, Karp compiles the body of define-X-verifier to a Rosette function that given an instance solves for a certificate. We return to the compilation from Karp to Rosette in Section 4; herein we focus on how Karp programs use these utilities.

As a final note about the utilities from decision-problem and define-X-verifier, all these functions are protected appropriately with the derived contracts from decision-problem, which reject ill-formed instances and certificates. Taking a leaf out of the book of gradual typing [23, 30, 31, 32], the contracts protect these functions that are produced by the typed definition language from their unsafe use in code written in the reduction language, which is untyped.

3.2 Reductions in Karp

The untyped reduction language, karp/reduction, comes with three main forms. Each of them corresponds to one of the three constructions of a reduction and its proof as described Section 2. First, the define-forward-instance-construction form consumes the names of from- and to-problem of the reduction, and its body defines a function that translates an instance of the from-problem to an instance of the to-problem.

The body of this form (and of the other two forms) is written in a simple terminating language that can only express polynomial time algorithms with respect to the size of the form's instance and certificate arguments. Specifically, the body consists of comprehensions over finite data structures or numbers (akin to for-loops), and local definitions that bind first-order data such as pieces of the given instance of the from-problem or the results of the comprehesions, and calls to a restricted set of primitive operations on numbers. Moreover, in order to emphasize the distinction between polynomial time and pseudo-polynomial time algorithms, karp/reduction treats differently cardinal numbers (for counting the size of sets) such as the k field of the problem definition for Independent-Set from Figure 6, and general numbers (whose size is equal to the length of their binary representation).[2] In detail, karp/reduction disallows

[2] By default, all numbers are cardinal except explicitly annotated numerical fields of numerical problems, e.g. Subset-Sum

comprehensions on general numbers, and as a result, pseudo-polynomial enumerations. The accompanying technical report [36] provides a formal model of karp/reduction together with a cost semantics and a proof of the asymptotic complexity bounds of the algorithms karp/reduction can express.

```
#lang karp/reduction
(require "3sat.karp"
         "iset.karp"
         karp/lib/cnf
         karp/lib/graph
         karp/lib/mapping-reduction)

(define-forward-instance-construction
 #:from 3sat #:to iset
 (3sat->iset a)

 (define Cs (clauses-of (φ a)))
 ; create the set of vertices (Fig. 1 Step 1)
 (define V (for/set {(el l i)
            for [l ∈ C]
             for [(C #:index i) ∈ Cs]}))
 ; create a 1st set of edges (Fig. 1 Step 2)
 (define E1
   (for/set
     {((el l1 i) . -e- . (el l2 j))
      for [l1 ∈ (literals-of C1)]
       for [l2 ∈ (literals-of C2)]
        for [(C1 #:index i) ∈ Cs]
         for [(C2 #:index j) ∈ Cs]
          if (literal-neg-of? l1 l2)}))
 ; create a 2nd set of edges (Fig. 1 Step 3)
 (define E2
   (for/set
     {((el (fst p) i)
       . -e- . (el (snd p) i))
      for [p ∈ (all-pairs-in
                (literals-of C))]
       for [(C #:index i) ∈ Cs]}))

 (create-iset-instance
   ([G (create-graph V (set-∪ E1 E2))]
    [k (set-size Cs)])))
```

Figure 7. 3-SAT to INDEPENDENT-SET forward instance construction in Karp

Figure 7 depicts the definition of the forward instance construction for the reduction from 3-SAT to INDEPENDENT-SET. In detail, the function 3sat->iset extracts the set of clauses Cs from the field φ of its argument a and uses them to construct the vertices and edges of an iset instance. The construction follows faithfully the corresponding three-step construction from Section 2. For the first step, comprehension for/set builds the set of vertices of a graph by creating abstract set elements (el l i) out of the literals l of the clauses C in Cs. The comprehension also enumerates the clauses C and embeds the index i of each clause in the vertices of its literals to distinguish appearances of the same literal in different clauses. For the second step, a for/set comprehension builds a first set of edges by creating vertices same as the previous step and packaging them with the edge constructor -e- from Karp's graph library if the underlying literals are a negation of each other. The third step makes the triangle gadgets by creating edges for all pairs of literals of each clause in Cs similar to the previous step. Finally, the create-3sat-instance constructor collects all these pieces in a new iset instance.

The define-forward-certificate-construction and define-backward-certificate-construction forms define the two certificate constructions from Section 2 that prove the correctness of a reduction. The forms consume the names of the from- and to- problem of the reduction, and their body defines a function of three arguments. For the forward certificate construction, the three arguments are a forward instance construction, an instance of the from-problem and a certificate for that instance. The forward certificate construction uses its arguments to compute a certificate for the to-problem instance obtained by applying the given forward instance construction on the given from-problem instance. Analogously, the three arguments of the backward certificate construction are a forward instance construction, an instance of the from-problem and a certificate for the to-problem instance translated from the from-problem instance via the given forward instance construction. The function uses the three arguments to compute a certificate for the given from-problem instance. In other words, the forward and backward certification constructions in Karp reflect their math counterparts from Section 2.

Figure 8 shows the definition of the forward certificate construction for the 3-SAT to INDEPENDENT-SET reduction in Karp. Following the corresponding construction from Section 2, the function 3sat->iset->>-cert constructs a set of vertices out of literals of the 3sat instance a that the certificate c^a maps to true. To achieve this, the function uses the find-one comprehension to obtain a single literal from each clause that satisfies the above property. Then it bundles the literal together with the index of the clause it appears in in an abstract element to form a vertex.

We omit the Karp code for the backward certificate construction; the interested reader can find the complete code for this reduction in the accompanying technical report.

3.3 Property Testing Reductions in Karp

Central to the design of Karp is that reductions are programs, and hence, should be tested. In general, the correctness of a reduction can be tested two ways. First, since a reduction is correct if and only if its forward instance construction maps

```
#lang karp/reduction
(require "3sat.karp"
         "iset.karp"
         karp/lib/cnf
         karp/lib/graph
         karp/lib/mapping-reduction)

(define-forward-certificate-construction
  #:from 3sat #:to iset
  (3sat->iset->>-cert f a c^a)

  (for/set
    {(el (find-one [l ∈ C] s.t.
           (or (and
                 (positive-literal? l)
                 (c^a (underlying-var l)))
               (and
                 (negative-literal? l)
                 (not
                  (c^a (underlying-var l))))))
         i)
     for [(C #:index i) in (φ a)]}))
```

Figure 8. 3-Sat to Independent-Set forward certificate construction in Karp

(no)yes-instances of the from-problem of the reduction to (no)yes-instances of the to-problem, we can test the instance construction. Second, since a proof of the correctness of a reduction is the existence of forward and backward certificate constructions that translate a certificate for an instance of the from-problem of the reduction to a certificate for the translated instance of the to-problem and vice versa, we can test the certificate constructions. The reduction language of Karp comes with an automated property testing procedure, `check-reduction`, that tests both the instance construction and the two certificate constructions. Put differently, Karp tests both the correctness of a reduction and its alleged proof of correctness.

In more detail, the `check-reduction` construct of the `karp/reduction` language consumes the names of the form- and to-problem of a reduction along with the forward instance construction, and the forward and backward certificate construction. For instance, the following snippet property tests the 3-Sat to Independent-Set reduction from this section:

```
(check-reduction #:from 3sat #:to iset
   3sat->iset
   3sat->iset->>-cert
   3sat->iset-<<-cert)
```

When run, this call to `check-reduction` initiates a number of rounds of testing[3] and each round of testing proceeds as follows:

1. The generator `generate-3sat-instance` produces a `3sat` instance `a`.
2. The instance solver `3sat-solver` returns a certificate `c^a` for `a` or declares it as a no-instance.
3. The forward instance construction `3sat->iset` translates `a` to an `iset` instance `b`.
4. The instance solver `iset-solver` attempts to find a certificate `c^b` for `b`. If it can not find one, it declares `b` a no-instance.
5. If `a` and `b` are not both yes- or no-instances the current round of testing has discovered an issue with the reduction because it detected a violation of the first correctness criterion. Hence, the testing procedure terminates and reports `a` as a counterexample.
6. If `a` and `b` are both no-instances, the current round of testing has succeeded and another one with a different `3sat` instance can begin.
7. If `a` and `b` are both yes-instances, the testing procedure moves to examining the second correctness criterion. For that, the forward certificate construction `3sat->iset->>-cert` translates `c^a` to a candidate `iset` certificate `c1` for `b`, and the verifier `iset-verifier` checks whether `c1` is indeed a certificate for `b`.
8. If `c1` is not a certificate for `b`, the current round of testing has discovered an issue with the reduction because it detected a violation of the second correctness criterion. Hence, the testing procedure terminates and reports `a` as a counterexample.
9. If `c1` is a certificate for `b`, the backward certificate construction `3sat->iset<<-cert` translates `c^b` to a candidate `3set` certificate `c2` for `a`, and the verifier `3set-verifier` checks whether `c2` is indeed a certificate for `a`.
10. If `c2` is not a certificate for `a`, the current round of testing has discovered an issue with the reduction because it detected a violation of the second correctness criterion. Hence, the testing procedure terminates and reports `a` as a counterexample.
11. If `c2` is a certificate for `a`,the current round of testing has succeeded and another one with a different `3sat` instance can begin.

The above procedure relies on the contracts that are derived from the from- and to-problem definitions. Verifiers and solvers are produced by `karp/problem-definition`, which is typed, but the testing process supplies to them the results of the untyped instance and certificate constructions. These results may be ill-formed; they may not conform to the expected shape of instances and certificates that the code

[3] The default is 10 but it can be configured with an optional argument of `check-reduction` that we leave out herein for simplicity.

of verifiers and solvers expects. Hence, the contracts that recognize instances and certificates enforce the expected type-level invariants at the boundary between verifiers and solvers, and the rest of the reduction code. When the contracts discover that an instance or certificate is ill-formed, the testing procedure terminates and reports which of the instance and certificate constructions is buggy along with the arguments to the construction that make the bug manifest.

Besides the instance and certificate contracts, the quality of the results of the testing procedure depends on the quality of the instance generators that are derived from the definition of the from-problem of the reduction. Specifically, it is necessary that the generators produce both yes- and no-instances for a problem in order for the testing procedure to exercise the first correctness criterion properly. However, due to the variety of decision problems, this is a challenging task. For example, graphs with way more edges than vertices are more likely to include a Hamiltonian cycle than an independent set. Put differently, the "semantics" rather than the structure of a decision problem determines the strategy for generating an effective set of instances, which makes difficult the derivation of effective generators from problem definitions.

To mitigate this issue, `Karp` takes two counter-measures. The first has been already discussed: `check-reduction` tests both the correctness of the reduction and its alleged proof even though the two ways of testing the reduction exercise the same notion of correctness. However, a testing procedure that checks for correctness both ways increases its chances of discovering an incorrect reduction. For example, `check-reduction` can discover the wrong reduction from Figure 4 either at step 5 or step 10 of a testing round.

As a second counter-measure, `Karp` comes with highly-fined tuned generators for built-in data structures such as CNF formulas and graphs. For example, the generator of the CNF library produces both satisfiable and unsatisfiable formulas. Similarly, the generator of the graph library produces a mix of graphs with cycles, cliques, isolated components etc. Moreover, to keep testing reductions between NP-Complete problems tractable, the built-in generators take advantage of known size thresholds for which random instances of a data structure are highly likely to either satisfy or not satisfy a given property. For example, random 3-CNF formulas with m clauses and n variables are highly likely to be either satisfiable or unsatisfiable when the ratio m/n is around 4.36 [8, 35]. Buidling on the fine-tuned built-in generators, `Karp` derives an instance generator for a problem based on the structure of the problem's definition. Such derived generators are effective for testing most reductions — Section 5 provides evidence of the pragmatic value of the derived generators and the out-of-the-box use of `check-reduction` for discovering incorrect reductions. For reductions where the out-of-the-box generators are not effective, `Karp` also offers a small language for the definition of custom generators.

4 Karp under the Hood

`Karp`'s problem definition and reduction languages are "little" languages built with Racket's language-oriented programming infrastructure [13]. This makes it possible for the two languages to inter-operate; a `Karp` program that defines two decision problems, spells out a reduction between them and tests the reduction is really a Racket program in disguise. In detail, the constructs and forms of the two languages are a mere domain-specific syntactic veneer on top of regular Racket data structures, functions and contracts implemented with Racket macros [15]. The design and implementation of the macros mostly follows well-known patterns and recipes. For instance, the implementation of the simple type system of `karp/problem-definition` is an adapation of the technique for building type systems with macros [11]. After the expansion [16] of the macros all that is left is plain Racket code.[4]

The exception to the above rule is the implementation of the macro `define-X-verifier`, which as Section 3 explains is produced itself by a use of the macro `decision-problem` that defines problem `X`. A use of `define-X-verifier`, such as the one in Figure 5, expands into two functions and binds them to the identifiers `X-verifier` and `X-solver` respectively. The first function is a Racket predicate that checks if its certificate argument is indeed a certificate for its problem instance argument. The second one is a Rosette function.

Rosette [33] consists of a language that extends a core of Racket with constraint solving utilities, and a symbolic virtual machine that evaluates programs (symbolically) to logical constraints. In more detail, a Rosette program can declare *symbolic* variables, state an assertion that involves some of these symbolic variables, and ask Rosette to solve, if possible, the assertion to obtain *concrete* values for its variables that make the assertion evaluate to true. Rosette's virtual machine attempts to translate the program to an appropriate formula and asks a constraint solver, such as Z3 [24], if there is an assignment of concrete variables that satisfies the formula, and hence, makes the program's assertion true. Assertions in Rosette programs can use regular Racket functions whose bodies, same as any other code in a Rosette program, Rosette's virtual machine attempts to interpret symbolically. At the same time, since Rosette is just another language in the Racket universe, Racket programs can call any Rosette function.

`Karp` takes advantage of exactly this interoperation between Rosette and Racket to obtain an instance solver for a problem `X`. A use of `define-X-verifier` expands to a Rosette function `X-solver` that has a concrete instance argument, declares symbolic variables for the certificate of the instance, asserts that the body of the verifier has to be true and calls Rosette's `solve` on the assertion. Rosette's virtual

[4]In fact, the built-in libraries of `Karp` are written directly in Racket.

machine interprets the body, along with any Racket functions it calls, to produce a query to the constraint solver for a value of the symbolic certificate that makes the assertion-body of `define-X-verifier` true. The resulting instance solver `X-solver` can be called by any Racket code, including the correctness contract of a reduction that involves problem `X`, and transitively Karp's testing procedure while it is property testing the correctness of a reduction from or to `X`.

However, there is a caveat. The types of Rosette's symbolic variables are limited to those that constraint solvers support while the types of symbolic certificates include the unsupported types of mappings, sets, and graphs. The implementation of Karp works around this issue by encoding its data structures as Racket's functional hash tables, dubbed hashes. For example, a Karp set is represented as a hash from the elements of its concrete domain to true or false; true if the element is in the set and false otherwise. Hence, Karp represents a symbolic certificate as a concrete hash whose range consists of symbolic boolean variables. In general terms, given that certificates of NP problems are always finite and constrained by a known concrete data structure, in order to solve a given problem instance, Karp encodes a symbolic certificate using the above technique and, via Rosette's `solve`, issues queries to a constraint solver that reference the symbolic boolean variables from the encoding of the symbolic certificate. In terms of a concrete example, the certificate of a Independent-Set is a subset of the set of vertices of a concrete Independent-Set instance. Hence, Karp encodes it as a hash from the vertices of the instance to symbolic boolean variables and calls Rosette's `solve` to determine which vertices need to be included in the certificate.

Alas, hashes are not part of safe Rosette, i.e., the core of Racket that Rosette's virtual machine can interpret symbolically. When the Rosette symbolic interpreter encounters a call to a hash lookup, such as `(hash-ref hash key)`, it delegates to the runtime of Racket whose behavior, is undefined when `key` is a symbolic value, i.e., an expression that contains symbolic variables and cannot be interpreted by Rosette's virtual machine further.

To turn the bodies of verifiers to Rosette code with well-defined behavior, the implementation of Karp generates uses of unsafe Rosette operations only when it can prove that they are safe. Otherwise, it elaborates them to code that is safe for Rosette but that, in contrast to the sub-linear asymptotic complexity of a hash lookup, has suboptimal linear asymptotic complexity with respect to the size of the hash. For instance, if the implementation of Karp cannot prove that `(set-∈ x S)` is safe, i.e., `x` is a concrete value, then it generates safe code that searches linearly through the hash representation of `S` instead of performing a hash lookup directly.

$\boxed{K}$

p	=	(**solve** pd (**instance** [x = $data$] ... [x = $data$]))
pd	=	(e **where** (**instance** [x : $ispec$] ... [x : $ispec$]) (**certificate** [x : $cspec$]))
$cspec$	=	Int | Bool | (**subset-of** x) | (**element-of** x) | (**mapping** x $cspec$)
$ispec$	=	$cspec$ | (**set** $ispec$) | Symbol | (**mapping** x $ispec$)
e	=	...
$data$	=	...
τ	=	Int | Bool | Symbol | (SetOf τ) | (Map τ τ)

Figure 9. Syntax of K

To validate this approach, we have developed K, a formal model of `karp/problem-definition`, together with a type-driven translation from K to R, a formal model of safe Rosette extended with hashes. To prove the correctness of K, we show that well-typed K programs translate to R programs that use unsafe operations such as hash lookups in a safe manner. In technical terms, inspired by Chang et al. [4], R has a *concreteness-aware* type system that distinguishes between expressions that evaluate to concrete and symbolic values and that guarantees that well-typed R programs avoid uses of symbolic values that endanger the safety of unsafe operations. Hence, the correctness of K follows from the fact that the translation maps well-typed K programs to well-typed R ones. The remainder of this section outlines the formal development; the complete definitions and proofs are in the accompanying technical report.

4.1 K, A Core Karp Problem Definition Language

Figure 9 shows the most interesting elements of the syntax of K. The top-level of a program in K represents a call to an instance solver derived from a problem definition and verifier. In detail, a program p consists of a problem definition pd and a problem instance (**instance** [x_1 = $data_1$] ... [x_n = $data_n$]). The problem definition pd has access to the fields $x_1, ..., x_n$ of the problem instance, and bundles up the body of a verifier e with the specification of the corresponding decision problem. The body of a verifier e, omitted from Figure 9, can use operations on the integers, boolean, symbols, sets and mappings that form the *data* of the given problem instance, also omitted from the figure.

Figure 10 demonstrates the expressiveness of K. It shows a program that solves an instance of Set-Cover, the problem that, given a set x_F of subsets of a set x_E, asks whether there are at most k elements of x_F such that every element x_e of x_E belongs to one of these k elements of X_F.

The same as `karp/problem-definition`, K comes with a standard simple type system. However, the type system also checks statically that an instance problem given to a verifier is well-typed, i.e., its type matches the specification of the

```
(solve
  ((and (<= (card x_c) x_K)
        (forall (x_e x_E) (exists (x_S x_c) (∈ x_e x_S))))
   where (instance [x_E : (set Symbol)]
                   [x_F : (set (subset-of x_E))]
                   [x_K : Int])
         (certificate [x_c : (subset-of x_F)]))
  (instance [x_E = ('x1 'x2 'x3 'x4 'x5)]
            [x_F = (set (set 'x1 'x2 'x3) (set 'x2 'x3)
                        (set 'x3 'x4) (set 'x4 'x5))]
            [x_K = 2]))
```

Figure 10. A K program solving a Set-Cover instance

fields of an instance of the corresponding problem. In the implementation of `karp/problem-definition`, contracts check that instances are well-typed at runtime as instances are provided by the untyped reduction language.

K does not come with a semantics. Instead, K programs translate to R, our model of core Rosette.

4.2 R, the Core Rosette Language

R has the necessary safe Rosette features in order to be an adequate target language for the translation of K programs. It comes with integers, booleans, symbols, their usual primitive operations and conditionals, symbolic variables of the above data types, and symbolic unions. Moreover it offers a collection of operations for creating and manipulating hashes that encode K's sets and mappings, and their operations. Due to lack of space, we omit a detailed discussion of the syntactic elements of R and their semantics and instead we focus on R hashes.

Hashes in R are of the form (**hash** *struct* $[e_k\ e_v]$...) where the tag *struct* is either Set or Map to allow R to distinguish whether a hash represents a K set or a mapping. The semantics of R uses the tag of a hash to determine how to simplify symbolic unions of hashes. The latter simplifications perform the so-called merging of symbolic values that is critical for avoiding path explosion during symbolic interpretation [27, 33], and its details depend on whether the hashes to be merged represent sets or mappings. Similarly, the tags determine how R compares hashes for equality as this operation is meaningful only between hashes that represent the same kind of K data structure.

R offers two lookup operations for hashes. The first one, (**hash-ref** h k), is akin to Racket's hash lookup and it's behavior is not defined when k is a symbolic value. Second, (**hash-branch** h k), aims to allow hash lookups for symbolic keys, and in essence evaluates to an expression that inspects all keys k' of h, compares them one by one with the symbolic k, and calls (**hash-ref** h k') for the first k' that is equal to k. In other words, when k is a symbolic value, (**hash-branch** h k) results in a symbolic union that search linearly through the keys of h.

The type system of R is concreteness-aware. A type of an R expression consists of two parts: a concreteness tag T and a base type τb. The tag can be either concrete • or symbolic ∘, and serves as a conservative overapproximation of whether an expression evaluates to a concrete or a symbolic value. The base type is standard simple type. The type system ensures that an expression with the tag • always evaluates to a concrete value of the appropriate simple type. In contrast, the result of an expression that typechecks to a type with the tag ∘ is not necessarily symbolic; it can be either a concrete or symbolic value of the appropriate type. Most other aspects of the type system are standard except that it rejects programs where expressions whose type has the tag ∘ appear at positions where only concrete values are expected. For instance, the type system rejects a program with an expression (**hash-ref** h k) where k has a type with the tag ∘.

The (reduction) semantics of R models the symbolic execution of programs in core Rosette. In particular, it attempts to evaluate a program to a query that can be issued to a constraint solver, but does not model the interaction with the solver. Formally the semantics induces an evaluation function that for any program gives one of three results: (i) the program evaluates to a concrete or symbolic value; or (ii) it evaluates to an error; or (iii) it gets stuck. Errors are the results of looking up nonexistent keys in hash tables. In contrast, programs get stuck when the semantics attempts to reduce an expression such as (**hash-ref** h k) where k is symbolic, i.e., stuck expressions correspond to undefined behavior.

The type system of R is sound with respect to its reduction semantics. That is, *a well-typed R program evaluates to value or an error.*

4.3 Translating K Programs to R

The translation from K to R is type-driven; when it translates a K expression, it produces not only its R image but also the type of the image. This enables the translation of a K expression, such as a mapping lookup, to inspect the concreteness tags of the types of the images of the arguments to the lookup, and based on the tags, to decide whether to produce a use of the unsafe operation **hash-ref** or to fall back to a use of the safe operation **hash-branch**. The final result of the translation is an open R expression, which encodes a query for solving an instance of a decision problem, and an environment from the expression's free variables to concrete and symbolic values, which are the encodings of the problem instance and the certificate respectively.

As an example, Figure 11 shows the result of translating the K program from Figure 10 to R. The interesting bit is that the translation replaces the (∈ x_S x_c) expression with (**hash-ref-def** x'_k x_k **false**). The **hash-ref-def** operation is a variant of **hash-ref** that also consumes a default value, here **false**, for when its x_k argument is not in x'_k. Hence, the

hash lookup is unsafe unless if x_k is concrete. However, the translation can prove that x_k is concrete since x_k ranges over the elements of x_E, the first field of the problem instance that the query attempts to solve. Hence, the hash lookup is indeed safe.

As a final note on Figure 11, it also demonstrates the encoding of sets with hashes in Karp. The two fields of the instance, x_E and x_F, turn into hashes that trivially map each element of the two sets to **true**. The certificate x_c, which as a set is a subset of x_E, turns into a symbolic hash that maps the keys of x_E to symbolic variables. The result of issuing the translated query to a solver is going to be concrete values for these symbolic variables and, hence, a selection of the elements of x_E that should appear in the certificate for the given instance.

```
(and (<= (h-summap (x_k x_v x_c) (if x_v 1 0)) x_K)
     (h-andmap (x_k x_v x_E)
       (if x_v (h-ormap (x'_k x'_v x_c)
                 (if x'_v (hash-ref-def x'_k x_k false) false))
               true)))

x_E ↦ (hash Set [`x1 true] [`x2 true] [`x3 true] [`x4 true] [`x5 true])
x_F ↦ (hash Set [(hash Set [`x1 true] [`x2 true] [`x3 true]) true]
                [(hash Set [`x2 true] [`x3 true]) true]
                [(hash Set [`x3 true] [`x4 true]) true]
                [(hash Set [`x4 true] [`x5 true]) true])
x_K ↦ 2
x_c ↦ (hash Set [(hash Set [`x1 true] [`x2 true] [`x3 true]) b°_1]
                [(hash Set [`x2 true] [`x3 true]) b°_2]
                [(hash Set [`x3 true] [`x4 true]) b°_3]
                [(hash Set [`x4 true] [`x5 true]) b°_4])
```

Figure 11. Solving an instance of Set-Cover translated to R

A key property of the translation is that it preserves typability. That is, *the translation maps a well-typed K program to a well-typed R program.* This property together with the soundness of the type system of K imply the central theorem about the correctness of Karp's problem definition language:

Theorem 1 (Uniform Evaluation for K)**.** *All well-typed K programs have well-defined behavior, that is they either evaluate to values or errors.*

5 Trying Karp in Practice

We put Karp to practice to mainly answer two questions:

- Q1: Is Karp sufficiently expressive to capture a wide range of reductions?
- Q2: Is Karp sufficiently performant to property test a wide range of reductions and find interesting bugs?

Secondarily, we also conducted a small formative user study to get a glimpse of what are the friction points for programming reductions in Karp.

5.1 On the Expressiveness of Karp

To evaluate the expressiveness of Karp, we attempted to implement a proportion of the NP-reductions from Kleinberg and Tardos [20]'s standard algorithm textbook. We first formulated in Karp five common reductions that appears as examples in the text. We then solved with Karp 20 of the 42 NP-reduction end-of-chapter exercises. From the 22 exercises that we did not solve, we conjecture that 15 are within reach. However, seven involve reductions that cannot be expressed in Karp as is.

Overall, we observe that the limiting expressiveness factor for Karp is the problem definition. Hence, the rest of this section analyzes how the design and features of `karp/problem-definition` measure up against the characteristics of the various NP-Hard problems from the book.

Set Problems. Abstractly, the certificate for a set problem is a subset of another set that satisfies some inclusion or exclusion constraints. Karp's built-in set operations are sufficient to define 10 problems in a similar manner to the definition of Set-Cover from Section 4.

Problems with Labeled Objects. Karp's finite mappings are essential to express five problems. We encode a set of labeled objects as a mapping from a base set to a set of labels. In some cases, labels are numerical values such as the start time of jobs in Interval-Scheduling. In other cases, they can also be abstract as the color of vertices in Graph-3-Coloring.

Basic Graph Problems. The six problems of this group resemble Independent-Set from Section 3. Besides operations on sets, they also depend on the basic operations from Karp's graph library.

Path and Connectivity Problems. These seven problems are similar to the Directed-Hamiltonian-Cycle in Section 1 and their certificates are paths, cycles or trees which are all represented in Karp as subgraphs. Furthermore, their verifiers check properties of the certificates related to connectivity or acyclicity. Hence, they lean on the corresponding predicates on graphs from Karp's graph library. The macro expansion of these predicates to Rosette is inspired by the way Gebser et al. [17] devise a solver for acyclicity constraints using off-the-shelf SAT solver.

Beyond Connectivity. Three problems are graph problems whose certificates have constraints that are not expressible with Karp's graph library. Two have certificates that are graph partitions of unspecified number of parts — Karp does support though partitions of constant number of parts. The certificate of the third requires that each vertex in the graph meets a constraint that involves properties of other vertices in the graph — such dependent properties are well outside the expressiveness of `karp/problem-definition`.

Problems with Unsupported Data Types. Four additional problems require data types such as strings, real numbers and functions on the real line that Karp does not support.

Table 1. The tables report the average time of 35 runs of the testing procedure, with the first 5 dropped. ± indicates the margin of error for the 95% confidence interval. C: classroom setting, E: evaluation setting

Reduction	Key Feauture(s)	Time (ms)
3Sat->3D-Matching	Set	1959±56
3Sat->Directed-Hamiltonian-Cycle (3SAT->D-HC)	Graph, Connectivity	8898±204
?->Directed-Edge-Disjoint-Paths (?->D-EDP)	Graph, Connectivity	35641±893
?->Fully-Compatible-Configuration	Graph, Mapping	8759±465
3Sat->Graph-Coloring	Graph, Mapping	6236±280
3Sat->Independent-Set (3SAT->ISET)	Graph	1231±19
?->Low-Diameter-Clustering	Mapping	1707±67
?->Plot-Fulfillment	Graph, Path	1722±165
?->Winner-Determination-for-Combinatorial-Auctions	Set, Mapping	916±35
?->Diverse-Subset	Set, Mapping	1109±41
?->Resource-Reservation	Set	901±26
?->Strongly-Independent-Set (?->SISET)	Graph	26401±889
Independent-Set->Vertex-Cover (ISET->VC)	Graph	855±49
?->Independent-Set	Graph, Mapping	1586±24
?->(a,b)-Skeleton	Set, Graph	3806±143
?->2-Partition (?->2-P)	Set	711±43
?->Galactic-Shortest-Path (?->GSP)	Mapping, Path	14381±3912
?->Dominating-Set (?->DS)	Graph	1470±36
?->Nearby-Electromagnetic-Observation	Set, Mapping	1329±25
?->Feedback-Vertex-Set (?->FVS)	Graph, Acyclicity	1660±30
?->Hitting-Set	Set	841±6
?->Monotone-Satisfiability	CNF	1728±14
Vertex-Cover->Set-Cover	Set	1049±24
?->Graphical-Steiner-Tree (?->GST)	Graph, Connectivity	56078±7305
?->Strategic-Advertising	Set	890±8

Reduction	Error	Source	Time (ms)
3SAT->D-HC	missing vertices that block the path from passing through all variables	E	615±85
?->D-EDP	not using a dedicated edge for each different part of the from-problem	E	3197±896
3SAT->ISET	missing triangle edges between vertices created from the literals of the same clause	C	109±13
?->SISET	missing the edges the different pieces of the from-problem	C	845±22
ISET->VC	incorrectly using the complement graph	E	314±93
?->2-P	not padding the values to ensure all numbers are nonnegative in all cases	E	692±28
?->GSP	missing extra edges with appropriate weight to balance out the weight accumulated	E	1022±445
?->DS	no consideration for isolated vertices	C	1619±72
?->FVS	no consideration for vertices that correspond to parts of the from-problem	C	270±62
?->GST	failure to maintain connectivity between all parts of the from-problem	C	1510±71

5.2 On the Performance of `Karp`

We conducted a performance evaluation with Karp that has two parts. Both parts involve using `check-reduction`, the testing procedure of Karp described in Section 3 to test a collection of reductions.

For the first part, we measured the time of running Karp's testing procedure for the 25 reductions[5] we solved from Kleinberg and Tardos [20]. Every run of the testing procedure tried 10 randomly generated instances of the from-problem of the reduction. Based on our observations, 10 generated instances discover most mistakes (see also the second part of the evaluation) and, hence, they are a sensible unit of property testing in Karp. The top table of Table 1 reports the results for this part of the evaluation; in the worst case it takes two minutes for the testing procedure to test a reduction.

For the second part of the evaluation, we implemented non-trivial incorrect solutions for 10 out of the 25 exercises and we measured the time it takes for Karp's testing procedure to detect the error. We sourced five of the incorrect

[5]The from-problems of some of these reductions are obfuscated for pedagogical reasons.

solutions (marked with C) from instructors of algorithms courses, who in turn reported mistakes made by their students. The remaining five incorrect solutions (marked with E) are mistakes that we encountered while trying to solve the corresponding reductions. Those include mistakes we did while working on the reductions together with incorrect solutions we discovered when comparing our solutions with those of others. One such mistake is described in Section 1. As the bottom table of Table 1 demonstrates, the testing procedure discovers all the mistakes in less than 1.5 seconds, i.e., in most cases within 10 testing rounds and in all cases within 20.

All measurements were performed on a typical set up for a CS undergrad student: a laptop with Intel Core i7-1165G7 2.80GHz × 4 and 16 GiB memory running 64-bit Arch Linux with kernel 5.16.12-arch1-1 and Racket 8.4 [cs]. Rosette was configured to use Z3 version 4.8.14. For most reductions, the testing procedure used the default instance generator for the corresponding from-problem. However, the default generators for Subset-Sum, Internal-Scheduling, Graph-Coloring and Hitting-Set do not produce a good mixture of yes- and no-instances. Hence, we replaced them with custom generators written in the small generator language that accompanies Karp.

5.3 On Programming in Karp

We organized a small formative user study with five CS Theory PhD students. We picked these students because of their expertise in theory and reductions. Hence, problems that they would run into during the study would be more likely due to Karp rather than proficiency about reductions. To help them get acquainted with Karp, we shared with the students a walk-through tutorial about the reduction from Vertex-Cover to Set-Cover in Karp. Then we asked them to define Graphical-Steiner-Tree in Karp and program the reduction to it.

Based on our observations, all students were able express problem definitions and verifiers in Karp with a few lines of code (in most cases less than 30). However, only four out of five the students successfully completed the exercise within four hours. The fifth student was successful in defining Graphical-Steiner-Tree in Karp but was not able to come up with a reduction to try to program it. The four successful students gave positive feedback about Karp citing the lack of detailed documentation as the biggest friction point. Moreover, three of the students discovered issues in the solution with Karp's testing procedure and the counterexamples it discovered helped the students get to the correct reduction. Despite the fact that Karp does not attempt to minimize counterexamples, the students were able to analyze them, and similar to debugging a program after a failed test, make progress towards a correct reduction.

6 Related Work

NP Reductions as Programs. The computational logic textbook of Gonczarowski and Nisan [18] comes with a Python library with programming exercises. One of these exercises asks students to complete a specific reduction. The Python library comes with a series of hard-coded tests to help students debug that reduction-as-programs. In a similar spirit, Enström and Kann [12] descibe a lab exercise where students have to implement a specific reduction given specific data definitions for the two problems involved. In addition, the exercise only concerns the forward instance construction part and not the two certificate constructions. Finally, Barak [2]'s textbook includes Python implementations of the proofs for a few reductions as a way to enhance the material of the book. Unlike Karp, these prior attempts to teach reductions via programming uses general purpose programming languages. As a result these attempts are limited to specific reductions for which the instructors have hand-rolled bootstrap and testing code.

Closer to our work, Creus et al. [10] propose REDNP, a domain specific language based on C for describing and testing NP reductions. In contrast to Karp, REDNP does not support the definition of new NP problems. Each new problem must be added to RASCO, the underlying runtime system, via a reduction to SAT (or to some other already supported problem). It is unclear whether one can use RASCO to test the correctness of such bootstrapping reductions, which are, in essence, RASCO's way of obtaining a solver for new problems by stitching together chains of reductions. Furthermore, the addition of a new problem to RASCO does not ask for defining a verifier for the problem. Pedagogically, this is important as the proof that a problem is NP is that it has a polynomial verifier. Nor does REDNP require certificate constructions alongside a forward instance construction. However, as described in Section 2 the certificate constructions are an important piece of a formal argument that a reduction is correct. As for testing the correctness of reductions, RASCO and REDNP do not offer automated property testing; the test cases (problem instances) need to be hand-crafted or generated outside the framework. Finally, REDNP is fundamentally imperative and does not provide complexity guarantees. Karp is declarative. And as discussed in Section 3, reductions in Karp have polynomial asymptotic complexity by construction.

Visualizing NP Reductions. Starting with the efforts of Pape and Schmitt [26] and Page [25], most software tools that aim to facilitate the instruction of NP reductions focus on visualization (e.g. [3], [9], [34]). The most recent examples include [22] from the OpenDSA project that designs online interactive visualizations of reductions. Unlike Karp, none of these tools help students formulate their own reductions and their proofs in an executable and testable manner.

Tools for Teaching CS Theory Besides Reductions. Despite the prevailing pen-and-paper tradition of theoretical

computer science, there are numerous efforts to build software tools that improve learning on topics besides reductions. Most of these tools target topics from theory of computation such as automata theory and formal languages, and offer automation for exercise generation, autograding and feedback generation (e.g. [28], [1], [29], [21]).

7 Conclusion

Karp is the product of an observation about teaching computer science and a conviction about programming languages. The observation is that the instruction of the theory of algorithms and programming go hand-in-hand. The conviction is that programming languages technology, especially of the domain-specific kind, is key to bring the two together without disrupting the habits of students and instructors. Specifically, the seed for Karp was planted when one of the authors saw an increase in student learning after replacing some pen-and-paper exercises on dynamic programming, the other notoriously difficult topic of Algorithms courses, with programming exercises in Python. The students had no problem switching their math scribblings for Python comprehensions and benefited greatly from being able to run, and most importantly, test their algorithms. Karp builds on this lesson and combines it with a range of exciting programming languages ideas. Preliminary evidence is encouraging that Karp can be the basis for an educational programming environment fit to NP reductions.

Plenty of work is necessary to get to a full-fledged production environment for teaching NP reductions. First, the effectiveness of Karp as a teaching aid would benefit significantly from visualization. A visual evaluation of a Karp reduction and, in particular, of its whole testing process will help students detect the root of issues in their reductions and how to fix them. In the direction of improving the effectiveness of testing, a suitable notion of testing coverage could guide the testing process and the generation of instances towards subtle corner cases. Integration with a theorem prover, which would take advantage of the restricted linguistic setting of Karp, will enable the formal verification of NP reductions closing the gap between what Karp offers and what students are asked to do in pen-and-paper reduction assignments and exams — in addition to the instance and certificate constructions, students also have to argue that the constructions are correct, which they cannot do currently in Karp. Moreover, our attempt to put Karp to use so far has revealed certain expressiveness deficiencies such as the lack of support for strings. We believe that further experience with Karp will bring up needs for additional libraries of data structures or, even, the need for relaxing the restricted design of the reduction language. In this latter case, an interesting question is how we can boost expressiveness without disrupting the complexity guarantees of Karp, and without having to resort to a complex type system that taxes students and instructors alike. But most importantly, it is critical to examine Karp systematically through the lenses of learning sciences and human-computer interaction to determine the friction points in achieving its goal of improving learning outcomes for algorithms students.

Acknowledgments

We thank the PLDI reviewers for their feedback. We also thank the theory group and the programming languages group at Northwestern for their help and support while we were working on this paper.

References

[1] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated Grading of DFA constructions. In *Proc. International Joint Conference on Artificial Intelligence*, pp. 1976–1982, 2013. https://dl.acm.org/doi/10.5555/2540128.2540412

[2] Boaz Barak. Introduction to Theoretical Computer Science. Online, 2019. https://introtcs.org

[3] Markus Andreas Brändle. GraphBench: Exploring the Limits of Complexity with Educational Software. PhD dissertation, ETH Zürich, 2006. https://doi.org/10.3929/ETHZ-A-005128663

[4] Stephen Chang, Alex Knauth, and Emina Torlak. Symbolic types for lenient symbolic execution. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 40:1–40:29, 2017. https://doi.org/10.1145/3158128

[5] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. ACM International Conference on Functional Programming*, pp. 268–279, 2000. https://doi.org/10.1145/351240.351266

[6] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proc. ACM Symposium on Theory of Computing*, pp. 151–158, 1971. https://doi.org/10.1145/800157.805047

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introductions to Algorithms. MIT Press, 1990.

[8] James M. Crawford and Larry D. Auton. Experimental Results on the Crossover Point in Random 3-SAT. *Artificial Intelligence* 81, pp. 31–57, 1996. https://doi.org/10.1016/0004-3702(95)00046-1

[9] Pilu Crescenzi. Using AVs to Explain NP-completeness. In *Proc. International Conference on Innovationand Technology in Computer Science Education*, pp. 93–97, 2010. https://doi.org/10.1145/1822090.1822175

[10] Carles Creus, Pau Fernández, and Guillem Godoy. Automatic Evaluation of Reductions between NP-Complete Problems. In *Proc. International Conference on Theory and Applications of Satisfiability Testing*, pp. 415–421, 2014. https://doi.org/10.1007/978-3-319-09284-3_30

[11] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proc. Workshop on Scheme and Functional Programming* volume 4, 2007.

[12] Emma Enström and Viggo Kann. Computer Lab Work on Theory. In *Proc. International Conference on Innovationand Technology in Computer Science Education*, pp. 93–97, 2010. https://doi.org/10.1145/1822090.1822118

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM* 61(3), pp. 62–71, 2018. https://doi.org/10.1145/3127323

[14] Robert B. Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM International Conference on Functional Programming*, pp. 48–59, 2002. https://doi.org/10.1145/581478.581484

[15] Matthew Flatt. Composable and Compilable Macros. In *Proc. ACM International Conference on Functional Programming*, pp. 72–83, 2002. https://doi.org/10.1145/581478.581484

[16] Matthew Flatt. Binding as Sets of Scopes. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 72–83, 2016. https://doi.org/10.1145/2837614.2837620

[17] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT Modulo Graphs: Acyclicity. In *Proc. European Workshop on Logics in Artificial Intelligence*, pp. 137–15, 2014. https://doi.org/10.1007/978-3-319-11558-0_10

[18] Yannai A. Gonczarowski and Noam Nisan. Mathematical Logic through Python. Cambridge University Press, 2021.

[19] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, pp. 85–103, 1972.

[20] Jon Kleinberg and Eva Tardos. Algorithm Design. Pearson Education, 2006.

[21] Loris D'Antoni, Martin Helfrich, Jan Kretinsky, Emanuel Ramneantu, and Maximilian Weininger. Automata Tutor v3. In *Proc. International Conference on Computer Aided Verification*, pp. 3–14, 2020. https://doi.org/10.1007/978-3-030-53291-8_1

[22] Nabanita Maji. An Interactive Tutorial for NP-Completeness. Master dissertation, Virginia Polytechnic Institute and State University, 2015.

[23] Jacob Matthews and Robert B. Findler. Operational Semantics for Multi-Language Programs. *ACM Transactions on Programming Languages and Systems* 31(3), pp. 12:1–12:44, 2009. https://doi.org/10.1145/1190216.1190220

[24] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008. https://doi.org/10.1007/978-3-540-78800-3_24

[25] Christian Page. Using interactive visualization for teaching the theory of NP-completeness. In *Proc. ED-MEDIA/ED-TELECOM*, pp. 1070–1075, 1998.

[26] Christian Pape and Peter H. Schmitt. Visualizations for proof presentation in theoretical computer science education. In *Proc. International Conference on Computers in Education*, pp. 229–236, 1997.

[27] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. A Formal Foundation for Symbolic Evaluation with Mergings. In *Proc. ACM Symposium on Principles of Programming Languages*, 2022. https://doi.org/10.1145/3498709

[28] Susan H. Rodger and Thomas Finley. JFLAP - An Interactive Formal Languages and Automata Package. Jones and Bartlett, 2006. https://www2.cs.duke.edu/csed/jflap/jflapbook/

[29] Varun Shenoy, Ullas Aparanji, K. Sripradha, and Viraj Kumar. Generating DFA Construction Problems Automatically. In *Proc. International Conference on Learning and Teaching in Computing and Engineering*, pp. 32–37, 2016. https://doi.org/10.1109/LaTiCE.2016.8

[30] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Workshop on Scheme and Functional Programming*, pp. 81–92, 2006.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 964–974, 2006. https://doi.org/10.1145/1176617.1176755

[32] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008. https://doi.org/10.1145/1328438.1328486

[33] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 135–152, 2014. https://doi.org/10.1145/2594291.2594340

[34] Steven Vegdahl. Visualizing NP-completeness through circuit-based widgets. *Journal of Computing Sciences in Colleges* 30(1), pp. 190–198, 2010.

[35] Ian P. Welsh and Toby Walsh. The SAT Phase Transition. In *Proc. European Conference on Artificial Intelligence* volume 94, pp. 105–109, 1994.

[36] Chenhao Zhang, Jason Hartline, and Christos Dimoulas. Karp: A Language for NP Reductions. Northwestern University, NU-CS-2022-03, 2022.