Projet Ajacs

Deliverable WP1

Tools to guarantee a program only uses features of the sub-language, and precise analyses relying on this language

August 2018

We propose two tools to prove analyses of languages: an interactive double-debugger and a systematic way to derive analyses from a semantic descriptions. These tools are described in the following papers.

JSExplain: A Double Debugger for JavaScript, Arthur Charguraud, Alan Schmitt, and Thomas Wood.

Skeletal Semantics and their Interpretations, Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt.

JSExplain: A Double Debugger for JavaScript

Arthur Charguéraud, Alan Schmitt, Thomas Wood

► **To cite this version:**

Arthur Charguéraud, Alan Schmitt, Thomas Wood. JSExplain: A Double Debugger for JavaScript. The Web Conference 2018, Apr 2018, Lyon, France. pp.1-9, <10.1145/3184558.3185969>. <hal-01745792>

HAL Id: hal-01745792

https://hal.inria.fr/hal-01745792

Submitted on 28 Mar 2018

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

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

JSExplain: A Double Debugger for JavaScript

Arthur Charguéraud
Inria & Université de Strasbourg,
CNRS, ICube UMR 7357
arthur.chargueraud@inria.fr

Alan Schmitt
Inria & Univ Rennes, CNRS, IRISA
alan.schmitt@inria.fr

Thomas Wood
Imperial College London
thomas.wood09@imperial.ac.uk

ABSTRACT

We present JSExplain, a reference interpreter for JavaScript that closely follows the specification and that produces execution traces. These traces may be interactively investigated in a browser, with an interface that displays not only the code and the state of the interpreter, but also the code and the state of the interpreted program. Conditional breakpoints may be expressed with respect to both the interpreter and the interpreted program. In that respect, JSExplain is a double-debugger for the specification of JavaScript.

ACM Reference Format:

Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *WWW '18 Companion: The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3184558.3185969

1 INTRODUCTION

1.1 A reference interpreter for JS

JavaScript (JS) has a complex semantics. As of 2017, its specification by ECMA consists of 885 pages of English prose [6] (details in §1.2). Unsurprisingly, this prose-based presentation of the specification does not meet the needs of the JavaScript designers and implementers. In particular, the JavaScript standardization committee (TC39) has repeatedly expressed the need for better tools for describing and interacting with the semantics (§1.3).

Prior work on the formalization of JS semantics, notably JSCert [3] and KJS [11], has made some progress, yet falls short of delivering several of the features needed by TC39 (§1.4). In this work, we aim at addressing these requests: we revisit the JSCert semantics by giving it a presentation more accessible to the JavaScript community. Our presentation aims to be well-suited for writing and reading the specifications, executing test cases, checking coverage, and interatively debugging the specification (§1.5).

The JS specification is essentially describing a reference interpreter. Although it consists of English prose, the ECMA standard reads almost like pseudo-code. Most ambiguities and unclear paragraphs that were present in ECMA3 and ECMA5 were progressively resolved in subsequent editions. Thus, turning ECMA pseudo-code into real code, i.e., code expressed in a real programming language, is not so hard. Yet, there are two nontrivial aspects: dealing with the representation of the state, and dealing with abrupt termination, such as exceptions, return, break, and continue statements.

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW'18 Companion, April 23–27, 2018, Lyons, France
© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.
ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3185969

Indeed, in JS, the evaluation of any sub-expression, of any type conversion, and of most internal operations from the specification may result in the execution of user code, hence the raising of an exception, interrupting the normal control flow. Throught its successive editions, the ECMA standard progressively introduced a notation akin to an exception monad (§1.2). This notation is naturally translated into real code by a proper monadic bind operator of the exception monad.

Regarding the state, the standard assumes a global state. A reference interpreter could either assume a global state, modified with side-effects, or thread the state explicitly in purely-functional style. We chose the latter approach for three reasons. First, we already need a monad for exceptions, so we may easily extend this monad to also account for the state. Second, starting from code with an explicit state would make it easier to generate a corresponding inductive definition in a formal logic (e.g., Coq), which we would like to investigate in the future. Third, to ease the reading, one may easily hide a state that is explicitly threaded; the converse, materializing a state that is implicit, would be much more challenging.

We thus write our reference interpreter in a purely-functional language extended with syntactic sugar for the monadic notation to account for the state and the propagation of abrupt termination (§2). For historical reasons, we chose a subset of OCaml as source syntax, but other languages could be used. In fact, we implemented a translator from our subset of OCaml to a subset of JS (a subset involving no side effects and no type conversions). We thereby obtain a JS interpreter that is able to execute JS programs inside a JS virtual machine—JS fans should be delighted. To further improve accessibility to JS programmers, we also translate the source code of our interpreter into a human-readable JS-style syntax, which we call pseudo-JS, and that essentially consists of JS syntax augmented with a monadic notation and with basic pattern matching.

Our reference semantics for JS is inherently executable. We may thus execute our interpreter on test suites, either by compiling and executing the OCaml code, or by executing the JS translation of that code. It is indeed useful to be able to check that the evaluation of examples from the JS test suites against our reference semantics produces the desired output.

Even more interesting is the possibility to investigate, step by step, the evaluation of the interpreter on a given test case. Such investigation allows to understand *why* the evaluation of a particular test case produces a particular output—given the complexity of JS, even an expert may easily get puzzled by the output value of a particular piece of code. Furthermore, interactive execution makes it easier for the contributor of a new JS feature to add new test cases and to check that these tests trigger the new features and correctly interact with existing features.

In this paper, we present a tool, called JSExplain, for investigating JS executions. This tool can be thought as a *double debugger*, which

displays both the state of the interpreted program and that of the interpreter program. In particular, our tool supports conditional breakpoints expressed simultaneously on the interpreter program and the interpreted program. To implement this tool, we generate a version of our interpreter that is instrumented for producing execution traces (§3), and we provide a web-based tool to navigate through such traces (§4). As far as we know, our tool is the first such double debugger, i.e., debugger with specific additions for dealing with programs that interpret other programs (§5).

1.2 English Specification of JS

To illustrate the style in which the JavaScript standard (ECMA) [6] is written, consider the description of addition, which will be our running example throughout the paper. In JS, the addition operator casts its arguments to integers and computes their sum, except if one of the two arguments is a string, in which case the other argument is cast to a string and the two strings are concatenated.

The ECMA5 presentation (prior to June 2016) appears in Figure 1. First, observe that the presentation describes both the parsing rule for addition and its evaluation rule. Presumably for improved accessibility, the JS standard does not make explicit the notion of an abstract syntax tree (AST). The semantics of addition goes as follows: first evaluate the left branch to a value, then evaluate the right branch to a value, then converts both values (which might be objects) into primitive values (e.g., string, number, ...), then test whether one of the two arguments is a string. If so, cast both arguments to strings and return their concatenation; otherwise cast both arguments to numbers and return their sum.

This presentation style used in ECMA5 gives no details about the propagation of exceptions. While the treatment of exceptions is explicit for statements, it is left implicit for expressions. For example, if the evaluation of the left branch raises an exception, the right branch should not be evaluated. It appeared that leaving the treatment of exceptions implicit could lead to ambiguities at what exactly should or should not be evaluated when an exception gets triggered. The ECMA committee hates such ambiguities, because it could (and typically does) result in different browsers exhibiting different behaviors—the nightmare of web-developers.

In ECMA6, such ambiguities were resolved by making the propagation of exceptions explicit. Figure 2 shows the updated specification for the addition operator. There are two main changes. First, each piece of evaluation is described on its own line, thereby making the evaluation order crystal clear. Second, the meta-operation ReturnIfAbrupt is invoked on every intermediate result. This meta-operation essentially corresponds to an exception monad. The ECMA6 standards, which aims to be accessible to a large audience, avoids the introduction of the word "monad". Instead, it specifies ReturnIfAbrupt as a "macro", as shown in Figure 3. Essentially, every result consists of a "completion record", which corresponds to a sum type distinguishing normal results from exceptions.

In all constructs except try-catch blocks, exceptions interrupt the normal flow of the evaluation. As a result, ECMA6 specification is scattered with about 1100 occurences of ReturnIfAbrupt operations. Realizing the impracticability of that style of specification, the standardization committee decided to introduce in ECMA7 an additional layer of syntactic sugar in subsequent versions of the

Evaluation of: *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   - Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*).
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*).

Figure 1: ECMA5 specification of addition.

Evaluation of: *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*rprim*).
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. let *lstr* be ToString(*lprim*).
    b. ReturnIfAbrupt(*lstr*).
    c. let *rstr* be ToString(*rprim*).
    d. ReturnIfAbrupt(*rstr*).
    e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. let *lnum* be ToNumber(*lprim*).
13. ReturnIfAbrupt(*lnum*).
14. let *rnum* be ToNumber(*rprim*).
15. ReturnIfAbrupt(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*.

Figure 2: ECMA6 specification of addition.

specification. As detailed in Figure 4, they define the question mark symbol to be a lightweight shorthand for ReturnIfAbrupt steps. The specification of addition in that new style is shown in Figure 5.

The presentation of ECMA7 and ECMA8 (Figure 5) is both more concise than that of ECMA6 (Figure 2) and more formal than that of ECMA5 (Figure 1). The use of question marks is to be compared in §2 with the monadic notation that we use for our formal semantics.

1.3 Requests from the JS Committee

The JavaScript standardization body, part of ECMA and known as TC39, includes representatives from browser vendors, major

Evaluation of: ReturnIfAbrupt

Algorithms steps that say

1. ReturnIfAbrupt(*argument*).

mean the same thing as:

1. If *argument* is an abrupt completion, return *argument*.
2. Else if *argument* is a Completion Record, let *argument* be *argument*.[[value]].

Figure 3: ECMA6 interpretation of ReturnIfAbrupt.

Algorithms steps that say or are otherwise equivalent to:

1. ReturnIfAbrupt(AbstractOperation()).

mean the same thing as:

1. Let *hygienicTemp* be AbstractOperation().
2. If *hygienicTemp* is an abrupt completion, return *hygienicTemp*.
3. Else if hygienicTemp is a Completion Record, let *hygienicTemp* be *hygienicTemp*.[[Value]].

Where *hygienicTemp* is ephemeral and visible only in the steps pertaining to ReturnIfAbrupt.

Invocations of abstract operations and syntax-directed operations that are prefixed by **?** indicate that ReturnIfAbrupt should be applied to the resulting Completion Record. For example, the step:

1. **?** OperationName().

is equivalent to the following step:

1. ReturnIfAbrupt(OperationName()).

Figure 4: ECMA7 and ECMA8 addition for ReturnIfAbrupt.

Evaluation of: *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. let *lstr* be ? ToString(*lprim*).
   b. let *rstr* be ? ToString(*rprim*).
   c. Return the String that is the result of concatenating *lstr* and *rstr*.
8. let *lnum* be ? ToNumber(*lprim*).
9. let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*.

Figure 5: ECMA7 and ECMA8 specification of addition.

actors of the web, and academics. It aims at defining a common semantics that all browsers should implement. TC39 faces major challenges. On the one hand, it must ensure full backward compatibility, to avoid "breaking the web". In particular, no feature used in the wild ever gets removed from the specification. On the other hand, the committee imposes the rule that no feature may be added to the standard before it has been implemented, shipped, and tested at scale in at least two distinct major browsers. Any member of the committee may propose new features, hence there are many proposals being actively developed, at different stages of formalization [14].

The rapid evolution of the standard stresses the need for appropriate tools to assist in the rewriting, testing, and debugging of the semantics. In particular, several members with whom we have had interactions expressed their need for several basic tools, such as:

- a tool for knowing whether all variables that occur in the specification are properly defined (bound) somewhere;
- a tool to perform basic type-checking of the meta-functions and of the variables involved in the specification;
- a tool for checking that effectful operations go on a line of their own, to avoid ambiguity in the order of evaluation;
- a tool for checking that the behavior is specified in all cases;
- a tool able to tell which lines from the specification are not covered by any test from the main test suite (test262 [15]);
- a tool able to execute step-by-step the specification on concrete JS programs, and to inspect the value of the variables.

In particular, step-by-step execution is critically needed to evaluate new features. When the committee decides that a feature proposal is worth integrating, it typically does not accept the proposal as is, but instead modifies the proposal in a way that is amenable to a simple, clear specification without corner cases, carefully trying to avoid harmful interactions with other existing features (or planned features). During this process, at some point the committee members have in their hand a draft of the extended semantics as well as a collection of test cases illustrating the new behaviors that should be introduced. Naturally, they would like to check that their formalization of the extended semantics assigns the expected behavior to each of the test cases.

One might argue that such a task could be performed by modifying one of the mainstream browsers. Yet, existing JS runtimes are built with efficiency in mind, with huge code bases involving numerous optimizations. As such, modifying the code in any way is too costly for committee members to investigate variations on a feature request. Even if they could invest the effort, the distance between the English prose specification and the implementation would be too large to have any confidence that the two match, i.e., that the behavior implemented in the code matches the behavior described by the prose.

An alternative approach to testing a new feature is to develop an elaboration (local translation) of that feature into plain JS. This can take the form of syntactic sugar adding a missing API, namely a *polyfill*, or the form of a source to source translation, namely a *transpiler*. For instance, one might translate so-called "template literals" into simple string concatenation.

```
/* new feature */            /* plain JavaScript */
var name = "me";               var name = "me";
`hello ${name}`;               "hello " + name;
```

While polyfills and transpilers are a simple approach to testing new features, they have two major limitations. First, the encoding might be very invasive. For instance, the 2015 version of ECMAScript added *proxies*, and as a consequence significantly changed the internal methods of the language; the Babel [1] transpiler for proxies [2] is able to simulate this feature in prior version of JS, but at the cost of replacing all field access operations with calls to wrapper functions. Second, the interaction of several new features implemented using these approaches is very difficult to anticipate.

1.4 Formal Specifications of JS

In recent years, two projects, JSCert [3] and KJS [11] have proposed a formal specification for a significant subset of JS. JSCert provides a big-step inductive definition for ECMA5, (technically, a pretty-big-step specification [5]), formalized in the Coq proof assistant [16]. JSCert comes along with a reference interpreter, called JSRef, that is proved correct with respect to the inductive definition. JSRef may be extracted into executable OCaml code for executing tests. KJS describes a small-step semantics for JS as a set of rewriting rules, using the K framework [13]. This framework has been used to formalize the semantics of several other real-world languages. It provides in particular tool support for executing (syntax-directed) transition rules on a concrete input program.

At first sight, it might seem that a formal specification addresses all the requests from the committee. Definitions are thoroughly type-checked; in particular, all variables must be properly bound. Definitions, being defined in a formal language, are ambiguity-free; in particular, the order of evaluation and the propagation of exceptions is precisely specified. The semantics can be executed on concrete input programs; moreover, with some extra tooling, one may execute a set of programs and report on the coverage of the specification by the tests.

Given all the nice features of formal semantics, why wouldn't the standardization committee TC39 consider one of these formal semantics as the reference for the language? After discussing with senior members from TC39, we understand that there are (at least) three main reasons why there is no chance for a formal semantics such as JSCert or KJS to be adopted as reference semantics.

1. Formal specifications in Coq or in K use syntax and concepts that are not easily accessible to JS programmers. Yet, the specification is meant to be read by a wide audience.
2. These formal languages have a cost of entry that is too high for committee members to reach the level of proficiency required for contributing new definitions all by themselves.
3. JSCert and KJS come with specifications that can be executed, yet provide no debugger-style support for interactively navigating through an execution and for visualizing the state and the values of the variables, and thus do not help so much in tuning the description of new features.

In the present work, we temporarily leave aside the motivation of giving a formal semantics to JS that one could use to formalize properties of the language (e.g., security properties), and rather focus on trying to provide a formal semantics that meets better the day-to-day needs of the TC39 committee.

1.5 Contribution

In this paper, we present a tool, called JSExplain, which aims at providing a formal semantics for JS that addresses the aforementioned limitations of prior work. Our contribution is two-fold. First, we present a specification for JS expressed in a language that, we argue, JS programmers can easily read and write (§2). Second, we present an interactive tool that supports step-by-step execution of the specification on an input JS program (§3 and §4). Our tool mimics the features of a debugger, such as navigation controls, state and variable visualization, and conditional breakpoints, but does so for both the interpreter program and the interpreted program.

The language in which we display the specification consists of a subset of JS extended with syntactic sugar for monads and basic pattern matching. This language, which we call pseudo-JS, could be the source syntax for our specification. However, for historical and technical reasons, we use as input syntax a subset of OCaml, which is processed using the OCaml type-checker. Our current tool automatically converts the OCaml AST into pseudo-JS code. In the future, we might as well have our reference interpreter be directly in pseudo-JS syntax, and we could typecheck that code either by converting it to OCaml or by reimplementing a basic ML type-checker. A third alternative would be to use the Reason syntax [12], a JS-like syntax for OCaml programs. The only difference between the approaches is whether TC39 committee members would prefer to write OCaml style or JS style code.

2 SPECIFICATION LANGUAGE

2.1 Constructs of the Language

The input syntax in which we write and display the specification is a purely-functional language that includes the following constructs: variables, constants, sequence, conditional, let-binding, function definition, function application (with support for prefix and infix functions), data constructors, records (including record projections, and the "record-with" construct to build a copy of a record with a number of fields updated), tuples (i.e., anonymous records), and simple pattern matching (only with non-nested patterns, restricted to data constructors, constants, variables, and wildcards). For convenience, let-bindings and functions may bind patterns (as opposed to only variables).

We purposely aim for a specification language with a limited number of constructs and a very standard semantics, to minimize the cost of entry into that language. Note that the input code is type-checked in ML. (Polymorphism is used mainly for type-checking options and lists, and operations on them.)

As explained earlier (§1.2), the semantics involves the propagation of exceptions and other abrupt behaviors (break, continue, and return). Their propagation can be described within our small language, using functions and pattern matching. Nevertheless, introducing a little bit of syntactic sugar greatly improves readability. For example, we write "`let%run x = e1 in e2`" to mean "`if_run e1 (fun x -> e2)`", where `if_run` is a function that implements our monad.

The monadic operator `if_run` admits a polymorphic type, hence functions from the specification may return objects of various types. Nevertheless, in practice most functions from the ECMA standard

```
and run_binary_op s c op v1 v2 =
  match op with
  | C_binary_op_add -> run_binary_op_add s c v1 v2
  ...
and run_binary_op_add s0 c v1 v2 =
  let%prim (s1, w1) = to_primitive_def s0 c v1 in
  let%prim (s2, w2) = to_primitive_def s1 c v2 in
  if  (type_compare (type_of (Coq_value_prim w1)) Coq_type_string)
   || (type_compare (type_of (Coq_value_prim w2)) Coq_type_string)
  then
    let%string (s3, str1) = to_string s2 c (Coq_value_prim w1) in
    let%string (s4, str2) = to_string s3 c (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_string (strappend str1 str2))))))
  else
    let%number (s3, n1) = to_number s2 c (Coq_value_prim w1) in
    let%number (s4, n2) = to_number s3 c  (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_number (n1 +. n2))))))
```

Figure 6: Current input syntax of our specification language: a subset of pure OCaml, extended with monadic notation.

are described as returning a "completion triple", which either describe abrupt termination or describe a value. In a number of cases, the value is in fact constrained to be of a particular type. For example, if `to_number` produces a value, then this value is necessarily a number. The standard exploits this invariant implicitly in formulation such as "let *n* be the number produced by calling `to_number`". In constrast, our code needs to explicitly project the number from the value returned. To that end, we introduce specialized monads such as `if_number`, written in practice "`let%number n = e1 in e2`". (An alternative approach would be to assign polymorphic types to completion triples, however following this route would require diverging slightly from ECMA's specification in a number of places.)

Figure 6 shows the specification of addition in our reference interpreter, in OCaml syntax extended with the monadic notation. This code implements its informal equivalent from Figure 2. In that code, `s` denotes the state, `c` denotes the environment (variable and lexical environment, in JS terminology), `op` corresponds to the operator (here, the constructor `C_binary_op_add` corresponds to the AST token describing the operator +), `v1` and `v2` corresponds to the arguments, and `w1` and `w2` to their primitive values. The function `strappend` denote string concatenation, whereas "`+.`" denotes addition on floating pointer numbers (i.e., JS's numbers).

First observe that, as explained earlier (§1.1), the state is threaded throughout the code. We show in the next section how to hide the state variables (§2.2). Observe also that the code also relies on a few auxiliary functions. The function `type_compare` implements comparison over JS types—to keep our language small and explicit, we do not want to assume a generic comparison function with nontrivial specification. The functions `to_primitive_def`, `to_string`, and `to_number` are internal functions from the specification that implement conversions. These operations might end up evaluating arbitrary user code, and thus could perform side-effects or raise exceptions, hence the need to wrap them in monadic let-bindings.

One important feature of this source language is that it does not involve any "implicit" mechanism. All type conversions are explicit in the code, so it is always perfectly clear what is meant. In particular, there is no need to type-check the code to figure out its semantics. In summary, the OCaml code of the interpreter (e.g., Figure 6) is well-suited as a non-ambiguous input language. Note that this code may be compiled using OCaml's compiler in order to run test cases; the current version of our interpreter passes more than 5000 test cases from the official test suite (test262).

```
var run_binary_op = function (op, v1, v2) {
 switch (op) {
  case C_binary_op_add:
    return (run_binary_op_add(v1, v2));
  ... }
};
var run_binary_op_add = function (v1, v2) {
  var%prim w1 = to_primitive_def(v1);
  var%prim w2 = to_primitive_def(v2);
  if ((type_compare(type_of(w1), Type_string)
      || type_compare(type_of(w2), Type_string))) {
    var%string str1 = to_string(w1);
    var%string str2 = to_string(w2);
    return strappend(str1, str2);
  } else {
    var%number n1 = to_number(w1);
    var%number n2 = to_number(w2);
    return (n1 + n2);
  }
};
```

Figure 7: Generated code for the interpreter in pseudo-JS syntax, with implicit environments, state, and casts.

2.2 Translation into Pseudo-JS Syntax

Although we believe that it is a desirable feature to have a source langage fully explicit, there is also virtue in pretty-printing the source code of our interpreter in a more concise syntax. The "noise" that appears in the formal specification (e.g. Figure 6) comes from three main sources:

(1) every function takes as argument the environment;
(2) every function takes as argument and returns a description of the mutable state (a.k.a. heap);

(3) values are typically built using numerous constructors, e.g. `C_value_prim`, which lifts a number (an OCaml value of type **`float`**) into a JS value (an OCaml value of type **`value`**).

Fortunately, we can easily eliminate these three sources of noises.

First, the environment is almost always passed unchanged. It may be modified only during the scope of a function call, a with construct, or a block. When it is modified, new bindings are simply pushed into the environment (which behaves like a stack), and subsequently popped. Thus, we may assume, like the ECMA specification does, that the environment is stored in a global state. This saves the need to pass an argument called "c" around.

Second, the description of the mutable state is threaded through the code. The "current state" is passed as argument to every function that might perform side-effects, and, symmetrically, the "updated state" is returned to the caller, which binds a fresh name for it. Considering that there is only one version of the state at any given point of an execution, we may assume, like the ECMA specification does, that state to be stored in a global variable. This saves the need to pass values called "s1", "s2"... around.

Third, the presence of many constructors is due to the need for casts. Many of these casts could, however, be viewed as "implicit casts" (or "coercions", in Coq's terminology). For a carefully chosen set of casts, defined once and for all, and for a well-typed program with implicit casts, there exists a unique (non-ambiguous) way to insert casts in order to make the program type-check. Although we have previously argued that explicit casts are useful, as they allow giving a semantics that does not depend on type-checking, we now argue that it may also be useful to pretty-print the code assuming implicit casts, in order to improve readability.

In summary, we propose to the reader of the specification a version that features implicit state, implicit context, and implicit casts. Given that we are playing the game of pretty-printing syntax, we take the opportunity to switch along the way to a JS-friendly syntax, using brackets and semicolons. This target language, called pseudo-JS syntax, consists of a subset of the JS syntax, extended with monadic notation, and an extended `switch` construct that is able to bind variables (like OCaml's pattern matching, but restricted to non-nested patterns for simplicity).

The pretty-printing of the addition operator in pseudo-JS syntax appears in Figure 7. To illustrate our extended pattern matching syntax feature of pseudo-JS, we show below an excerpt from the main switch that interprets an expression.

```
switch (t) {
 case Coq_expr_identifier(x):
   var%run r = identifier_resolution(x); return (r);
 case Coq_expr_binary_op(e1, op, e2): ...
```

3 TRACE-PRODUCING EXECUTIONS

JSExplain is a tool for interactively investigating execution traces of our JS interpreter executing example JS programs. The interface consists of a web page [8] that embeds a JS parser and a trace-producing version of our interpreter implemented in standard JS.

So far, we have shown how to translate the OCaml source into pseudo-JS syntax (§2.2). In this section, we explain how to translate the OCaml source into proper JS syntax, and then how to instrument the JS code in a systematic way for producing execution traces.

Figure 8 illustrates the output of translating from our OCaml subset towards JS. Note that this code is not meant for human consumption. We implement monadic operators as function calls, introduce the `return` keyword where necessary, encode sum types as object literals with a `tag` field, encode tuples as arrays (encoding tuples as object literals would work too), turn constructor applications into functions calls, implement pattern matching by first switching on the `tag` field then binding fresh variables to denote the arguments of constructors.

We thus obtain an executable JS interpreter in JS which, like our JS interpreter in OCaml, may be used for executing test cases. One interest of the JS version is that it may be easily executed inside a browser, a set up that might be more convenient for a number of users. One limitation, however, is that the number of steps that can be simulated may be limited on JS virtual machines that do not optimize tail-recursive function calls. Indeed, the execution of monadic code involves repeated calls to continuations, whose (tail-call) invocation unnecessarily grows the call stack.

To set up our interactive debugger, we produce, from our OCaml source code, an instrumented version of the JS translation. This instrumented code produces execution traces as a result of interpreting an input JS program. These traces store information about all the states that the interpreter goes through. In particular, each event in the trace provides information about the code pointer and the instantiations of local variables from the interpreter code.

More precisely, we log events at every entry point of a function, every exit point, and on every variable binding. Each event captures the state, the stack, and the values of all local variables in scope of the interpreter code at the point where the event gets triggered. To reduce noise in the trace, we only log events in the core code of the interpreter, and not the code from the auxiliary libraries. Overall, an execution of the instrumented interpreter on some input JS program produces an array of events. This array can then be investigated using our double debugger (§4).

Figure 9 shows an example snippet of code, giving an idea of the mechanisms at play. Note, again, that this code is not meant for human consumption. The function `log_event` augments the trace. Consider for instance `log_event("Main.js", 4033, ctx_747, "enter")`. The first two arguments identify the position in the source file, as a file name and a unique token used to recover the line numbers. The third argument is a context describing values of the local variables, and the fourth argument describes the type of event.

When investigating the trace, we need to be able to highlight the corresponding line of the interpreter code. We wish to be able to do so for the three versions of the interpreter code: the OCaml version, the pseudo-JS version, and the plain JS version. To implement this feature, our generator, when processing the OCaml source code, also produces a table that maps, for each version and for each file of the interpreter, event tokens to line numbers.

The contexts stored in events are extended each time a function is entered, a new variable is declared, or the function returns (so as to capture the returned value). Contexts are represented as a purely-functional linked list of mappings between variable names and values. This representation maximizes sharing and thus minimize the memory footprint of the generated trace. The length of the trace grows linearly with the number of execution steps performed. For example, the simple program "`var i = 0; while (i < N) { i++ }`"

```
var run_binary_op_add = function (s0, c, v1, v2) {
  return (if_prim(to_primitive_def(s0, c, v1), function(s1, w1) {
    return (if_prim(to_primitive_def(s1, c, v2), function(s2, w2) {
      if ((type_compare(type_of(Coq_value_prim(w1)), Coq_type_string())
        || type_compare(type_of(Coq_value_prim(w2)), Coq_type_string()))) {
          return (if_string(to_string(s2, c, Coq_value_prim(w1)), function(s3, str1) {
            return (if_string(to_string(s3, c, Coq_value_prim(w2)), function(s4, str2) {
              return (res_out(Coq_out_ter(s4, res_val(
                           Coq_value_prim(Coq_prim_string(strappend(str1, str2))))))); }));}));
      } else { ... }})); }));
};
```

Figure 8: Snippet of generated code for the interpreter in standard JS syntax, without trace instrumentation.

```
var run_binary_op_add = function (s0, c, v1, v2) {
  var ctx_747 = ctx_push(ctx_empty, [{key: "s0", val: s0}, {key: "c", val: c}, {key: "v1", val: v1}, {key: "v2", val: v2}]);
  log_event("JsInterpreter.js", 4033, ctx_747, "enter");
  var _return_1719 = if_prim((function () {
      log_event("JsInterpreter.js", 3985, ctx_747, "call");
      var _return_1700 = to_primitive_def(s0, c, v1);
      log_event("JsInterpreter.js", 3984, ctx_push(ctx_747, [{key: "#RETURN_VALUE#", val: _return_1700}]), "return");
      return (_return_1700); }()),
    function(s1, w1) { ... });
   log_event("JsInterpreter.js", 4028, ctx_push(ctx_748, [{key: "#RETURN_VALUE#", val: _return_1718}]), "return");
   return (_return_1718);
   });
  log_event("JsInterpreter.js", 4032, ctx_push(ctx_747, [{key: "#RETURN_VALUE#", val: _return_1719}]), "return");
  return (_return_1719);
};
```

Figure 9: Snippet of generated code for the interpreter in standard JS syntax, with trace instrumentation.

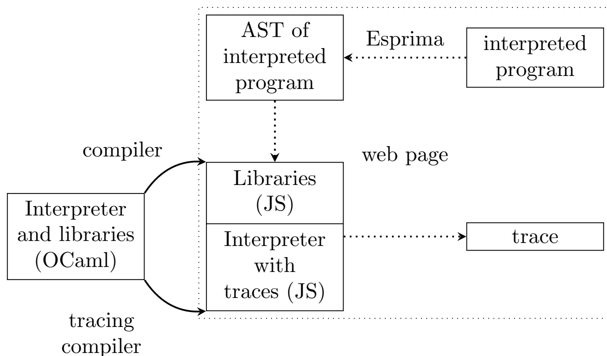

Figure 10: Architecture of JSExplain.

generates a trace of size 2 990 for $N = 1$, of size 14 166 for $N = 10$, of size 126 126 for $N = 100$, and of size 1 245 726 for $N = 1000$.

The fact that these numbers are large reflects the fact that the reference interpreter is inherently vastly inefficient, as it follows the specification faithfully, without any optimization. Due to our use of functional data structures, the memory footprint of the trace should be linear in the length of the trace. We have not observed the memory footprint to be a limit, but if it were we could more carefully select which events should be stored.

4 JSEXPLAIN: A DOUBLE DEBUGGER FOR JS

The global architecture of JSExplain is depicted in Figure 10. Starting from our JS interpreter in OCaml, we generate a JS interpreter in JS. We instrument the JS code to produce a trace of events. This compilation is done ahead of time and depicted by solid arrows.

When hitting the `run` button, the flow depicted by the dotted arrows occurs. The web page parses the code from the text area, using the Esprima library [7]. This parser produces an AST, with nodes annotated with locations. This AST is then provided as input to the instrumented interpreter, which generates a trace of events. This trace may then be inspected and navigated interactively.

For a given event from the execution trace, our interface highlights the corresponding piece of code from the interpreter, and shows the values of the local variables, as illustrated in Figure 11. It also highlights the corresponding piece of code in the interpreted program, as illustrated at the top of Figure 13, and displays the state and the environment of the program at that point of the execution, as illustrated in Figure 12.

Recovering the information about the interpreted code is not completely straightforward. For example, to recover the fragment of code to highlight, we find in the trace the closest previous event that contains a call to function with an argument named `_term_`. This argument corresponds to the AST of a subexpression, and this AST is decorated (by the parser) with locations. Note that, for efficiency reasons, we associate to each event from the trace its corresponding `_term_` argument during a single pass, performed immediately after the trace is produced.

Similarly, we are able to recover the state and environment associated with the event. The state of the interpreted program consists of four fields: the strictness flag, the value of the `this` keyword,

JsInterpreter.js JsInterpreter.pseudo JsInterpreter.ml

```
2217 var arguments_object_map_loop = function
     (l, xs, len, args, x, str, lmap, xsmap) {
2218   return (
2219     function (fO, fS, n) {
2220         if (int_eq(n, 0)) {
2221           return (fO({}));
2222         } else {
2223           return (fS((n - 1)));
```

s: <state-object>
c: <execution-ctx-object>
l:
  – [[Prototype]]: <Object>(object_proto)
  – length: 1
xs: List:
  • "j"
len: 1
args: List:
  • 0
x: List:
  • 3
  • 0
str: false
lmap:
  – [[Prototype]]: <Object>(object_proto)
xsmap: List:

Figure 11: Display of the variables from the interpreter code

strictness: false
this: <Object>(global)
lexical-env:
  • **environment-record-object**:<Object>(global)
    – [[Prototype]]: null
    – i: 3
    – t: <Object>(0)
      – [[Prototype]]: <Object>(array_proto)
      – 0: <Object>(1)
        – [[Prototype]]: <Object>(function_proto)
        – length: 0
        – prototype: <Object>(2)
        – <Function>
          – scope:
            • **environment-record-object**:<Object>(global)
              – i: 3
              – t: <Object>(0)
      – 1: <Object>(3)
      – 2: <Object>(5)
      – length: 3
variable-env:
  • **environment-record-object**:<Object>(global)

Figure 12: Display of the state and environment of the interpreted code. The environment includes the local variables.

the lexical environment, and the variable environment. We implemented a custom display for these elements, and also for values of the languages, in particular for objects: one may click on an object to reveal its contents and recursively explore it.

We provide several ways to navigate the trace. First, we provide buttons for reaching the beginning or end of the execution, and buttons for stepping one event at a time. Second, we provide, similarly to debuggers, *next* and *previous* buttons for skipping function calls, as well as a *finish* button to reach the end of the current function. These features are implemented by navigating the trace, keeping track of the number of *enter* and *return* events. Third, we provide buttons for navigation based not on steps related to the interpreter program but instead based on steps of the interpreted program: *source previous* and *source next* find the closest event which induces

```
14 var t = [];
15 for (var i = 0; i < 3; i++) {
16   t[i] = (function(j) {
17       return function() { return j+5; };
18     })(i);
19 };
20 t[0]();
21 t[1]();
```

RUN Step: 24033 / 24186 (enter)
Begin End Backward Forward Prev Next Finish
Condition: I_line() == 2769 && S('j') == 1 Reach

JsInterpreter.js JsInterpreter.pseudo JsInterpreter.ml

```
2769 var run_binary_op_add = function (v1, v2) {
2770   var%prim w1 = to_primitive_def(v1);
2771   var%prim w2 = to_primitive_def(v2);
2772   if ((type_compare(type_of(w1), Type_string)
2773       || type_compare(type_of(w2), Type_string))) {
2774     var%string str1 = to_string(w1);
2775     var%string str2 = to_string(w2);
2776     return (strappend(str1, str2));
2777   } else {
2778     var%number n1 = to_number(w1);
2779     var%number n2 = to_number(w2);
2780     return ((n1 + n2));
2781   }
2782 };
```

s0: <state-object>
c: <execution-ctx-object>
v1: 1
v2: 5

Figure 13: Example of a conditional breakpoint, constraining the state of both the interpreter and the interpreted code.

a change in the location on the subexpression evaluated in the interpreted code, and *source cursor* finds the last event in the trace for which the associated subexpression contains the active cursor in the "source program" text area.

The aforementioned tools are sufficient for simple explorations of the trace, yet we have found that it is sometimes useful to reach events at which specific conditions occur, such as being at a specific line in the interpreter, in the interpreted code, with variables from the interpreter or interpreted code having specific values. We thus provide a text box to enter arbitrary breakpoint conditions to be evaluated on events from the trace. For example, the condition in Figure 13 reaches the next occurrence of a call to `run_binary_op_add` in a context where the source variable `j` has value 1. The break point condition may be any JS expression using the following API: `I_line()` returns the current line of the interpreter, `S_line()` returns the current line of the source, `I('x')` returns the value of `x` in the interpreter, `S_raw('x')` returns the value of `x` in the source (e.g. the JS object `{tag: "value_number", arg: 5}`), and `S('x')` returns the JS interpretation of the value of `x` in the source (e.g. the JS value `5`).

5 RELATED WORK

There are many formal semantics of JavaScript, from pen-and-paper ones [10], to the aforementioned JSCert [3] and KJS [11]. As described in §1.4, these semantics are admirable but lack crucial features to be actively used in the standardization effort.

To our knowledge, the closest work to the double-debugger approach is the multi-level debugging approach of Kruck et al. [9]. They present a debugger for an interpreter for domain-specific languages that lets developers choose the level of abstraction at which they debug their program. An abstraction is a way to display some values (encoded in the host language, or as present in the DSL) as well as showing only stack frames that represent computation

at the DSL level. Our technique is more general as it does not focus on domain-specific languages.

In fact, our double-debugger approach could be easily adapted to interpreter for other languages than JS. To that end, it suffices to implemented an interpreter for the desired language in the subset of OCaml that we support, and to provide code for extracting and displaying the term and state associated with a given event. We have recently followed that approach and adapted our framework to derive a double-debugger for (a significant subset of) the OCaml programming language.

Regarding the translation from OCaml to JS that we implement, one might consider using an existing, general-purpose tool. Js_of_ocaml [17] converts OCaml bytecode into efficient JS code. Presumably, we could implement the logging instrumentation as an OCaml source-to-source translation and then invoke Js_of_ocaml. Yet, with that approach, we would need to convert the representation of trace events from the encoding of these values performed by Js_of_ocaml into proper JS objects that we can display in the interactive interface. This conversion is nontrivial, as some information, such as the name of constructors, is lost in the process. As we already implemented a translator from OCaml to pseudo-JS, it was simpler to implement a translator from OCaml to plain JS.

Another translator from OCaml to JS is Bucklescript [4], which was released after we started our work. Similarly to our translator, Bucklescript converts OCaml code into JS code advertised as readable. Bucklescript also has the limitation that the names of constructors are lost, although presumably this could be easily fixed. Besides, if we wanted to revisit our implementation to base it on Bucklescript, for trace generation we would need to either modify Bucklescript, which is quite complex as it covers the full OCaml language, or to reimplement trace instrumentation at the OCaml level, which should be doable yet would involve a bit more work than at the level of untyped JS code.

6 CONCLUSION AND FUTURE WORK

We presented JSExplain to TC39[1] and the committee expressed strong interest. They would like us to extend our specification to cover all of the specification. We have almost finished the formalization of proxies, which are a challenging addition to the language as they change many internal methods. Although all members seem to agree that the current toolset for developing the specification is inappropriate, it requires a strong leadership and a consensus to commit to a new toolchain. Our goal is to cover the current version of ECMAScript, we currently cover ECMAScript 5, and to help committee members use it to formalize new additions to JavaScript.

There are numerous directions for future work. (1) We plan to set up a modular mechanism for describing unspecified behaviors (e.g. "for-in" enumeration order) as well as browser-specific behaviors (sometimes browsers deviate from the specification, for historical reasons). (2) We could investigate the possibility of extending the formalization of the standard by also covering the parsing rules of JS; currently, our semantics is expressed with respect to the AST of the input program. (3) To re-establish a link with the original JSCert inductive definition, which is useful for conducting formal proofs about the metatheory of the langage, we would like to investigate the possibility of automatically generating pretty-big-step [5] definitions from the reference semantics expressed in our small language, possibly using some amount of annotation to guide the process. (4) To close even further the gap between a formal language and the English prose, we could also investigate the possibility of automatically generating English sentences from the code. Indeed, the prose from the ECMAScript standard is written in such a systematic manner that this should be doable, at least to some extent.

[1] https://tc39.github.io/tc39-notes/2016-05_may-25.html#jsexplain-as--tw

ACKNOWLEDGMENTS

We acknowledge funding from the ANR project AJACS ANR-14-CE28-0008 and the CominLabs project SecCloud.

REFERENCES

[1] 2017. Babel. (October 25, 2017). https://babeljs.io/.

[2] 2017. Babel proxy plugin. (October 25, 2017). https://www.npmjs.com/package/babel-plugin-proxy.

[3] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised javascript specification. In *Proceedings of POPL 2014*, 87–100.

[4] 2017. Bucklescript. (October 27, 2017). https://bucklescript.github.io/bucklescript/.

[5] Arthur Charguéraud. 2013. Pretty-big-step semantics. In *Proceedings of ESOP 2013* (LNCS). Volume 7792. Springer, 41–60.

[6] ECMA. 2017. Ecmascript 2017 language specification (ecma-262, 8th edition). (June 2017). https://www.ecma-international.org/ecma-262/8.0/index.html.

[7] 2017. ECMAScript parsing infrastructure for multipurpose analysis. (October 26, 2017). http://esprima.org/.

[8] 2017. JSExplain. (October 26, 2017). https://jscert.github.io/jsexplain/branch/master/driver.html.

[9] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. 2016. Multi-level debugging for interpreter developers. In *MODULARITY (Companion)*. ACM, 91–93.

[10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An operational semantics for javascript. In *Proceedings of APLAS 2008* (LNCS). Volume 5356. Springer, 307–325.

[11] Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a complete formal semantics of javascript. In *Proceedings of PLDI 2015*. ACM, 346–356.

[12] 2017. Reason. (October 27, 2017). https://reasonml.github.io/.

[13] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79, 6, 397–434.

[14] 2017. TC39 proposals. (October 26, 2017). https://github.com/tc39/proposals.

[15] 2017. Test262. (October 26, 2017). https://github.com/tc39/test262.

[16] The Coq development team. 2014. *The Coq proof assistant reference manual*. Version 8.4. http://coq.inria.fr.

[17] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Exprerience*, 44, 8, 951–972.

Skeletal Semantics and their Interpretations

ANONYMOUS AUTHOR(S)

Many meta-languages have been proposed for writing rule-based operational semantics, in order to provide general interpreters and analysis tools. We take a different approach. We develop a meta-language for a *skeletal semantics* of a language, where each skeleton describes the complete semantic behaviour of a language construct. We define a general notion of *interpretation*, which provides a systematic and language-independent way of deriving semantic judgements from the skeletal semantics. We provide four generic interpretations of our skeletal semantics to yield: a simple well-formedness interpretation; a concrete interpretation; an abstract interpretation; and a constraint generator for flow-sensitive analysis. We prove general *consistency results*, establishing that the concrete and abstract interpretations are consistent and that any solution to the constraints generated by the constraint generator must be a correct abstract semantics.

1 INTRODUCTION

Plotkin's Structural Operational Semantics [Plotkin 1981] gave momentum to a movement in programming languages in which a language's semantics is defined via inference rules. A number of tool-assisted formalisms for defining such rule-based semantics have since then emerged, such as Twelf [Pfenning and Schürmann 1999] based on the Edinburgh Logical Framework [Harper et al. 1987] and, more recently, $\mathbb{K}$ [Roşu and Şerbănuţă 2010], Ott [Sewell et al. 2010] and Lem [Mulligan et al. 2014]. These formalisms propose a general meta-language for writing inference rules, and machine support for deriving interpreters and analysis tools for the language so defined.

We have been inspired by this agenda to provide a general meta-theory for developing interpreters and analysis tools. We have, however, significantly moved away from the approach of providing a meta-language for writing rule-based operational semantics. Instead, we introduce a meta-language for writing *skeletal semantics*. A skeletal semantics is not a semantics in its own right. It comprises a set of *skeletons*, where each skeleton provides a syntactic formulation of the complete semantic behaviour of a language construct. Associated with skeletons is a definition of *interpretation*, which provides a systematic way of deriving semantic judgements from the skeletal semantics. For example, in this paper, we provide four generic interpretations of our skeletal semantics to yield: a simple well-formedness interpretation based on sorts; a concrete interpretation; an abstract interpretation; and a constraint generator for flow-sensitive analysis. We also prove generic *consistency results*, depending only on simple lemmas for the basic constructs of a language: for example, establishing a general result that the concrete and abstract interpretations are consistent.

An instantiation of our meta-theory to a particular language involves: (1) writing skeletons for each language construct, which is comparatively straightforward from a big-step operational (or natural) semantics [Kahn 1987]; (2) providing language-dependent interpretations of the basic constructs of the language; and (3) proving simple lemmas for these basic constructs to prove generic consistency results. We have shown that a number of semantic results about a simple While language can indeed be stated and proved generally using our skeletal semantics and their interpretations, and then instantiated to many languages, including While.

Our work provides a general theory for relating many forms of semantics at a language-independent level, and is thus related to the theory abstract interpretation [Cousot and Cousot 1977]. Abstract interpretation provides general definitions to capture what it means for an abstract semantics to be consistent with respect to the concrete semantics, using Galois connections. These definitions guide the building of an abstract semantics, but they leave large parts to be dealt with

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

50
51
52

$$\frac{\sigma, e \Downarrow true \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad \frac{\sigma, e \Downarrow false \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o}$$

53
54

Fig. 1. Usual concrete rules for the *if* construct

55
56
57
58

$$\frac{\sigma, e \Downarrow true^{\#} \qquad \sigma, t_1 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad \frac{\sigma, e \Downarrow false^{\#} \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o}$$

59
60
61

$$\frac{\sigma, e \Downarrow \top_{bool} \qquad \sigma, t_1 \Downarrow x_o \qquad \sigma, t_2 \Downarrow x_o}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow x_o} \qquad \frac{\sigma, e \Downarrow \bot_{bool}}{\sigma, \mathit{if}\ e\ t_1\ t_2 \Downarrow \bot}$$

62
63
64

Fig. 2. Usual abstract rules for the *if* construct

65
66 manually. Over the years, the challenge has been to generate such an abstract semantics from
67 the concrete semantics and prove consistency. There has been much work on providing recipes
68 for such an endeavour. For example, Van Horn and Might [Van Horn and Might 2010] provide
69 a recipe for constructing abstract semantics from well-known abstract machines. Midtgaard and
70 Jensen [Midtgaard and Jensen 2008] describe a systematic derivation of control flow analyses.
71 The initial work on abstract interpretation [Cousot and Cousot 1977] assumed that the concrete
72 semantics was given by a transition system. Schmidt [Schmidt 1995, 1997a] instead proposed a
73 rule-based approach, assuming that the concrete semantics was given as a big-step operational
74 semantics. He lifted the Galois connection from concrete and abstract domains to concrete and
75 abstract derivations, and proved consistency. This approach produced an abstract semantics close
76 to the concrete semantics. It was easier to build, but it was still not systematic. Recently, Bodin *et*
77 *al.* [Bodin et al. 2015] applied Schmidt's approach to a restricted setting called *pretty-big-step*
78 semantics [Charguéraud 2013]. They identified a general pretty-big-step rule format, demonstrated
79 that it was possible to generate abstract rules from concrete rules in a systematic way, and provided
80 a general consistency result. In contrast, we have introduced a meta-language for skeletal semantics,
81 that can be instantiated to any language with a big-step operational semantics, with several general
82 interpretations consistency results.
83 We demonstrate our skeletal semantics in action using the simple conditional statement from
84 the While language. Consider the usual concrete rules associated with the conditional statement in
85 Figure 1, and the abstract rules in Figure 2, supposing that the booleans are abstracted by the usual
86 four-valued lattice given by $\{true^{\#}, false^{\#}, \top_{bool}, \bot_{bool}\}$. These abstract rules are intuitively correct,
87 but they are first built in an ad hoc way and then shown to be related to the concrete rules using a
88 Galois connection. More generally, the systematic construction of abstract rules from concrete rules
89 requires a deep understanding of how the analysed programming language evaluates expressions:
90 in a case like a vanilla While language, this is quite straightforward; but for a complex language
91 such as JavaScript [Bodin et al. 2014; ECMA 2018; Maffeis et al. 2008], the relationship between the
92 concrete and abstract semantics can be difficult to get right.
93 We define the skeletal semantics, providing a general meta-theory for defining language se-
94 mantics, in Section 2. Figure 3 shows the skeleton associated with the *if* construct, with generic
95 subterms denoted by x_{t_1}, x_{t_2}, and x_{t_3}, input state x_σ and output state x_o. The H construct identi-
96 fies the required subcomputations associated with x_{t_1}, x_{t_2}, and x_{t_3}. The skeleton stitches these H
97 constructs together, using the internal symbolic variable x_{f_1} and the branching which identifies
98

99
100
101
102
103
104
105
106
107
108
109
110
111
112

$$\textsc{If}\left(if\ x_{t_1}\ x_{t_2}\ x_{t_3}\right) := \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \begin{pmatrix} \texttt{isTrue}\left(x_{f_1}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_1}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]$$

Fig. 3. Skeleton for the *if* construct

$$\frac{\sigma^\#, e \Downarrow v^\# \qquad \left(\llbracket\texttt{isTrue}\rrbracket^\#\left(v^\#\right) \implies \sigma^\#, t_1 \Downarrow \sigma_o^\#\right) \qquad \left(\llbracket\texttt{isFalse}\rrbracket^\#\left(v^\#\right) \implies \sigma^\#, t_2 \Downarrow \sigma_o^\#\right)}{\sigma^\#, if\ e\ t_1\ t_2 \Downarrow \sigma_o^\#}$$

Fig. 4. The abstract interpretation of the the *if* construct: intuitive description.

113 paths through the skeleton using the filters `isTrue` and `isFalse`. Such a skeleton thus explicitly
114 describes both the data flow and the control flow associated with a language construct, identifying
115 the common pattern underlying the concrete and abstract rules.
116 We provide a general definition of interpretation for our skeletal semantics in Section 3 and
117 study four generic interpretations:

118 119
- A simple well-formedness interpretation which states that the stitching of the skeleton in Figure 3 respects the sorting of the basic constructs.

120 121
- The concrete interpretation (Section 4) which intuitively picks one path from each branching of the skeleton, corresponding to the two rules of Figure 1.

122 123 124 125
- The abstract interpretation (Section 5), whose complex definition (Figure 11) boils down to the intuitive description given by the rule of Figure 4: a rule with optional branches, considering all paths compatible with the return value of the expression e. This rule naturally subsumes the four rules of Figure 2.

126 127 128 129
- A constraint generator for flow-sensitive static analysis (Section 7). Although these constraints are different in nature to the abstract semantics, they are expressed in our meta-theory using the same mechanism, i.e., an interpretation of the skeletal semantics. This provides a strong connection between them.

130
131 We also define notions of *consistency* between interpretations (Section 3.2). The shared structure of
132 our different interpretations greatly eases the proof process. We use our notions of consistency to
133 show that the abstract interpretation is correct in relation to the concrete interpretation, and that
134 any solution to the constraints generated must be a correct abstract semantics.
135 We instantiate our skeletal semantics to a While language throughout the paper, and demonstrate
136 how classic proof techniques based on an abstract interpretation of While can be captured with
137 our approach (Section 6). We however emphasise that skeletons and interpretations, as well as their
138 consistency proofs, are generic and can be applied to any programming language. Most proofs
139 have been moved to the anonymous supplementary material for space reasons.

140
141 ## 2 SKELETAL SEMANTICS

142 ### 2.1 Terms

143 The first ingredient of a skeletal semantics is the syntactic terms and their sorts. We assume given a
144 countable set of *sorts*, ranged over by s, separated into *base sorts* and *program sorts*. We also assume
145 given a countable set of term variables, ranged over by x_t, and a finite set of constructors, ranged
146 over by c. The *signature* of a constructor c, written $sig(c)$, is of the form $(s_1..s_n) \longrightarrow s$, where n is the
147

c	Signature
$const$	$lit \longrightarrow expr$
var	$ident \longrightarrow expr$
$+$	$(expr \times expr) \longrightarrow expr$
$=$	$(expr \times expr) \longrightarrow expr$
$\neg$	$expr \longrightarrow expr$
$skip$	$stat$
$:=$	$(ident \times expr) \longrightarrow stat$
$;$	$(stat \times stat) \longrightarrow stat$
if	$(expr \times stat \times stat) \longrightarrow stat$
$while$	$(expr \times stat) \longrightarrow stat$

Fig. 5. Constructors for While

arity of c, s_n are sorts, and s is a program sort. Note that there is no term of base sort, they will be instantiated for each interpretation.

Let Γ be a mapping from term variables to sorts. Sorted terms are either term variables x_t of sort $\Gamma(x_t)$ or a term $c(t_1..t_n)$ of sort s, where c has signature $sig(c) = (s_1..s_n) \longrightarrow s$ and the subterms $t_1..t_n$ have the appropriate sort. We write $Sort_\Gamma(t)$ for the sort of t. Let E be a mapping from term variables to terms such that $\forall x_t \in dom(E), Sort_\Gamma(E(x_t)) = \Gamma(x_t)$. We extend it to terms as follows: $E(c(t_1..t_n)) = c(E(t_1)..E(t_n))$ when defined. We write $Sort(t)$ for $Sort_\emptyset(t)$

We write $Tvar(t)$ to denote the set of term variables occurring in t. We say t is *closed* if $Tvar(t) = \emptyset$. In that case, we write $t : s$ for $Sort(t) = s$.

Lemma 2.1. *Let t a term, E an environment mapping term variables to closed terms, and Γ a sorting environment such that $Tvar(t) \subseteq dom(E)$, $Tvar(t) \subseteq dom(\Gamma)$, and for any $x_t \in Tvar(t)$ we have $\Gamma(x_t) = Sort(E(x_t))$. Then we have $Sort_\Gamma(t) = Sort(E(t))$.*

Proof. By induction on the structure of t. If it is a term variable then the result is immediate. If it has the shape $c(t_1..t_i)$ then by n inductions we have $Sort_\Gamma(t_i) = Sort(E(t_i))$, and we conclude that $Sort_\Gamma(c(t_1..t_n)) = Sort(E(c(t_1..t_n)))$. □

Running Example. Base sorts are *ident* for *program variables* and *lit* for *literals*. Program sorts are *expr* for *expressions* and *stat* for *statements*. The signature of constructors is given in Figure 5.

2.2 Skeletons

We assume a countable set of *flow variables*, ranged over by x_f, which are used in the skeleton bodies to hold semantic values (states, intermediate values,...). Among flow variables, we distinguish two of them: x_σ holds the semantic state at the start of a skeleton, and x_o is supposed to hold the semantic result at the end of a skeleton. We let *skeletal variables*, ranged over by x or y, be the union of term variables and flow variables. A *skeleton* has the shape $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$, where Name is the skeleton name, t is a term, and S is the *skeleton body*.

$$\begin{array}{rl} \textsc{Skeleton Body} & S ::= [\,] \mid B; S \\ \textsc{Bone} & B ::= H\left(x_{f_1}, t, x_{f_2}\right) \mid F\left(x_1..x_n\right) ?\triangleright \left(y_1..y_m\right) \mid \left(S_1..S_n\right)_V \end{array}$$

A skeleton body is a sequence of *bones*. A bone is either a *hook* $H(x_{f_1}, t, x_{f_2})$, consisting of an input flow variable x_{f_1}, a term t to be hooked during interpretation, and an output flow variable

$$\textsc{LitInt}(\mathit{const}\,(x_t)) \coloneqq [\texttt{litToVal}\,(x_t) \mathrel{?\!\triangleright} x_o]$$

$$\textsc{Var}(\mathit{var}\,(x_t)) \coloneqq [\texttt{read}\,(x_t, x_\sigma) \mathrel{?\!\triangleright} x_o]$$

$$\textsc{Add}\left(x_{t_1} + x_{t_2}\right) \coloneqq \left[\begin{array}{l} H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isInt}\left(x_{f_1}\right) \mathrel{?\!\triangleright} x_{f_{1'}}; H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \texttt{isInt}\left(x_{f_2}\right) \mathrel{?\!\triangleright} x_{f_{2'}}; \\ \quad \texttt{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\!\triangleright} x_o \end{array}\right]$$

$$\textsc{Eq}\left(x_{t_1} = x_{t_2}\right) \coloneqq \left[\begin{array}{l} H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isInt}\left(x_{f_1}\right) \mathrel{?\!\triangleright} x_{f_{1'}}; H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \texttt{isInt}\left(x_{f_2}\right) \mathrel{?\!\triangleright} x_{f_{2'}}; \\ \quad \texttt{eq}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\!\triangleright} x_o \end{array}\right]$$

$$\textsc{Neg}(\neg x_t) \coloneqq \left[H\left(x_\sigma, x_t, x_{f_1}\right); \texttt{isBool}\left(x_{f_1}\right) \mathrel{?\!\triangleright} x_{f_2}; \texttt{neg}\left(x_{f_2}\right) \mathrel{?\!\triangleright} x_o\right]$$

$$\textsc{Skip}(\mathit{skip}) \coloneqq [\texttt{id}\,(x_\sigma) \mathrel{?\!\triangleright} x_o]$$

$$\textsc{Asn}\left(x_{t_1} := x_{t_2}\right) \coloneqq \left[H\left(x_\sigma, x_{t_2}, x_{f_1}\right); \texttt{write}\left(x_{t_1}, x_\sigma, x_{f_1}\right) \mathrel{?\!\triangleright} x_o\right]$$

$$\textsc{Seq}\left(x_{t_1};\ x_{t_2}\right) \coloneqq \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); H\left(x_{f_1}, x_{t_2}, x_o\right)\right]$$

$$\textsc{If}\left(\mathit{if}\ x_{t_1}\ x_{t_2}\ x_{t_3}\right) \coloneqq \left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isBool}\left(x_{f_1}\right) \mathrel{?\!\triangleright} x_{f_{1'}}; \left(\begin{array}{l} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{array}\right)_{\{x_o\}}\right]$$

$$\textsc{While}\left(\mathit{while}\, x_{t_1}\ x_{t_2}\right) \coloneqq \left[\begin{array}{l} H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \texttt{isBool}\left(x_{f_1}\right) \mathrel{?\!\triangleright} x_{f_{1'}}; \\ \left(\begin{array}{l} \texttt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_{f_2}\right); H\left(x_{f_2}, \mathit{while}\, x_{t_1}\ x_{t_2}, x_o\right) \\ \texttt{isFalse}\left(x_{f_{1'}}\right); \texttt{id}\left(x_\sigma\right) \mathrel{?\!\triangleright} x_o \end{array}\right)_{\{x_o\}} \end{array}\right]$$

Fig. 6. Skeletal semantics for While

s	*in*(*s*)	*out*(*s*)
expr	*state*	*val*
stat	*state*	*state*

Fig. 7. Input and output flow sorts for program sorts

x_{f_2}; or a *filter* $F(x_1..x_n) \mathrel{?\!\triangleright} (y_1..y_m)$ which tests if the values bound to its input skeletal variables $(x_1..x_n)$ satisfy a condition specified by F, and in that case outputs values to be bound to $(y_1..y_m)$; or a set of *branches* $(S_1..S_n)_V$ which represent the different behavioural pathways, where V declares the skeletal variables that are shared and must be defined by all branches.

A filter with no output skeletal variables is simply written $F(x_1..x_n)$. It then acts as a predicate.

Requirement 2.2. We require that there exists exactly one skeleton for any given constructor c.

Running Example. The skeletons of our While example are given in Figure 6. Requirement 2.2 is trivially satisfied.

2.3 Flow Sorts

We extend the sorts with *flow sorts*, that are the sorts of values in interpretations. In our running example, flow sorts are *state*, *val*, *int*, and *bool*. We relate flow sorts to hooks and filters as follows.

In a hook $H(x_{f_1}, t, x_{f_2})$, the flow variable x_{f_1} stands for an input state that fits with t, and x_{f_2} stands for a result. Given a program sort s, we define $in(s)$ as its input flow sort and $out(s)$ as its output flow sort. Their definitions are for our running example are given in Figure 7.

246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294

f	$fsort(f)$
litToVal	$lit \longrightarrow val$
read	$(ident, state) \longrightarrow val$
isInt	$val \longrightarrow int$
add	$(int, int) \longrightarrow val$
eq	$(int, int) \longrightarrow val$
isBool	$val \longrightarrow bool$
neg	$bool \longrightarrow val$
write	$(ident, state, val) \longrightarrow state$
id	$state \longrightarrow state$
isTrue	$bool \longrightarrow ()$
isFalse	$bool \longrightarrow ()$

Fig. 8. Filter sorts

Similarly, a filter $F(x_1..x_n) \ ?\triangleright \ (y_1..y_m)$ is assigned a signature, written $fsort(F)$, of the form $(s_1..s_n) \longrightarrow (s'_1..s'_m)$. We write $()$ for the output sort of a filter if $m = 0$ and omit the enclosing parentheses when n or m is 1. Filter signatures for our running example are given in Figure 8.

We check the consistency of the hook and filters with the skeletons in our well-formedness interpretation, introduced in Section 3.1.

3 INTERPRETATIONS

An *interpretation* I specifies how to interpret the empty skeleton body, hooks, filters, and branches. It defines a set of *interpretation states*, ranged over by Σ in this section but with specific notations for each interpretation, and a set of *interpretation results*, ranged over by O in this section, as well as the following relations:

- $[\![[]]\!]^I(\Sigma) \Downarrow O$ defining the interpretation of the empty skeleton body;
- $\left[\!\left[H(x_{f_1}, t, x_{f_2})\right]\!\right]^I(\Sigma) \Downarrow \Sigma'$ defining the interpretation of a hook;
- $[\![F(x_1..x_n) \ ?\triangleright \ (y_1..y_m)]\!]^I(\Sigma) \Downarrow \Sigma'$ defining the interpretation of a filter, for each filter F;
- $[\![\bigoplus_n]\!]^I_V(O, \Sigma) \Downarrow \Sigma'$ defining the merging of the interpretation of branches, where O is a partial function from $[1..n]$ to interpretation results, and where V is the set of skeletal variables defined and shared by all branches.

Given a skeleton body S and the relations above, we define the remaining cases for the interpretation of S as follows.

$$\begin{pmatrix} [\![B]\!]^I(\Sigma) \Downarrow \Sigma' \\ [\![S]\!]^I(\Sigma') \Downarrow O \end{pmatrix} \implies [\![B; S]\!]^I(\Sigma) \Downarrow O$$

$$\begin{pmatrix} \forall i \in dom(O) \,.\, [\![S_i]\!]^I(\Sigma) \Downarrow O(i) \\ \left[\!\left[\bigoplus_n\right]\!\right]^I_V(O, \Sigma) \Downarrow \Sigma' \end{pmatrix} \implies [\![(S_1..S_n)_V]\!]^I_V(\Sigma) \Downarrow \Sigma'$$

Interpretations enable us to define the meaning of skeletons by only specifying the parts that matters. Interpretations apply to any skeletons and are thus independent of the language. The rest of the paper presents different interpretation and their relations.

3.1 Well-Formedness Interpretation

The first interpretation we consider is a *well-formedness* interpretation, to verify that every skeleton is well formed. More precisely, we verify that every skeletal variable used has been first defined, that every variable defined in a skeleton is fresh (with an exception for branches, see below), and that the sorting of filters, hooks, and branches are consistent.

Intuitively, in the hook $H(x_{f_1}, t, x_{f_2})$, flow variable x_{f_1} is *used* and flow variable x_{f_2} is *defined*. Similarly, in the filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$, skeletal variables $(x_1..x_n)$ are used and skeletal variables $(y_1..y_m)$ are defined.

The case for branches $(S_1..S_n)_V$ is special. First, each branch S_i *must* define the skeletal variables in V. Second, every variable defined in the whole set of branches must be distinct, with the exception of the variables in V as they have to be defined in every branch. And third, the only variables defined by the branches that may be used in the rest of the skeleton body are those in V.

We define when the well-formedness (WF) interpretation in Figure 9. Its interpretation states and result consist of a pair of a sorting environments Γ, mapping term variables to base and program sorts, and flow variables to flow sorts, and a set $\mathcal{D}$ of skeletal variables that have been defined at that point. In this interpretation, we write $x : s$ to state that the kind of variable and sort match, namely term variables with base or program sorts, and flow variables with flow sorts.

The interpretation for the empty skeleton body is trivial, it simply returns its arguments. The interpretation of a hook $H(x_{f_1}, t, x_{f_2})$ checks that x_{f_1} is in Γ, that every term variable of t is also in Γ, and that variable x_{f_2} is fresh (i.e., not in $\mathcal{D}$). In addition, it checks that the sort for x_{f_1} is what t expects as input sort and that x_{f_2} is latter bound to an output sort of t.

The interpretation for a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ is similar. It ensures that the input skeletal variables $(x_1..x_n)$ are in Γ, that the number and kind of both input and output variables match the signature of F, that the output variables are fresh, that the sort of the input variable correspond to the input signature of F, and it continues binding the output variables to the output signature of F.

Finally, the interpretation of the merging of branches checks that every branch is well formed, that the variables in V are exactly those shared by the branches (neither less nor more than those), and that the sorting environments returned by the branches all agree when restricted to V. In that case, the returned sorting environment is the concatenation of the input environment and the one shared by the branches. The $n \geq 2$ constraint is to have a more concise way of stating that the variables shared by the branches are exactly those in V, it is not a restriction as an empty set of branches is useless, it prevents the skeleton from being interpreted as offering no pathway, and a singleton set of branches can be inlined.

Let $t = c(t_1..t_n)$ be a closed term such that $Sort(t) = s$, where s is a program sort. There are two ways to assign an output sort to t: directly, as $out(s)$, or using the WF interpretation of the skeleton for c to compute the associated sort x_o. If both coincide, we say the skeleton is *well formed*.

Definition 3.1. A skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$ is *well formed* iff for any closed term $t = c(t_1..t_n)$ such that $Sort(t) = s$, we have $[\![S]\!]^{\mathsf{wf}}\ (\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$ and $\Gamma'(x_o) = out(s)$, with the initial sorting environment Γ being $\left\{x_\sigma \mapsto in(s) + x_{t_1} \mapsto Sort(t_1)..x_{t_n} \mapsto Sort(t_n)\right\}$, and with $\mathcal{D} = dom(\Gamma)$.

In the following we only consider well-formed skeletons. For instance, the skeletons for While are well formed.

3.2 Interpretation Consistency

We now define how to relate interpretations. Given interpretations I_1 and I_2, we assume given a relation $OKst(\Sigma_1, \Sigma_1)$ between the interpretation states, and a relation $OKout(O_1, O_2)$ between their results. Intuitively, consistency is the progagation of these relations along interpretations.

344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374

$$\implies [\![[]]\!]^{\mathsf{wf}} (\Gamma, \mathcal{D}) \Downarrow (\Gamma, \mathcal{D})$$

$$\begin{pmatrix} x_{f_1} \in dom(\Gamma) \subseteq \mathcal{D} \\ Tvar(t) \subseteq dom(\Gamma) \\ \Gamma(x_{f_1}) = in(Sort_\Gamma(t)) \\ x_{f_2} \notin \mathcal{D} \\ \Gamma' = \Gamma + x_{f_2} \mapsto out(Sort_\Gamma(t)) \\ \mathcal{D}' = \mathcal{D} \cup \{x_{f_2}\} \end{pmatrix} \implies [\![H(x_{f_1}, t, x_{f_2})]\!]^{\mathsf{wf}} (\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$$

$$\begin{pmatrix} (x_1..x_n) \subseteq dom(\Gamma) \subseteq \mathcal{D} \\ (x_1..x_n) : (\Gamma(x_1)..\Gamma(x_n)) \\ (y_1..y_m) \cap \mathcal{D} = \emptyset \\ fsort(F) = (\Gamma(x_1)..\Gamma(x_n)) \to (s_1..s_m) \\ (y_1..y_m) : (s_1..s_m) \\ \Gamma' = \Gamma + (y_1..y_m) \mapsto (s_1..s_m) \\ \mathcal{D}' = \mathcal{D} \cup (y_1..y_m) \end{pmatrix} \implies [\![F(x_1..x_n) ?\triangleright (y_1..y_m)]\!]^{\mathsf{wf}} (\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$$

$$\begin{pmatrix} n \geq 2 \\ \forall i \in [1..n]. O(i) = (\Gamma_i, \mathcal{D}_i) \\ \forall i \in [1..n]. dom(\Gamma_i) \subseteq \mathcal{D}_i \\ \forall ij. i \neq j \implies (\mathcal{D}_i \setminus \mathcal{D}) \cap (\mathcal{D}_j \setminus \mathcal{D}) = V \\ \forall i \in [1..n]. \Gamma + \Gamma_i|_V = \Gamma' \\ \mathcal{D}' = \bigcup_{i \in [1..n]} \mathcal{D}_i \end{pmatrix} \implies \left[\!\!\left[\bigoplus_n \right]\!\!\right]_V^{\mathsf{wf}} (O, (\Gamma, \mathcal{D})) \Downarrow (\Gamma', \mathcal{D}')$$

Fig. 9. WF Interpretation

375 We define two kinds of consistency: one about where interpretations are defined, i.e, whether
376 they return a result, and one about their results.

377
378 *Definition 3.2.* Interpretations I_1 and I_2 are *existentially consistent* if for any S, Σ_1, Σ_2, and O_1, such
379 that $OKst(\Sigma_1, \Sigma_2)$ and $[\![S]\!]^{I_1}(\Sigma_1) \Downarrow O_1$, there exists a O_2 such that $[\![S]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$.

380 *Definition 3.3.* Interpretations I_1 and I_2 are *universally consistent* if for any S, Σ_1, Σ_2, O_1, and O_2,
381 if $OKst(\Sigma_1, \Sigma_2)$, $[\![L]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ and $[\![L]\!]^{I_2}(\Sigma_2) \Downarrow O_2$, then $OKout(O_1, O_2)$.

382
383
3.3 Proving Consistency

384 Both consistency properties can be stated at the level of the building block of interpretations.
385 Formally, we have the following two lemmas.

386
387 Lemma 3.4. *Let I_1 and I_2 be two interpretations, OKst a relation between their input states, and*
388 *OKout a relation between their output states. If for any Σ_1 and Σ_2 such that $OKst(\Sigma_1, \Sigma_2)$ we have*

389 (1) *$[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ implies there is an O_2 such that $[\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$;*
390 (2) *$[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$ implies there is an Σ_2' such that $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$ and*
391 *$OKst(\Sigma_1', \Sigma_2')$;*

392

393 (3) $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma'_1$, *implies there is an* Σ'_2 *such that* $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow$
394 Σ'_2 *and* $OKst(\Sigma'_1, \Sigma'_2)$;
395 (4) *for any* O_1 *and* O_2 *such that* $dom(O_1) = dom(O_2) \subseteq \{1..n\}$ *and* $\forall i \in dom(O_1).OKout(O_1(i), O_2(i))$,
396 $[\![\bigoplus_n]\!]^I_V(O_1, \Sigma_1) \Downarrow \Sigma'_1$ *implies there is an* Σ'_2 *such that* $[\![\bigoplus_n]\!]^I_V(O_2, \Sigma_2) \Downarrow \Sigma'_2$ *and* $OKst(\Sigma'_1, \Sigma'_2)$;
397 *then* I_1 *and* I_2 *are existentially consistent.*
398
399 Lemma 3.5. *Let* I_1 *and* I_2 *be two interpretations, and OKst a relation between their input states. If*
400 *for any* Σ_1 *and* Σ_2 *such that* $OKst(\Sigma_1, \Sigma_2)$ *we have*
401 (1) $[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ *and* $[\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ *implies* $OKout(O_1, O_2)$;
402 (2) $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma'_1$ *and* $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma'_2$ *implies* $OKst(\Sigma'_1, \Sigma'_2)$;
403
404 (3) $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma'_1$ *and* $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma'_2$, *implies* $OKst(\Sigma'_1, \Sigma'_2)$;
405 (4) *for any* O_1, O_2 *of domain a subset of* $[1..n]$ *and such that* $\forall i \in dom(O_1) \cap dom(O_2).OKout(O_1(i), O_2(i))$,
406 $[\![\bigoplus_n]\!]^I_V(O_1, \Sigma_1) \Downarrow \Sigma'_1$ *and* $[\![\bigoplus_n]\!]^I_V(O_2, \Sigma_2) \Downarrow \Sigma'_2$ *implies* $OKst(\Sigma'_1, \Sigma'_2)$;
407 *then* I_1 *and* I_2 *are universally consistent.*
408

409 4 CONCRETE INTERPRETATION

410 We now define an interpretation used to compute a big-step evaluation semantics in the form of a
411 *triple set*: a set of triples (also called *judgements*) of the form *(state, term, result)*. For each base sort
412 we assume given a set of *base terms*, and for each flow sort a set of *values*. We write $t : s$ to state
413 that base term t has base sort s, and $v : s$ to state that value v has flow sort s.
414 For each filter $F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)$ such that $fsort(F) = (s_1..s_n) \to (s'_1..s'_m)$, we assume given
415 an interpretation $[\![F]\!]$ which is a relation between elements of $(s_1..s_n)$ and elements of $(s'_1..s'_m)$. We
416 write $[\![F]\!](v_1..v_n) \Downarrow (v'_1..v'_m)$ to state it relates $(v_1..v_n)$ to $(v'_1..v'_m)$.
417 The input state of a concrete interpretation is a pair comprising
418 - an environment Σ mapping term variables to closed terms and flow variables to values,
419 - a set T of triples of value, closed term, and value, representing already known judgements
420 and used to give meaning to the sub-derivations $H(x_{f_1}, t, x_{f_2})$.
421 The interpretation result maps term variables to closed terms and flow variables to values.
422 We define the concrete interpretation in Figure 10. For the empty skeleton body, it simply returns
423 its environment. For a hook $H(x_{f_1}, t, x_{f_2})$, it looks up in the triple set a known computation for
424 $\Sigma(x_{f_1})$ and $\Sigma(t)$ whose result is v, and it continues binding x_{f_2} to v. Note that if the language is
425 non-deterministic, there may be several such values and one is picked. For a filter F, one uses its
426 interpretation with the input $(\Sigma(x_1)..\Sigma(x_n))$. As filter interpretations are relations, there may be
427 several results as well. Finally, to merge branches, the interpretation picks a branch that successfully
428 returned a result and extends its environment accordingly.
429
430 *Running Example.* We instantiate the base sort *ident* with strings and *lit* with integers. We
431 instantiate the flow sort *int* with integers, *bool* with booleans, *val* with the disjoint union *int* + *bool*,
432 and *state* with a partial function from strings to *val*. The concrete interpretation of the filters are
433 the following partial functions: `litToVal`(i): injects i from *int* to *int* + *bool*, `read`(id, st): applies
434 st to id (since st is a partial function, it may not return a result), `isInt`(v): matches v in the
435 disjoint union *int* + *bool*, returns v if it is in *int*, `add`(i_1, i_2): returns the integer addition of i_1 and
436 i_2 injected in *int* + *bool*, `eq`(i_1, i_2): returns `true` injected in *int* + *bool* if $i_1 = i_2$, `false` injected in
437 *int* + *bool* otherwise, `isBool`(v): matches v in the disjoint union *int* + *bool*, returns v if it is in *bool*,
438 `write`(id, st, v): returns the partial function mapping id to v and any other id' to $st(id')$, `id`(st):
439 returns st, `isTrue`(b): returns () if b = `true`, `isFalse`(b): returns () if b = `false`. In the rest of the
440 paper, we directly write `x` for *var*(`x`) and n for *const*(n) in the examples.
441

442 443 444 445 446 447 448 449 450 451 452 453

$$\Longrightarrow [\![[]]\!] (\Sigma, T) \Downarrow \Sigma$$

$$\begin{pmatrix} (\Sigma(x_{f_1}), \Sigma(t), v) \in T \\ \Sigma' = \Sigma + x_{f_2} \mapsto v, T \end{pmatrix} \Longrightarrow [\![H(x_{f_1}, t, x_{f_2})]\!] (\Sigma, T) \Downarrow (\Sigma', T)$$

$$\begin{pmatrix} [\![F]\!] (\Sigma(x_1)..\Sigma(x_n)) \Downarrow (v_1..v_m) \\ \Sigma' = \Sigma + y_1 \mapsto v_1..y_m \mapsto v_m \end{pmatrix} \Longrightarrow [\![F(x_1..x_n) \mathbin{?\triangleright} (y_1..y_m)]\!] (\Sigma, T) \Downarrow (\Sigma', T)$$

$$\begin{pmatrix} \mathcal{O}(i) = \Sigma_i \\ V \subseteq dom(\Sigma_i) \\ \Sigma' = \Sigma + \Sigma_i|_V \end{pmatrix} \Longrightarrow \left[\!\!\left[\bigoplus_n \right]\!\!\right]_V (\mathcal{O}, (\Sigma, T)) \Downarrow (\Sigma', T)$$

454 Fig. 10. Concrete Interpretation

455 456

4.1 Consistency of WF and Concrete Interpretations

457 458 459 *Definition 4.1.* We say a triple set T is *well formed* if all its elements are well formed, i.e., if $(\sigma, t, v) \in T$, then $t = c(t_1..t_n)$ and there is a sort s such that $Sort(t) = s$, $\sigma : in(s)$, and $v : out(s)$.

460 461 We define $OKst((\Gamma, \mathcal{D}), (\Sigma, T))$ as follows: T is well-formed, $dom(\Gamma) = dom(\Sigma)$, and for any $x \in dom(\Gamma)$ we have $\Sigma(x) : \Gamma(x)$.

462 We define $OKout(\Gamma, \Sigma)$ as follows: $dom(\Gamma) = dom(\Sigma)$ and for any $x \in dom(\Gamma)$ we have $\Sigma(x) : \Gamma(x)$.

463 464 Lemma 4.2. *The well-formedness and concrete interpretations are universally consistent.*

465

4.2 Concrete Derivations

466 467 468 469 470 471 The concrete interpretation describes how sketelons can be interpreted from a set of hooks. The *immediate consequence* $\mathcal{H}$ describes how sketelons can be assembled. It starts from a set of well-formed triple (that is, of Hoare triples) T, and derives a new set of judgments using the concrete interpretation. Intuitively, from the set of triples generated by derivations of depth at most n, it builds the set of triples generated by derivations of depth at most $n + 1$. It is defined as follows.

472 473 474 475 476 477 478

$$\mathcal{H}(T) = \left\{ (\sigma, t, v) \left| \begin{array}{c} t = c(t_1..t_n) \land Sort(t) = s \\ \textsc{Name}(c(x_{t_1}..x_{t_n})) := S \in Rules \\ \sigma : in(s) \\ \Sigma = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n \\ [\![S]\!] (\Sigma, T) \Downarrow \Sigma' \\ \Sigma'(x_o) = v \end{array} \right. \right\}$$

479 480 Lemma 4.3. *The functional $\mathcal{H}$ is monotonic.*

481 482 Proof. This is immediate by inspecting the interpretation of skeletal bodies, as the only one where T is used is for hooks, and a bigger T does not remove results. □

483 484 Lemma 4.4. *If T is a well-formed triple set, then $\mathcal{H}(T)$ is a well-formed triple set.*

485 486 The concrete semantics $\Downarrow$ is the smallest fixpoint of $\mathcal{H}$. This corresponds to the set of triples generated by any finite derivation, or in other words, an inductive definition of the concrete rules.

487 488 *Definition 4.5.* The concrete semantics $\Downarrow$ is the smallest fixpoint of $\mathcal{H}$.

489 Lemma 4.6. *We have $\Downarrow = \bigcup_n \mathcal{H}^n(\emptyset)$.*

490

491 Proof. The set of triple sets ordered by inclusion is a CPO, and $\mathcal{H}$ is continuous on this CPO.
492 We conclude by Kleene fixpoint theorem. □
493
494 Lemma 4.7. *The concrete semantics $\Downarrow$ is well-formed.*

495 Proof. Let $(\sigma, t, v) \in \Downarrow$. By Lemma 4.6, there exists a finite number n such that $(\sigma, t, v) \in \mathcal{H}^n(\emptyset)$.
496 We prove by induction on n that (σ, t, v) has the expected properties. It is immediate for 0, and for
497 $n + 1$ we simply apply lemma 4.4. □
498

499 ## 5 ABSTRACT INTERPRETATION

500 This section describes how a set of skeletons defining a programming language can be re-interpreted
501 over an abstract domain of properties to obtain an abstract interpretation of the language. This
502 defines a program logic in which the abstract semantic derivations are built using the same inference
503 rules as the concrete semantics, but using abstract versions of the filters.
504

505 ### 5.1 Abstract Domains

506 An abstract interpretation of a set of skeletons must define abstract domains for all the terms and
507 flow sorts used in the skeleton bodies, ending with abstract semantic states and abstract results.
508 Elements in the abstract domains represent sets of values in the corresponding concrete domain
509 (they are related through the concretion function γ introduced below). The abstract interpretation
510 framework is designed to be parametric in the choice of abstract domains for base values such as
511 integers, booleans, and program states. All we require is that each abstract domain for sort s is a
512 partial order $\sqsubseteq$ with a least element, denoted $\bot_s$, representing the empty set. For example, the lattice
513 of intervals can be used as an abstract domains for integers, with $\bot_{int}$ being the empty interval.
514 Similarly, a state that maps program variables to integer values can be abstracted as a mapping from
515 variables to intervals, or as a polyhedron that defines linear relations between program variables.
516 Skeletal variables can also range over terms. For each program or base sort s, we define an
517 abstract domain by imposing a flat partial order on the set of terms of that sort (*i.e.*, we relate a term
518 to itself and no other term) and by adding a $\bot_s$ element, smaller than all terms of that sort. Abstract
519 base terms include every concrete base term, they may also include additional terms that denote
520 sets of concrete base terms. To lighten notations, we sometimes omit the sort in $\bot$ in an equality.
521 In this case, $v^\# = \bot$ should be read $v^\# = \bot_{Sort(v^\#)}$ and, $v^\# \neq \bot$ should be read $v^\# \neq \bot_{Sort(v^\#)}$.
522

523 ### 5.2 Abstract Interpretation of Skeletons

524 In addition to the abstract domains, an abstract interpretation must specify its input and output
525 states, and how the empty skeleton body, hooks, filters, and the merging of branches are interpreted.
526 For each filter symbol F of signature $(s_1..s_n) \rightarrow (s'_1..s'_m)$ we assume give a total function $[\![F]\!]^\#$ from
527 the domain corresponding to $(s_1..s_n)$ to the domain corresponding to $(s'_1..s'_m)$. A filter interpretation
528 may return $\bot$ to state it is not defined for that input.
529 The input state of an abstract interpretation is a triple $(f, \Sigma^\#, T^\#)$ comprising a *flag* f, an abstract
530 environment $\Sigma^\#$ (a mapping from skeletal variables to abstract terms and values), and a set of
531 abstract semantic triples $T^\#$ that gives a semantics to hooks. A flag is either $\bot$ or $\top$ and it indicates
532 whether it has been determined the current skeleton does not apply ($\bot$) or that it may still apply ($\top$).
533 The output state of an abstract interpretation is a flag and an abstract semantic where skeletal
534 variables hold the result of the abstract interpretation. Figure 11 defines the abstract semantics.
535 The abstract interpretation of an empty list of hypotheses just returns the flag and environment
536 from its input. There are three cases for the interpretation of a hook $H(x_{f_1}, t, x_{f_2})$. If we have
537 determined that the skeleton does not apply, we just set x_{f_2} to $\bot$ of the correct sort. In the two
538 other cases, we need to have a triple $(\sigma^\#, t^\#, v^\#)$ from $T^\#$ such that $\Sigma^\#(x_{f_2}) \sqsubseteq \sigma^\#$ and $\Sigma^\#(t) \sqsubseteq t^\#$.
539

$$\Longrightarrow \llbracket [] \rrbracket^{\#}\left(f, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(f, \Sigma^{\#}\right)$$

$$\Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \Longrightarrow \left\llbracket H\left(x_{f_1}, t, x_{f_2}\right)\right\rrbracket^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}(t) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, \bot\right) \in T^{\#} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^{\#}(t)))} \end{array}\right) \Longrightarrow \left\llbracket H\left(x_{f_1}, t, x_{f_2}\right)\right\rrbracket^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} \Sigma^{\#}\left(x_{f_1}\right) \sqsubseteq \sigma^{\#} \\ \Sigma^{\#}(t) \sqsubseteq t^{\#} \\ \left(\sigma^{\#}, t^{\#}, v^{\#}\right) \in T^{\#} \\ v^{\#} \sqsubseteq v^{\#\prime} \\ \Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto v^{\#\prime} \end{array}\right) \Longrightarrow \left\llbracket H\left(x_{f_1}, t, x_{f_2}\right)\right\rrbracket^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} fsort(F) = \left(s_1' .. s_n'\right) \to \left(s_1 .. s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1} .. y_m \mapsto \bot_{s_m} \end{array}\right) \Longrightarrow \llbracket F\left(x_1 .. x_n\right) \mathbin{?\!\triangleright} \left(y_1 .. y_m\right) \rrbracket^{\#}\left(\bot, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} \left(\Sigma^{\#}\left(x_1\right) .. \Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#} .. v_n^{\#}\right) \\ \llbracket F \rrbracket^{\#}\left(v_1^{\#} .. v_n^{\#}\right) = \bot \\ fsort(F) = \left(s_1' .. s_n'\right) \to \left(s_1 .. s_m\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto \bot_{s_1} .. y_m \mapsto \bot_{s_m} \end{array}\right) \Longrightarrow \llbracket F\left(x_1 .. x_n\right) \mathbin{?\!\triangleright} \left(y_1 .. y_m\right) \rrbracket^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} \left(\Sigma^{\#}\left(x_1\right) .. \Sigma^{\#}\left(x_n\right)\right) \sqsubseteq \left(v_1^{\#} .. v_n^{\#}\right) \\ \llbracket F \rrbracket^{\#}\left(v_1^{\#} .. v_n^{\#}\right) \sqsubseteq \left(v_1^{\#\prime} .. v_m^{\#\prime}\right) \\ \Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto v_1^{\#\prime} .. y_m \mapsto v_m^{\#\prime} \end{array}\right) \Longrightarrow \llbracket F\left(x_1 .. x_n\right) \mathbin{?\!\triangleright} \left(y_1 .. y_m\right) \rrbracket^{\#}\left(\top, \Sigma^{\#}, T^{\#}\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} n \geq 1 \\ \forall i \in [1..n]. O(i) = \left(\bot, \Sigma_i^{\#}\right) \\ \forall i \in [1..n]. V \subseteq dom\left(\Sigma_i^{\#}\right) \\ \forall i, j \in [1..n].\ \Sigma_i^{\#}\big|_V = \Sigma_j^{\#}\big|_V \\ \Sigma^{\#\prime} = \Sigma^{\#} + \Sigma_1^{\#}\big|_V \end{array}\right) \Longrightarrow \left\llbracket \bigoplus_n \right\rrbracket_V^{\#}\left(f, O, \left(\Sigma^{\#}, T^{\#}\right)\right) \Downarrow \left(\bot, \Sigma^{\#\prime}, T^{\#}\right)$$

$$\left(\begin{array}{r} dom(O) = [1..n] \\ \mathcal{E} = \left\{\Sigma_i^{\#} \middle| O(i) = \left(\top, \Sigma_i^{\#}\right)\right\} \neq \emptyset \\ \forall \Sigma_i^{\#} \in \mathcal{E}. V \subseteq dom\left(\Sigma_i^{\#}\right) \\ \Sigma_i^{\#} \in \mathcal{E} \implies \Sigma^{\#\prime} = \Sigma^{\#} + \Sigma_i^{\#}\big|_V \end{array}\right) \Longrightarrow \left\llbracket \bigoplus_n \right\rrbracket_V^{\#}\left(\top, O, \left(\Sigma^{\#}, T^{\#}\right)\right) \Downarrow \left(\top, \Sigma^{\#\prime}, T^{\#}\right)$$

Fig. 11. Abstract Interpretation

This loss of precision gives some flexibility for this derivation. We then have two (non exclusive) cases: if $v^{\#} = \bot$, then we know the skeleton does not apply, and we set the flag to $\bot$ and x_{f_2} to the appropriate $\bot$. For the last case, we do not restrict what $v^{\#}$ is (it may still be $\bot$), and we bind in the resulting environment x_{f_2} to some $v^{\#\prime}$ that may be less precise than $v^{\#}$, again to gain flexibility.

589
590
591
592
593
594
595
596
597
598
599
600

$$\mathcal{H}^\#\left(T^\#\right)=\left\{\left(\sigma^\#,t^\#,v^\#\right)\middle|\begin{array}{c}t^\#=c\left(t_1^\#..t_n^\#\right)\wedge Sort\left(t^\#\right)=s\\ \textsc{Name}\left(c\left(x_{t_1}..x_{t_n}\right)\right):=S\in Rules\\ \sigma^\#:in(s)\\ \Sigma^\#=x_\sigma\mapsto\sigma^\#+x_{t_1}\mapsto t_1^\#..x_{t_n}\mapsto t_n^\#\\ [\![S]\!]^\#\left(\top,\Sigma^\#,T^\#\right)\Downarrow\left(f,\Sigma^{\#\prime}\right)\\ \Sigma^{\#\prime}\left(x_o\right)=v^\#\end{array}\right\}$$

Fig. 12. The abstract immediate consequence operator

601 The abstract interpretation of a filter $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$ also has three cases. If we know the
602 skeleton does not apply, we just bind the output variables $(y_1..y_m)$ to the appropriate $\bot$ depending
603 on the signature of F. Otherwise, we apply the filter interpretation to an approximation of the
604 arguments as given by the environment. If the result is $\bot$, we know the skeleton does not apply
605 and switch the flag to $\bot$, as well as extend the environment with $\bot$ of the correct sort. Otherwise,
606 we keep the flag as $\top$ and extend the environment to an approximation of the result of the filter.
607 For the merging operator, we interpret every possible branch and collect their results in $\mathcal{O}$. If all
608 branching have the $\bot$ flag (either because the $\bot$ flag was set before their interpretation, which would
609 then be propagated, or because they newly returned it), then the skeleton does not apply and we set
610 the flag accordingly, extending the environment with mappings from the shared skeletal variables
611 V to $\bot$ of the correct sort. Otherwise, we collect all branches that have a $\top$ flag. They must all return
612 abstract environments that agree on the shared variables (which is why the approximations in the
613 filter and hook cases are useful, to ensure this is possible), and we extend the current environment
614 with this common environment.
615 The key difference between the abstract and concrete interpretations is how the different results
616 are merged in case of branching. The concrete semantics picks one of them, whereas the abstract
617 semantics requires all branches that provided a result to agree. This is because the goal of the
618 abstract semantics is to infer abstract semantic triples that are valid statements about all possible
619 resulting states, i.e., about all possible concrete choices in case of branching.
620

621 5.3 Consistency of WF and Abstract Interpretations

622 We define $OKst((\Gamma,\mathcal{D}),(f,\Sigma^\#,T^\#))$ as follows: $T^\#$ is well formed, $dom(\Gamma)=dom(\Sigma^\#)$, and for any
623 $x\in dom(\Gamma)$ we have $\Sigma^\#(x):\Gamma(x)$.
624 We define $OKout(\Gamma,(f,\Sigma^\#))$ as follows: $dom(\Gamma)=dom(\Sigma^\#)$ and for any $x\in dom(\Gamma)$ we have
625 $\Sigma^\#(x):\Gamma(x)$.
626
627 Lemma 5.1. *The well-formedness and abstract interpretations are universally consistent.*

628 5.4 Abstract Derivations

629
630 We define the abstract immediate consequence operator from well-formed triple sets to triple sets in
631 Figure 12. As in the concrete case, the immediate consequence describes how to assemble skeletons.
632 Lemma 5.2. *The functional $\mathcal{H}^\#$ is monotonic.*
633
634 Proof. This is immediate by inspecting the interpretation of skeletal bodies, as the only one
635 where $T^\#$ is used is for hooks, and a bigger $T^\#$ does not remove results. □
636 Lemma 5.3. *If $T^\#$ is a well-formed triple set, then $\mathcal{H}^\#(T^\#)$ is a well-formed triple set.*
637

In our setting, an abstract semantics $\Downarrow^\#$ is a set of facts of the form $(\sigma^\#, t^\#, v^\#)$ stating that from state $\sigma^\#$ term $t^\#$ evaluates to $v^\#$. A correct abstract semantics is one where such triples correspond to triples in the concrete semantics (see Section 5.5). The more facts an abstract semantics contains, the more useful it is, as it provides more information about the behaviour of terms. This is why we choose to take as abstract semantics the one with most facts, i.e., the greatest fixpoint of $\mathcal{H}^\#$. In addition, this gives us a proof technique: since the greatest fixpoint is the union of all sets such that $T^\# \subseteq \mathcal{H}^\#(T^\#)$, to prove that a fact is correct, one can propose a candidate set $T^\#$ containing this fact, then show that $T^\# \subseteq \mathcal{H}^\#(T^\#)$. This amounts to proving that the facts $T^\#$ are an invariant of the semantics. If we were to translate such invariants into a derivation, the resulting derivation may be infinite.[1]

We could also define the abstract semantics $\Downarrow^\#$ as the smallest fixpoint of $\mathcal{H}^\#$. This would be sound, but having fewer facts, we would then miss valuable abstract results. More precisely, if a triple $(\sigma^\#, t, v^\#)$ stands in the smallest fixpoint of $\mathcal{H}^\#$, then (as abstract triple sets form a CPO ordered by inclusion and $\mathcal{H}^\#$ is continuous on this CPO), there exists a finite number n such that $(\sigma^\#, t, v^\#) \in {\mathcal{H}^\#}^n(\emptyset)$. In other words, there exists a finite abstract derivation yielding the triple $(\sigma^\#, t, v^\#)$. This implies that for all concrete state $\sigma \in \gamma(\sigma^\#)$, the program t terminates. We would have thus lost all facts for which the abstract semantics cannot prove termination. Defining the abstract semantics as the greatest fixpoint of $\mathcal{H}^\#$ solves this issue.

Lemma 5.4. *$\Downarrow^\#$ is well formed.*

Proof. As $\Downarrow^\#$ is the largest fixpoint of $\mathcal{H}^\#$, it is the union of all well-formed triple sets $T^\#$ such that $T^\# \subseteq \mathcal{H}^\#(T^\#)$. Let $(\sigma^\#, t^\#, v^\#) \in \Downarrow^\#$, there is $T^\# \subseteq \mathcal{H}^\#(T^\#)$ where $(\sigma^\#, t^\#, v^\#) \in T^\#$ and $T^\#$ is well formed. Hence $(\sigma^\#, t^\#, v^\#)$ has the requested properties. □

5.5 Consistency of Concrete and Abstract Interpretations

We suppose given a *concretion function* γ for the abstract domain, from abstract terms to sets of concrete terms, and from abstract values to sets of concrete values. We impose several constraints on γ. First, γ must be compatible with $\sqsubseteq$: if $t \in \gamma(t^\#)$ and $t^\# \sqsubseteq t^{\#\prime}$, then $t \in \gamma(t^{\#\prime})$, and if $v \in \gamma(v^\#)$ and $v^\# \sqsubseteq \gamma(v^{\#\prime})$, then $v \in \gamma(v^{\#\prime})$. Second, for any abstract term $t^\#$ of sort s, the set $\gamma(t^\#)$ must only contain terms of sort s. In addition, $\gamma(c(t_1^\#..t_n^\#)) = \left\{c(t_1..t_n) \middle| t_i \in \gamma(t_i^\#)\right\}$. Conversely, for any concrete term t, we have $\gamma(t) = \{t\}$, as abstract base terms are extensions of concrete base terms.

Lemma 5.5. *Let Σ be a mapping from term variables to concrete terms, and $\Sigma^\#$ be a mapping from term variables to abstract terms. If $Tvar(t) \subseteq dom(\Sigma)$, $Tvar(t) \subseteq dom(\Sigma^\#)$, and $\forall x_t \in Tvar(t), \Sigma(x_t) \in \gamma(\Sigma^\#(x_t))$, then $\Sigma(t) \in \gamma(\Sigma^\#(t))$.*

Proof. By induction on the structure of t. If it is a base term, then the result holds by hypothesis on base terms, if it is a term variable, then the result is immediate, and otherwise we prove the property by induction on the subterms. □

Regarding values, we have similar restrictions: for any abstract value $v^\#$ of sort s, the concrete values in $\gamma(v^\#)$ all have sort s. We also require the abstract interpretation of filters to be consistent with the concrete one: if $[\![F]\!]\,(v_1..v_n) \Downarrow (v_1'..v_m')$ and $\forall i \in [1..n].v_i \in \gamma(v_i^\#)$, then $[\![F]\!]^\#\,(v_1^\#..v_n^\#) = (v_1^{\#\prime}..v_m^{\#\prime})$ and $\forall i \in [1..n].v_i' \in \gamma(v_i^{\#\prime})$. In particular, if the concrete filter relate its input to an output, the abstract filter cannot return $\bot$.

Definition 5.6. Let T a concrete triple set and $T^\#$ an abstract triple set. We say they are *consistent* if for any $(\sigma, t, v) \in T$ and $(\sigma^\#, t^\#, v^\#) \in T^\#$, if $\sigma \in \gamma(\sigma^\#)$ and $t \in \gamma(t^\#)$, then $v \in \gamma(v^\#)$.

[1] See the Figure 10 of [Schmidt 1997a] for an example of such representation.

687 We define $OKst(\Sigma, T)(f, \Sigma^\#, T^\#)$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, for any $x \in dom(\Sigma)$, we
688 have $\Sigma(x) \in \gamma(\Sigma^\#(x))$, and T and $T^\#$ are well formed and consistent.
689 We define $OKout\Sigma(f, \Sigma^\#)$ as follows: $f = \top$, $dom(\Sigma) = dom(\Sigma^\#)$, and for any $x \in dom(\Sigma)$, we
690 have $\Sigma(x) \in \gamma(\Sigma^\#(x))$.
691
692 Lemma 5.7. *The concrete and abstract interpretations are universally consistent.*
693
694 Lemma 5.8. *Let T and $T^\#$ well formed and consistent triple sets, then $\mathcal{H}(T)$ and $\mathcal{H}^\#(T^\#)$ are well*
695 *formed and consistent triple sets.*

696 We finally show that the abstract semantics is correct in relation to the concrete semantics. In a
697 nutshell, for any triple in the concrete semantics $\Downarrow$ (the smallest fixpoint of $\mathcal{H}$) and any triple in
698 the abstract semantics $\Downarrow^\#$ (the largest fixpoint of $\mathcal{H}^\#$), if the input states and terms are related, then
699 the output values are related. Formally, we have the following.
700
701 *Definition 5.9.* An abstract triple set $T^\#$ is *correct* if it is well-formed and consistent with $\Downarrow$.
702
703 Theorem 5.10. *$\Downarrow^\#$ is correct.*

704 Proof. We prove by induction on k that $\mathcal{H}^k(\emptyset)$ and $\Downarrow^\#$ are well formed and consistent. The
705 check that $\Downarrow^\#$ is well formed is simply Lemma 5.4.
706 The result is immediate for $k = 0$ since $\emptyset$ is well formed, and there is nothing else to check.
707 Let $k = n + 1$, by induction we have $\mathcal{H}^n(\emptyset)$ and $\Downarrow^\#$ are well formed and consistent. By Lemma 5.8
708 we have $\mathcal{H}^{n+1}(\emptyset)$ and $\mathcal{H}^\#(\Downarrow^\#) = \Downarrow^\#$ are well formed and consistent, as required.
709 To conclude, we apply Lemma 4.6. □
710
711 Note that in the previous theorem we only use the fact that $\Downarrow^\#$ is a fixpoint: it does not have to
712 be the greatest fixpoint.
713
714 ## 6 DERIVING PROOF TECHNIQUES FROM AN ABSTRACT SEMANTICS

715 We present in this section several proof techniques derived from an abstract semantics and instan-
716 tiated in our While language.
717
718 ### 6.1 Example: Interval analysis of While

719 To give a concrete example of an abstract interpretation we design a value analysis of the language
720 While in the style of Schmidt's Abstract Interpretation of Natural Semantics [Schmidt 1995]. We
721 have the following flow sorts in the semantic definition (cf. Figure 7): *int*, *bool*, *val*, and *state*.
722 We describe an analysis in which integers are approximated by intervals, ordered by inclusion.
723 Writing $[n, m]$ for the interval of integers between n and m (with the convention that $[n, m] = \emptyset$ if
724 $m < n$), we can define the abstract domains for each of the flow sort as follows:
725
726–727
$$int^\# = ([n, m] : n \in \mathbf{Z} \cup \{-\infty\} \wedge m \in \mathbf{Z} \cup \{+\infty\}) \qquad val^\# = int^\# \times bool^\#$$
$$bool^\# = \{\bot_{bool}, true^\#, false^\#, \top_{bool}\} \qquad state^\# = ident^\# \to val^\#$$
728
729 We abstract identifiers by themselves, thus $ident^\# = ident$, with only the trivial (reflexive) ordering.
730 The abstract domain of Booleans is (isomorphic to) the set of subsets of Booleans, ordered by inclu-
731 sion. The abstract domain of values is the defined as the Cartesian product, ordered component-wise,
732 of the abstract domain of integers and Booleans, where each component gives an approximation
733 of the concrete value, *provided* that the value is of the corresponding sort. States are represented
734 as mappings from identifiers to values, ordered pointwise. Identifiers that have not been defined
735

736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763

$$\begin{aligned}
[\![\texttt{litToVal}]\!]^\# &= \lambda(l).\ \begin{cases}([n,n],\bot_{bool}) & \text{if } l = n\\ ([n,m],\bot_{bool}) & \text{if } l = [n,m]\end{cases}\\
[\![\texttt{read}]\!]^\# &= \lambda\left(\sigma^\#,\mathsf{x}\right).\ \sigma^\#(\mathsf{x})\\
[\![\texttt{isInt}]\!]^\# &= \lambda(i,b).\ i\\
[\![\texttt{add}]\!]^\# &= \lambda(i_1,i_2).\ ([l_1+l_2,u_1+u_2],\bot_{bool}) \text{ if } i_1=[l_1,u_1] \text{ and } i_2=[l_2,u_2]\\
[\![\texttt{eq}]\!]^\# &= \lambda(i_1,i_2).\ \begin{cases} true & \text{if } i_1=[n,n]=i_2\\ false & \text{if } i_1\cap i_2=\emptyset\\ \top_{bool} & \text{otherwise}\end{cases}\\
[\![\texttt{isBool}]\!]^\# &= \lambda(i,b).\ b\\
[\![\texttt{neg}]\!]^\# &= \lambda b.\ \begin{cases} false^\# & \text{if } b = true^\#\\ true^\# & \text{if } b = false^\#\\ b & \text{otherwise}\end{cases}\\
[\![\texttt{write}]\!]^\# &= \lambda(\mathsf{x},\sigma^\#,v^\#).\ \sigma^\#\left[\mathsf{x}\leftarrow v^\#\right]\\
[\![\texttt{id}]\!]^\# &= \lambda\sigma^\#.\ \sigma^\#\\
[\![\texttt{isTrue}]\!]^\# &= \lambda b.\ \begin{cases}\bot & \text{if } b\in\left\{\bot_{bool}, false^\#\right\}\\ () & \text{otherwise}\end{cases}\\
[\![\texttt{isFalse}]\!]^\# &= \lambda b.\ \begin{cases}\bot & \text{if } b\in\left\{\bot_{bool}, true^\#\right\}\\ () & \text{otherwise}\end{cases}
\end{aligned}$$

Fig. 13. Abstract interpretation of filters

764
765
766
767
768
769
770

are mapped to the undefined value $\bot_{val^\#}$. The concretisation function γ from abstract domains to concrete domains formalises the relation between concrete and abstract values.

$$\gamma([n,m]) = \{i \mid n \le i \le m\} \qquad \gamma(i,b) = \gamma(i)\cup\gamma(b) \qquad \gamma(\top_{bool}) = \{true, false\}$$
$$\gamma(\bot_{bool}) = \emptyset \qquad \gamma\left(true^\#\right) = \{true\} \qquad \gamma\left(false^\#\right) = \{false\}$$
$$\gamma\left(\sigma^\#\right) = \left\{\Sigma \,\middle|\, \forall \mathsf{x}.\Sigma(\mathsf{x}) \in \gamma\left(\sigma^\#(\mathsf{x})\right)\right\}$$

771
772
773
774
775
776
777
778
779

The abstraction of the basic filters used in the definition of While is given in Figure 13 The definition of `litToVal` takes into account that literals in our abstract interpretation may be integers or intervals. The reader may be surprised of the definition of the abstract interpretation of the filters `isInt` and `isBool` as they respectively return an abstract integer and an abstract boolean instead of a boolean stating whether their argument can respectively be an integer and a boolean. This is correct because an abstract value that is only an integer has the shape $(i, \bot_{bool})$, and applying `isBool` to it returns $\bot_{bool}$, indicating it contains no boolean.

6.2 Abstract rules for analysing While

780
781
782
783
784

Given the instantiation of the filters used in the abstract semantic of While, we can now derive an abstract interpretation of While programs. The result of an abstract interpretation of a program is a set of abstract triples that correctly describes the program behaviour. We shall present the analysis through a set of syntax-directed inference rules for inferring such triples. For a given term

785 $c(t_1..t_n)$, we take the corresponding skeleton $\textsc{Name}(c(x_{t_1}..x_{t_n})) := S$ in the semantics and apply the
786 general abstract interpretation to the skeleton body S. This results in a series of conditions for a
787 triple to be valid that will form the hypotheses of the inference rules.
788
789 *Rule for addition.* As a first example, we derive a rule for analysing arithmetic expressions such
790 as $t_1 + t_2$. A triple $(\sigma^\#, t_1 + t_2, v^\#)$ is valid if it belongs to a fixpoint $T^\#$ of $\mathcal{H}^\#$. Unfolding definitions,

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in T^\# = \mathcal{H}^\#\left(T^\#\right) \\
\Leftrightarrow\quad & \left[\!\!\left[\begin{array}{l} H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \mathtt{isInt}\left(x_{f_1}\right) \mathrel{?\triangleright} x_{f_{1'}}; \\ H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \mathtt{isInt}\left(x_{f_2}\right) \mathrel{?\triangleright} x_{f_{2'}}; \mathsf{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\triangleright} x_o \end{array}\right]\!\!\right]^\# \left(\top, \Sigma_1^\#, T^\#\right) \Downarrow \left(f, \Sigma_o^\#\right) \\
& \quad \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 \quad \wedge \quad v^\# = \Sigma_o^\#\left(x_o\right)
\end{aligned}$$

797 For simplicity, We here make the choice of ignoring the non-$\top$-case for the flag f. In other words,
798 we are ignoring the possibility of shortcutting the abstract interpretation of the rule if a $\bot$ is found
799 during the abstract execution. We also do not include the many weakening opportunities, as it
800 significantly burdens the rules.

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow\quad & \left(\Sigma_1^\#\left(x_\sigma\right), \Sigma_1^\#\left(x_{t_1}\right), v_1^\#\right) \in T^\# \\
& \wedge \quad \left[\!\!\left[\begin{array}{l} \mathtt{isInt}\left(x_{f_1}\right) \mathrel{?\triangleright} x_{f_{1'}}; \\ H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \mathtt{isInt}\left(x_{f_2}\right) \mathrel{?\triangleright} x_{f_{2'}}; \mathsf{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\triangleright} x_o \end{array}\right]\!\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \wedge \quad \Sigma_1^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 \quad \wedge \quad \Sigma_2^\# = \Sigma_1^\# + x_{f_1} \mapsto v_1^\# \quad \wedge \quad v^\# = \Sigma_o^\#\left(x_o\right)
\end{aligned}$$

810 Interpreting the filter `isInt` makes us consider the integer projection of the abstract value $v_1^\#$.
811 We can thus rewrite the implication as follows.

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow\quad & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, b_1^\#\right) \\
& \wedge \quad \left[\!\!\left[H\left(x_\sigma, x_{t_2}, x_{f_2}\right); \mathtt{isInt}\left(x_{f_2}\right) \mathrel{?\triangleright} x_{f_{2'}}; \mathsf{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\triangleright} x_o\right]\!\!\right]^\# \left(\top, \Sigma_2^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \wedge \quad \Sigma_2^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto i_1^\# \quad \wedge \quad v^\# = \Sigma_o^\#\left(x_o\right)
\end{aligned}$$

819 We can continue unfolding the abstract interpretation of the rule. We eventually reaches the
820 following implication:

$$\begin{aligned}
& \left(\sigma^\#, t_1 + t_2, v^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow\quad & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, v_2^\#\right) \in T^\# \quad \wedge \quad v_1^\# = \left(i_1^\#, b_1^\#\right) \quad \wedge \quad v_2^\# = \left(i_2^\#, b_2^\#\right) \quad \wedge \quad v^\# = \Sigma_o^\#\left(x_o\right) \\
& \wedge \quad \left[\!\!\left[\mathsf{add}\left(x_{f_{1'}}, x_{f_{2'}}\right) \mathrel{?\triangleright} x_o\right]\!\!\right]^\# \left(\top, \Sigma_4^\#, T^\#\right) \Downarrow \left(\top, \Sigma_o^\#\right) \\
& \wedge \quad \Sigma_4^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{f_1} \mapsto v_1^\# + x_{f_{1'}} \mapsto i_1^\# + x_{f_2} \mapsto v_2^\# + x_{f_{2'}} \mapsto i_2^\# \\
\Leftarrow\quad & \left(\sigma^\#, t_1, \left(i_1^\#, b_1^\#\right)\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \left(i_2^\#, b_2^\#\right)\right) \in T^\# \quad \wedge \quad [\![add]\!]^\#\left(i_1^\#, i_2^\#\right) = v^\#
\end{aligned}$$

829 By writing $\sigma^\# \vdash t : v^\#$ for $(\sigma^\#, t, v^\#) \in T^\#$ we get the familiar rule below.

$$\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, b_1^\#\right) \qquad \sigma^\# \vdash t_2 : \left(i_2^\#, b_2^\#\right) \qquad [\![add]\!]^\#\left(i_{1'}^\#, i_{2'}^\#\right) = v^\#}{\sigma^\# \vdash t_1 + t_2 : v^\#}$$

Rule for conditionals. In the case of the addition, the structure of the skeleton was linear. We have seen that we ignored some branches (the ones triggering $\bot$), but these were not very important. We now show the example of conditionals, were branches are more visible. A triple $(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma^\#_o)$ is valid if it belongs to a fixpoint $T^\#$ of $\mathcal{H}^\#$. Unfolding definitions, and passing through the linear part of the skeleton, we get:

$$\begin{aligned}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma^\#_o\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftrightarrow\ & \left[\!\!\left[H\left(x_\sigma, x_{t_1}, x_{f_1}\right); \mathtt{isBool}\left(x_{f_1}\right)\ ?\!\triangleright\ x_{f_{1'}}; \begin{pmatrix} \mathtt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \mathtt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\!\right]^\# \left(\top, \Sigma^\#_1, T^\#\right) \Downarrow \left(f, \Sigma^\#_o\right) \\
& \quad \wedge \quad \Sigma^\#_1 = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 \quad \wedge \quad \sigma^\#_o = \Sigma^\#_o\left(x_o\right) \\
\Leftarrow\ & \left(\sigma^\#, t_1, v^\#_1\right) \in T^\# \quad \wedge \quad v^\#_1 = \left(i^\#_1, b^\#_1\right) \quad \wedge \\
& \left[\!\!\left[\begin{pmatrix} \mathtt{isTrue}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_2}, x_o\right) \\ \mathtt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right) \end{pmatrix}_{\{x_o\}} \right]\!\!\right]^\# \left(\top, \Sigma^\#_2, T^\#\right) \Downarrow \left(f, \Sigma^\#_o\right) \\
& \quad \wedge \quad \Sigma^\#_1 = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1 + x_{t_2} \mapsto t_2 + x_{t_3} \mapsto t_3 + x_{f_1} \mapsto v^\#_1 + x_{f_{1'}} \mapsto b^\#_1
\end{aligned}$$

From this stage, we continue the analysis in each of the two subbranches to build a map O representing the outputs of both branches. We consider two cases, depending on the value of $b^\#_1$.

First, if $b^\#_1$ is $\top_{bool}$. We then have both `isTrue` and `isFalse` holding on $b^\#_1$. By unfolding definitions and using weakening for the results of the two hooks, we get the following implication:

$$\begin{aligned}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma^\#_o\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow\ & \left(\sigma^\#, t_1, v^\#_1\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma^\#_2\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_3, \sigma^\#_3\right) \in T^\# \\
& \quad \wedge \quad v^\#_1 = \left(i^\#_1, \top_{bool}\right) \quad \wedge \quad \sigma^\#_2 \sqsubseteq \sigma^\#_o \quad \wedge \quad \sigma^\#_3 \sqsubseteq \sigma^\#_o
\end{aligned}$$

Using the same notations as above we can simplify this rule as below.

$$\frac{\sigma^\# \vdash t_1 : \left(i^\#_1, \top_{bool}\right) \qquad \sigma^\# \vdash t_2 : \sigma^\#_2 \qquad \sigma^\#_2 \sqsubseteq \sigma^\#_o \qquad \sigma^\# \vdash t_3 : \sigma^\#_3 \qquad \sigma^\#_3 \sqsubseteq \sigma^\#_o}{\sigma^\# \vdash \mathit{if}\ t_1\ t_2\ t_3 : \sigma^\#_o}$$

This case was really imprecise (we assumed that we got $\top_{bool}$ when evaluating the conditional's expression), but shows how our equivalent of concrete rules are merged in the abstract interpretation. We now consider a more precise case, where we could infer that the conditional's expression evaluated as $\mathit{true}^\#$. The other cases $\mathit{false}^\#$ and $\bot_{bool}$ are similar. In this case, the `isTrue` filter holds, but not `isFalse`: we can derive the judgment below when $\Sigma^\#(x_{f_{1'}}) = \mathit{true}^\#$.

$$\left[\!\!\left[\mathtt{isFalse}\left(x_{f_{1'}}\right); H\left(x_\sigma, x_{t_3}, x_o\right)\right]\!\!\right]^\# \left(\top, \Sigma^\#, T^\#\right) \Downarrow \left(\bot, \Sigma^\#_o\right)$$

Following the rules for abstract interpretation (see Figure 11), this removes the second branch from the $\mathcal{E}$ set, only leaving constraints from the first branch. We thus get the following implication, where we no longer need the weakening for the result of the hook.

$$\begin{aligned}
& \left(\sigma^\#, \mathit{if}\ t_1\ t_2\ t_3, \sigma^\#_o\right) \in \mathcal{H}^\#\left(T^\#\right) \\
\Leftarrow\ & \left(\sigma^\#, t_1, v^\#_1\right) \in T^\# \quad \wedge \quad \left(\sigma^\#, t_2, \sigma^\#_o\right) \in T^\# \quad \wedge \quad v^\#_1 = \left(i^\#_1, \mathit{true}^\#\right)
\end{aligned}$$

We can rewrite as above this implication as a rule.

$$\frac{\sigma^\# \vdash t_1 : \left(i^\#_1, \mathit{true}^\#\right) \qquad \sigma^\# \vdash t_2 : \sigma^\#_2}{\sigma^\# \vdash \mathit{if}\ t_1\ t_2\ t_3 : \sigma^\#_2}$$

883 *Rule for loops.* The skeleton for loops is close to the one for conditionals. We can similarly derive
884 abstract rules such as the ones below.

885–887
$$\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \top_{bool}\right) \qquad \sigma^\# \vdash t_2 : \sigma_2^\# \qquad \sigma_2^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\# \qquad \sigma_3^\# \sqsubseteq \sigma^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

888–890
$$\frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \mathit{true}^\#\right) \qquad \sigma^\# \vdash t_2 : \sigma_2^\# \qquad \sigma_2^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma_3^\#} \qquad \frac{\sigma^\# \vdash t_1 : \left(i_1^\#, \mathit{false}^\#\right)}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

891 We can also use the fact that any fixpoint of $\mathcal{H}^\#$ is considered valid. The following implication
892 (which we can prove in a way similar to above) is valid for any well-formed set $T^\#$.

893–897
$$\begin{array}{ll} & \left(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma_o^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \\ \Leftarrow & \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \quad \wedge \quad v_1^\# \sqsubseteq \left(i_1^\#, \top_{bool}\right) \\ & \quad \wedge \quad \left(\sigma^\#, t_2, \sigma_2^\#\right) \in T^\# \quad \wedge \quad \left(\sigma_2^\#, \mathit{while}\, t_1\, t_2, \sigma_3^\#\right) \in T^\# \quad \wedge \quad \sigma_3^\# \sqsubseteq \sigma_o^\# \quad \wedge \quad \sigma^\# \sqsubseteq \sigma_o^\# \end{array}$$

898 In particular, as the condition $v_1^\# \sqsubseteq (i_1^\#, \top_{bool})$ is vacuously true, we can weaken this implication as
899 follows (forcing all intermediate states to be the same).

900–901
$$\left(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#\right) \in \mathcal{H}^\#\left(T^\#\right) \quad \Leftarrow \quad \left(\sigma^\#, t_1, v_1^\#\right) \in T^\# \wedge \left(\sigma^\#, t_2, \sigma^\#\right) \in T^\# \wedge \left(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#\right) \in T^\#$$

902 This implication means that given any $T_0^\#$ such that $T_0^\# \subseteq \mathcal{H}^\#(T_0^\#)$, that associates t_1 in the state $\sigma^\#$
903 with a result (that is that there exists $v^\#$ such that $(\sigma^\#, t_1, v^\#) \in T_0^\#$), and such that $(\sigma^\#, t_2, \sigma^\#) \in T_0^\#$,
904 we can extend $T_0^\#$ into $T_1^\# = T_0^\# \cup \left\{(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#)\right\}$. By monotonicity of $\mathcal{H}^\#$, we get $T_0^\# \subseteq \mathcal{H}^\#(T_1^\#)$,
905 and by the above implication, we get $(\sigma^\#, \mathit{while}\, t_1\, t_2, \sigma^\#) \in \mathcal{H}^\#(T_1^\#)$. Hence, $T_1^\# \subseteq \mathcal{H}^\#(T_1^\#)$, and every
906 triple in $T_1^\#$ is correct in relation to $\Downarrow$. In other words, the following familiar rule is admissible.

907–909
$$\frac{\sigma^\# \vdash t_1 : v^\# \qquad \sigma^\# \vdash t_2 : \sigma^\#}{\sigma^\# \vdash \mathit{while}\, t_1\, t_2 : \sigma^\#}$$

910
911 ### 6.3 State Splitting

912 As another example of the use of the abstract interpretation, we show how to extend the abstract
913 semantics to obtain more precise results. Our motivating example is t: $\mathit{while}\, \neg(\mathsf{x} = 0)\, \mathsf{x} := \mathsf{x} - 1$ for
914 which we want to show that the triple $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)$ is correct (we simplify notation and
915 write n for $([n, n], \bot_{bool})$, and $[n, m]$ for $([n, m], \bot_{bool})$). Proving this is not possible as such. To see this,
916 observe that in the rule for While, the same state is used to run the expression and the statement,
917 hence the return value of the expression is not reflected in the state (it may only prevent a branch
918 from being taken). Communicating information from an expression back to a state is a non-trivial
919 problem which depends on the language considered, but we can help the abstract interpretation by
920 splitting the state in three parts: $\{(\mathsf{x} \mapsto 0, t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [1, \infty], t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)\}$.
921 Let $T^\#$ be the set of triples (listed below) obtained from adding triples for every sub-expression
922 of t We can show that $\{(\mathsf{x} \mapsto 0, t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [1, \infty], t, \mathsf{x} \mapsto 0)\} \subset \mathcal{H}^\#(T^\#)$ (the second triple uses
923 $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0)$ to evaluate the recursive while term). However there is still one of the three
924 triples that cannot be derived, *viz.*, $(\mathsf{x} \mapsto [0, \infty], t, \mathsf{x} \mapsto 0) \in \mathcal{H}^\#(T^\#)$.
925 To derive this third triple, we introduce a proof technique called *state splitting* to obtain a
926 more precise abstract semantic. The core idea of the technique is that if the state $\sigma^\#$ of a triple
927 $(\sigma^\#, t^\#, v^\#)$ is covered by the states of some triples $(\sigma_1^\#, t^\#, v^\#)..(\sigma_n^\#, t^\#, v^\#)$, in the sense that $\gamma(\sigma^\#) \subseteq$
928 $\gamma(\sigma_1^\#) \cup .. \cup \gamma(\sigma_n^\#)$, then we may use $(\sigma^\#, t^\#, v^\#)$ in the input triple set $T^\#$ of $\mathcal{H}^\#(T^\#)$ *without* having
929 to show that $(\sigma^\#, t^\#, v^\#)$ is in the resulting triple set.
930 Formally, we first define a function *Sp* from triple sets to triple sets that add such triples.
931

932 *Definition 6.1.* Let $T^\#$ an abstract triple set. We define the state splitting function $Sp(T^\#)$ as:

$$Sp\left(T^\#\right) = \left\{ \left(\sigma^\#, t^\#, v^\#\right) \middle| \begin{array}{c} \left\{\left(\sigma_1^\#, t^\#, v^\#\right) .. \left(\sigma_n^\#, t^\#, v^\#\right)\right\} \subseteq T^\# \text{ with } n \geq 1 \\ \forall i \in [1..n]. Sort\left(\sigma^\#\right) = Sort\left(\sigma_i^\#\right) \\ \gamma\left(\sigma^\#\right) \subseteq \gamma\left(\sigma_1^\#\right) \cup .. \cup \gamma\left(\sigma_n^\#\right) \end{array} \right\}$$

Lemma 6.2. *For any $T^\#$, $T^\# \subseteq Sp(T^\#)$, and Sp is monotonic.*

Lemma 6.3. *Let $T^\#$ a well-formed triple set, then $Sp(T^\#)$ is well formed.*

Proof. Let $(\sigma^\#, t^\#, v^\#) \in Sp(T^\#)$, then there is some $\sigma_1^\#, t^\#, v^\# \in T^\#$ such that $Sort(\sigma^\#) = Sort(\sigma_1^\#) = in(t^\#)$ and $Sort(v^\#) = out(t^\#)$. □

We next show that the functional $Sp(\mathcal{H}^\#(Sp(\cdot)))$ has the same consistency property as $\mathcal{H}^\#(\cdot)$.

Lemma 6.4. *Let T and $T^\#$ well formed and consistent triple sets, then $\mathcal{H}(T)$ and $Sp(\mathcal{H}^\#(Sp(T^\#)))$ are well formed and consistent triple sets.*

We finally state that the proof technique is correct.

Lemma 6.5. *Let $T^\#$ a well-formed abstract triple set. If $T^\# \subseteq \mathcal{H}^\#(Sp(T^\#))$, then $Sp(T^\#)$ is correct.*

We turn back to our example. Consider the following triple set.

$$T^\# = \left\{ \begin{array}{c} (\mathsf{x} \mapsto 0, 0, 0), (\mathsf{x} \mapsto [1,\infty], 0, 0), (\mathsf{x} \mapsto 0, -1, -1), (\mathsf{x} \mapsto [1,\infty], -1, -1), \\ (\mathsf{x} \mapsto 0, \mathsf{x}, 0), (\mathsf{x} \mapsto [1,\infty], \mathsf{x}, [1,\infty]), \\ \left(\mathsf{x} \mapsto 0, \mathsf{x} = 0, true^\#\right), \left(\mathsf{x} \mapsto [1,\infty], \mathsf{x} = 0, false^\#\right), \\ \left(\mathsf{x} \mapsto 0, \neg(\mathsf{x} = 0), false^\#\right), \left(\mathsf{x} \mapsto [1,\infty], \neg(\mathsf{x} = 0), true^\#\right), \\ (\mathsf{x} \mapsto [1,\infty], \mathsf{x} - 1, [0,\infty]), (\mathsf{x} \mapsto [1,\infty], \mathsf{x} := \mathsf{x} - 1, \mathsf{x} \mapsto [0,\infty]), \\ (\mathsf{x} \mapsto 0, t, \mathsf{x} \mapsto 0), (\mathsf{x} \mapsto [1,\infty], t, \mathsf{x} \mapsto 0) \end{array} \right\}$$

We can show that $T^\# \subseteq Sp(\mathcal{H}^\#(Sp(T^\#)))$, hence every triple of $Sp(T^\#)$ is correct, in particular $(\mathsf{x} \mapsto [0,\infty], t, \mathsf{x} \mapsto 0)$.

Note that this proof technique does not depend on the programming language considered. The difficulty is transferred to the choice of how to split the state, but as long as the splitting is correct (the added triple are covered by the existing ones), the resulting technique is sound.

7 CONSTRAINT GENERATION

As a final interpretation, we show how the abstract interpretation can be used to construct an actual program analyser. We define the analyser as an interpretation that generate data flow constraints to analyse a given program [Nielson et al. 1999].[2] Constraint-based program analysis is a well-known technique for defining analyses. We show how this technique can be lifted and defined entirely as an interpretation, by generating constraints over all the flow variables used in a semantic definition.

We first need to formalise (and extend) the standard notion of *program point*. We take a program point pp to be a list of integers denoting a position in a term. Program points form a monoid with concatenation operator · and neutral element ϵ. We define a subterm operator $t@\mathsf{pp}$ as follows.

$$t@\epsilon \triangleq t \qquad c\,(t_1..t_n)\,@k\cdot\mathsf{pp} \triangleq \begin{cases} t_k@\mathsf{pp} & \text{if } k \in [1..n] \\ \text{undefined} & \text{otherwise} \end{cases}$$

[2]We impose the technical restriction that any hook used in a skeleton can be matched to a program point of the program (closed term) t_0 under consideration. Thus constraint-based analysis of code-generating code is not considered here.

981 We assume given a function *Gent* that for a given term t_0 states the set of program points for
982 which constraints will be generated. It typically consists of the set of executable subterms of t_0. We
983 require the program points of $Gent(t_0)$ to be executable: if $\mathsf{pp} \in Gent(t_0)$, then $t_0@\mathsf{pp} = c(t_1..t_n)$.
984 Requirement 2.2 enforces the existence of a skeleton for this term.
985 We next define a partial operator PP_{t_0} that associates program points to the terms occurring in the
986 hooks of a skeleton. Formally, if $PP_{t_0}(\mathsf{pp}, \mathrm{N}, t) = \mathsf{pp}'$, then (1) skeleton N is applicable: $\mathsf{pp} \in Gent(t_0)$,
987 $t_0@\mathsf{pp} = c(t_1..t_n)$, and N is of the form $\mathrm{N}(c(x_{t_1}..x_{t_n})) := S$, (2) the hook $H(x_{f_1}, t, x_{f_2})$ occurs in S,
988 and (3) the resulting program point is part of the set of explored program points: $\mathsf{pp}' \in Gent(t_0)$
989 and $t_0@\mathsf{pp}' = (x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n)(t)$.
990 Constraints are either of the form $[x = x']$, $[x \sqsubseteq x']$, or $[x : s]$, where x and x' are variables and
991 s a sort. We generate variable names in constraints of the form $\mathsf{pp}\text{-}x$. The constraint generation
992 function *Gen* that takes a program t_0 and returns the set of constraints generated by t_0 is defined as
993
994–998
$$Gen(t_0) \triangleq \bigcup C \cup \left\{ \begin{array}{c} [\mathsf{pp}\text{-}x_\sigma : in(Sort(t_0@\mathsf{pp}))], \\ [\mathsf{pp}\text{-}x_o : out(Sort(t_0@\mathsf{pp}))], \\ \forall i \in [1..n].\ [\mathsf{pp}\text{-}x_{t_i} = t_i] \end{array} \right\} \left| \begin{array}{c} \mathsf{pp} \in Gent(t_0) \wedge t_0@\mathsf{pp} = c(t_1..t_n) \\ \mathrm{N}\left(c\left(x_{t_1}..x_{t_n}\right)\right) := S \in Rules \\ [\![S]\!]^c(\mathrm{N}, \mathsf{pp}, \emptyset) \Downarrow C \\ \mathcal{D}_\mathrm{N}(\emptyset) = \left\{x_{t_1}..x_{t_n}, x_\sigma\right\} \wedge x_o \in \mathcal{D}_\mathrm{N}(C) \end{array} \right.$$

999 For each skeleton N we define a function $\mathcal{D}_\mathrm{N}$ that maps sets of constraints to sets of skeletal
1000 variables. This is not necessary for the constraint generation but is used to prove consistency
1001 between constraints and the abstract semantics.
1002 The constraint generation interpretation of skeletons $[\![S]\!]^c$ is given in Figure 14. The rule for
1003 hooks generates constraints for connecting the input state $\mathsf{pp}'\text{-}x_\sigma$ with the flow variable holding
1004 the input state in the hook $\mathsf{pp}\text{-}x_{f_1}$, and the resulting output state of the hook with the output of
1005 the hook. Each filter comes with a constraint generation function $[\![F]\!]^c$ specific to the analysis
1006 of that filter. We require that the constraints generated for that filter agree with the abstract
1007 semantics: if $\mathcal{S}$ is a solution to the constraints $[\![F]\!]^c(\mathsf{pp}\text{-}x_1..\mathsf{pp}\text{-}x_n, \mathsf{pp}\text{-}y_1..\mathsf{pp}\text{-}y_m)$, then following
1008 holds: $[\![F]\!]^\#(\mathcal{S}(\mathsf{pp}\text{-}x_1)..\mathcal{S}(\mathsf{pp}\text{-}x_n)) \sqsubseteq (\mathcal{S}(\mathsf{pp}\text{-}y_1)..\mathcal{S}(\mathsf{pp}\text{-}y_m))$. For analysing a set of branches, we
1009 generate constraints for each branch and return the union of these constraint sets.
1010
1011 *Correctness.* A *solution* $\mathcal{S}$ of a set of constraints C is a mapping from the variables in C to abstract
values and terms such that every constraint in C holds.
1012
1013 LEMMA 7.1. *Let t_0 be a term and $\mathcal{S}$ be a solution of $Gen(t_0)$. Let $T^\#$ be defined as follows:*
1014–1018
$$T^\# = \left\{ (\sigma^\#, t, v^\#) \left| \begin{array}{c} \mathsf{pp} \in Gent(t_0) \\ t = t_0@\mathsf{pp} \\ \mathcal{S}(\mathsf{pp}\text{-}x_\sigma) = \sigma^\# \\ \mathcal{S}(\mathsf{pp}\text{-}x_o) = v^\# \end{array} \right. \right\}$$
1019 *Then $T^\#$ is well typed and $T^\# \subseteq \mathcal{H}^\#(T^\#)$.*
1020
1021 *Discussion.* The constraints we generate are *path-insensitive*: they do not capture the fact that
1022 when a filter does not hold, the rest of the skeleton does not matter. Constraints can be path-
1023 sensitive by letting the state of the interpretation be a pair consisting of a set *Stop* of constraint
1024 sets representing pathways in the skeleton that are stopped, similar to the $\perp$ flag in the abstract
1025 interpretation, and one constraint set *Run* representing all the running paths. When a filter is
1026 encountered, the *Run* set is added to *Stop* with the additional constraint that the filter returns $\perp$.
1027 The usual constraints for the filter are added to *Run*. In a nutshell, we duplicate constraints for each
1028 filter: once when it does not hold, and once when it may hold. At the end of the interpretation, the
1029

$$\implies [\![[]]\!]^c(\mathrm{N},\mathsf{pp},C) \Downarrow C$$

$$\left(\begin{array}{r} PP_{t_0}(\mathsf{pp},\mathrm{N},t) = \mathsf{pp}' \\ x_{f_1} \in \mathcal{D}_{\mathrm{N}}(C) \\ C' = C \cup \left\{\begin{array}{c}[\mathsf{pp}\text{-}x_{f_1} \sqsubseteq \mathsf{pp}'\text{-}x_\sigma], \\ [\mathsf{pp}'\text{-}x_o \sqsubseteq \mathsf{pp}\text{-}x_{f_2}]\end{array}\right\} \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup \{x_{f_2}\}\end{array}\right) \implies [\![H(x_{f_1},t,x_{f_2})]\!]^c(\mathrm{N},\mathsf{pp},C) \Downarrow (\mathrm{N},\mathsf{pp},C')$$

$$\left(\begin{array}{r} \{x_1..x_n\} \subseteq \mathcal{D}_{\mathrm{N}}(C) \\ [\![F]\!]^c\left(\begin{array}{c}\mathsf{pp}\text{-}x_1..\mathsf{pp}\text{-}x_n, \\ \mathsf{pp}\text{-}y_1..\mathsf{pp}\text{-}y_m\end{array}\right) = C_f \\ C' = C \cup C_f \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup \{y_1..y_m\}\end{array}\right) \implies [\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^c(\mathrm{N},\mathsf{pp},C) \Downarrow (\mathrm{N},\mathsf{pp},C')$$

$$\left(\begin{array}{r} i \geq 1 \\ \forall i \in [1..n].\mathcal{O}(i) = C_i \\ \forall i \in [1..n].V \subseteq \mathcal{D}_{\mathrm{N}}(C_i) \\ C' = C \cup \bigcup_{i\in[1..n]} C_i \\ \mathcal{D}_{\mathrm{N}}(C') = \mathcal{D}_{\mathrm{N}}(C) \cup V\end{array}\right) \implies \left[\!\!\left[\bigoplus_n\right]\!\!\right]_V^c(\mathcal{O},(\mathrm{N},\mathsf{pp},C)) \Downarrow (\mathrm{N},\mathsf{pp},C')$$

Fig. 14. Constraint Generation

global constraint to be satisfied is the disjunction of the *Run* constraint set with the disjunction of all constraint sets in *Stop*, each constraint set interpreted as a conjunction of its atomic constraints.

Example. Consider $t_0 = \mathit{while}\,\neg(\mathsf{x} = 0)\,\mathsf{x} := \mathsf{x} - 1$. Its executable subterms are

$$\mathit{Gent}(t_0) = \{\epsilon, 1, 1{\cdot}1, 1{\cdot}1{\cdot}1, 1{\cdot}1{\cdot}2, 2, 2{\cdot}2, 2{\cdot}2{\cdot}1, 2{\cdot}2{\cdot}2\}\,.$$

Note that the subterm x appears both as program points $1{\cdot}1{\cdot}1$ and $2{\cdot}2{\cdot}1$ of t_0. For the different filters we generate symbolic constraints that will reuse abstract filters: $[\![\mathtt{isBool}]\!]^c(x,y) = \{[y = \mathtt{isBool}(x)]\}$. A mapping $\mathcal{S}$ is then a solution of such a symbolic constraint if $\mathcal{S}(y) = [\![\mathtt{isBool}]\!]^\#(\mathcal{S}(x))$.

The definition of $\mathit{Gen}(t_0)$ generates a large number of constraints. We focus on a selection of them: those generated by the initial program point ϵ. The associated skeleton is

$$\textsc{While}(\mathit{while}\,x_{t_1}\,x_{t_2}) := [H(x_\sigma, x_{t_1}, x_{f_1})\,;\mathtt{isBool}(x_{f_1}) \mathrel{?\triangleright} x_{f_{1'}}; \ldots]\,.$$

The constraint generation will produce the constraints

$$\begin{array}{lll}[\epsilon\text{-}x_\sigma : \mathit{state}], & [\epsilon\text{-}x_\sigma = \epsilon\text{-}\textsc{While}\text{-}x_\sigma], & [\epsilon\text{-}\textsc{While}\text{-}x_{t_1} = \neg(\mathsf{x} = 0)], \\ [\epsilon\text{-}x_o : \mathit{state}], & [\epsilon\text{-}x_o = \epsilon\text{-}\textsc{While}\text{-}x_o], & [\epsilon\text{-}\textsc{While}\text{-}x_{t_1} = \mathsf{x} := \mathsf{x} - 1]\end{array}$$

as well as the constraints given by $[\![H(x_\sigma, x_{t_1}, x_{f_1}); \mathtt{isBool}(x_{f_1}) \mathrel{?\triangleright} x_{f_{1'}}; \ldots]\!]^c(\mathrm{N},\mathsf{pp},\emptyset)$. The hook case links the variable ϵ-$\textsc{While}$-x_σ to the input of x_{t_1}, which here represents $\neg(\mathsf{x} = 0)$: we have $PP_{t_0}(\epsilon, \textsc{While}, x_{t_1}) = 1$ and we thus generate the two constraints

$$[\epsilon\text{-}\textsc{While}\text{-}x_\sigma \sqsubseteq 1\text{-}x_\sigma], [1\text{-}x_o \sqsubseteq \epsilon\text{-}\textsc{While}\text{-}x_{f_1}]\,.$$

1079 The constraints on 1-x_σ and 1-x_o are generated when considering the program point 1, corresponding
1080 to the evaluation of ¬(x = 0) (corresponding to the skeleton Neg). As stated, the set of all generated
1081 constraints is large; it is provided in the supplementary material.
1082
1083

8 RELATED WORK

1084 *Rule formats for operational semantics.* Ott [Sewell et al. 2010] is a formalism for describing
1085 language semantics and type systems. Ott proposes a meta-language with a humanly readable
1086 syntax for writing semantic definitions as inference rules, and has facilities for translating these
1087 definitions into executable interpreters and specifications in proof assistants such as Coq and HOL.
1088 Lem [Mulligan et al. 2014] offers a core functional language extended with logical features from
1089 proof assistants for writing semantic models. Ott can be used to describe static type systems but
1090 neither Ott nor Lem has been used to derive program analyses.

1091 [Churchill et al. 2015] addresses the issue of the reusability operational semantics. Their approach
1092 is based on structures called *fundamental constructs*, or *funcons*, which only specify the changed parts
1093 of the state for a given construct. For instance, the funcon for *if* does not mention environments,
1094 but only the boolean part of the value which is needed. Funcons can then be combined to build a
1095 programming language. There is a connection between these funcons and our rules as they are both
1096 meant to capture the whole behaviour of a given language construct. For these funcons to make
1097 sense, they had to precisely separate each syntactic sorts in a similar way than that in Section 2.1.
1098 To the extend of our knowledge, the work on funcons has been focused on building extendable
1099 concrete semantics, and has never been used to build an abstract semantics.

1100 Views [Dinsdale-Young et al. 2013] has a concrete operational semantics for control flows, but
1101 is parameterised on the state model and basic commands. It proposes a program logic for this
1102 language, which is parameterised on the actions of the basic commands. They prove a general
1103 soundness result stating that suffices to check soundness for the basic commands. This corresponds
1104 in our framework by the fact that only simple properties on filters need to be checked.

1105 Iris [Jung et al. 2017] is parameterised by a small-step reduction relation. It proposes a logic to
1106 reason about resources. This logic is parameterised by a representation of resources in the form
1107 of an algebraic structure called "camera". Cameras comes with local properties about resources
1108 that users have to check. These local constraints yield the soundness of the Iris logic. To be used in
1109 practice, Iris requires its users to provides lemmas about weakest preconditions for each language
1110 construct. These lemmas are easy to find and prove in simple example (such as a vanilla While),
1111 but they require a deep understanding about how one reasons about the considered language.
1112 Such lemmas can be much complex to express (let alone proving) in complex languages such as
1113 JavaScript [Gardner et al. 2012]. We believe that our framework guides the proof effort by local
1114 reasoning: at each step, the abstract interpretation naturally considers every applicable branches.

1115 The $\mathbb{K}$ framework [Roşu and Şerbănuţă 2010] proposes a formalism for writing operational
1116 semantics and for constructing program verifiers directly on top of the semantic definition, as
1117 opposed to using an intermediate representation and/or a verification generator linked to a specific
1118 program logic. The semantic rules are given as rewriting rules over terms of semantic state. The $\mathbb{K}$
1119 framework has been used to write semantic definition of several real-world languages, including C,
1120 Java, and JavaScript. The program verifiers are based on matching logic [Roşu 2017], a formalism
1121 for reasoning about patterns and the set of terms that they match. A language-independent set of
1122 proof rules defines a Reachability Logic which can reason about the set of reachable states of a
1123 program. This has been instantiated to obtain program verifiers reasoning about data structures of
1124 heap-manipulating programs in C, Java, and JavaScript [Ştefănescu et al. 2016]

1125 The $\mathbb{K}$ framework has goals similar to ours: derive verifiers from operational semantics, correct
1126 by construction. A key difference is that the semantics of the $\mathbb{K}$ specification tool is complex and
1127

not clearly documented [Li and Gunter 2018]. In this work, we have focused on crystallising a general yet simple rule format for which we have developed a rich meta-theory. Our format enables a general definition of when a semantics is well-defined and provides a generic correctness theorem for the derived program verifiers that can be machine-checked in the Coq proof assistant.

Derivational abstract interpretation. Schmidt initiated the abstract interpretation of big-step operational semantics [Schmidt 1995] by showing how to abstract derivation trees (using co-induction to harness infinite derivations) and derived classical data flow and control flow analyses as abstract interpretations. Other systematic derivations of static analyses have taken small-step operational semantics as starting point. Schmidt [Schmidt 1997b] discusses the general principles for such an approach and compares small-step and big-step operational semantics as foundations for abstract interpretation. Cousot [Cousot 1999] shows how to derive static analyses for an imperative language using the principles of abstract interpretation. Midtgaard and Jensen [Midtgaard and Jensen 2008] use a similar approach for calculating control-flow analyses for functional languages from operational semantics in the form of abstract machines. Van Horn and Might [Van Horn and Might 2010, 2011] show how a series of analyses for higher-order functional languages can be derived from operational semantics formulated as abstract machines. The atomic operations of the machines are given an abstract interpretation and it is shown that the "abstract abstract machines" can simulate all the transitions of the concrete abstract machine. The abstract machines used by Van Horn and Might can be expressed in our rule format: the atomic operations correspond to our filters and the simulation result corresponds to our consistency result for concrete and abstract interpretations. The two works differ slightly in scope in that we are interested in a general semantic rule format and its meta-theory whereas Van Horn and Might are concerned with giving a systematic derivation of advanced analyses for higher-order languages with state.

Bodin *et al.* [Bodin et al. 2015] apply Schmidt's framework to a particular format (called pretty-big-step semantics [Charguéraud 2013]), a restriction of big-step operational semantics. They show how it can be given a concrete and an abstract interpretation in the setting of a While language. The soundness result for this particular format is formalised in the Coq proof assistant. The present paper generalises this work to a richer language and to a larger class of abstract domains. More precisely, the present formalism proposes a general rule format to describe many kinds of operational semantics together with a general definition of interpretation. We establish a soundness result of one interpretation with respect to another by proving simple lemmas on filters. This soundness result generalises that of [Bodin et al. 2015].

9 CONCLUSION AND FUTURE WORK

We have defined a syntax for programming language semantics, called skeletal semantics. A skeleton is a *simple* way to describe the *complete* semantic behaviour of a language construct *as a single definition.* Skeletons can be given different *generic* interpretations, independently of a particular programming language. These generic interpretation capture the essence of notions such as concrete semantics and abstract interpretation. The major advantage of skeletal semantics is that several general results about relating semantic interpretations can be established at a language-independent level once and for all. These generic results can then be instantiated to the specifics of a given programming language, thereby structuring and simplifying their proofs. In particular, our approach provides a systematic way of building abstract interpretations and their correctness proof. We have also shown how proof techniques can be derived from an abstract semantics, and how constraint-based program analyses can be obtained as an interpretation of skeletal semantics.

In this paper we have focused on proving the fundamental properties of skeletal semantics. There are a number of promising research directions to pursue in order to further develop the

1177 skeletal approach to semantics. With skeletal semantics we can capture many language constructs,
1178 including higher-order and object-oriented features. We have so far not studied how distributed
1179 and interactive computation fit into the framework, nor have we studied how the formalism scales
1180 to bigger languages. For instance, the specification of JavaScript [ECMA 2018] is written in a style
1181 where the whole behaviour of each construct of the language is a single definition, making it
1182 straightforward to express as a skeletal semantics.

1183 A distinguishing feature of skeletal semantics is that interpretations can be used to characterise
1184 several styles of semantics, independently of the language considered. In this paper we have focused
1185 on big step semantics. We plan to show how we can accommodate other flavors of semantics, such
1186 as small step operational semantics or abstract machines.

1187 Concerning proof techniques, we have shown how to add an abstract rule for state splitting.
1188 However, the proof principle used to validate this abstract rule, namely that abstract interpretation
1189 is a greatest fixpoint, is not specific to state splitting. We want thus to explore other abstract rules
1190 validated by this greatest fixpoint. In particular, we conjecture that we can use this approach to
1191 obtain a frame rule for skeletal semantics, paving the way for the integration of separation logic as
1192 an abstract interpretation. It is also possible to generate better (more precise) constraints than what
1193 we propose in Section 7, as well as constraints for other analyses, such as control flow analysis.
1194 We plan to study how the constraint generation such as that of [Palsberg 1995] for 0-CFA can be
1195 expressed in our framework.

1196 Finally, we have experimented with the mechanization of the skeletal semantics by implementing
1197 an interpretation that generates an interpreter in OCaml from a skeletal semantics. However, we can
1198 go further because our framework is simple enough to be formalised in a proof assistant such as Coq.
1199 Having a Coq formalisation of skeletal semantics would provide a number of benefits. In particular,
1200 it would assist the designer of program analyses in structuring and building machine-checkable
1201 proofs of correctness of abstract interpretations with respect to concrete semantics.

1202

1203 ## REFERENCES

1204 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt,
1205 and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT*
1206 *Symposium on Principles of Programming Languages (POPL 2014)*. San Diego, CA, USA, 87–100.
1207 Martin Bodin, Thomas Jensen, and Alan Schmitt. 2015. Certified Abstract Interpretation with Pretty-Big-Step Semantics. In
Proceedings of the 2015 ACM Conference on Certified Programs and Proofs (CPP'15). ACM, 29–40.
1208 Arthur Charguéraud. 2013. Pretty-big-step Semantics. In *European Symposium on Programming*. Springer, 41–60.
1209 Martin Churchill, Peter D Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications.
1210 In *Transactions on Aspect-Oriented Software Development XII*. Springer, 132–179.
1211 Patrick Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*. NATO ASI
Series F. IOS Press, Amsterdam.
1212 Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by
1213 Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of*
1214 *programming languages*. ACM, 238–252.
1215 Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for
1216 All Languages. In *Proc. of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications*
(OOPSLA'16). ACM, 74–91.
1217 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Composi-
1218 tional Reasoning for Concurrent Programs. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles*
1219 *of Programming Languages (POPL '13)*. ACM, 287–300.
1220 ECMA. 2018. ECMAScript 2018 Language Specification (ECMA-262, 9th edition). (June 2018). https://www.
1221 ecma-international.org/ecma-262/9.0/index.html
1222 Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. *ACM SIGPLAN Notices*
47, 1 (2012), 31–44.
1223 Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1987. A Framework for Defining Logics. In *Proceedings of the*
1224 *Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society,

1225

194–204.
Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the Ground Up. *Submitted to JFP* (2017).
Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.), Vol. 247. Springer, 22–39. https://doi.org/10.1007/BFb0039592
Liyi Li and Elsa L. Gunter. 2018. *IsaK: A Complete Semantics of* $\mathbb{K}$. Technical Report.
Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Proceedings of APLAS'08 (LNCS)*, Vol. 5356. 307–325. See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.
Jan Midtgaard and Thomas Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *SAS (LNCS)*, Vol. 5079. Springer Verlag, 347–362. https://doi.org/10.1007/978-3-540-69166-2_23
Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-world Semantics. In *Proc. of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, 175–188.
Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.
Jens Palsberg. 1995. Closure Analysis in Constraint Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 1 (Jan. 1995), 47–62.
Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings (Lecture Notes in Computer Science)*, Harald Ganzinger (Ed.), Vol. 1632. Springer, 202–206. https://doi.org/10.1007/3-540-48660-7_14
Gordon Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report FN-19. DAIMI, Aarhus University.
Grigore Roşu. 2017. Matching Logic. *Logical Methods in Computer Science* 13, 4 (December 2017), 1–61. https://doi.org/abs/1705.06312
Grigore Roşu and Traian Florin Şerbănuţă. 2010. An Overview of the $\mathbb{K}$ Semantic Framework. *Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434. https://doi.org/10.1016/j.jlap.2010.03.012
David A. Schmidt. 1995. Natural-semantics-based Abstract Interpretation (preliminary version). In *SAS*. Springer LNCS vol. 983, 1–18. https://doi.org/10.1007/3-540-60360-3_28
David A. Schmidt. 1997a. Abstract Interpretation in the Operational Semantics Hierarchy. *BRICS Report Series* 4, 2 (1997).
David A. Schmidt. 1997b. Abstract Interpretation of Small-Step Semantics. In *Proc. 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages (Springer LNCS vol. 1192)*. 76–99.
Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok StrniÅąa. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of Functional Programming* 20, 1 (2010), 71–122.
David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proc. of ACM 2010 Int. Conf. on Functional Programming (ICFP'10)*. ACM, 51–62.
David Van Horn and Matthew Might. 2011. Abstracting Abstract Machines: A Systematic Approach to Higher-order Program Analysis. *Commun. ACM* 54, 9 (Sept. 2011), 101–109.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

Skeletal Semantics and their Interpretations, Supplementary Material

ANONYMOUS AUTHOR(S)

We present the detailed proofs for the paper “Skeletal Semantics and their Interpretations”

1 PROOFS OF SECTION 3

1.1 Proof of Lemma 3.4

Lemma. Let I_1 and I_2 be two interpretations, $OKst$ a relation between their input states, and $OKout$ a relation between their output states. If for any Σ_1 and Σ_2 such that $OKst(\Sigma_1, \Sigma_2)$ we have

(1) $[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ implies there is an O_2 such that $[\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$;
(2) $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$ implies there is an Σ_2' such that $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$ and $OKst(\Sigma_1', \Sigma_2')$;
(3) $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$, implies there is an Σ_2' such that $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$ and $OKst(\Sigma_1', \Sigma_2')$;
(4) for any O_1 and O_2 such that $dom(O_1) = dom(O_2) \subseteq \{1..n\}$ and $\forall i \in dom(O_1).OKout(O_1(i), O_2(i))$, $[\![\bigoplus_n]\!]_V^I(O_1, \Sigma_1) \Downarrow \Sigma_1'$ implies there is an Σ_2' such that $[\![\bigoplus_n]\!]_V^I(O_2, \Sigma_2) \Downarrow \Sigma_2'$ and $OKst(\Sigma_1', \Sigma_2')$;

then I_1 and I_2 are existentially consistent.

Proof. We show by induction on S that if $OKst(\Sigma_1, \Sigma_2)$ and $[\![S]\!]^{I_1}(\Sigma_1) \Downarrow O_1$, then there exists a O_2 such that $[\![S]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$.

The case where $S = []$ is exactly (1).

We now turn to the case where $S = B; S'$. We thus have $[\![B]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$ and $[\![S']\!]^{I_1}(\Sigma_1') \Downarrow O_1$. If B is a hook $H(x_{f_1}, t, x_{f_2})$, we apply (2) to deduce there is a Σ_2' such that $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$ and $OKst(\Sigma_1', \Sigma_2')$. By induction on S' there exists an O_2 such that $[\![S']\!]^{I_2}(\Sigma_2) \Downarrow O_2$ and $OKout(O_1, O_2)$, and we can conclude with $[\![S']\!]^{I_2}(\Sigma_2) \Downarrow O_2$.

The case for a filter F is identical, using hypothesis (3).

We finally turn to the branches case: $S = (S_1..S_n)_V; S'$. We apply the induction hypothesis for each S_i to obtain two functions O_1 and O_2 that are thus defined on the same domain, and for which $\forall i \in dom(O_1).OKout(O_1(i), O_2(i))$. We may then apply (4) to show $\exists \Sigma_2'. [\![\bigoplus_n]\!]_V^I(O_2, \Sigma_2) \Downarrow \Sigma_2'$ and $OKst(\Sigma_1', \Sigma_2')$, and we conclude by induction with S'. □

1.2 Proof of Lemma 3.5

Lemma. Let I_1 and I_2 be two interpretations, and $OKst$ a relation between their input states. If for any Σ_1 and Σ_2 such that $OKst(\Sigma_1, \Sigma_2)$ we have

(1) $[\![[]]\!]^{I_1}(\Sigma_1) \Downarrow O_1$ and $[\![[]]\!]^{I_2}(\Sigma_2) \Downarrow O_2$ implies $OKout(O_1, O_2)$;
(2) $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$ and $[\![H(x_{f_1}, t, x_{f_2})]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$ implies $OKst(\Sigma_1', \Sigma_2')$;
(3) $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$ and $[\![F(x_1..x_n) \mathrel{?\triangleright} (y_1..y_m)]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$, implies $OKst(\Sigma_1', \Sigma_2')$;
(4) for any O_1, O_2 of domain a subset of $[1..n]$ and such that $\forall i \in dom(O_1) \cap dom(O_2).OKout(O_1(i), O_2(i))$, $[\![\bigoplus_n]\!]_V^I(O_1, \Sigma_1) \Downarrow \Sigma_1'$ and $[\![\bigoplus_n]\!]_V^I(O_2, \Sigma_2) \Downarrow \Sigma_2'$ implies $OKst(\Sigma_1', \Sigma_2')$;

then I_1 and I_2 are universally consistent.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

50 Proof. We show by induction on S that if $OKst(\Sigma_1, \Sigma_2)$, $[\![S]\!]^{I_1}(\Sigma_1) \Downarrow O_1$, and $[\![S]\!]^{I_2}(\Sigma_2) \Downarrow O_2$,
51 then $OKout(O_1, O_2)$.
52 The case where $S = []$ is exactly (1).
53 We now turn to the case where $S = B; S'$. We thus have $[\![B]\!]^{I_1}(\Sigma_1) \Downarrow \Sigma_1'$, $[\![S']\!]^{I_1}(\Sigma_1') \Downarrow O_1$,
54 $[\![B]\!]^{I_2}(\Sigma_2) \Downarrow \Sigma_2'$, and $[\![S']\!]^{I_2}(\Sigma_2') \Downarrow O_2$. If B is a hook or a filter, we apply (2) or (3) to deduce that
55 $OKst(\Sigma_1', \Sigma_2')$. We conclude by induction on S'.
56 We finally turn to the branches case: $S = (S_1..S_n)_V; S'$. For any $i \in dom(O_1) \cap dom(O_2)$, we
57 have by induction on S_i that $OKout(O_1(i), O_2)(i)$. Hence by (4) we have $OKst(\Sigma_1', \Sigma_2')$ and we can
58 conclude by induction on S'. □
59

60 ## 2 PROOF OF SECTION 4

61 ### 2.1 Proof of Lemma 4.2

62 **Lemma.** The well-formedness and concrete interpretations are universally consistent.
63

64 Proof. We check all the conditions to apply Lemma 3.5 are met.
65 Condition (1) is trivially met.
66 We turn to Condition (2). Let $s = Sort_\Gamma(t)$, $\Gamma' = \Gamma + x_{f_2} \mapsto out(s)$, $\mathcal{D}' = \mathcal{D} \cup \{x_{f_2}\}$, and $\Sigma' =$
67 $\Sigma + x_{f_2} \mapsto v$ with $(\Sigma(x_{f_1}), \Sigma(t), v) \in T$. By hypothesis of typing we have $Tvar(t) \subseteq dom(\Gamma) = dom(\Sigma)$,
68 and by the condition hypothesis we have $\Sigma(x_t) : \Gamma(x_t)$ for any $x_t \in Tvar(t)$. By Lemma 2.1 we thus
69 have $Sort_\Gamma(t) = Sort(\Sigma(t))$. As T is well-formed, we thus have $v : out(s)$, and we can thus conclude
70 that $OKst((\Gamma', \mathcal{D}'), (\Sigma', T))$.
71 We turn to condition (3). Let $\Gamma' = \Gamma + y_1 \mapsto s_1..y_m \mapsto s_m$, $\mathcal{D}' = \mathcal{D} \cup \{y_1..y_m\}$, $\Sigma' = \Sigma +$
72 $y_1 \mapsto v_1..y_m \mapsto v_m$, where $fsort(F) = (\Gamma(x_1)..\Gamma(x_n)) \rightarrow (s_1..s_m)$ and $[\![F]\!](\Sigma(x_1)..\Sigma(x_n)) \Downarrow (v_1..v_m)$.
73 Since the interpretation of F is consistent with sorting, we have $\Sigma'(y_i) : \Gamma'(s)[i]$ for any $i \in [1..m]$.
74 We can thus conclude that $OKst((\Gamma', \mathcal{D}'), (\Sigma', T))$.
75 We finally turn to condition (4), writing O_Γ and O_Σ for the two functions. As merging is defined
76 for the WF interpretation, we have $dom(O_\Gamma) = [1..n]$. As merging is defined for the concrete
77 interpretation, there is an i such that $O_\Sigma(i) = \Sigma_i$ with $dom(V) \subseteq dom(\Sigma_i)$. Let $(\Gamma_j, \mathcal{D}_j)$ be $O_\Gamma(i)$ for
78 $j \in [1..n]$, $\Gamma' = \Gamma + \Gamma_i|_V$, $\mathcal{D}' = \bigcup_j \mathcal{D}_j$, and $\Sigma' = \Sigma + \Sigma_i|_V$. We conclude from $OKout((\Gamma_i, \mathcal{D}_i), \Sigma_i)$
79 that the environments restricted to V agree, hence $OKst((\Gamma', \mathcal{D}'), (\Sigma', T))$. □
80

81 ### 2.2 Proof of Lemma 4.4

82 **Lemma.** If T is a well-formed triple set, then $\mathcal{H}(T)$ is a well-formed triple set.
83

84 Proof. Let $(\sigma, t, v) \in \mathcal{H}(T)$, then $t = c(x_{t_1}..x_{t_n})$, $Sort(t) = s$, there is a skeleton $\mathrm{N}(c(x_{t_1}..x_{t_n})) :=$
85 $S, \sigma : in(s)$, $[\![S]\!](\Sigma, T) \Downarrow (\Sigma', T)$ with $\Sigma = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n$, and $\Sigma'(x_o) = v$. We only
86 need to check that $v : out(s)$.
87 As the skeleton is well formed, we know that $[\![S]\!]^{\mathrm{wf}}(\Gamma, \mathcal{D}) \Downarrow (\Gamma', \mathcal{D}')$ and $\Gamma'(x_o) = out(s)$, with
88 $\Gamma = \{x_\sigma \mapsto in(s) + x_{t_1} \mapsto Sort(t_1)..x_{t_n} \mapsto Sort(t_n)\}$, and with $\mathcal{D} = dom(\Gamma)$. Since T is well formed,
89 and since the environments Γ and Σ match, we have $OKst((\Gamma, \mathcal{D}), (\Sigma, T))$. By Lemma 4.2, we have
90 $OKout((\Gamma', \mathcal{D}'), \Sigma')$, hence $v = \Sigma'(x_o) : \Gamma'(x_o) = out(s)$. □
91

92 ## 3 PROOF OF SECTION 5

93 ### 3.1 Proof of Lemma 5.1

94 **Lemma.** The well-formedness and abstract interpretations are universally consistent.
95

96 Proof. We check all the conditions to apply Lemma 3.5 are met.
97 Condition (1) is trivially met.
98

99 We turn to Condition (2) with $H(x_{f_1}, t, x_{f_2})$. Let $s = Sort_\Gamma(t)$, $\Gamma' = \Gamma + x_{f_2} \mapsto out(s)$, and $\mathcal{D}' =$
100 $\mathcal{D} \cup \{x_{f_2}\}$. By hypothesis of typing we have $Tvar(t) \subseteq dom(\Gamma) = dom(\Sigma^\#)$, and by the condition
101 hypothesis we have $\Sigma^\#(x_t) : \Gamma(x_t)$ for any $x_t \in Tvar(t)$. By Lemma 2.1 we thus have $Sort_\Gamma(t) =$
102 $Sort(\Sigma^\#(t))$. We now proceed by case on the abstract rule used. In the first two cases, we have
103 $\Sigma^{\#\prime} = \Sigma^\# + x_{f_2} \mapsto \bot_{out(Sort(\Sigma^\#(t)))}$, and we can immediately conclude since $\Sigma^\#(x_{f_2}) = \bot_{out(Sort(\Sigma^\#(t)))}$:
104 $out(Sort(\Sigma^\#(t))) = out(Sort_\Gamma(t)) = \Gamma(x_{f_2})$. In the third case, we have $\Sigma^\#(x_{f_1}) \sqsubseteq \sigma^\#$ and $\Sigma^\#(x_{f_1}) : \Gamma(x_{f_1})$,
105 hence $\sigma^\# : \Gamma(x_{f_1}) = in(s)$ as $\sqsubseteq$ is only defined on things of identical sorts. By the same reasoning,
106 we have $t^\# : s$. As $T^\#$ is well formed, we have $v^\# : out(s)$ hence $v^{\#\prime} : out(s)$, as required.
107 We turn to condition (3) for filter $F(x_1..x_n)$ $?\triangleright$ $(y_1..y_m)$. Let $\Gamma' = \Gamma + y_1 \mapsto s_1..y_m \mapsto s_m$,
108 $\mathcal{D}' = \mathcal{D} \cup \{y_1..y_m\}$, and $fsort(F) = (\Gamma(x_1)..\Gamma(x_n)) \to (s_1..s_m)$. As before, we proceed by case on
109 the abstract semantic rule used. The first two cases are immediate as we explicitly have $\Sigma^{\#\prime}(y_i) =$
110 $\bot_{s_i} : \Gamma'(y_i)$ for all $i \in [1..m]$. Otherwise let $(\Sigma^\#(x_1)..\Sigma^\#(x_n)) \sqsubseteq (v_1^\#..v_n^\#)$, $[\![F]\!]^\# (v_1^\#..v_n^\#) = (v_1^{\#\prime}..v_m^{\#\prime}) \sqsubseteq$
111 $(v_1^{\#\prime\prime}..v_m^{\#\prime\prime})$, and $\Sigma^{\#\prime} = \Sigma^\# + y_1 \mapsto v_1^{\#\prime\prime}..y_m \mapsto v_m^{\#\prime\prime}$. By sorting hypothesis on $\Sigma^\#$ and filter F, we
112 deduce that $v_i^{\#\prime\prime} : s_i$ for all $i \in [1..m]$, as required.
113 We finally turn to condition (4), writing $\mathcal{O}_\Gamma$ and $\mathcal{O}_{\Sigma^\#}$ for the two functions. As merging is defined
114 for the WF interpretation, we have $dom(\mathcal{O}_\Gamma) = [1..n]$. As merging is defined for the abstract
115 interpretation, we also have $dom(\mathcal{O}_{\Sigma^\#}) = [1..n]$. Hence we have $OKout(\mathcal{O}_\Gamma(i), \mathcal{O}_{\Sigma^\#}(i))$ for every
116 $i \in [1..n]$. In both cases for the abstract interpretation, there is an i such that $\Sigma^{\#\prime} = \Sigma^\# + \Sigma_i^\#|_V$ ($i = 1$
117 in the first case, otherwise choose one in $\mathcal{E} \neq \emptyset$). We immediately conclude using the hypothesis
118 $OKout(\mathcal{O}_\Gamma(i), \mathcal{O}_{\Sigma^\#}(i))$. □
119
120
3.2 Proof of Lemma 5.3

121 **Lemma.** If $T^\#$ is a well-formed triple set, then $\mathcal{H}^\#(T^\#)$ is a well-formed triple set.
122
123 Proof. Let $(\sigma^\#, t^\#, v^\#) \in \mathcal{H}^\#(T^\#)$. We have $t^\# = c(t_1^\#..t_n^\#)$, $Sort(t^\#) = s$, and $\sigma^\# : in(s)$. Let
124 $\mathrm{N}(c(x_{t_1}..x_{t_n})) := S$ the skeleton for constructor c. As it is well formed, we have $[\![S]\!]^{\mathsf{wf}} (\Gamma, \mathcal{D}) \Downarrow$
125 $(\Gamma', \mathcal{D}')$ and $\Gamma'(x_o) = out(s)$, with $\Gamma = \{x_\sigma \mapsto in(s) + x_{t_1} \mapsto Sort(t_1)..x_{t_n} \mapsto Sort(t_n)\}$ and $\mathcal{D} =$
126 $dom(\Gamma)$. Let $\Sigma^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1^\#..x_{t_n} \mapsto t_n^\#$, we also have $[\![S]\!]^\# (\top, \Sigma^\#, T^\#) \Downarrow (f, \Sigma^{\#\prime})$ with
127 $\Sigma^{\#\prime}(x_o) = v^\#$. As $\Sigma^\#(x_{t_i}) = t_i : Sort(t_i) = \Gamma(x_{t_i})$ for all $i \in [1..n]$, we have $OKst((\Gamma, \mathcal{D}), (\top, \Sigma^\#, T^\#))$.
128 By Lemma 5.1, we then have $OKout((\Gamma', \mathcal{D}'), (f, \Sigma^{\#\prime}))$, hence $v^\# = \Sigma^{\#\prime}(x_o) : \Gamma'(x_o) = out(s)$, as
129 needed. □
130
131
3.3 Proof of Lemma 5.7

132 **Lemma.** The concrete and abstract interpretations are universally consistent.
133
134 Proof. We check the hypotheses to apply Lemma 3.5.
135 Hypothesis (1) is trivially true.
136 We now turn to hypothesis (2) with $H(x_{f_1}, t, x_{f_2})$. Let $(\sigma, t', v) \in T$ such that $\Sigma(x_{f_1}) = \sigma$ and
137 $\Sigma(t) = t'$, and $(\sigma^\#, t^\#, v^\#) \in T^\#$ such that $\Sigma^\#(x_{f_1}) \sqsubseteq \sigma^\#$ and $\Sigma^\#(t) \sqsubseteq t^\#$. We have $\sigma = \Sigma(x_{f_1}) \in$
138 $\gamma(\Sigma^\#(x_{f_1}))$, hence $\sigma \in \gamma(\sigma^\#)$. As T and $T^\#$ only contain closed terms, we have $Tvar(t) \subseteq dom(\Sigma)$ and
139 $Tvar(t) \subseteq dom(\Sigma^\#)$. By hypothesis, we also have $\forall x_t \in Tvar(t), \Sigma(x_t) \in \gamma(\Sigma^\#(x_t))$. We may thus
140 apply Lemma 5.5 and conclude that $t' = \Sigma(t) \in \gamma(\Sigma^\#(t)) \subseteq \gamma(t^\#)$. By hypothesis on the triple sets,
141 we thus have $v \in \gamma(v^\#)$. This implies that $v^\# \neq \bot$, hence the only abstract rule that may apply
142 is the one where there is some $v^{\#\prime}$ such that $v^\# \sqsubseteq v^{\#\prime}$ and the resulting state is $(\top, \Sigma^{\#\prime}, T^\#)$ with
143 $\Sigma^{\#\prime} = \Sigma^\# + x_{f_2} \mapsto v^{\#\prime}$. On the concrete side we have as resulting state (Σ', T) with $\Sigma' = \Sigma + x_{f_2} \mapsto v$.
144 To conclude we need to show they are consistent, namely that $v \in \gamma(v^{\#\prime})$. This is immediate as
145 $v \in \gamma(v^\#)$ and $v^\# \sqsubseteq v^{\#\prime}$.
146
147

We now turn to hypothesis (3) with $F(x_1..x_n)\ ?\triangleright\ (y_1..y_m)$. On the concrete side we have $[\![F]\!]\,(\Sigma(x_1)..\Sigma(x_n)) \Downarrow (v'_1..v'_m)$ and the new state is (Σ', T) with $\Sigma' = \Sigma + y_1 \mapsto v'_1..y_m \mapsto v'_m$. On the abstract side, we have $(\Sigma^\#(x_1)..\Sigma^\#(x_n)) \sqsubseteq (v^\#_1..v^\#_n)$. We thus have $\Sigma(x_i) \in \gamma(v^\#_i)$ for any $i \in [1..n]$, hence $[\![F]\!]^\#\,(v^\#_1..v^\#_n) = (v^{\#\prime}_1..v^{\#\prime}_m)$ with $v'_i \in \gamma(v^{\#\prime}_i)$ for any $i \in [1..m]$ by filter consistency. In particular the result of the abstract filter is not $\bot$, hence the only rule that may apply is the one where the resulting abstract state is $(\top, \Sigma^{\#\prime}, T^\#)$ where $\Sigma^{\#\prime} = \Sigma^\# + y_1 \mapsto v^{\#\prime\prime}_1..y_m \mapsto v^{\#\prime\prime}_m$ and $(v^{\#\prime}_1..v^{\#\prime}_m) \sqsubseteq (v^{\#\prime\prime}_1..v^{\#\prime\prime}_m)$. We immediately conclude that the resulting concrete and abstract states are consistent.

We finally turn to hypothesis (4) with O and $O^\#$ such that for any $i \in dom(O) \cap dom(O^\#)$, we have $OKout(O(i), O^\#(i))$. On the concrete side, there is some i such that $O(i) = \Sigma_i$, $V \subseteq dom(\Sigma_i)$, and $\Sigma' = \Sigma + \Sigma_i|_V$. On the abstract side, we cannot have $O^\#(i) = (\bot, \Sigma^\#_i)$, since the flag is not $\top$ and we have $OKout(O(i), O^\#(i))$, hence only the second merge rule may apply and there is some $\Sigma^\#_i \in \mathcal{E}$ such that $O^\#(i) = (\top, \Sigma^\#_i)$. We thus have $\Sigma^{\#\prime} = \Sigma^\# + \Sigma^\#_i\big|_V$ and we conclude by $OKout(O(i), O^\#(i))$ that the resulting concrete and abstract state are consistent. □

3.4 Proof of Lemma 5.8

Lemma. Let T and $T^\#$ well formed and consistent triple sets, then $\mathcal{H}(T)$ and $\mathcal{H}^\#(T^\#)$ are well formed and consistent triple sets.

Proof. Let $(\sigma, t, v) \in \mathcal{H}(T)$ and $(\sigma^\#, t^\#, v^\#) \in \mathcal{H}^\#(T^\#)$ such that $\sigma \in \gamma(\sigma^\#)$ and $t \in \gamma(t^\#)$. By Lemma 4.4 (σ, t, v) is well formed, and we have $t = c(t_1..t_n)$. By Lemma 5.3, $(\sigma^\#, t^\#, v^\#)$ is well formed and we have $t^\# = c'(t^\#_1..t^\#_m)$. As $t \in \gamma(t^\#)$, we have $c = c'$, $n = m$, $\forall i \in [1..n], t_i \in \gamma(t^\#_i)$, and $Sort(t) = Sort(t^\#) = s$ for some sort s such that $Sort(\sigma) = Sort(\sigma^\#) = in(s)$. Let $\mathrm{N}(c(x_{t_1}..x_{t_n})) := S$ be the skeleton for c. We thus have $[\![S]\!]\,(\Sigma, T) \Downarrow \Sigma'$ with $\Sigma' = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n$ and $\Sigma'(x_o) = v$. We also have $[\![S]\!]^\#\,(\top, \Sigma^\#, T^\#) \Downarrow (f, \Sigma^{\#\prime})$ with $\Sigma^\# = x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t^\#_1..x_{t_n} \mapsto t^\#_n$ and $\Sigma^{\#\prime}(x_o) = v^\#$. As we have $OKst((\Sigma, T), (\top, \Sigma^\#, T^\#))$, we conclude by Lemma 5.7 that $OKout(\Sigma', (f, \Sigma^{\#\prime}))$, hence $v = \Sigma'(x_o) \in \gamma(\Sigma^{\#\prime}(x_o)) = \gamma(v^\#)$, as required. □

4 PROOFS OF SECTION 6

4.1 Proof of Lemma 6.4

Lemma. Let T and $T^\#$ well formed and consistent triple sets, then $\mathcal{H}(T)$ and $Sp(\mathcal{H}^\#(Sp(T^\#)))$ are well formed and consistent triple sets.

Proof. Let $(\sigma, t, v) \in \mathcal{H}(T)$ and $(\sigma^\#, t^\#, v^\#) \in Sp(\mathcal{H}^\#(Sp(T^\#)))$ such that $\sigma \in \gamma(\sigma^\#)$ and $t \in \gamma(t^\#)$. By Lemma 4.4 (σ, t, v) is well formed, and we have $t = c(t_1..t_n)$. By Lemmas 5.3 and 6.3, $(\sigma^\#, t^\#, v^\#)$ is well formed, thus $t^\# = c'(t^\#_1..t^\#_m)$. As $t \in \gamma(t^\#)$, we have $c = c'$, $n = m$, $\forall i \in [1..n], t_i \in \gamma(t^\#_i)$, and $Sort(t) = Sort(t^\#) = s$ for some sort s such that $Sort(\sigma) = Sort(\sigma^\#) = in(s)$.

By the definition of Sp, there are $\left\{(\sigma^\#_1, t^\#, v^\#)..(\sigma^\#_m, t^\#, v^\#)\right\} \subseteq \mathcal{H}^\#(Sp(T^\#))$ with $m \geq 1$, $\forall i \in [1..m].Sort(\sigma^\#) = Sort(\sigma^\#_i)$, and $\gamma(\sigma^\#) \subseteq \gamma(\sigma^\#_1) \cup .. \cup \gamma(\sigma^\#_m)$. As $\sigma \in \gamma(\sigma^\#)$, there is a $j \in [1..m]$ such that $\sigma \in \gamma(\sigma^\#_j)$. Let $\mathrm{N}(c(x_{t_1}..x_{t_n})) := S$ be the skeleton for c. We thus have $[\![S]\!]\,(\Sigma, T) \Downarrow \Sigma'$ with $\Sigma' = x_\sigma \mapsto \sigma + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n$ and $\Sigma'(x_o) = v$. We also have $[\![S]\!]^\#\,(\top, \Sigma^\#, Sp(T^\#)) \Downarrow (f, \Sigma^{\#\prime})$ with $\Sigma^\# = x_\sigma \mapsto \sigma^\#_j + x_{t_1} \mapsto t^\#_1..x_{t_n} \mapsto t^\#_n$ and $\Sigma^{\#\prime}(x_o) = v^\#$. We show that we have $OKst((\Sigma, T), (\top, \Sigma^\#, Sp(T^\#)))$. We only need to check that T is consistent with $Sp(T^\#)$. Let $(\sigma', t', v') \in T$ and $(\sigma^{\#\prime}, t^{\#\prime}, v^{\#\prime}) \in Sp(T^\#)$ such that $\sigma' \in \gamma(\sigma^{\#\prime})$ and $t' \in \gamma(t^{\#\prime})$. As before, there is some $(\sigma^{\#\prime}_k, t^{\#\prime}, v^{\#\prime}) \in T^\#$ such that $\sigma' \in \gamma(\sigma^{\#\prime}_k)$. As $T^\#$ is consistent with T, with thus have $v' \in \gamma(v^{\#\prime})$, as needed.

We apply Lemma 5.7 to deduce that $OKout(\Sigma', (f, \Sigma^{\#\prime}))$, hence $v = \Sigma'(x_o) \in \gamma(\Sigma^{\#\prime}(x_o)) = \gamma(v^\#)$, as required. □

197
4.2 Proof of Lemma 6.5

Lemma. Let $T^\#$ a well-formed abstract triple set. If $T^\# \subseteq \mathcal{H}^\#(Sp(T^\#))$, then $Sp(T^\#)$ is correct.

PROOF. Let $\Downarrow_s^\#$ the largest fixpoint on well-formed abstract triple sets of $Sp(\mathcal{H}^\#(Sp(\cdot)))$. We first prove by induction on k that $\mathcal{H}^k(\emptyset)$ and $\Downarrow_s^\#$ are well formed and consistent. As $\Downarrow_s^\#$ is the union of all well formed $T^\#$ such that $T^\# \subseteq Sp(\mathcal{H}^\#(Sp(T^\#)))$, it is immediately well formed.

The result is immediate for $k = 0$ since $\emptyset$ is well formed, and there is nothing else to check.

Let $k = n + 1$, by induction we have $\mathcal{H}^n(\emptyset)$ and $\Downarrow_s^\#$ are well formed and consistent. By Lemma 6.4 we have $\mathcal{H}^{n+1}(\emptyset)$ and $Sp(\mathcal{H}^\#(Sp(\Downarrow_s^\#))) = \Downarrow_s^\#$ are well formed and consistent, as required. Hence by Lemma 4.6, $\Downarrow_s^\#$ is correct.

Let $T^\# \subseteq \mathcal{H}^\#(Sp(T^\#))$. As Sp is monotonic, we have $Sp(T^\#) \subseteq Sp(\mathcal{H}^\#(Sp(T^\#)))$. Hence $Sp(T^\#) \subseteq \Downarrow_s^\#$, thus it is correct. □

5 PROOFS OF SECTION 7

5.1 Proof of Lemma 7.1

Lemma. Let t_0 be a term and $\mathcal{S}$ be a solution of $Gen(t_0)$. Let $T^\#$ be defined as follows:

$$T^\# = \left\{ (\sigma^\#, t, v^\#) \;\middle|\; \begin{array}{c} \mathsf{pp} \in Gent\,(t_0) \\ t = t_0@\mathsf{pp} \\ \mathcal{S}\,(\mathsf{pp}\text{-}x_\sigma) = \sigma^\# \\ \mathcal{S}\,(\mathsf{pp}\text{-}x_o) = v^\# \end{array} \right\}$$

Then $T^\#$ is well typed and $T^\# \subseteq \mathcal{H}^\#(T^\#)$.

PROOF. First, it is straightforward to show that $T^\#$ is well typed, because for any $\mathsf{pp} \in Gent(t_0)$, the solution must satisfy the constraints that $\mathsf{pp}.x_\sigma : in(Sort(t_0@\mathsf{pp}))$ and $\mathsf{pp}.x_o : out(Sort(t_0@\mathsf{pp}))$.

We want to show correctness by proving that $T^\# \subseteq \mathcal{H}^\#(T^\#)$. Let $(\sigma^\#, t, v^\#) \in T^\#$ where $t = c(t_1..t_n)$ and $Sort(t) = s$. We show that for rule $\mathrm{N}(cx_{t_1}..x_{t_n}) := S$, we have $[\![S]\!]^\#\,(\top, x_\sigma \mapsto \sigma^\# + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n, T^\#) \Downarrow (\top, \Sigma^{\#\prime})$ with $\Sigma^{\#\prime}(x_o) : out(s)$ and $\Sigma^{\#\prime}(x_o) = v^\#$.

By hypothesis, we have $t = t_0@\mathsf{pp}$ with $\mathsf{pp} \in Gen(t_0)$.

We define $OKst((\mathrm{N}, \mathsf{pp}, C), (f, \Sigma^\#, T^\#))$ as follows: if $C \subseteq Gen(t_0)$ then N, pp, and $T^\#$ are fixed as above, $f = \top$, $\mathcal{D}_\mathrm{N}(C) = dom(\Sigma^\#)$, $\{\mathsf{pp}\text{–}x | x \in dom(\Sigma^\#)\} \subseteq dom(C)$, $\{x_{t_1}..x_{t_n}\} \subseteq dom(\Sigma^\#)$, $\forall i \in [1..n].\Sigma^\#(x_{t_i}) = t_i$, and $\forall x \in dom(\Sigma^\#)$ we have $\Sigma^\#(x) = \mathcal{S}(\mathsf{pp}\text{–}x)$.

We define $OKout(C, (f, \Sigma^\#))$ as follows: if $C \subseteq Gen(t_0)$, then $f = \top$, $\mathcal{D}_\mathrm{N}(C) = dom(\Sigma^\#)$, $\{\mathsf{pp}\text{–}x | x \in dom(\Sigma^\#)\} \subseteq dom(C)$, and $\forall x \in dom(\Sigma^\#)$ we have $\Sigma^\#(x) = \mathcal{S}(\mathsf{pp}\text{–}x)$.

We now prove that the constraint generation is defined-consistent with the abstract interpretation by applying Lemma 3.4.

In the following we rely on the fact that if $C \subseteq Gen(t_0)$, then every constraint of C is satisfied by $\mathcal{S}$.

Property (1) is immediate by $[\![[]]\!]^\#\,(\top, \Sigma^\#, T^\#) \Downarrow (\top, \Sigma^\#)$.

In all cases we assume we have both $C \subseteq Gen(t_0)$ and $C' \subseteq Gen(t_0)$. Indeed, if we don't have $C \subseteq Gen(t_0)$, then as the set of constraints increases, we also do not have $C' \subseteq Gen(t_0)$ and we can conclude vacuously. We can also conclude vacuously if we don't have $C' \subseteq Gen(t_0)$.

We now prove property (2). We have $[\![H(x_{f_1}, t', x_{f_2})]\!]^c\,(\mathrm{N}, \mathsf{pp}, C) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$ with the following hypotheses: $PP_{t_0}(\mathsf{pp}, \mathrm{N}, t') = \mathsf{pp}'$ and $C' = C \cup \{\mathsf{pp}\text{–}x_{f_1} \sqsubseteq \mathsf{pp}'\text{-}x_\sigma, \mathsf{pp}'\text{-}x_o \sqsubseteq \mathsf{pp}\text{–}x_{f_2}\}$. By the hypotheses on the generated program points, we have $\mathsf{pp}' \in Gent(t_0)$ and $t_0@\mathsf{pp}' = \Sigma^\#(t')$. By the definition of $T^\#$, we have $(\sigma^{\#\prime}, t_0@\mathsf{pp}, v^{\#\prime}) \in T^\#$ with $\sigma^{\#\prime} = \mathcal{S}(\mathsf{pp}'\text{-}x_\sigma)$ and $v^{\#\prime} = \mathcal{S}(\mathsf{pp}'\text{-}x_o)$. As $x_{f_1} \in \mathcal{D}_\mathrm{N}(C)$, we have $x_{f_1} \in dom(\Sigma^\#)$, hence $\Sigma^\#(x_{f_1}) \sqsubseteq \mathcal{S}(\mathsf{pp}\text{–}x_{f_1}) \sqsubseteq \mathcal{S}(\mathsf{pp}'\text{-}x_\sigma) = \sigma^{\#\prime}$. We use the

abstract semantics for hooks where the result has flag $\top$, and we let $\Sigma^{\#\prime} = \Sigma^{\#} + x_{f_2} \mapsto \mathcal{S}(\mathsf{pp}\text{–}x_{f_2})$. We now verify the hypotheses for the abstract semantic are met and that $\mathit{OKst}((\mathrm{N}, \mathsf{pp}, C'), (\top, \Sigma^{\#\prime}, T^{\#}))$. Most conditions are straightforward with the exception of $v^{\#\prime} \sqsubseteq \Sigma^{\#\prime}(x_{f_2}) = \mathcal{S}(\mathsf{pp}\text{–}x_{f_2})$. This holds because $v^{\#\prime} = \mathcal{S}(\mathsf{pp}'\text{-}x_o) \sqsubseteq \mathcal{S}(\mathsf{pp}\text{–}x_{f_2}) = \Sigma^{\#\prime}(x_{f_2})$.

We now prove property (3) for filter $F(x_1..x_i) \mathrel{?\triangleright} (y_1..y_j)$. We have $[\![F(x_1..x_i) \mathrel{?\triangleright} (y_1..y_j)]\!]^c\, (\mathrm{N}, \mathsf{pp}, C) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$. As $\{x_1..x_i\} \subseteq \mathcal{D}_{\mathrm{N}}(C) = \mathit{dom}(\Sigma^{\#})$, we have $(\Sigma^{\#}(x_1)..\Sigma^{\#}(x_i)) = (\mathcal{S}(\mathsf{pp}\text{–}x_1)..\mathcal{S}(\mathsf{pp}\text{–}x_n))$. Let $(v_1^{\#\prime}..v_j^{\#\prime}) = (\mathcal{S}(\mathsf{pp}\text{–}y_1)..\mathcal{S}(\mathsf{pp}\text{–}y_j))$. We have $[\![F]\!]^{\#}\, (\mathcal{S}(\mathsf{pp}\text{–}x_1)..\mathcal{S}(\mathsf{pp}\text{–}x_i)) \sqsubseteq (v_1^{\#\prime}..v_j^{\#\prime})$, by hypothesis on the filters. Let $\Sigma^{\#\prime} = \Sigma^{\#} + y_1 \mapsto v_1^{\#\prime}..y_j \mapsto v_j^{\#\prime}$, we easily check that we can use the abstract semantic concluding with $\top$ for filters, and that $\mathit{OKst}((\mathrm{N}, \mathsf{pp}, C'), (\top, \Sigma^{\#\prime}, T^{\#}))$.

We finally prove property (4). We have $[\![\bigoplus_m]\!]_V^c\, (O, (\mathrm{N}, \mathsf{pp}, C)) \Downarrow (\mathrm{N}, \mathsf{pp}, C')$. For any $i \in [1..m]$ we have $O_1(i) = C_i$, hence O_2 is supposed to be defined on the same domain, thus $O_2(i) = (f_i, \Sigma_i^{\#})$ and $\mathit{OKout}(C_i, (f_i, \Sigma_i^{\#}))$. If C or one of the C_i is not included in $\mathit{Gen}(t_0)$, we can immediately conclude vacuously. Otherwise, we have $f_i = \top$, $V \subseteq \mathcal{D}_{\mathrm{N}}(C_i) = \mathit{dom}(\Sigma_i^{\#})$ for any i. In addition, we also have $\Sigma_i^{\#}(x) = \mathcal{S}(\mathsf{pp}\text{–}x)$ for any i and $x \in V$, hence all abstract environments agree on V. Since $i \geq 1$, let $\Sigma^{\#\prime} = \Sigma^{\#} + \Sigma_1^{\#}\big|_V$. All conditions are met to apply the abstract merge semantic and conclude.

By Lemma 3.4 we have result-consistency of constraint generation in relation to abstract interpretation. Let C' be such that $[\![S]\!]^c\, (\mathsf{pp}, \mathrm{N}, \emptyset) \Downarrow C'$. We now show that the initial state are related. We have $\mathcal{D}_{\mathrm{N}}(\emptyset) = \mathit{dom}(\Sigma^{\#})$, where $\Sigma^{\#} = x_\sigma \mapsto \sigma^{\#} + x_{t_1} \mapsto t_1..x_{t_n} \mapsto t_n, T^{\#}$, and for any $x \in \mathit{dom}(\Sigma^{\#})$, we have $\Sigma^{\#}(x) = \mathcal{S}(\mathsf{pp}\text{–}x)$ because of the constraints in $\mathit{Gen}(t_0)$ that $\mathsf{pp}\text{-}x_\sigma = \mathsf{pp}\text{–}x_\sigma$ and $\forall i \in [1..n].\mathsf{pp}\text{–}x_{t_i} = t_i$. Hence there is a $\Sigma^{\#\prime}$ such that $[\![S]\!]^{\#}\, (\top, \Sigma^{\#}, T^{\#}) \Downarrow (f, \Sigma^{\#\prime})$ and $\mathit{OKout}(C', (f, \Sigma^{\#\prime}))$, thus $f = \top$. Since we have $x_o \in \mathcal{D}_{\mathrm{N}}(C')$, we can conclude that $\Sigma^{\#\prime}(x_o) = \mathcal{S}(\mathsf{pp}\text{–}x_o) = \mathcal{S}(\mathsf{pp}\text{-}x_o) = v^{\#}$, as required.

We thus conclude that $T^{\#} \subseteq \mathcal{H}^{\#}(T^{\#})$: any solution of the generated constraints is correct. □

6 FULL EXAMPLE OF SECTION 7

Due to space reasons, it was not possible to show all the constraints generated by the example of Section 7 from the program $t_0 = \mathit{while}\, \neg(\mathsf{x} = 0)\, \mathsf{x} := \mathsf{x} - 1$. They are shown below.

$$\begin{gathered}
[\epsilon\text{-}x_\sigma : \mathit{state}] \\
[\epsilon\text{-}x_{t_1} = \neg(\mathsf{x} = 0)] \\
[\epsilon\text{-}x_\sigma \sqsubseteq 1\text{-}x_\sigma] \\
[\epsilon\text{-}x_{f_{1'}} = \mathtt{isBool}(\epsilon\text{-}x_{f_1})] \\
[\epsilon\text{-}x_\sigma \sqsubseteq 2\text{-}x_\sigma] \\
[\epsilon\text{-}x_\sigma \sqsubseteq \epsilon\text{-}x_\sigma] \\
[() = \mathtt{isFalse}(\epsilon\text{-}x_{f_{1'}})] \\
[1\text{-}x_\sigma : \mathit{state}] \\
[1\text{-}x_{t_1} = (\mathsf{x} = 0)] \\
[1{\cdot}1\text{-}x_o \sqsubseteq 1\text{-}x_{f_1}] \\
[1\text{-}x_o = \mathtt{neg}(1\text{-}x_{f_2})] \\
[1{\cdot}1\text{-}x_o : \mathit{val}] \\
[1{\cdot}1\text{-}x_{t_2} = 0] \\
[1{\cdot}1{\cdot}1\text{-}x_o \sqsubseteq 1{\cdot}1\text{-}x_{f_1}] \\
[1{\cdot}1\text{-}x_\sigma \sqsubseteq 1{\cdot}1{\cdot}2\text{-}x_\sigma] \\
[1{\cdot}1\text{-}x_{f_{2'}} = \mathtt{isInt}(1{\cdot}1\text{-}x_{f_2})] \\
[1{\cdot}1{\cdot}1\text{-}x_\sigma : \mathit{state}] \\
[1{\cdot}1{\cdot}1\text{-}x_{t_1} = \mathsf{x}] \\
[1{\cdot}1{\cdot}2\text{-}x_\sigma : \mathit{state}] \\
[1{\cdot}1{\cdot}2\text{-}x_{t_1} = 0] \\
[2\text{-}x_\sigma : \mathit{state}] \\
[2\text{-}x_{t_1} = \mathsf{x}] \\
[2\text{-}x_\sigma \sqsubseteq 2{\cdot}2\text{-}x_\sigma] \\
[2\text{-}x_o = \mathtt{write}(2\text{-}x_{t_1}, 2\text{-}x_\sigma, 2\text{-}x_{f_1})] \\
[2{\cdot}2\text{-}x_o : \mathit{val}] \\
[2{\cdot}2\text{-}x_{t_2} = -1] \\
[2{\cdot}2{\cdot}1\text{-}x_o \sqsubseteq 2{\cdot}2\text{-}x_{f_1}] \\
[2{\cdot}2\text{-}x_\sigma \sqsubseteq 2{\cdot}2{\cdot}2\text{-}x_\sigma] \\
[2{\cdot}2\text{-}x_{f_{2'}} = \mathtt{isInt}(2{\cdot}2\text{-}x_{f_2})] \\
[2{\cdot}2{\cdot}1\text{-}x_\sigma : \mathit{state}] \\
[2{\cdot}2{\cdot}1\text{-}x_{t_1} = \mathsf{x}] \\
[2{\cdot}2{\cdot}2\text{-}x_\sigma : \mathit{state}] \\
[2{\cdot}2{\cdot}2\text{-}x_{t_1} = -1]
\end{gathered}$$

$$\begin{gathered}
[\epsilon\text{-}x_o : \mathit{state}] \\
[\epsilon\text{-}x_{t_2} = \mathsf{x} := \mathsf{x} - 1] \\
[1\text{-}x_o \sqsubseteq \epsilon\text{-}x_{f_1}] \\
[() = \mathtt{isTrue}(\epsilon\text{-}x_{f_{1'}})] \\
[2\text{-}x_o \sqsubseteq \epsilon\text{-}x_{f_2}] \\
[\epsilon\text{-}x_o \sqsubseteq \epsilon\text{-}x_o] \\
[\epsilon\text{-}x_o = \mathtt{id}(\epsilon\text{-}x_\sigma)] \\
[1\text{-}x_o : \mathit{val}] \\
[1\text{-}x_\sigma \sqsubseteq 1{\cdot}1\text{-}x_\sigma] \\
[1\text{-}x_{f_2} = \mathtt{isBool}(1\text{-}x_{f_1})] \\
[1{\cdot}1\text{-}x_\sigma : \mathit{state}] \\
[1{\cdot}1\text{-}x_{t_1} = \mathsf{x}] \\
[1{\cdot}1\text{-}x_\sigma \sqsubseteq 1{\cdot}1{\cdot}1\text{-}x_\sigma] \\
[1{\cdot}1\text{-}x_{f_{1'}} = \mathtt{isInt}(1{\cdot}1\text{-}x_{f_1})] \\
[1{\cdot}1{\cdot}2\text{-}x_o \sqsubseteq 1{\cdot}1\text{-}x_{f_2}] \\
[1{\cdot}1\text{-}x_o = \mathtt{eq}(1{\cdot}1\text{-}x_{f_{1'}}, 1{\cdot}1\text{-}x_{f_{2'}})] \\
[1{\cdot}1{\cdot}1\text{-}x_o : \mathit{val}] \\
[1{\cdot}1{\cdot}1\text{-}x_o = \mathtt{read}(1{\cdot}1{\cdot}1\text{-}x_{t_1}, 1{\cdot}1{\cdot}1\text{-}x_\sigma)] \\
[1{\cdot}1{\cdot}2\text{-}x_o : \mathit{val}] \\
[1{\cdot}1{\cdot}2\text{-}x_o = \mathtt{litToVal}(1{\cdot}1{\cdot}2\text{-}x_{t_1})] \\
[2\text{-}x_o : \mathit{state}] \\
[2\text{-}x_{t_2} = (\mathsf{x} - 1)] \\
[2{\cdot}2\text{-}x_o \sqsubseteq 2\text{-}x_{f_1}] \\
[2{\cdot}2\text{-}x_\sigma : \mathit{state}] \\
[2{\cdot}2\text{-}x_{t_1} = \mathsf{x}] \\
[2{\cdot}2\text{-}x_\sigma \sqsubseteq 2{\cdot}2{\cdot}1\text{-}x_\sigma] \\
[2{\cdot}2\text{-}x_{f_{1'}} = \mathtt{isInt}(2{\cdot}2\text{-}x_{f_1})] \\
[2{\cdot}2{\cdot}2\text{-}x_o \sqsubseteq 2{\cdot}2\text{-}x_{f_2}] \\
[2{\cdot}2\text{-}x_o = \mathtt{add}(2{\cdot}2\text{-}x_{f_{1'}}, 2{\cdot}2\text{-}x_{f_{2'}})] \\
[2{\cdot}2{\cdot}1\text{-}x_o : \mathit{val}] \\
[2{\cdot}2{\cdot}1\text{-}x_o = \mathtt{read}(2{\cdot}2{\cdot}1\text{-}x_{t_1}, 2{\cdot}2{\cdot}1\text{-}x_\sigma)] \\
[2{\cdot}2{\cdot}2\text{-}x_o : \mathit{val}] \\
[2{\cdot}2{\cdot}2\text{-}x_o = \mathtt{litToVal}(2{\cdot}2{\cdot}2\text{-}x_{t_1})]
\end{gathered}$$