Programmers frequently maintain implicit data invariants, which are relations between different data structures in a program. Traditionally, such invariants are manually enforced and checked by programmers. This ad-hoc practice is difficult because the programmer must manually account for all the locations and configurations that break an invariant. Moreover, implicit invariants are brittle under code-evolution: when the invariants and data structures change, the programmer must repeat the process of manually repairing all of the code locations where invariants are violated.

A much better approach is to introduce data invariants as a language feature and rely on language support to maintain invariants. To handle this challenge, we introduce Targeted Synthesis, a technique for integrating data invariants with invariant-agnostic imperative code at compile-time. This technique is nontrivial due to the complex structure of both invariant specifications, as well as general imperative code.

The key insight is to take a language co-design approach involving both the language of data invariants, as well as the imperative language. We leverage this insight to produce two high-level results: first, we support a language with iterators without requiring general quantified reasoning, and second, we infer complicated invariant-preserving patches. We evaluate these claims through a language termed Spyder, a core calculus of data invariants over imperative iterator programs. We evaluate the expressiveness and performance of Spyder on a variety of programs inspired by web applications, and we find that Spyder efficiently compiles and maintains data invariants.

1 INTRODUCTION

Programmers routinely face the task of enforcing data invariants. Prominent examples of data invariants include well-formedness of data structures, model-view relations in interactive GUI applications, and consistency between application data and the database. Failure to properly enforce invariants is a common source of serious bugs and security vulnerabilities [5]. Traditionally, programmers do not state invariants explicitly. Instead, they tacitly maintain invariants by sprinkling invariant-restoring snippets across their code. This ad-hoc practice is error-prone because the programmer must maintain a mental model of what invariants are broken and how to restore them. In addition, these snippets are brittle under software evolution: when data structures and their invariants change, the programmer must go over the entire code base to modify, remove, or add invariant-restoring snippets.

An attractive alternative to this traditional model is to let programmers state the desired invariants explicitly, and have the programming language take responsibility for both checking the invariant satisfaction, as well as enforcing the invariants by updating the necessary data structures. This is the approach taken in declarative constraint programming [24, 38]. The downside of existing constraint languages, however, is that they solve constraints at run time, which is both unpredictable and inefficient. Wouldn’t it be great if instead we could compile declarative constraints into imperative code? Importantly, this would make the semantics of constraints more predictable, since any...
ambiguity would have to be resolved at compile time, when the language can ask the programmer for help. In this work, we propose using \textit{program synthesis} technology to compile declarative data invariants into imperative, invariant-enforcing patches.

Program synthesis is an active area of research \cite{14, 20, 37, 41, 45, 47} that tackles the problem of generating programs from declarative constraints. In particular, synthesis from logical specifications \cite{12, 21, 26, 35, 43} takes as input a logical predicate over a program’s inputs and outputs, and searches for a program that satisfies the predicate. We describe how program synthesis enables language support for data invariants through a motivating example.

1.1 Motivating Example: Budgeting

Consider a budget builder application for recurring expenses and incomes, shown in Fig. 1. Amounts are stored in two different formats, \textit{Weekly} and \textit{Daily}, so that the end-user can provide input in the most relevant period. For example, a budget for meals can be given in \textit{Daily} units, rent can be given in \textit{Weekly} units, etc. To give an overall sense of the budget, the running daily total is stored in \textit{Totals}; the final entry of \textit{Totals} contains the overall surplus or deficit. Revenues are distinguished from Expenses by rendering Revenues black and Expenses red. Finally, to visually distinguish alternate rows, the application renders them in different colors as specified by the user in two color pickers.

Each of these application properties is a data invariant that the programmer has to maintain: (1) \textit{Weekly} and \textit{Daily} are unit-conversions of each other, (2) \textit{Totals} is a running sum of the \textit{Daily} values, (3) if an entry is negative, its font color is red, and (4) each of the row’s background colors are determined by the color pickers.

Consider a function that adjusts the income in an existing budget for an increased cost of living. This function, shown in Fig. 1b, multiplies each positive daily income by the Cost-of-Living-Adjustment (COLA) constant. The loop in Fig. 1b immediately breaks invariants (1) and (2): the \textit{Weekly} and \textit{Total} values are stale. Our goal is to synthesize an \textit{invariant patch}, i.e. a code snippet that, when inserted into the function body, will provably restore the broken data invariants.

At a first glance, it seems natural to insert the patch at the end of the function, using the programmer-provided data invariants as the specification for synthesis. Unfortunately generating such a function-level patch is nontrivial even for this simple example. Since each row of the table is modified, the patch must involve a loop over the rows of the table. Synthesizing loops is challenging,
Targeted Synthesis for Programming with Data Invariants

because the synthesis algorithm must generate an inductive loop invariant. Note, that the original data invariant is not suitable because it does not hold on entry to the new loop – the programmer’s loop broke the data invariant in the first place. Moreover, even if the synthesis algorithm is clever enough to generate a loop, it must be careful to preserve the programmer’s original logic. The simplest solution is to update the Daily field of each row using the Weekly values. Such a patch would be disastrous – the data invariant is erroneously “maintained” by undoing the programmer’s changes!

More generally, this simple example highlights the two main research problems for synthesis-based language support of data invariants: (a) complex patches: even for simple data invariants, the synthesis algorithm must calculate both inductive invariants and complex control flow, and (b) the frame problem: without frame conditions, the synthesis algorithm can enforce the invariant by simply reverting the programmer’s changes.

1.2 Targeted Synthesis and the Spyder Language

The technical contribution of this paper is a solution to the above two research problems. Our solution consists of co-designing a programming language with a novel targeted synthesis algorithm, which generates patches locally – as close as possible to the invariant violation – as opposed to at the function boundaries.

Targeted synthesis addresses the problem of complex patches, because local patches are typically much smaller; moreover, pushing a patch inside a loop often results in preserving the original data invariant between loop iterations, turning it into an inductive loop invariant. For example, in Fig. 1b, a local patch updates r.week and r.total inside the loop. Not only is this patch a short, straight-line code snippet, but also it maintains the data invariant (1) as an inductive loop invariant. Targeted synthesis also addresses the frame problem: enforcing invariants at basic block boundaries enables a simple syntactic check that disallows patching variables modified by the programmer in that block and thereby ensures that all programmer’s changes are preserved.

This paper presents Spyder, a core language with iterators and data invariants, which is designed to be amenable to targeted synthesis. In particular, Spyder offers a restricted form of loops and quantified data invariants, which allows the synthesis algorithm to exploit their structural similarity and push synthesis specifications inside loops, in order to generate local patches.

The remainder of the paper is structured as follows. We use the domain of web GUI applications to give a high-level overview of Spyder in Sec. 2. Sec. 3 formalizes the semantics of the Spyder language, and Sec. 4 presents our targeted synthesis algorithm for extending Spyder programs with invariant-preserving patches. As part of our formalisms, we contribute a soundness guarantee that the targeted synthesis algorithm preserves the original invariants; this is summarized in Sec. 4. Sec. 5 evaluates our Spyder compiler on a series of benchmark and case studies. Finally, we conclude by reviewing related work in Sec. 6.

2 OVERVIEW

We begin with an overview of Targeted Synthesis by an illustrative example, in which the programmer uses data invariants to author an interactive GUI application. Consider a budgeting application, as shown in Fig. 1. The rendering and logic of application is relatively easy to express using traditional techniques, as we will discuss in Sec. 6, but traditional techniques don’t offer language support for statically enforcing the application’s data invariants. We will demonstrate how Spyder supports data invariants by iteratively building the interactive logic for this example.
2.1 Data Invariants

The programmer starts with the logic for the Weekly and Daily columns, shown in Fig. 2. To do this, the programmer declares a collection of `int`s termed `weeks`, shown on line 1 of Fig. 2a, as well as a collection of `int`s termed `days` (line 2). These two declarations introduce new mutable global variables `days` and `weeks`.

One invariant of the system is the unit-conversion invariant – each of the elements of `weeks` is 7 times greater than the corresponding element of `days`. This invariant should always hold and in particular, needs to be enforced whenever either `weeks` or `days` is mutated. To implement the unit-conversion invariant, the programmer uses a `foreach` construct on line 4, binding the elements of `weeks` to the local iterator `w` and the elements of `days` to `d`. Using these local bindings, they express the unit-conversion invariant using the formula on line 5:

\[ d \times 7.\text{val} = w.\text{val} \]

Because this unit-conversion invariant is defined over elements of collections, traditional techniques would model collections as arrays and require a quantified relation over the indices of the arrays. Such relations are notoriously tricky to build by hand (and indeed, to verify), but in Spyder, the programmer can use the `foreach` abstraction. This abstraction builds an element-wise product relation by introducing fresh iterator bindings over the abstracted collections.

2.2 Maintaining Data Invariants

Next, the programmer writes imperative code that Spyder patches to maintain the unit-conversion invariant. In the application, recall that the budget-builder needs to adjust all of the revenues (and only the revenues) in the budget by the Cost-of-Living-Adjustment (COLA). To implement this modification, the programmer writes a `procedure` called `adjustForCOLA` on line 7. This function iterates over the elements of `days` using the `for` loop on line 8, which binds each element of `days` to a local iterator variable `d`.

Since the COLA should only be applied to revenues, the programmer checks the value of the element `d` using a conditional on line 9, and then scales the daily revenue by an iterator update on line 10. The iterator semantics of Spyder are standard for object-oriented iterators; in particular, notice that the value of the iterator (e.g. `d.val`) is implicitly given by the iterator variable itself (e.g. `d` in the expression `d > 0`).

In this code snippet, the programmer has directly assigned an updated value to `d`, and by extension the values of `days`. On its own, this update breaks the unit-conversion invariant – in particular, the Weekly value of this row of the application depends on the concrete value of `d`. Using
Targeted Synthesis for Programming with Data Invariants

traditional techniques, the programmer would have to manually maintain the invariant by setting the corresponding value of \( \text{weeks} \), i.e. by adding an extra snippet for correctly updating \( \text{weeks} \).

Fortunately for the programmer, invariants are statically maintained in \textit{Spyder} and the compiler synthesizes and inserts a invariant-restoring snippet automatically, as shown in Fig. 2b. In this case, the compiler extends the original loop over \( \text{days} \) with an extra binding over \( \text{weeks} \); in \textit{Spyder}, this has the semantics of a simultaneous iteration (analogous to a functional zip) so that \( d \) and \( w \) refer to elements of \( \text{days} \) and \( \text{weeks} \) at the same index.

More generally, in contrast to traditional programming, \textit{Spyder} enables the programmer to write modifications that are \textit{agnostic} to the existing invariants. In this case, the programmer simply writes a direct update to the elements of \( \text{days} \) and \textit{Spyder} ensures that the overall system’s state is correct.

\subsection*{2.3 Program Composition Through Data Invariants}

Now that we have discussed \textit{Spyder}’s process for maintaining simple invariants, we show how \textit{Spyder}’s invariant-driven paradigm enables compositional programs. So far, the programmer has implemented some basic functionality for a budgeting application with \textit{Daily} and \textit{Weekly} values. In Fig. 3, they will now use invariants to extend the program with two cross-cutting concerns: (1) visual rendering logic for ensuring positive numbers are black and negative numbers are red, and (2) a new column of \textit{Total} values, which holds a running total of the \textit{Daily} column. For brevity, we omit the definition of \texttt{adjustForCOLA}, but rest assured that as the programmer adds invariants, \textit{Spyder} maintains the correctness of \texttt{adjustForCOLA} as well.

The programmer starts with a procedure modeling an update to a single cell of \( \text{days} \), termed \texttt{editDaily} on lines 7 through 10 of Fig. 3a. This procedure takes two parameters: \texttt{idx}, the modified location, and \texttt{val}, the new value. In a traditional programming language, the most natural way to implement this function would be a simple subscription-based update, \( \text{weeks}[\text{idx}] := \text{val} \). In \textit{Spyder}, however, the programmer must give up expressiveness for the ability to automatically maintain invariants. In this case, \textit{Spyder} does not allow collection subscriptions (e.g. \( \text{weeks}[\text{idx}] \)) to be l-values in assignments.

Instead, the programmer can use the \texttt{for} abstraction to iterate over the values of \( \text{weeks} \), and modify the correct value (as shown in lines 8 through 10). Notice that the programmer’s implementation of \texttt{editDaily} is \textit{agnostic} to the unit-conversion invariant on lines 4 and 5. More generally, \textit{Spyder} enables programmers to update data structures without having to propagate necessary changes to maintain the data invariants. Moreover, as we will shortly demonstrate, \textit{Spyder} enables the programmer’s invariants to modularly \textit{compose}.

The programmer next extends the application to implement the font-color invariant, that negative cells are rendered red while positive cells are rendered black. To do this, they add a font-color collection on line 12, \texttt{rowFontColors}, for the colors of \( \text{weeks} \) and \( \text{days} \). To encode the font-color invariant, they quantify over the elements of \( \text{days} \) and \texttt{rowFontColors} on line 13, and require that positive and negative elements are colored properly on lines 14 and 15.

In traditional programming languages, the implementation of \texttt{editDaily} would now be incorrect. In particular, the edit on line 10 might change the required font color for the modified \textit{Daily} and \textit{Weekly} cells. Traditionally, it would now be up to the programmer to recognize that \texttt{editDaily} is subject by this new invariant. Moreover, the programmer would also have to manually patch and extend \texttt{editDaily} to handle the font-color invariant.

In contrast, \textit{Spyder} patches the implementation of \texttt{editDaily} automatically; the programmer does not need to write any more code. As a result, the implementation of \texttt{editDaily} is agnostic to the font-color and unit-conversion invariants; the procedure and the invariants modularly \textit{compose}. 

, Vol. 1, No. 1, Article . Publication date: April 2018.
So far, the program’s invariants have been relatively straightforward. We will next show how to encode a complicated, cumulative invariant. Consider the running-sum invariant, in which each element of Totals is a running-sum of the corresponding elements in Daily. In each of the previous invariants, the relation for a collection did not depend on other elements of the collection. However, the most natural way to express a sum, through a loop or a reduction, relies on an accumulator to store the sum e.g. \( \text{accum} := \text{accum} + \text{next} \).

To implement the running-sum invariant for Totals, the programmer adds a new collection \( \text{totals} \) on line 17. Next, the programmer uses \texttt{foreach} and the \texttt{prev} iterator method to build a sum invariant over the elements of \( \text{totals} \) on lines 18 and 19. Of note here is the \texttt{prev} method; to enable relations with accumulators, \textsc{Spyder} provides \texttt{prev} which references the previous element of the iterator. The access of \texttt{prev} takes a single argument for the default value, at the beginning of iteration. In this example, the programmer uses \( \text{total}.\text{prev}(0) \) in a relation to access the previous value of \( \text{total} \) if it is defined and 0 otherwise.

Finally, the programmer finishes by adding a font-color invariant for the elements of \( \text{totals} \) on lines 21 through 24. At this point, the programmer is finished with \texttt{editDaily}. Using traditional techniques, the programmer would have to be mindful of all of the invariants of the system as they programmed \texttt{editDaily}. In particular, as they added each invariant to the system, they would have to refine and patch the implementation of \texttt{editDaily} to support each new invariant.
In contrast, Spyder enables support for invariants by patching editDaily to maintain the invariants; the code generated by Spyder is given in Fig. 3b. We’ll explain how Spyder calculates this implementation later in the paper. Not only does Spyder generate invariant-aware code, but it actually generates exactly the code that an expert programmer would write for this system. Moreover, the fixes introduced by Spyder are complex in several ways: (1) the fixes are extensive, accounting for the majority of the code in Fig. 3b (2) the fixes are nonlocal, meaning that each fix is spread out over (and interleaved with) the original code (3) the fixes have to add fresh new variables just to maintain the invariants (4) each invariant requires multiple fixes.

2.4 Generalization of Technique

From the programmer’s perspective, the process of invariant patching is invisible – Spyder accepts the original, invariant-oblivious code. More generally, the cognitive load of imperative programming with invariants in Spyder is significantly less than traditional techniques. Using Targeted Synthesis, when writing imperative code, the programmer needs to only reason about local code properties (e.g. the value of $d$) and does not need to reason about global code invariants (e.g. the relation between days and rowFontColors).

For the purpose of exposition, we have so far highlighted the novel properties of Targeted Synthesis without delving into the technical details. The remainder of this paper will focus on explaining how Spyder works. We first describe the Spyder source language in Sec. 3, and we show how to translate from Spyder terms to a well-studied imperative language. We also give a hoare-style axiomatic semantics to Spyder programs, and we provide a formal guarantee that Spyder verification triples are equivalent to standard triples in Sec. 3. Next, we present synthesis rules for patching and extending Spyder programs in Sec. 4, and we provide a formal guarantee that our synthesis rules are sound with respect to our Spyder verification triples. Finally, we present several case studies and a benchmarking evaluation in Sec. 5, and we conclude with a discussion of related work in Sec. 6.

3 THE SPYDER LANGUAGE

We present the syntax and semantics of the Spyder source language. In this section, we do not describe how to maintain data invariants in the setting of imperative programs with collections. Instead, we just provide the formal framework for both expressing collection-manipulation programs, as well as axiomatically verifying properties over programs. We will build on the results of this section in Sec. 4 to show how to maintain data invariants through Targeted Synthesis.

First, we introduce the core syntax of Spyder in Sec. 3.1. Then, we give a semantics to the syntax by translating Spyder terms to a well-studied standard imperative language in Sec. 3.2. Next, we demonstrate how to mechanically verify when invariants are maintained or violated by defining a Hoare-style [16, 22] axiomatic logic for Spyder in Sec. 3.4. Finally, we give a proof of soundness for this verification logic by reduction to the standard axiomatic semantics for imperative array programs (i.e. Hoare logic) in Theorem 3.1.

3.1 Surface Syntax for Spyder

At its core, Spyder is an imperative collection-manipulation language. The focus in Spyder is to support data invariants for mutable, finite collections. To this end, we formalize and define a core calculus for iterating over and mutating collections, which we present in Fig. 4a.

Values and Types. Spyder has three datatypes: integers, collections, and iterators. Integers are standard and we denote a variable declaration of type int as data $x: \text{int}$. 
Collections. Collections hold elements and are analogous to ordered containers, e.g. lists or arrays. For variable declarations, we denote a collection of \( T \) by \texttt{data col : T[]} . Collections are homogeneous and for clarity of presentation, our core syntax and formalisms assume that all collections are 1-dimensional (i.e. collections of integers). In our implementation, however, collections can nest arbitrarily (and extending the formalisms to arbitrary nesting is straightforward). For example, the list \([1,2,3] \) is a collection of integers and the list \([[1,2],[3,4]] \) is a collection of integer collections. In contrast, the list \([1,[2,3]] \) has mixed element types and is not valid. Collections expose a single method, size, which returns the number of elements in the collection. For simplicity, we assume all collections have a statically known size which does not vary at runtime. We also assume that collection sizes are homogeneous, for example, the list \([[1,2],[3,4,5]] \) would not be a valid SPYDER collection. A key difference between collections and arrays is that collections do not support subscription (i.e. \texttt{col[idx]} is not a valid SPYDER term). Instead, to access the elements of a collection, SPYDER exposes the \texttt{for} (\( x, y \)) statement, which iterates over the values of the collection \( y \). In addition to iteration, the \texttt{for} statement creates a new variable binding for an iterator variable.

Iterators. Iterators allow access to the underlying elements of a collection. Iterator variables are not explicitly declared using \texttt{data} but are instead created in for loops. SPYDER supports several standard iterator methods: \texttt{val}, which returns the value of the iterated collection; \texttt{idx}, which returns the current iteration index; \texttt{prev}, which returns the previous value; and the \texttt{x ← E} operator (termed “put”), which destructively updates the value of the iterator \( x \) with the expression \( E \). For example, after the evaluation of the term \texttt{for x in xs: x ← x + 1; each element of xs is incremented by exactly one.}

Statements. For control-flow, SPYDER has mostly standard imperative statements. A key exception is the \texttt{for} term, which as discussed above, iterates over a collection. In addition, the \texttt{for} loop iterates over multiple collections simultaneously, similar to a zip in function programming. For example the term \texttt{for x in xs, y in ys: x ← x.val + 1; y ← y.val + 1;} replaces each element in \( y \) with the corresponding element in \( x \). Furthermore, iteration is only well-defined when the iterated collections have the same size.

Specifications. To express specifications for SPYDER terms, SPYDER exposes a rich specification language, the \texttt{Spec} terms. To ease the burden of synthesis and verification, we syntactically phrase specifications in a conjunctive normal form. At the top level are conjunctions of specifications using the \( ∧ \) operator. Each conjunct can be either a bare expression, or a quantification term. For quantification, SPYDER supports two quantifiers: (1) An existential quantifier through the \texttt{exists} keyword. This quantifier is not present in the surface syntax of SPYDER and is only used in the axiomatic semantics, which we present in Sec. 3.4. (2) A universal quantifier through the \texttt{foreach} keyword, which quantifies over the elements of a collection. For example, the specification \texttt{foreach x in xs, y in ys: x.val > y.val} states that each element of \( xs \) is greater than the corresponding element of \( ys \). Similar to the \texttt{for} statement, the \texttt{foreach} term is only well-defined when the bound collections have the same size. We discuss the details of specifications more in Sec. 3.4.

3.2 Imperative Target Language
We formalize the semantics of SPYDER by translating to an idealized imperative verification language, which we call \texttt{Imp-Array}. The syntax of this verification target language is shown in Fig. 4b. This language is very similar to Boogie [27] and indeed, in our implementation, we compile and synthesize to Boogie.
Targeted Synthesis for Programming with Data Invariants

\[ v, u \in \text{Vars}, \quad i \in \mathbb{Z} \]

\[ \text{Spec} ::= \text{foreach } (v_i, u_i) \text{ Spec} \]
| \exists v . Spec
| | Spec \land Spec
| \text{Expr}\

\[ \text{Block} ::= \text{skip} \mid \text{Stmt} ; \text{Block} \]

\[ \text{Stmt} ::= v ::= \text{Expr} \]
| \text{if } \text{Expr} \text{ then } \text{Stmt} \text{ else } \text{Stmt} \]
| while \text{Expr} \text{Stmt}\

\[ \text{Expr} ::= v \mid i \mid \text{true} \mid \text{false} \]
| \text{Expr bop Expr} \]
| \text{if } \text{Expr} \text{ then } \text{Expr} \text{ else } \text{Expr} \]
| \text{uop Expr} \]
| Expr [ Expr ] \]
| size(v) \]
| \forall v . Expr \]
| \exists v . Expr\]

\[ \text{bop} ::= + \mid \times \mid \% \mid \Rightarrow \mid \leftarrow \mid \ldots \]

\[ \text{uop} ::= \neg \mid ! \]

(a) Syntax for the Spyder language.

(b) Syntax for the Imp-Array language.

Fig. 4. Syntax for Spyder and Imp-Array.

Although Spyder and Imp-Array have similar syntax, there are several major differences. Broadly speaking, Imp-Array does not have language support for either collections or iterators. Imp-Array instead offers mutable low-level arrays, which map (integer) indices to values. At the statement level Imp-Array supports mutable updates to both variables and arrays, as well as general while loops. For expressions, Imp-Array enables rich quantification through the \( \forall \) quantifier, but in contrast to Spyder, does not support iterator methods.

To support collections and iterators, the translation from Spyder to Imp-Array must implement collection and iterator logic in terms of arrays and indices. We show an example of this in Fig. 5, in which a Spyder program for calculating a product is translated into Imp-Array. In this case, the integer collections values and product in Spyder map 1-to-1 to arrays in Imp-Array, and the for loop in Spyder is desugared into a while loop with an explicit index in Imp-Array. At a high-level, collections in Spyder correspond 1-to-1 with arrays in Imp-Array, and iterator variables in Spyder correspond to indices in Imp-Array.

3.3 Overview of Translation Semantics

We formalize translation as a syntax-directed recursive function over Spyder terms given in Fig. 6. Since for loops bind iterator variables, the translation must be stateful. We choose to explicitly pass the state using finite mathematical maps, which we term translation contexts and we generally denote as \( \Gamma \). We denote the translation of a term \( t \) using the context \( \Gamma \) as the Imp-Array term \( \text{trans}(t, \Gamma) \); we refer to this as “the translation of \( t \) in the context of \( \Gamma \)”.

Well-formedness of Translation Contexts. In general, the translation process is only well-defined if the translation context \( \Gamma \) is well-formed. Intuitively, there must be no name-collisions; a collection must not be iterated over multiple times; an iterator variable must not be directly written to (i.e. using assignment := instead of the iterator ← operator); etc. We formalize these well-formedness
data values : [int];
data product : [int];

foreach v in values, fact in product:
  fact.val = fact.prev(1) * v.val

procedure multValues():
  for v in values, fact in product:
    v <- v.val * 1.05;
    if (index == 0) {
      product[fact] := val[v];
    } else {
      product[fact] := product[fact-1] * val[v];
    }
    fact := fact + 1; v := v + 1;
)

(a) Spyder source code for a product invariant.

(b) Translated Imp-Array code for a product invariant.

Fig. 5. Source code and translation for maintaining a product invariant. In contrast to the overview example, the source code already maintains the invariant and the translation step must soundly produce ImpArray code which also maintains the invariant. This example is similar to a subproblem of the Targeted Synthesis algorithm, which reasons about candidate programs such as multValues.

3.4 Verification in Spyder and ImpArray

We next define an axiomatic semantics for Spyder that Targeted Synthesis to mechanically verify when invariants are preserved or violated by a statement. To that end, we define and present an axiomatic semantics for verifying properties about Spyder statements. In addition, we prove that Spyder axiomatic semantics are sound with respect to the standard axiomatic semantics for Imp-Array (i.e. Hoare triples).

Hoare Triples for Imp-Array. We start by briefly reviewing axiomatic semantics in Imp-Array which are well-known [22]. The standard approach, called Hoare triples, are deduction rules for relating three terms: a precondition $P$, a statement $S$, and a postcondition $Q$, denoted by $\{P\} S \{Q\}$. Intuitively, the rules derive a triple if and only if given the precondition $P$, the postcondition $Q$ holds after executing the statement $S$. We replicate these rules in Fig. 13.

Notice that in standard axiomatic semantics, the loop rule requires an inductive invariant $I$ to be maintained on every iteration. Furthermore, the axiomatic rules do not contain a notion of
Appendix A.3. The key parts of the proof are the soundness of the Theorem 3.1 (Relative Soundness).

\[ \forall S, Q, \Gamma . \text{wf}(P \land Q, \Gamma) \land \text{wf}(S, \Gamma) \implies \langle P \rangle S \langle Q \rangle \implies \{\text{trans}(P, \Gamma)\} \\text{trans}(S, \Gamma) \{\text{trans}(Q, \Gamma)\} \]

We prove this property by induction over the derivation of the SPYDER Triple \( \langle P \rangle S \langle Q \rangle \), given in Appendix A.3. The key parts of the proof are the soundness of the Put and For rules which we discuss in detail below.
V ∈ globals
Var-Global V ∉ Γ
wf(v, Γ)

V ∉ Γ
Var-Bound
wf(v, Γ)

Prob-Int
wf(i, Γ)
– Prob-BT
wf(true, Γ)
– Prob-BF
wf(false, Γ)

Bop
wf(E₁, Γ)
wf(E₂, Γ)

Uop
wf(E, Γ)

Elem
v ∈ Γ
wf(v.val, Γ)

Prev
v ∈ Γ
wf(v.prev(E), Γ)

Idx
v ∈ Γ
wf(v.idx, Γ)

Size
v ∈ range(Γ)
wf(v.size, Γ)

(a) Well-formedness rules for Spyder Expressions.

u ∉ Γ u ∉ range(Γ) u ∉ globals
Foreach I @ v := u
wf( foreach (v, u), I, Γ )

Exists ( fresh x )\  \ \  wf(I, Γ)

Conjunct
wf(exists x . I, Γ)

wf(I₁ ∩ I₂, Γ)

(b) Well-formedness rules for Spyder Invariants.

Blk-Skip
wf(skip, Γ)

Blk-Seq
wf(S, Γ)

wf(B, Γ)

Stmt-Assign
v ∉ Γ
wf(E, Γ)

wf(v := E, Γ)

Stmt-Put
v ∈ Γ
wf(E, Γ)

wf(v := E, Γ)

Stmt-Cond
wf(E, Γ)

wf(I₁, Γ)

wf(B₁, Γ)

wf(B₂, Γ)

wf(if E then B₁ else B₂, Γ)

Stmt-For
y ∉ range(Γ)

x ∉ assign(B₁)
y ∈ globals

wf(B₁, Γ @ x := y)

wf(for (x, y) B₁, Γ)

(c) Well-formedness rules for Spyder Statements.

Fig. 7. Well-formedness rules for Spyder. For exposition, when rules bind a variable (e.g. for) we only formalize the well-formedness for a single binding. The extension to multiple bindings is straightforward.

P \implies P' Q' \implies Q
Consequence

(P') S (Q') \implies Q

(P) S (Q)

Skip

(P) skip (P)

Sequence

(P) S ; B (R)

(P ∧ E) B₁ (Q)

(P ∧ ¬ E) B₂ (Q)

Conditional

(P) if E then B₁ else B₂ (Q)

(P) := E (exists v'. P[ v' -> v' ] ∧ v = E[v -> v'])

Assign

(fresh v')

(P) v := E (exists v'. P[ val(v) -> v' ] ∧ val(v) = E[ val(v) -> v' ])\n
Put

mod(B₁, I) ∩ free(I) = 0

(weaken pred(I) ∧ 0 ≤ idx(x) < size(y)) B₁ (I)

For

(foreach (x, y) I) for (x, y) B₁ (foreach (x, y) I)

Fig. 8. Hoare-style verification logic for Spyder. For exposition, we only formalize the relation loops with a binding. Since loops are only well-defined when the iterated collections have the same statically known size, the extension to multiple bindings is straightforward.
**Strong Iterator Updates.** The Put case is interesting because under the hood, the update $x \leftarrow E$ translates to an array write (namely $\Gamma(x)[x] := E$). This is potentially problematic because the standard array semantics assumes indices can alias and so all information about the collection $\Gamma(x)$ is lost after the update. In Spyder however, there is no variable aliasing and moreover, the well-formedness rules ensure that the only way to reference the values of the collection $\Gamma(x)$ is through exactly one iterator $x$ and one expression $x.\text{val}$.

As a consequence, in the Put rule we reason about the value of $x.\text{val}$ while soundly retaining information about the collection $\Gamma(x)$.

**Quantifier introduction and maintenance.** A key requirement of the axiomatic semantics is to soundly reason about when loops maintain (or violate) universally quantified invariants (i.e. foreach terms). To that end, we provide a For rule, which is similar to a standard while rule in that the inductive invariant is on both sides of the statement. Unlike the Hoare while rule, however, the For rule for a loop $\text{for } x \text{ in } xs$ requires a top-level $\text{foreach } x \text{ in } xs$ as well.

In order to show that a foreach invariant is maintained by a for loop, it suffices to reason about each iteration of the loop in isolation. Due to the well-formedness constraints, the only way to modify the elements of a collection is through the $\leftarrow$ operator. As a consequence the execution of a loop iteration cannot invalidate the results of previous iterations. Since the loop is guaranteed to execute for each element of the collection, the rule introduces a foreach quantifier after the loop is complete.

Furthermore, it’s tempting to assume the specialized invariant as a precondition to verifying the loop body. If the invariant does not contain the prev method, this is completely valid. However, the prev method complicates matters because each iteration does not necessarily establish prev for the next iteration. To address this situation, we use the weaken_prev helper function to soundly weaken an expression with respect to prev. As a result, the For rule retains as much information as is soundly possible, and enables automated verification and synthesis by removing a layer of quantification.

## 4 Targeted Synthesis for Spyder

With an axiomatic semantics for Spyder programs at hand, we now consider the program of automatically enforcing data invariants. First, we motivate and formalize the problem in Sec. 4.1. Then, in Sec. 4.2 we present the Targeted Synthesis algorithm for solving this problem. We prove the algorithm sound in Sec. 4.3.

### 4.1 Automatic Enforcement of Data Invariants

Let $\Pi$ be a Spec term, and $S$ be a Spyder statement (i.e. a Stmt term). We say that $\Pi$ is a data invariant for $S$ if and only if $S$ maintains $\Pi$:  

$$\langle \Pi \rangle S \langle \Pi \rangle.$$ 

For example, the specification $\text{foreach } x \text{ in } xs: x.\text{val} > 0$ is a data invariant for a loop which increments each value of $xs$, $\text{for } x \text{ in } xs: x \leftarrow x.\text{val} + 1$, but it is not a data invariant for decrement loop $\text{for } x \text{ in } xs: x \leftarrow x.\text{val} - 1$. This definition extend straightforwardly to statement blocks $B$.

Let $B, B'$ be two Spyder blocks. We say that a block $B'$ is an extension of $B$ ($B \prec B'$) if $B$ and $B'$ have identical semantics on variables modified by $B$.

---

1. In particular the well-formedness relation prohibits a foreach quantifier over a collection $y$ from entering the body of a loop over $y$.
2. If a top-level term is not in this form but is equivalent under renaming and quantifier shuffling, the system can use the Consequence rule to rewrite the term as needed to make progress.
An invariant enforcement problem is a pair \( \langle B, \Pi \rangle \) of a block \( B \) and a specification \( \Pi \). A solution to the enforcement problem is a block \( B' \) such that \( B \prec B' \) and \( \langle \Pi \rangle B' \langle \Pi \rangle \). In other words, the goal is to find an extension of \( B \) that maintains \( \Pi \).

To find a solution, our algorithm analyses \( B \) and insert local patches whenever the invariant needs to be restored. Since there are many candidate patches to explore, the key challenge is to make the search efficient. To this end, our algorithm: (1) a-priori restricts the search to extensions of \( B \), by keeping track of the set of variables that a patch is allowed to modify; (2) targets the invariant \( \Pi \) to \( B \), producing a specification for each patch that is as local as possible.

### 4.2 Targeted Synthesis Algorithm

We formalize Targeted Synthesis as a completion judgment \( \mathit{md} \vdash \langle \Pi \rangle B \langle \Phi \rangle \rightarrow B' \). Intuitively, it means that given a pre- and post-condition \( \Pi \) and \( \Phi \), and the set of variables \( \mathit{md} \) modified so far, an input block \( B \) should be completed into \( B' \). In this case, we say that \( B' \) is a completion for \( B \), and the intension is that \( B' \) satisfies the specification \( (\langle \Pi \rangle B' \langle \Phi \rangle) \) and does not modify any variables in \( \mathit{md} \cap \mathit{md} = \emptyset \). We present the inference rules for this judgment in Fig. 9.

**Patch Generation.** The rule \text{Synth-Base} fires once we reach the end of the input block and performs the actual patch generation. It non-deterministically picks a patch that satisfies the specification, and can only update “stale” variables, which are not themselves modified but depend on modified variables via the specification \( \Pi \) (we formalize this notion of dependency in Fig. 10). Our implementation realizes the non-deterministic choice via constraint-based synthesis in the space of all blocks that only contain assignments and put-statements. \text{Synth-Loop} is similar to \text{Synth-Base} but allows generating looping patches when the postcondition contains quantification.

**Accumulating Modifications.** \text{Assign} and \text{Put} simply accumulate modifications made by the input block. In these rules, the variable modified by the current statement is added to \( \mathit{md} \), and the precondition of the subproblem is updated to reflect the result of the modification. Note that while the top-level completion problem is always symmetric (i.e. of the form \( \mathit{md} \vdash \langle \Pi \rangle B \langle \Pi \rangle \)), where \( \Pi \) is the data invariant we are trying to enforce), the pre- and the post-condition might become different as a result of applying \text{Assign} or \text{Put}. Sometimes these differences must be reconciled, because rules like \text{For-Specialize} only apply to symmetric goals. The rule \text{Inv} allow us to do just that: restore the invariant \( \Phi \) by inserting a patch in the middle of a block.

**Targeting.** The central rule of our system is \text{For-Specialize}. If a data invariant and a loop have the same syntactic structure (i.e. iterate over the same collections), this rule targets the data invariant to the loop body: i.e. strips both the loop and the quantification from the subgoal. One complication here is the role of prev terms. As discussed in Sec. 3, terms with prev cannot be used as an assumption for the body of a targeted loop. In this case, we first patch the current loop iteration into the term \( B_{pre} \), and then continue to the remainder of the loop body.

**Alignment.** Finally, a crucial necessity for the \text{For-Specialize} rule is that the data invariant and the loop are syntactically similar. To reach this state, the \text{Foreach-Extend} and \text{For-Extend} rules syntactically search for an alignment. Both of these rules are semantics-preserving and are performed so that the Targeting rule can be applied.

### 4.3 Soundness of Synthesis Rules

In all cases, if the \text{Spyder} extension rules produce a new program, the program must satisfy the input data invariants. We formalize the synthesis soundness using the axiomatic semantics of Sec. 3:
Theorem 4.1
Targeted Synthesis for Programming with Data Invariants

More detail is in the appendix (in Appendix A.5) and the proof is straightforward.

Fig. 9. Inference rules for SPYDER algorithm, with explicit blocks.

Fig. 10. Inference rules for variable data-dependency relation. We relate two variables \( x \) and \( y \) by \( \sim \) if a modification to \( x \) might affect \( y \).

\[ \forall \Pi, B, B'. \emptyset \vdash (\Pi) B \langle \Phi \rangle \iff B' \implies (\Pi) B' \langle \Pi \rangle \]

We prove this by generalizing to \( md \vdash (\Pi) B \langle \Phi \rangle \iff B' \) and then by induction on the derivation. More detail is in the appendix (in Appendix A.5) and the proof is straightforward.
5 EVALUATION

The research claims of this paper are threefold: (1) data invariants are broadly applicable and pervasive abstractions, (2) it’s easier to program using data invariants as a language feature, (3) Targeted Synthesis enables fast, scalable synthesis. We contribute a prototype compiler for Spyder which targets Boogie [28]. In this prototype, we implement the contents of Sec. 3 by compiling to Boogie, and we implement the contents of Sec. 4 by extending Spyder terms using our own synthesis and CEGIS algorithms.

We evaluate all of these research claims, 1-3, through benchmarking and case studies using our prototype. For claim 1), we assessed the expressiveness of Spyder qualitatively through several case studies. We evaluated claims 2) and 3) quantitatively: for claim 2), we measured the usability of programming with data invariants by measuring the extra code overhead of invariant maintenance, and for claim 3), we measured the performance of our compiler.

5.1 Case Studies

The invariant language of Spyder, targeted towards expressing relations over collections, is a perfect fit for many useful idioms in web programming. Using Spyder, we implemented three applications inspired by real-life web programs.

5.1.1 Game of Life. John Conway’s Game of Life [8] is a popular visualization of a cellular automaton with applications in Chemistry, Physics, Math, and Computer Science. In this game, a discrete world of cells obeys particular evolutionary behavior. At each time step of the application, the cells in the world change state according to the rules of the game. We looked at several interactive applications of the game of life online, such as [1]. In all of these applications, the programmer manually maintained an invariant between the visual cells of the board and the internal data structure for the cells. To implement this in Spyder, we encoded the internal state of the game and its visual state as two integer arrays. An element-wise invariant relates the internal state of the game to its visual state. We implemented procedures for (1) making transitions in the internal state according to the rules of the game, (2) interactive logic that allows the user to change the state of a cell by clicking on the board, and (3) a button for starting and stopping the game. Spyder was able to synthesize a patch that re-synchronizes the model and the view for each of these procedures.

5.1.2 Budgeting Application. Our second case study is a spreadsheet-style budgeting application, described in detail in Sec. 2. For this benchmark, the programmer builds a financial application which takes in periodic revenues and deficits. This application takes amounts in three periodic intervals—weekly, monthly, and yearly—and converts between the amounts. In this way, the end-user can input data in the most convenient format.

A difficult feature of this benchmark was summing up the rows of the budget and presenting a total value. In traditional programming, this would require a procedure and would not be easy to compose. In contrast, in Spyder, this invariant is easily expressible using the prev calculus and indeed composes very well with the other invariants of the system.

5.1.3 Shared Expenses Application. Our final case study is an extension of the Budgeting Application. Anecdotally, one of the co-authors actually uses this type of application in real-life. The idea here is that two people who live in the same household want to split shared expenses equally at the end of the month. In this application, each row has 4 entries: in the first two cells store the expenses paid by person A and person B, respectively; in the third cell, stores the average cost for the expense (i.e. the final cost for each person), and in the fourth cell, the amount person A owes to person B (i.e. how much person B over/underpaid on the particular expense). Similar to
the budgeting application, we can express each row of this application in Spyder and further, we can conditionally render the amount owed between the participants.

5.2 Empirical Evaluation

In addition to the qualitative expressiveness evaluation, we empirically evaluate the code savings that Spyder provides and the performance of our targeted synthesis algorithm, all relative to traditional techniques. Fig. 11 shows the result of our evaluation. In Fig. 11, we distinguish between the synthetics benchmarks and the larger case studies. The various columns of Fig. 11 are described in the caption. We discuss the results in more detail below, but in general, our experiments demonstrate that Spyder compiles a variety of programs with data invariants, significantly easing the programmer’s burden, and does so in a relatively efficient manner.

Code Savings. We measure the amount of code writing that Spyder saves through the use of invariants. In particular, Fig. 11 shows for each benchmark, the size of the source code excluding invariants (Source), the size of the invariant specification code (Inv), and size of the code that would have to be written by hand to enforce the invariants (ByHand). We measured ByHand by writing the enforcement code by hand for the largest benchmarks. The ByHand column should be compared with the Inv column: by writing the code in the Inv column and using Spyder to generate the enforcement code, the programmer does not have to write the enforcement code in the ByHand column. The results show that Spyder invariants provide a succinct way of specifying what would otherwise be a much larger piece of enforcement code.

Performance. We evaluated the scalability of the Spyder compiler (and by extension, the Targeted Synthesis algorithm) by compiling our benchmarks and comparing the performance against a standard synthesis technique, Sketch [42]. For each of our benchmarks, we reimplemented the benchmark in Sketch and compared the performance. In contrast to Targeted Synthesis, Sketch performs bounded enumeration for verification, and as a consequence, quantified invariants scale in proportion to the size of the verified array. To measure the scalability of bounded verification, for Sketch programs with arrays we varied the number of elements in the concrete Sketch arrays from 3 elements to 50 elements.

Overall, we find that Sketch outperforms Spyder on the synthetic benchmarks but does not complete within the time limit in all three of our case studies. For the case studies, Sketch could solve these problems if the programmer wrote a synthesis template tailored to the specific study. In contrast, Spyder programmers do not have to develop an application-specific sketch.

6 RELATED WORK

This paper builds upon two lines of prior work, which until now have developed independently: declarative constraint programming, where the goal is to enforce global constraints at run time, and program synthesis and repair, which enforces traditionally local, end-to-end functional specifications at compile time. We first discuss the trade-offs between static and dynamic constraint solving, and then we detail each of these areas.

6.1 Static and Dynamic Constraint Solving

Two of the longstanding research problems for constraint solving are performance [4, 13, 18], as well as debugging over- and under-constrained systems [13, 23, 30, 39]. In essence, the choice of static vs dynamic constraint solving boils down to a tradeoff between issues at compile time vs issues at run time.

For performance, solving constraint statically results in (notoriously) long synthesis and compilation times, but produces fast code. Conversely, dynamic constraint solving does not require an
<table>
<thead>
<tr>
<th>Group</th>
<th>Benchmark</th>
<th>Program Size (#AST nodes)</th>
<th>Synthesis Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Source Inv ByHand</td>
<td>Spyder SK-3 SK-10 SK-50</td>
</tr>
<tr>
<td>Numerical</td>
<td>midpoint</td>
<td>31 7 14</td>
<td>27.80 1.03 N/A N/A</td>
</tr>
<tr>
<td>Programs</td>
<td>distinct mid</td>
<td>34 10 24</td>
<td>129.07 0.90 N/A N/A</td>
</tr>
<tr>
<td></td>
<td>1D equal arrays</td>
<td>47 8 30</td>
<td>21.68 0.98 0.97 1.19</td>
</tr>
<tr>
<td></td>
<td>2D equal arrays</td>
<td>45 13 18</td>
<td>11.79 0.99 1.10 4.57</td>
</tr>
<tr>
<td>Web Applications</td>
<td>1D GoL overview</td>
<td>237 39 30</td>
<td>66.85 - - -</td>
</tr>
<tr>
<td></td>
<td>split costs</td>
<td>140 26 56</td>
<td>156.04 - - -</td>
</tr>
</tbody>
</table>

Fig. 11. Benchmarks and Spyder results. Source: the #AST nodes in the Spyder source code (excluding invariant specifications); Inv: the #AST nodes in Spyder invariant specification code; ByHand: the #AST nodes in the code that a programmer would have to write by hand to maintain invariants without using our synthesis algorithm; Spyder: the synthesis time of Spyder; and SK-3, SK-10 and SK-50: the synthesis times for SKetch, on collections of size 3, 10, and 50, respectively. We report a timeout (−) after ten minutes and we use N/A to denote a Sketch program that doesn’t use collections.

expensive compilation pass but results in large runtime overheads, as high as 10x-100x (as reported in [13]). Consequently, the choice of static vs. dynamic for performance is a tradeoff between compilation time and runtime performance.

Debugging constraint systems is a similar story in that static systems can report a compile-time error when the system is over- or under-constrained. Conversely, dynamic systems generally attempt to resolve ill-posed systems anyway, using techniques such as constraint hierarchies [6], which results in unintuitive solutions – unintuitive because the solution does not satisfy the constraints. In either case, the ill-posed system must be debugged. In the static case, it is strictly the programmer who debugs the system, while in the dynamic case, the end user might be exposed to the ill-posed system. Consequently, the choice of static vs. dynamic for debugging is a tradeoff between programmer time and user time.

### 6.2 Dynamic Invariant Enforcement

There are two closely related research arcs on dynamically enforcing invariants: the field of constraint imperative programming, and the work of functional reactive programming. Both of these areas provide mechanisms for dynamically solving invariants, and both are orthogonal to our efforts because we solve constraints statically through program synthesis.

#### 6.2.1 Constraint Imperative Programming

The field of constraint solving is rich and storied [3, 11], as constraint solvers excel at calculating global solutions. Despite their power, constraint solvers are traditionally relegated to libraries. The field of Constraint Imperative Programming aims to provide first-class language support for constraint solving [13, 19, 33], but again, fundamentally our work is orthogonal because we solve constraints statically.

#### 6.2.2 Functional Reactive Programming

The field of Functional Reactive Programming (FRP) provide a dataflow language for building graphical systems [46]. Although inspired by animations, FRP quickly became popular as a tool for taming web application logic [9, 31]. The most popular recent work in this field are Elm [10] and its imperative cousin React [44], which provide a language and runtime for building client-side web applications. Although popular and powerful, FRP is a general, dynamic technique for abstracting over dataflow – in contrast, our work focuses on the problem of first-class data invariants, and solves for invariant patches statically.
6.3 Program Synthesis and Repair

In recent years, program synthesis has emerged as a promising technique for automating tedious and error-prone aspects of programming [20, 41, 45]. The two main directions in this area are synthesis from informal descriptions (such as examples, natural language, or hints) [2, 7, 14, 15, 32, 34, 37, 40, 47] and synthesis from formal specifications, where the goal is to synthesize a program that is provably correct relative to the specification [12, 21, 26, 29, 35, 43]. Both directions have been mainly focusing on synthesizing standalone programs from complete end-to-end functional specifications of their inputs and outputs. Instead, the present work focuses on synthesizing code snippets from incomplete, global specifications (data invariants) and integrating them with hand-written code.

The only prior techniques we are aware of for generating snippets from declarative specifications and inserting them into hand-written code is in the context of information-flow security [17, 36]. Enforcement of data invariants brings a different set of challenges, since invariants are deep semantic properties.

Our work is related to sound program repair [25], where the problem is, given a formal specification and a program that violates it, modify the program so that it provably satisfies the specification. Program repair, however, is a very general problem, and hence lacks a-priori restrictions on what modifications the repair algorithm is allowed to make. As a result, if the given specification is incomplete, the problem is ill-defined. In this work we show that in the specific setting of enforcing data invariants, the space of possible modifications can be sufficiently restricted to make the repair both predictable and efficient. As far as efficiency is concerned, the deductive program repair technique of [25] does not scale well with the number of patches to be generated within the same function. Spyder, on the other hand, leverages the restricted nature of the problem to solve each synthesis task completely independently, hence avoiding the combinatorial explosion with the number of patches.

REFERENCES


A.1 ImpArray Semantics

\[
\begin{align*}
\text{[v]}_\sigma &= \sigma[v] \\
\text{[i]}_\sigma &= i \\
\text{[true]}_\sigma &= \top \\
\text{[false]}_\sigma &= \bot \\
\text{[E_1 \text{bop} E_2]}_\sigma &= \text{[E_1]}_\sigma \text{bop} \text{[E_2]}_\sigma \\
\text{[uop E]}_\sigma &= \text{uop} \text{[E]}_\sigma \\
\text{[v[E]}_\sigma &= \text{[v]}_\sigma \text{[E]}_\sigma \\
\text{[size(E)}_\sigma &= \|\text{[E]}_\sigma\| \\
\text{[\forall v . E]}_\sigma &= \forall x \in \sigma . \text{[E[v \mapsto x]}_\sigma = \top
\end{align*}
\]

(a) Denotational semantics for Imp-Array expressions

(b) Operational semantics for Imp-Array statements

Fig. 12. Semantics for Imp-Array terms.

A.2 Spyder Semantics

Let \(\sigma\) be a Imp-Array state, \(E\) a Spyder Expression, and \(\Gamma\) a well-formed translation context with respect to \(E\). We define the denotational semantics of \(E\) as the denotational semantics of the corresponding Imp-Array expression:

**Definition 1 (Spyder Expression Semantics).**

\[
\text{[E]}_\sigma \overset{\text{def}}{=} \text{[trans}(E, \Gamma)\text{]}_\sigma
\]

We similarly define the operational semantics of a Spyder statement \(S\) as the operational semantics of the corresponding Imp-Array statement \(\text{trans}(S, \Gamma)\):

**Definition 2 (Spyder Statement Semantics).**

\[
\text{trans}(S, \Gamma), \sigma \rightsquigarrow \sigma' \quad S, \sigma \rightsquigarrow \sigma'
\]

A.3 Soundness of Spyder Triples

**Lemma A.1 (Bindings).**

\[
\forall P, B, \Gamma . \text{wf}(\text{for}(x, y)B, \Gamma) \land \text{wf}(P, \Gamma) \implies x \notin \text{free}(P)
\]

**Proof.** Induction over the derivation of \(\text{wf}(P, \Gamma)\). \qed
\[
P \implies P' \quad Q' \implies Q
\]

**Consequence**

\[
\frac{P' \triangleright S \{Q'\}}{P \triangleright S \{Q\}}
\]

**Skip**

\[
\frac{\{P\} \triangleright \text{skip} \{P\}}{\begin{array}{l}
\{P\} \triangleright S_1 \{Q\} \\
\{Q\} \triangleright S_2 \{R\}
\end{array}}
\]

**Sequence**

\[
\frac{\{P\} \triangleright \{Q\}}{\{P\} \triangleright S_1 \{Q\} ; S_2 \{R\}}
\]

**Conditional**

\[
\frac{\{P \land e\} \triangleright S_t \{Q\} \quad \{P \land \neg e\} \triangleright S_f \{Q\}}{\{P\} \begin{cases} \text{if} e \text{ then } S_t \else } S_f \{Q\} \end{cases}}
\]

**Assign-Var**

\[
\frac{\text{(fresh } v') \quad \{P\} \triangleright v := E \{ \exists v'. P[v \mapsto v'] \land v = E[v \mapsto v'] \}}{\{P\} \triangleright \text{assign} \{E_i \} \{P\} \triangleright \text{assign} \{E_r \}}
\]

**Assign-Array**

\[
\frac{\text{(fresh } v') \quad \{P\} \triangleright v[ E_i \] := E_r \{ \exists v'. P[v \mapsto v'] \land v = v'[E_i \[v \mapsto v'] := E_r[v \mapsto v'] \}}{\{I \land E\} \triangleright S \{I\} \quad \{I\} \triangleright \text{while} \; E \; S \{I \land \neg E\}}
\]

Fig. 13. Standard axiomatic semantics (Hoare logic) for `IMP-ARRAY`.

**Lemma A.2** (Assignment).

\[
\forall B, \Gamma \cdot \text{wf(for(x, y))}_{B, \Gamma} \implies x \notin \text{assign}(B)
\]

**Proof.** Induction over the derivation of \(\text{wf(for(x, y))}_{B, \Gamma}\). \(\square\)

**Lemma A.3** (Array substitution).

\[
\forall P, x, y, \Gamma \cdot \text{wf(P, } \Gamma) \land \Gamma(x) = y \implies \forall \sigma \cdot \sigma(\text{trans}(P, \Gamma)[y \mapsto y']) \implies \sigma(\text{trans}(P, \Gamma)[y[x] \mapsto x'])
\]

**Proof.** Structural induction over \(P\). \(\square\)

**Theorem A.4** (Relative Soundness).

\[
\forall P, Q, S, \Gamma \cdot \text{wf}(P \land Q, \Gamma) \land \text{wf}(S, \Gamma) \implies \langle P \rangle S \langle Q \rangle \implies \{\text{trans}(P, \Gamma)\} \triangleright \langle S \rangle \{\text{trans}(Q, \Gamma)\}
\]

**Proof.** By induction over the derivation of \(\langle P \rangle S \langle Q \rangle\); for each case of \(S\), we build a corresponding derivation for \(\{\text{trans}(P, \Gamma)\} \triangleright \langle S \rangle \{\text{trans}(Q, \Gamma)\}\).

In all cases we start by assuming \(\text{wf}(P \land Q, \Gamma) \land \text{wf}(S, \Gamma)\).

Cases of \(S\):

1. Base case, in which the last step of the derivation is \(\text{Skip; } \langle P \rangle \text{skip } \langle Q \rangle\). From the structure of \(\text{Skip}\), it must be the case that \(P\) and \(Q\) are structurally identical, i.e. the derivation is \(\langle P \rangle \text{skip } \langle P \rangle\). Since \(\text{trans}\) is a function, it maps \text{skip} to exactly one statement (namely \text{skip}).
and $P$ to exactly one expression $\text{trans}(P, \Gamma)$. Finally, we apply the Skip Hoare rule to obtain $\{\text{trans}(P, \Gamma)\} \text{skip} \{\text{trans}(P, \Gamma)\}$.

(2) Inductive case, in which the last step of the derivation is Consequence: $⟨P⟩ S ⟨Q⟩$. We will use the corresponding Consequence rule of Hoare logic to build a derivation for $\{\text{trans}(P, \Gamma)\} \text{trans}(S, \Gamma) \{\text{trans}(Q, \Gamma)\}$. Since the case is Consequence, there must be $P'$ and $Q'$ such that $P \Rightarrow P', Q' \Rightarrow Q$, and $⟨P'⟩ S ⟨Q'⟩$. From $\Rightarrow$, we know that $\text{trans}(P, \Gamma) \Rightarrow \text{trans}(P', \Gamma)$ and $\text{trans}(Q', \Gamma) \Rightarrow \text{trans}(Q, \Gamma)$. From the inductive hypothesis, we have the ImpArray triple

$$\{\text{trans}(P', \Gamma)\} \text{trans}(S, \Gamma) \{\text{trans}(Q', \Gamma)\},$$

and so we apply the Consequence ImpArray rule to obtain

$$\{\text{trans}(P, \Gamma)\} \text{trans}(S, \Gamma) \{\text{trans}(Q, \Gamma)\}.$$  

(3) Inductive case, in which the last step of the derivation is Conditional: $⟨P⟩ \text{if } E \text{ then } S_1 \text{ else } S_f ⟨Q⟩$. This follows from the inductive hypothesis applied to $E, S_1,$ and $S_f$, as well as the Conditional ImpArray rule.

(4) Inductive case, in which the last step of the derivation is Sequence: $⟨P⟩ S_1 ; S_2 ⟨Q⟩$. This follows from the inductive hypothesis applied to $S_1$ and $S_2$, as well as the Sequence ImpArray rule.

(5) Inductive case, in which the last step of the derivation is Assign: $⟨P⟩ v := E ⟨Q⟩$. In this case, the translation produces an Imp assignment to $v$. Since the last step is Assign, there must be a fresh variable $v'$ such $Q$ is the strongest postcondition of the assignment to $v$:

$$\exists v'. P[v \mapsto v'] \land v = E[v \mapsto v']$$

From the inductive hypothesis, we know that translating the Spyder triple produces an equivalent Imp Hoare triple

$$\{\text{trans}(P, \Gamma)\} v := \text{trans}(E, \Gamma) \{\exists v'. P[v \mapsto v'] \land v = E[v \mapsto v'], \Gamma)\}.$$

If you consider the translated term $\text{trans}(\exists v' . P[v \mapsto v'] \land v = E[v \mapsto v'], \Gamma)$, using $\Rightarrow$ and the definition of translation, you’ll find that it is exactly the ImpArray postcondition for assignment with $\text{trans}(P, \Gamma)$ as a precondition:

$$\exists v' . \text{trans}(P, \Gamma)[v \mapsto v'] \land v = \text{trans}(E, \Gamma)[v \mapsto v'].$$

So, we apply Assign with $P$ as a precondition to obtain

$$\{\text{trans}(P, \Gamma)\} v := \text{trans}(E, \Gamma) \{\exists v' . \text{trans}(P, \Gamma)[v \mapsto v'] \land v = \text{trans}(E, \Gamma)[v \mapsto v']\},$$

(6) Inductive case, in which the last step of the derivation is Put: $⟨P⟩ v \leftarrow E ⟨Q⟩$. For this, we will show that the translation of the put $v \leftarrow E$ takes the precondition $\text{trans}(P, \Gamma)$ to the translation of the Spyder post-condition $\exists v' . \text{weaken_foreach}(P, v, \Gamma)[\text{val}(v) \mapsto v'] \land \text{val}(v) = E[\text{val}(v) \mapsto v'].$

Consider the translation of $\text{val}(v)$ in the context of $\Gamma$. Since $\Gamma$ is well-formed with respect to the Put to $v$, it must be the case that $v \in \Gamma$ and $\Gamma(v) = y$ for some variable $y$. Furthermore, the Spyder expressions $\text{val}(v)$ and $\text{iter}(v)$ are translated to $y[v]$ and $v$ respectively.

Next, consider the Hoare postcondition of the translated put statement. The Put statement is translated to $y[v] := \text{trans}(E, \Gamma)$, and we can apply the Assign-Array rule to obtain the postcondition of $\text{trans}(P, \Gamma)$:

$$\{\text{trans}(P, \Gamma)\} y[v] := \text{trans}(E, \Gamma) \{\exists y' . \text{trans}(P, \Gamma)[y \mapsto y'] \land y = y'[v := \text{trans}(E, \Gamma)[y \mapsto y']\},$$
where $y'$ is some fresh variable.
Because the case is $\text{Put}$, we have just derived the Spyder triple
\[
\langle P \rangle \; v \leftarrow E \; \langle \exists v'. \; P[\text{val}(v) \mapsto v'] \land \text{val}(v) = E[\text{val}(v) \mapsto v'] \rangle,
\]
where $v'$ is some free variable.
Let $\sigma$ be a $\text{ImpArray}$ state such that
\[
[\exists y'. \; \text{trans}(P, \; \Gamma)[y \mapsto y'] \land y = y'[v := \text{trans}(E, \; \Gamma)[y \mapsto y']]]_{\sigma} = t
\].
Consider the Hoare term $P'$, $\exists v'. \; \text{trans}(P[\text{val}(v) \mapsto v'] \land \text{val}(v) = E[\text{val}(v) \mapsto v']$, $\Gamma$, or equivalently,
\[
\exists v'. \; \text{trans}(P, \; \Gamma)[y[v] \mapsto v'] \land y[v] = \text{trans}(E, \; \Gamma)[y[v] \mapsto v'].
\]
We claim that $[P']_{\sigma} = t$. Since $P$ is well-formed with respect to $\Gamma$, and $\Gamma(x) = y$, it must be the case that the substitution of $y \mapsto y'$ only affects translations of $\text{val}(v)$. As a result, if $y'$ is an (array) witness for $P'$, we can use the value $y[v]$ as a (variable) witness for $P'$.
Since $[P']_{\sigma} = t$, we can apply Consequence to obtain the triple
\[
\{\text{trans}(P, \; \Gamma)[x \leftarrow E, \; \Gamma}\} \langle P \rangle
\].
(7) Inductive case, in which the last step of the derivation is For: $\langle P \rangle$ for $(x, y)B_i \langle P \rangle$, where $P$ is of the form \text{foreach}(x, y)P_i. S
At a high-level, this rule is introducing a quantification over the elements of $y$. This is sound because the body $B_i$ can only adjust the elements at the current iteration, because the loop cannot modify variables captured in $I$, and because the translated loop is guaranteed to execute exactly once for every element of $y$.
Let $\Gamma'$ be $\Gamma$ extended with the loop binding $x \mapsto y$. Since $\Gamma$ is well-formed with respect to the loop, it must be the case that $\Gamma'$ is well-formed as well. Recall that the translated loop is
\[
x := 0; \; \text{while}(x < \text{size}(y)) \text{trans}(B_i, \; \Gamma') \; ; \; x := x + 1.
\]
Consider the translated foreach predicate $I$
\[
\forall x'. \; 0 \leq x' < \text{size}(y) \implies \text{trans}(P_i, \; \Gamma')[x \mapsto x'].
\]
We will use the While rule with three helper predicates: intuitively, we keep three predicates around to 1) quantify $I$ for previous iterations 2) safely weaken $I$ for the current iteration and 3) quantify $I$ for future iterations. Let $I_{pre}$ restrict $I$ up to the current iteration,
\[
\forall x'. \; 0 \leq x' < x \implies \text{trans}(P_i, \; \Gamma')[x \mapsto x'].
\]
Let $I_{post}$ weaken $I$ using weaken$_{\text{prev}}(P_i)$ for future iterations:
\[
\forall x'. \; x < x' < \text{size}(y) \implies \text{trans}($\text{weaken\_prev}(P_i)$, $\Gamma')[x \mapsto x'].
\]
Finally, let $I_{curr}$ be the weakening of $I$ for the current iteration: \text{trans}($\text{weaken\_prev}(P_i)$, $\Gamma'$). We will use the While rule with the combined predicate $I_{pre} \land I_{post} \land I_{curr}$ as the loop invariant, and in particular, we will show the following Hoare triple holds:
\[
\{I_{pre} \land I_{post} \land I_{curr} \land 0 \leq x < \text{size}(y)\} \text{trans}(B_i, \; \Gamma') \; ; \; x := x + 1\{I_{pre} \land I_{post} \land I_{curr}\}.
\]
Theorem A.6. Targeted Synthesis for Programming with Data Invariants: Block Append

Lemma A.5

A.4 Soundness of Targeted Synthesis

Lemma A.5 (Block Append).

\[ \forall B, B', P, Q, R. \langle P \rangle B \langle Q \rangle \land \langle Q \rangle B' \langle R \rangle \implies \langle P \rangle B \oplus B' \langle R \rangle. \]

Proof. By structural induction over the arguments of \( \oplus \).  

Theorem A.6.

\[ \forall \Pi, \Phi, B, B'. cn; md \vdash \langle \Pi \rangle B \langle \Phi \rangle \iff B' \implies \langle \Pi \rangle B' \langle \Phi \rangle \]

Proof. Induction over the derivation of \( \langle \Pi \rangle B \langle \Phi \rangle \).

(1) Base case, in which the last step is Synth-Base: \( cn; md \vdash \langle \Pi \rangle \text{skip} \langle \Phi \rangle \iff B \) Because a side-condition for Synth-Base is \( \langle \Pi \rangle B \langle \Phi \rangle \), this is trivially true.

(2) Base case, in which the last step is Synth-Loop: \( cn; md \vdash \langle \Pi \rangle \text{foreach}(v_i, u_i)\Phi \land \Phi \iff B \). This is true from the inductive hypothesis.

\[ \text{Vol. 1, No. 1, Article } \]
(3) Recursive case, in which the last step is Consequence: \( cn ; md \vdash \langle \Pi \rangle B \langle \Phi \rangle \leftrightarrow B' + B'' \).
From Lemma A.5 and the inductive hypothesis, it is the case that \( \langle \Pi \rangle B' + B'' \langle \Phi \rangle \).
(4) Recursive case, in which the last step is \texttt{Assign}: \( cn ; md \vdash \langle \Pi \rangle v := E ; B \langle \Phi \rangle \leftrightarrow v := E ; B' \).
We apply the Hoare rule for \texttt{Assign} to the inductive hypothesis.
(5) Recursive case, in which the last step is \texttt{Put}: \( cn ; md \vdash \langle \Pi \rangle v \leftarrow E ; B \langle \Phi \rangle \leftrightarrow v \leftarrow E ; B' \).
This is analogous to \texttt{Assign}.
(6) Recursive case, in which the last step is one of the Extension rules. These are all trivially sound from the inductive hypothesis.
(7) Recursive case, in which the last step is \texttt{Foreach-Specialize}:
\[
\{ \} \vdash \langle \text{foreach}(v_i, u_i) \phi \land \Phi \rangle \text{ for } (x_i, y_i) B_i ; B \langle \text{foreach}(v_i, u_i) \phi \land \Phi \rangle \leftrightarrow \langle x_i, y_i \rangle (B_{pre} + B'_i) ; B'.
\]
In this case, we use the inductive hypothesis to establish the triple for \( B'_i \). Next, we use the inductive hypothesis and Lemma A.5 to establish the triple for \( B'_i \) and \( B_{pre} \):
\[
\langle \text{weaken_prev}(\Phi) \rangle (B_{pre} + B'_i) \langle \Phi \rangle.
\]
On this, we apply the For Hoare logic rule to introduce the \texttt{foreach} term, and we appeal to the inductive hypothesis for the remainder \( B' \).
(8) Recursive case, in which the last step is \texttt{Conditional}: \( \{ \} \vdash \langle \Phi \rangle \) if \( E \) then \( B_f \) else \( B_f ; B \langle \Phi \rangle \leftrightarrow \langle \Phi \rangle \) if \( E \) then \( B'_f \) else \( B''_f ; B' \).
This follows from the inductive hypothesis and the Conditional Hoare rule.

\[ \Box \]

A.5 Soundness of Targetted Synthesis

Theorem A.7.
\[ \forall \Pi, \Phi, B, B'. cn ; md \vdash \langle \Pi \rangle B \langle \Phi \rangle \leftrightarrow B' \quad \implies \quad \langle \Pi \rangle B' \langle \Phi \rangle \]

Proof. Induction over the derivation of \( cn ; md \vdash \langle \Pi \rangle B \langle \Phi \rangle \leftrightarrow B' \). In all cases we show that \( \langle \Pi \rangle B' \langle \Phi \rangle \).
(1) Base case, in which the last step is \texttt{Synth-Base}: \( cn ; md \vdash \langle \Pi \rangle \text{ skip} \langle \Phi \rangle \leftrightarrow B \). Because a side-condition for \texttt{Synth-Base} is \( \langle \Pi \rangle B \langle \Phi \rangle \), this is trivially true.
(2) Base case, in which the last step is \texttt{Synth-Loop}: \( cn ; md \vdash \langle \Pi \rangle \text{ skip} \langle \text{foreach}(v_i, u_i) \phi \land \Phi \rangle \leftrightarrow B \). This is true from the inductive hypothesis.
(3) Recursive case, in which the last step is Consequence: \( cn ; md \vdash \langle \Pi \rangle B \langle \Phi \rangle \leftrightarrow B' + B'' \).
From Lemma A.5 and the inductive hypothesis, it is the case that \( \langle \Pi \rangle B' + B'' \langle \Phi \rangle \).
(4) Recursive case, in which the last step is \texttt{Assign}: \( cn ; md \vdash \langle \Pi \rangle v := E ; B \langle \Phi \rangle \leftrightarrow v := E ; B' \).
We apply the Hoare rule for \texttt{Assign} to the inductive hypothesis.
(5) Recursive case, in which the last step is \texttt{Put}: \( cn ; md \vdash \langle \Pi \rangle v \leftarrow E ; B \langle \Phi \rangle \leftrightarrow v \leftarrow E ; B' \).
This is analogous to \texttt{Assign}.
(6) Recursive case, in which the last step is one of the Extension rules. These are all trivially sound from the inductive hypothesis.
(7) Recursive case, in which the last step is \texttt{Foreach-Specialize}:
\[
\{ \} \vdash \langle \text{foreach}(v_i, u_i) \phi \land \Phi \rangle \text{ for } (x_i, y_i) B_i ; B \langle \text{foreach}(v_i, u_i) \phi \land \Phi \rangle \leftrightarrow \langle x_i, y_i \rangle (B_{pre} + B'_i) ; B'.
\]
In this case, we use the inductive hypothesis to establish the triple for \( B'_i \). Next, we use the inductive hypothesis and Lemma A.5 to establish the triple for \( B'_i \) and \( B_{pre} \):
\[
\langle \text{weaken_prev}(\Phi) \rangle (B_{pre} + B'_i) \langle \Phi \rangle.
\]
On this, we apply the For Hoare logic rule to introduce the foreach term, and we appeal to the inductive hypothesis for the remainder $B'$. 

(8) Recursive case, in which the last step is Conditional: \{ \} + $\langle \Phi \rangle$ if $E$ then $B_t$ else $B_f$ ; $B$ $\langle \Phi \rangle$ $\rightarrow$ if $E$ then $B'_t$ else $B'_f$ ; $B'$. This follows from the inductive hypothesis and the Conditional Hoare rule.

□